Thread-Pooling Architecture in a Custom-Made Game Engine

COMP 8045 – Major Project 1

JJ – A00000000 6-9-2019

Table of Contents

1 Introduction

1.1 Student Background

A Junior programmer with 4 years' experience at British Columbia Institute of Technology. I possess a number of skills that I will utilize with various programs and languages. I am a hard-worker and coupled with a positive attitude, can work well independently or cooperatively within a team environment.

1.2 Skills

- Languages: C++, C, Objective C, C#, Java, HTML, JavaScript
- Tools: Visual Studio, Net Beans, Git, Android Studio, XCode
- Editors: Unity3D, Maya, Blender
- Created multiple Video Game projects during enrollment
- Skills with developing Mobile apps on Android and Apple devices
- Considerable team-working experience

1.3 Education

British Columbia institute of Technology (BCIT) - Burnaby

- **Bachelor of Technology** **Graduating January 2019**
  - Games Development Option
- **Diploma of Technology** **Graduated 2016**
  - Computer System Digital Processing

1.4 Work Experience

- **Gentek Building Supplies** **May – November 2013, May – Sept 2014,**
  - Delivery Man / Supply Associate **April – September 2015, April – Sept 2016**
  - Dealt with customers and orders **April – September 2017, April - Present**

1.5 Project Description

The purpose of this project is to explore one of the many concepts that are commonly used within the gaming industry today. Specifically, I refer to the concept of the Thread-Pool design pattern. My objective is to use it as the base architecture to build a game engine from inception. With this design pattern, my engine will be taking advantage of modern computers multiple CPU cores and provide greater performance than currently on a basic game engine. Along-side the engine, I will be making a demonstration by using a plethora of 3D game objects to be interacting with each other in the game world.

1.6 Essential Problems

The main problem that this engine will be solving is the utilization of multiple cores on a single computer to greatly improve a games performance on higher end pcs. With some engines all calculations are completed on a single core one after another and that is good enough with smaller

games. However, as games get bigger and the number of objects in the game's world grows, a single core isn't enough to pump out the necessary calculations that need to be computed. This can lead to numerous issues on the game engines part such as frame-drops and skipped calculations.

Using the Thread-Pool Concept, the game engine will be able to utilize more of the computer's hardware to give the game engine greater computing power alleviating slow processing. We can break down heavy calculation areas that to workable chunks and complete certain calculations asynchronously. With a greater proficiency on utilizing the cores of the system we can do more functionally with games such as Artificial Intelligence, Pathfinding, Particle Effects and much more.

1.7 Goals and Objectives

The goal of this project is to create a fully functioning game engine using the Thread-Pool design pattern. Along with a physics-based environment to allow the player to play around in.

- Threaded Architecture
- Worker – Job System
- Physics Calculations and Handling
- Rendering and Input Capabilities
- Modular Game Object Design (Entity – Component – Systems)

2 Body

2.1 Background

In the project for the game's development course, our groups were tasked with creating a game engine that utilized multi-threading. Our engines had many different components (a rendering, physics, A.I and many more components) that needed to work together. Most of the engines we made used a different form of threading. These forms comprised of assigning threads to each single component. When the game demanded a certain function requiring physics, it would send a message to the appropriate thread that runs the function. This proved not to be the best way of using the threads on the computer as there was a lot of wait time in between all of the functions. However, it did provide a good introduction of how to deal with threads in an engine. The best way of handling the threads would be to create all the threads that are needed on the start of the engine. Next would be to run the threads continuously on idle and wait for a "Job" to be handed to the thread. With multiple threads working on these Jobs, the system will utilize more of the computer's hardware and increase the number of calculations that it can do.

Another way of dealing with the abundance of calculations that games need to do is to put most of them on the GPU. The GPU can do a lot of calculations rapidly and can easily surpass the CPU cores abilities. However, one of the major draw backs is that we would want to keep the calculations off the graphics card because it needs to handle all the rendering when it needs to display on the screen. If this were a simple demonstration of physics only then I could see the need to do GPU calculations. Nonetheless, I want this to be a full-fledged game engine by the end of the project time that can do a myriad of different systems, like physics, AI, networking, rendering and much more.

2.2 Project Statement and Overview

This project will be a demonstration of the capabilities of using threads in a game engine. While building this engine I will be using the Bullet 3D library to compute all of the physics calculations to update the game world. With Bullet, the engine will create and run a simple physics-based game that will use threading to update in-game objects and render them to the screen giving a realistic simulation.

2.3 Alternate Solutions

Currently there are a few different solutions that I could have used during this project instead of what I went with. Some of them are

2.3.1 Threading

Fork and Join – This method requires the computer to create threads at run-time and execute functions as they are needed. Then, they are deleted and removed from memory. Some advantages to this method are the unnecessary waste of time some threads can use up during there operation when other programs could be using the CPU. The disadvantages are having to create and destroy threads are costly when it comes to performance which overshadows the advantage.

One Thread to Each System – This method requires a thread to be placed at a system within the game engine for example one thread for rendering, another for physics, so on and so forth. The advantage to this method is having a thread always available for when a specific system function needs to be completed. The Disadvantage is that a lot of time can be wasted on systems that aren't using a thread currently and are just waiting for a task to complete. When games get very complex this can actually hinder the engine rather than help it.

Producer and Consumer – This method requires all threads to be created at the same time and given tasks to complete at any different system. The advantages are that with a great number of threads there will most likely be one available for when a quick calculation has to occur. Furthermore, Tasks can be broken down into more sizable chunks that can be computed faster. The disadvantage is that this is a very complex system that requires a great understanding of thread operation and scheduling.

Out of three the different ways of threading a system, the producer-consumer method seemed to be the best out of all of them. It's actually a hybrid version of the first two methods and provides both the advantages that they give.

2.3.2 Physics

Bullet – An expansive free physics library that has a great variety of functionality when trying to replicate physics in a game setting. It has everything that a game creator could us in their game from soft bodies (cloth, rope), rigid bodies, collision detection, collision shapes (cylinder, sphere, box) and more.

Box 2D – A free 2D physics library that can generate 2D physics-based objects in game. This library is used mostly in mobile games such as Angry Birds and small independent games like Shovel Knight. It is not as versatile as Bullet but still has the base needs for a 2D physics game. My original idea for this project was going to be a 2d physics game using this library but I decided to go with bullet 3D because I thought that 3D environment would provide for more complex tasks.

Creating my own – Another solution would be to create my own physics library where I could control every aspect. However, I really wanted to focus most of my attention with the threading in the engine rather than pouring a lot of time into building a physics engine where many have already been created. Furthermore, a lot of engine have many years of optimizations for handling on a wide variety of computers. Something that I would have to work on beyond the project limitations. It would be an interesting idea to work on after the project though.

2.4 Chosen Solution

The base of the project will be written in C++ code because it is a powerful language that many libraries support including the bullet and SDL libraries. It is also one of the most widely used languages in the world right now and used in large number of projects to this day. I have had extensive practice with this language and a great opportunity to learn even more about this language.

On the Threading side of the project, I have chosen to go with the Producer and Consumer methodology because it is the most ideal way of dealing with multi-threading and utilizing as much computing power that multi-core systems. However, it is also the most complex of all the multi-threading methods as it requires a lot more than just a base knowledge of threading. Which is perfect for the project that I am working on.

SDL2 is a development library that provides access to many different types of input such as joystick, keyboard and mouse, and game controllers. I will be using this library for input but only for the keyboard and mouse. Additionally, SDL2 also comes with some render capabilities through the use of OpenGL. The graphics library, OpenGL, is used in many projects and applications, and fully supports both 2D and 3D rendering to computers.

Physics will be handled by the library Bullet 3D because of the amount of functionality and optimizations that it has built into it. Furthermore, with the time constraints I can't be used on making a whole physics system by myself that is a whole project in of itself.

2.5 Design Details

2.5.1 Deliverables

Thread Pool Architecture

A fully functioning Job system that multiple threads can use to run functions.

Game Engine

Working game engine that renders, responds and plays.

Final Report

A final report on the engine and the development from the project.

2.5.2 Feasibility Assessment Report

Many big video game companies make use of multi-threading to enhance their games and to better utilize a computers hardware. With better usage of the computers CPU this engine could really be something that could create a great game. So, this project is more of an exploration of what it is like to make a big budget game engine.

2.5.3 Use Case Diagram

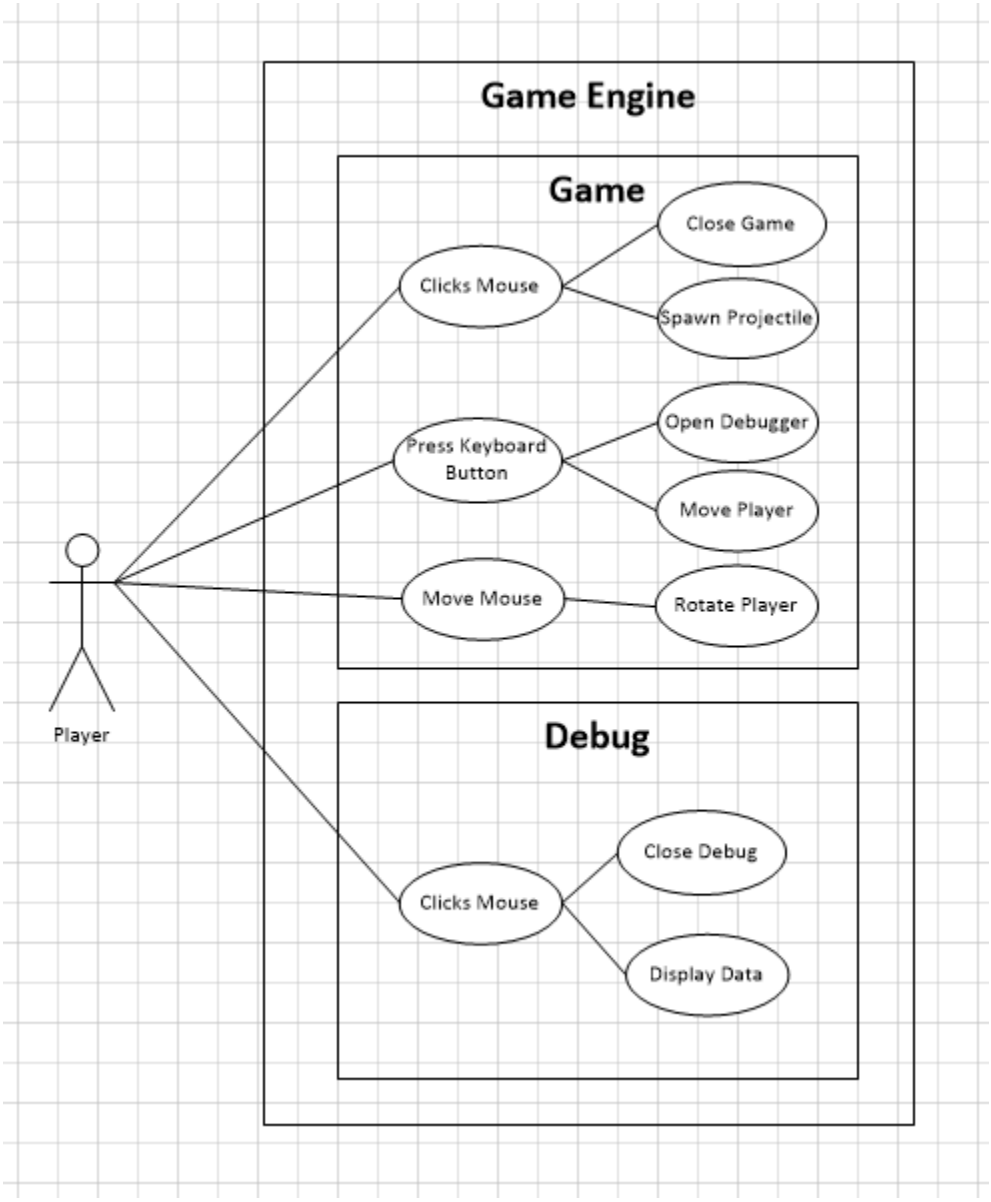


Figure 1

My Engine doesn't interact with the player too much as it is more of a test of the threading architecture with a basic game attached to it. I will be explaining the controls in the Instruction and User Manuals section of the report. However, some of the use cases listed are:

- While playing game
    - Player presses button on keyboard.
        - If movement keys, then move player object in game.

        - Else if Debug button, then open debug window.
        - Else if close button, then close game.
        - Else if pause button, then pause game.
    - Player moves mouse.
        - Update the players rotation to look in direction player wants to face.
    - Player clicks mouse button.
        - Spawn projectile
            - Which includes loading creating physics and render components and registering them to corresponding systems.
            - Adjust velocity and position based on where player is looking.
        - If close button
            - Close the game
- While in debug window
    - Player clicks mouse button
        - Check if player clicked within boundaries of data sections
            - Display data at top of screen
            - If not reset data
        - If player clicks close
            - Close debug window
    - Player presses button on keyboard
        - Exit debugger

2.5.4 Sequence Diagrams

2.5.4.1 Loading Game Diagram

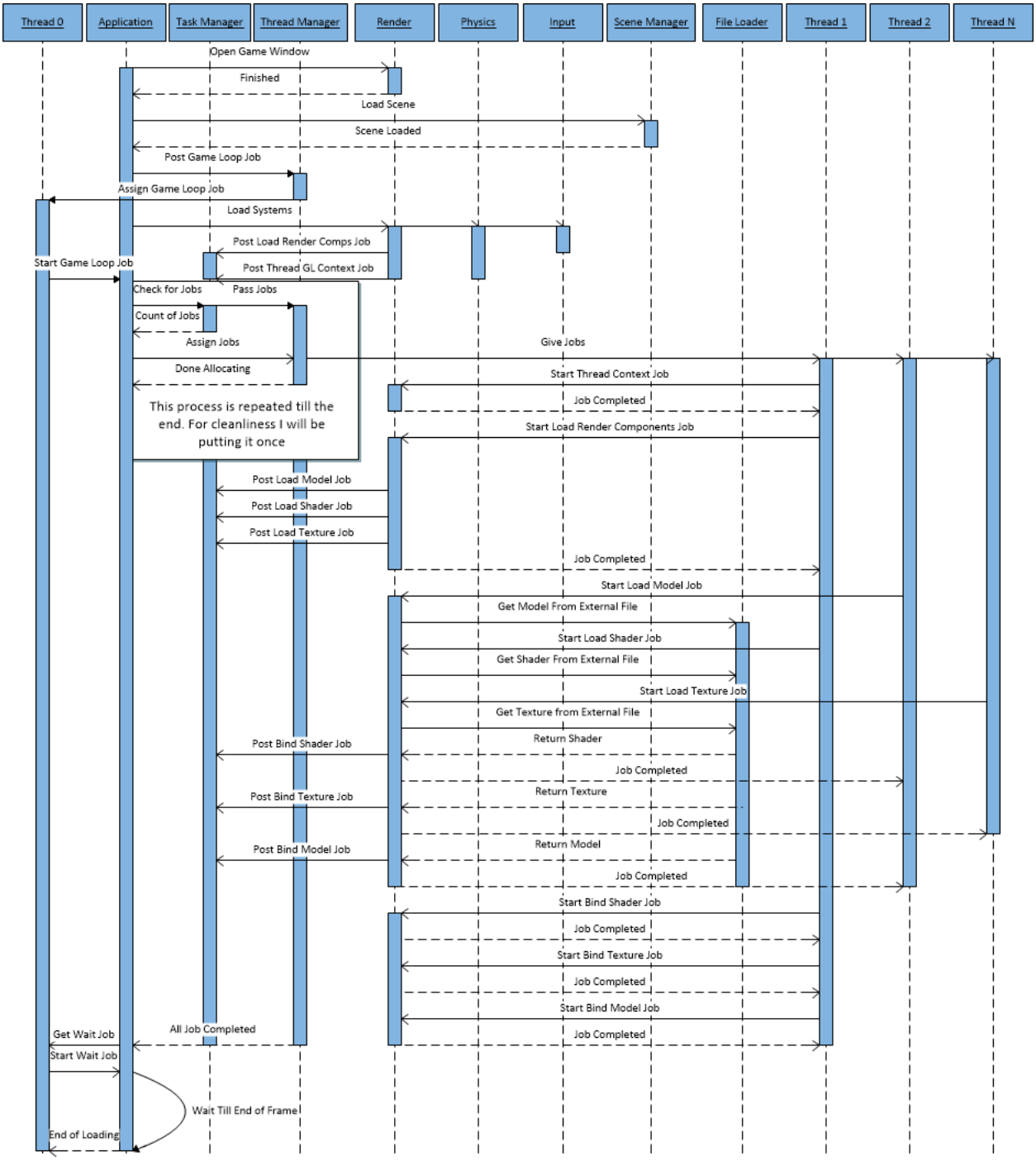


Figure 2

2.5.4.2 Playing Game Diagram

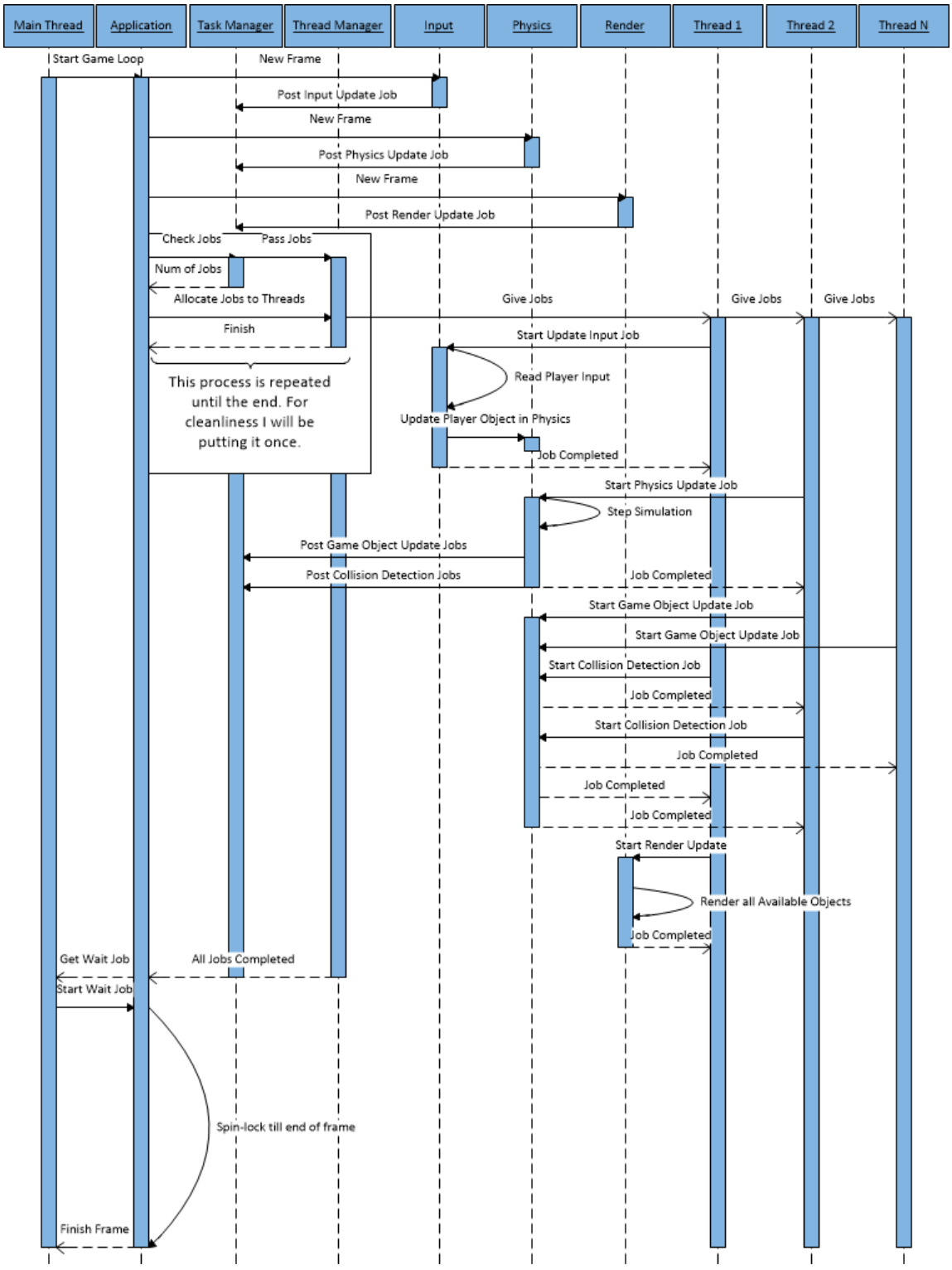


Figure 3

I have included both the loading and playing sequence in order to show the utilization of threads that I do with my game engine. It does look a bit convoluted, however it is hard to show an asynchronous design pattern on a purely linear diagram. A lot of the tasks in the diagrams will overlap and calculate at the same time.

An issue that I found with SDL only happens when a window is created. The Render System needs to create a context for the OpenGL library in order to render the graphics. However, only the thread that creates the context can handle it. This means that whenever the engine is using OpenGL code it is restricted to a single thread (Thread 1, in Figure 3). We can still multi-thread certain tasks that don't require OpenGL code like Loading Models and Loading Textures, but tasks like Bind Model and Bind Texture that initialize data on the OpenGL library can only be done on a single thread.

When my engine is creating the threads, it creates one more thread than what is called for. For example, if a computer has 8 cores in the hardware then my computer will be using 7 threads (number of cores minus one). The main reason for doing this is because windows actually halts the main thread (the one that created the application) when the window is dragged around. Furthermore, windows doesn't allow the main thread to run until the window is released which means a complete stop to my program. In order to avoid this problem, my engine runs all of the main functionality on a separate thread completely detached from the main thread. This allows the game to render and calculate even while the main thread is being disrupted. So, the thread object that you see at the far left in the diagram, which I will be calling Alternate Thread for simplicity, (Figure 2 + Figure 3) is actually a detached thread separate from the main application thread.

Tasks that can be completed on other threads are referred to as "Jobs". Whenever a function needs to be done, the engine will post a Job to the task manager. Then, it is processed either given to the thread manager or kept in the task manager. When a Job is kept in the task manager, it is going to be waiting for another job to finish in order to start itself. For example, the engine needs to update the Input system (Figure 3) and needs to update the transform with the new data then pass it in to the Physics simulation in order to calculate the simulation correctly. Therefore, an Input Update Job should happen before the Physics Update Job.

Alternate thread will be going through the task manager and thread manager to organize and distribute jobs to the other threads in the program. It will be doing this continuously throughout the whole program. Whenever a job is posted to the task manager it is then handled by the alternate thread and distributed.

The sequence diagram (Figure 3) shows what a single frame of what my engine will be doing while the game is playing. The time it takes for a single frame to complete is dependant on the refresh rate of the computer it runs on. For my computer, the engine runs at 60 frames per second (FPS) while other computers may run it at 30 FPS. Which is 16.667 milliseconds and 33.333 milliseconds respectively.

2.5.5 Class Diagram

2.5.5.1 Application, Scene & Event Architecture

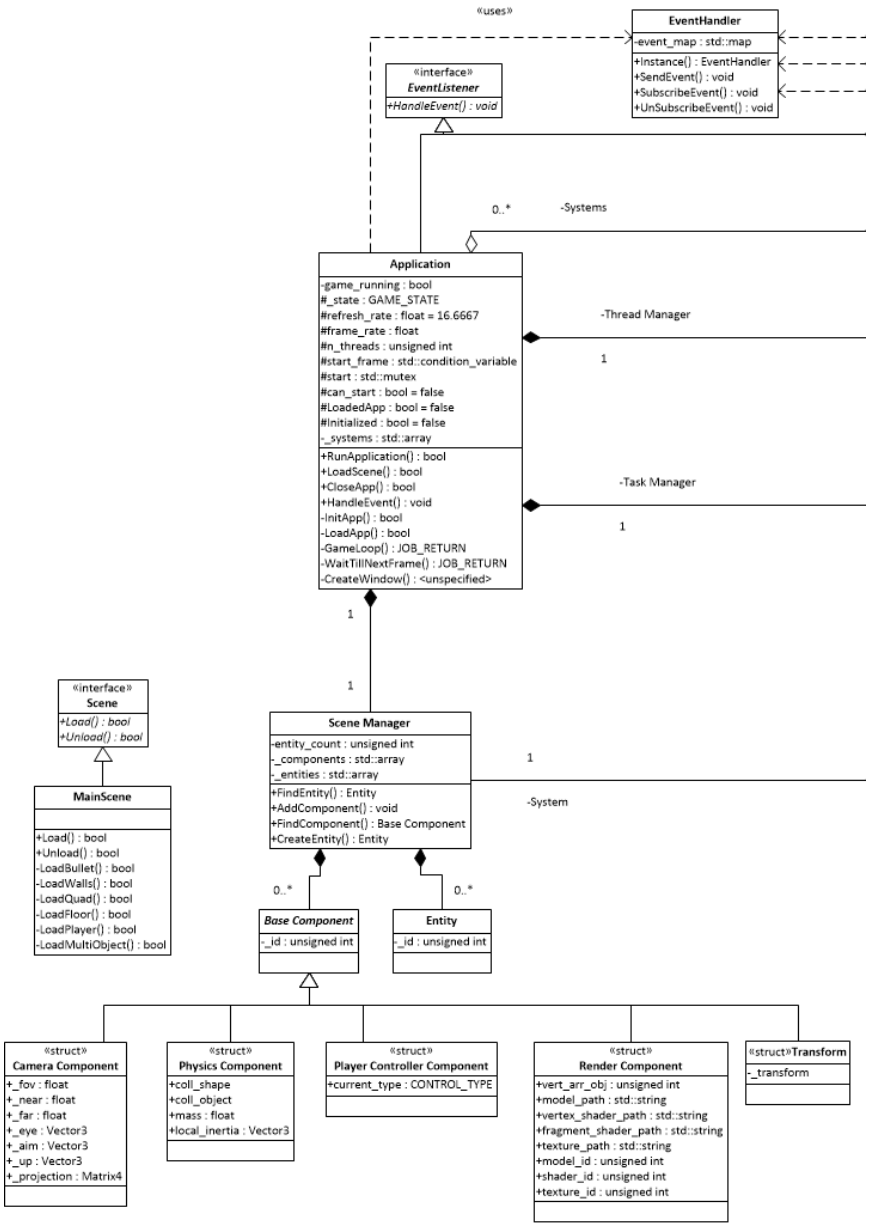


Figure 4

This is the class diagram for the base application, event system and the scene interface. With all of the class diagrams, I tried to make it as tidy and legible as possible. Any lines that are going off the picture are a connection with another class in another figure below. I tried to fit everything on one page unfortunately the program is too big for it. Now to go into the details of the class diagram. The diagram above (Figure 4) has:

- *Application*: This class is the backbone of the whole program and it stores most of the main systems that the engine will be using. It has a list for all the systems (Render, Physics and Input) and the managers (Task, Thread, Scene). This class initializes and loads all of the game data when the game starts. However, some levels in the game may only need certain systems. For example, a main menu probably won't need a physics system in order to run. This class inherits from the *Event Listener* class which means that it will listen for events that it is subscribed to. It also controls the state of the game which include playing, pausing, exiting and debugging mode.
- *Scene Manager*: This class contains all data pertaining to the components and the entities that are created when the new scene loads in. Currently, there is a maximum number (1024) of entities in the game world that can be created just so the engine does go overboard with the number of objects in the game world.
    - *Entity*: A container class that holds an identification number so that we can look up the corresponding components if necessary. It also holds a name for the object so a developer can identify the object easier.
    - *Base Component*: The interface class that all components derive from. This class only holds an identification number which is the same number as the *Entity* id.
        - *Camera Component*: This component controls the camera based on the Entities position and rotation. Attach to any object to follow it with a camera, however only one camera will work in my engine.
        - *Render Component*: This component contains the information pertaining to the Render system. It has all the path locations for the models, shaders and texture as well as identification numbers to look up the corresponding data.
        - *Physics Component*: This component contains the information pertaining to the Physics system. Bullet uses its own objects in order to calculate the simulation. So, each physics component has to start with a Collision Object and Collision Shape. These define how the object will react in the simulation.
        - *Player Controller Component*: This component contain the information pertaining to the Input system. For now, this is a basic structure that only has a single variable. The control type stores whether the controller is a keyboard and mouse or a game controller. This has not been fully implemented yet.
        - *Transform*: Bullet uses its own transform objects to update their simulations. This is a wrapper class so that I can use it anywhere the game engine needs it to be used.
    - *Scene*: The interface class that allows any scene to be loaded in by the scene manager. Any scene that inherits this class will have a basic load and unload function that will be called in the engine when starting.
        - *Main Scene*: We load in this scene to create all objects in the game world and store them into the *Scene Manager*. All of the other load functions are purely to keep the class clean and not jumbled in a single function.
- *Event Handler*: This class is using a singleton design pattern which only keeps a single object around in memory when it is initially created. There is no need to have multiple instances of

this object as it is only a messaging service. It stores all of the *Event Listeners* in a map connected to the list of events that are possible to send.

- *Event Listeners*: Classes that want to listen for events inherit from this class and can receive events when they are sent. However, classes that do inherit have to create their own *HandleEvent* functions in order to deal with the incoming events.

2.5.5.2 Systems Architecture

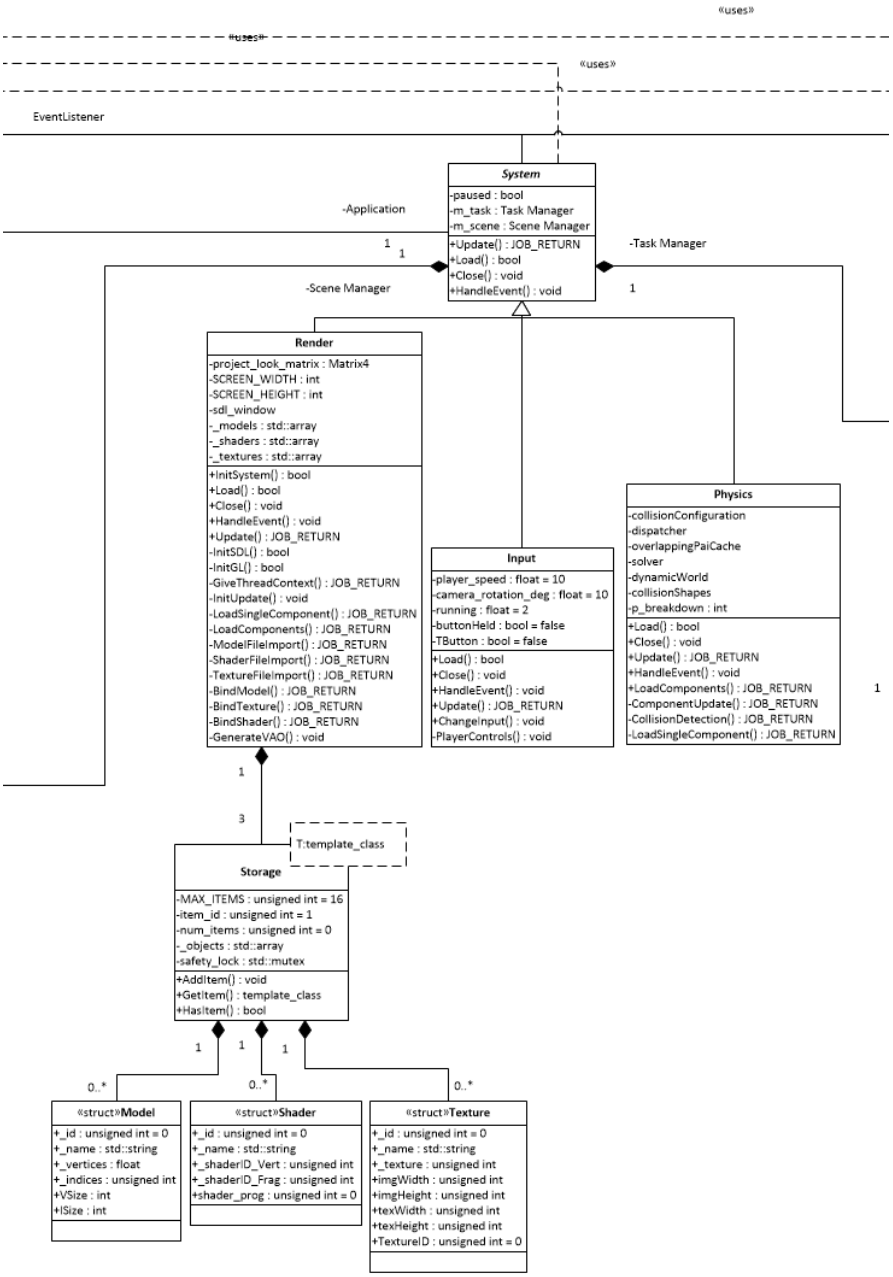


Figure 5

This is the diagram for all the systems in the engine that will be using the corresponding components. This diagram has:

- *System*: The interface class that all systems have to inherit from. I tried to make the systems modular so we can attach any new systems into the game whenever it is need. It provides all of the necessary functions initialize and play in my game engine. All systems inherit the

Event Listener class to wait for events to happen. Furthermore, they also use the singleton class *Event Handler* in order to send messages out to each other. In my engine, all components that are necessary to the game are, for the most part, are completely separated. There is no need for *Physics* to know anything about *Render*. The only component that will be needed by all of the system is *Transform* as most of them either update or read it. Which is why each class has a reference to the *Scene Manager* class. Moreover, all system take a reference to the *Task Manager* in order to post jobs to the queue.

- *Render*: This class provides for all the rendering that my engine will have to do. It uses OpenGL and SDL to render the game to the screen. A lot of the functionality that this class has is used by the multi-threading such as loading an external file and loading the components. A single thread will be identified as the Render Thread as SDL and OpenGL have a limitation on which ever thread created the context will have to work on all rendering.
  - *Storage*: We store all the external files that are import into three separate compartments.
    - *Model*: All models that are loaded from external files are brought into the *Storage* class and given an identification number to look up when it is time to render. We load in the model objects in with the use of the file loader functions.
    - *Shader*: All shaders that are loaded from external files are brought into the *Storage* class and given an identification number to look up when it is time to render. Vertex and fragment shaders are combined when being loaded into memory by OpenGL shader compiler, however this is only a temporary solution.
    - *Texture*: All textures that are loaded from external files are brought into the *Storage* class and given an identification number to look up when it is time to render. The engine uses the DevIL (Developer's Image Library) library in order to load texture in from memory.
- *Physics*: The Bullet library is used in this system in order to calculate the physics of all objects in the game world. All objects that created using the Bullet API are stored within the dynamic world. Essentially, Bullet is a black box program where calculations go in and a result can be read. This makes it very restrictive when I have to try and multi-thread it. Furthermore, the dynamic world only works on data that is stored within itself, so that means that any external objects, for example the physics and transform components, do not get updated with new data once the calculations are done. I have to update the data afterwards using a specific function in the bullet library.
- *Input*: This class handles the input the player enters such as player movement and shooting a projectile. The engine can also open the debug window when it called for by the player using the *Event Handler*.

2.5.5.3 Threading Architecture

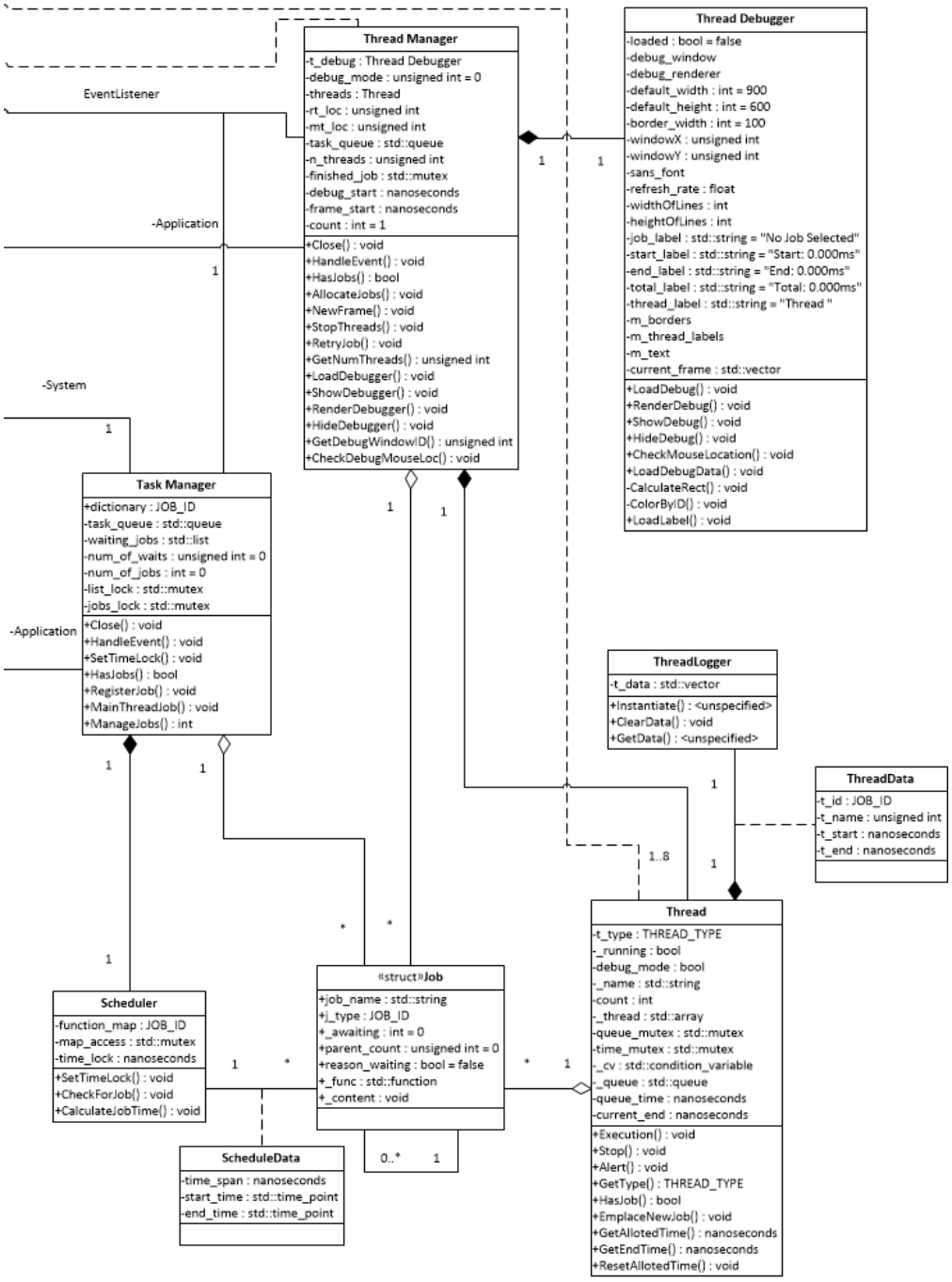


Figure 6

This is the diagram for all of the Threading and scheduling that my engine will be doing. The diagram contains:

- *Thread Manager*: This is the general manager for all things related to the threading in this engine. It takes all the Jobs that are given to by the *Task Manager* and distributes them to the *Thread*'s queues. Furthermore, the threads can be stops, paused and started when they are called upon by the *Thread Manager*. When this class distributes jobs to the threads, it uses the time that is stored within each thread and calculates if this is the best option by when the job would theoretically end. Although this does provide for a close selection of the thread, there are some issues with this method. Not every job will end at the same time because of the nature of how threads work and the operating system. It will skew the data and cause errors for thread selections.
    - *Thread Debugger*: The engine displays all the information of the threads through the debugger and SDL window. Using the *Thread Data* class, it reads all the data that is stored in each *Thread* object and displays them in graph form. The graph has a number of data points that can show how each thread is working at a given time. We can calculate the duration of the jobs from the start and end times from the *Thread Data* as well as find when the job was started by the start of the frame as an objective start point. Once we have the correct data points, the debugger can calculate and display the information.
    - *Thread*: This class controls the threads, using the standard C++ threads, and completes jobs as they come into the queue. Each thread uses a condition variable to wait for a new job to come in and the *Thread Manager* alerts them when one does come. Each thread comes with a simple type to denote how the threads are being used such as a render thread for rendering, a main thread for controlling application class and a general thread for working on any type of job. This class also uses the *Event Handler* to message the *Thread Manager* rather than using a forward declaration.
        - *Thread Logger*: The logger controls the data for when a thread has finished a job. It instantiates a new *Thread Data* object that can record the data of the job that the thread is currently working on. The data that is recorded is passed onto the debugger and used there when it is ready to display the data. This class only starts recording the data when debug mode has been turned on by the player.
        - *Thread Data*: The engine records all data when the debug mode has been turned on and is stored in this class. The data includes the id, name, start and end time of the job.
    - *Job*: This structure is a generic class that any class can create and send off to the *Task Manager*. I wanted to keep it generic because it is easier to use a single class than having to rely on polymorphic classes. Data can be passed through the class through the use of the void pointer. However, issues can come up as anything can be put into a void pointer object. Which means that any data can go through the job class even unintentional objects. All jobs have a list of parent jobs that they could wait on to start working themselves.

- *Task Manager*: This is the class which all jobs are posted to when they need to be completed. Jobs that are sent through are first evaluated in the scheduler to see if any other jobs are awaiting on this job to finish before they can start. Otherwise, this job is kept in either one of two queues the waiting queue or the task queue. The waiting queue is for all jobs that are awaiting on another job to complete while the task queue is taken directly to the *Thread Manager* to be distributed.
    - *Scheduler*: This class holds all of the data containing pertaining to the time of a jobs by their id. We can use this data to determine the best time to slot this job based on the length of time it takes as well as the finishing time of the job. When new jobs are brought to the *Scheduler*, it looks in the map to find the corresponding function id. If the data is found in the map, the data is then placed in the job for the *Thread Manager* to use. Otherwise, the map creates a new data point and does the same as before. When it returns, we calculate a new average time between the data points and come with a close average.
    - *Schedule Data*: This is a simple struct that holds the all the data and can be read by the Scheduler once the job has been completed. It includes start, end time and time taken.

2.5.6 Object Diagram

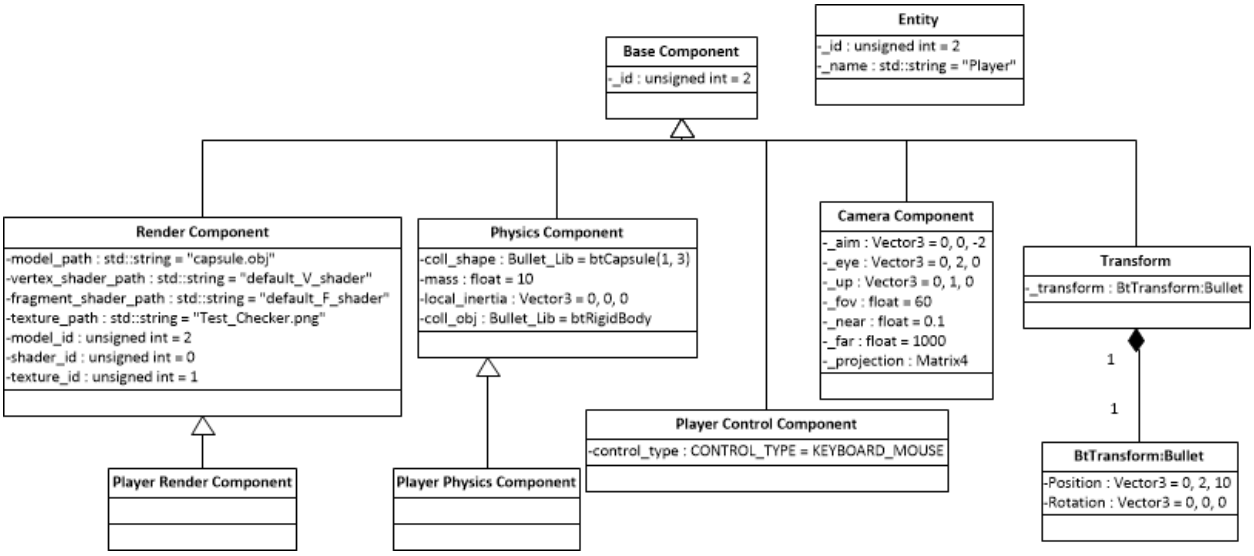


Figure 7

This is the object diagram for the entity called “Player” which is what the player is controlling throughout the game. When the new entity is created in the engine, an id is given to it which is passed onto all of the components that are used with this object. Each component is initialized in the Scene class (Main Scene for this example) and is put into the Scene Manager to handle. Render and Physics components are both created with specific variables, so I constructed separate sub-classes that are specific to the player.

Player Render Component initializes the paths for the model, shaders and textures that the player object will be using. Once the Render Component is passed into the Render System for loading, it loads all the assets and gives an id based on the id of the asset.

Player Physics Component initializes with Bullet’s Collision Shape defined as a capsule shaped hitbox. Furthermore, the Collision Object is defined as Bullet’s Rigid Body and a defined mass. When the Physics system loads this component, Bullet puts the Rigid Body and Collision Shape into its own memory stores.

Camera Component calculates the projection matrix on initialization based on the eye, aim and up vectors. The project matrix is then used by the Render system.

Player Control Component is how the player moves the entity around the world. This component is used in the Input and it updates the transform based on what controller type the player is using. Controller type is not fully implemented, so it is only the keyboard and mouse that work.

Lastly, the Transform is updated in both the Physics and Input systems, then it is read in the Render system. The order of the updating is important as we do not want to override the new input information with the physics data and vice versa. As Bullet handles everything in a black box, we need to update the players transform that is in Bullet’s dynamic world before stepping the simulation.

2.5.7 State Diagram

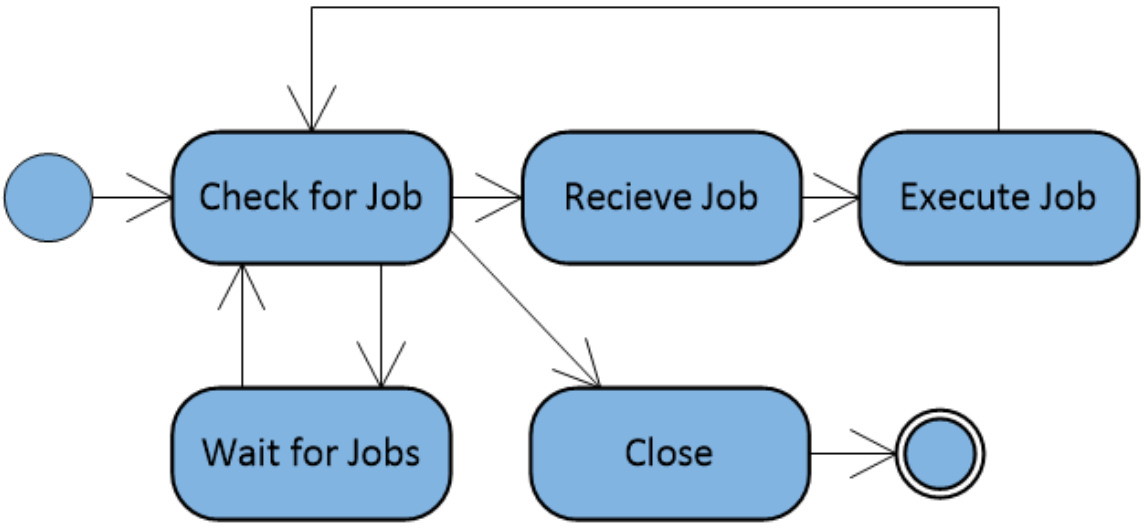


Figure 8

The State diagram above is for the Threads of the game engine. It is fairly simple in its execution so that there is no lag between job switching. The only thing that is different from the proposal is that there is no sleeping if Check for Job fails. Instead, the thread uses a condition variable so it can be woken up faster instead of waiting for the thread to finish its sleep cycle.

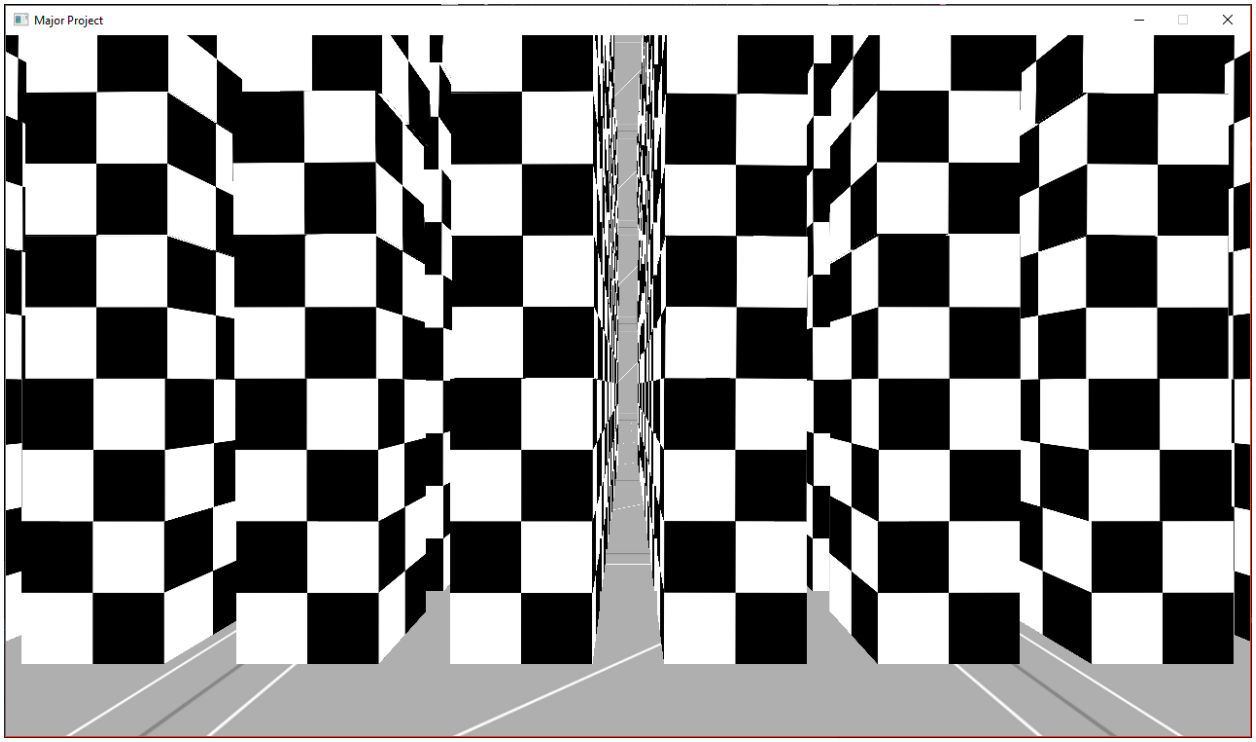
For example, if we set the threads to sleep for every 2 milliseconds and a Job just happen to arrive when it goes to sleep. The thread would have to wait for the sleep cycle to complete in order to wake up. Furthermore, using a condition variable allows the engine to pause the threads by having no jobs being sent to update the game therefore the thread never exits the condition variable.

2.5.8 Installation and Instruction Manual

The game should run all by itself. There are a couple of DLL files in the release folder of the game in that are needed to run the game. If there are any missing, they can be found in “Major Project Engine/Libraries”. This is where all the external libraries that are used in my project are kept.

One thing I did find that could cause problems are some errors about not being able to find MSVCP140.dll, ucrtbase.dll and VCRUNTIME140.dll. If any or all these files are missing, this means that you don’t have the most recent version of the Microsoft C++ Redistributable 2019 (Here is the link). Hopefully, that should allow the program to run.

When run the game it should look something like this:



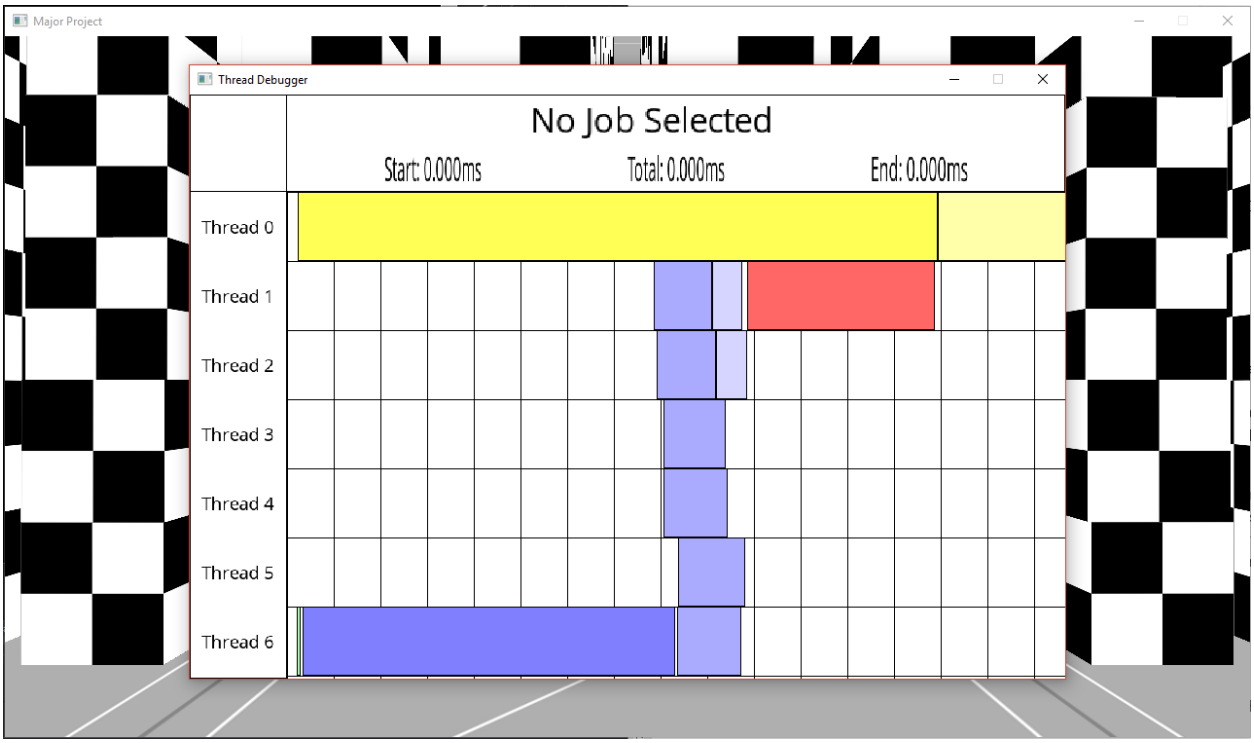
The controls for the game are:

- A – Strafe Right
- D – Strafe Left
- W – Walk Forward
- S – Walk Backwards
- Right Mouse Click – Shoot Projectile
- Move Mouse – Rotate Character
- Right Shift – Move Faster (Sprint)
- P – Pause Physics

- In this mode everything except the physics is still running. The Player is able to look around the area and shoot projectiles. Projectiles that are shot at this time will hover in mid-air until physics are reinstated. To un-pause press the P button again.

- Escape Key – Pause Game
  - Everything except the rendering is paused and the mouse is able to be moved outside of the window. To enter game again press the Escape key. In order to close the window, the player must hit the Close button in the top right corner of the window.
- T – Open up the Debug Menu
  - This button brings up the debug window. To close it press the Close button in the top right corner of the debug window.

The Debug Menu should look something like this:



Depending on how many cores that you have the number of threads that are labelled on the left side may be different. I have 7 threads because I have 8 cores in my computer. However, someone with 4 cores in their computer may only see 3 threads. Computers with newer CPUs will run the jobs faster than older computers, so jobs may look smaller or larger depending on the age of the computer.

The player can select any rectangular box within in the screen to get a text telling what the job is. Below the title of the job is information about the time taken for the currently selected job which includes start time, end time and total time taken. In order to tell the jobs apart, each job is coloured different based on their id and the system that they contribute to. For example, the Render type jobs are colour red while the Physics jobs are coloured blue.

2.6 Tests

Test # 1 – Job Finish Verification

This test was to make sure that a thread could finish a Job and was verified with a start and end time. Functionally, this was to test the thread-pool manager and distributing jobs.

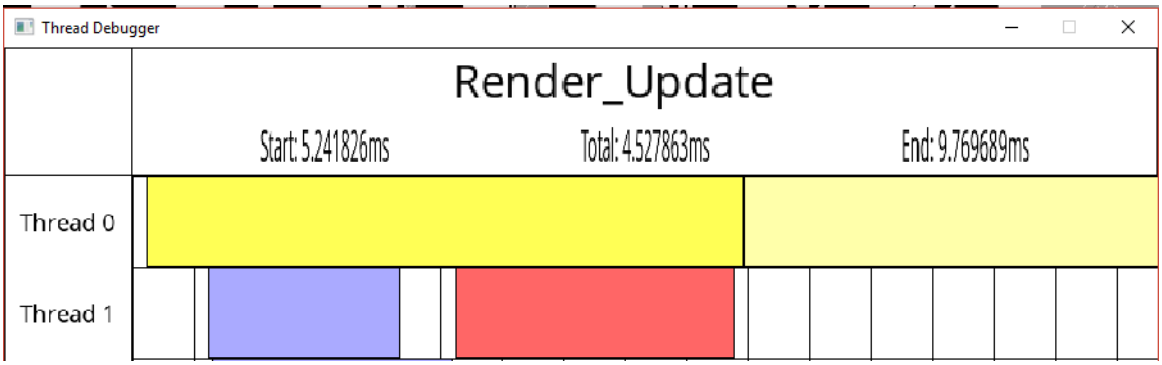


Figure 11

```
Number of Multi-Objects being created: 512
Albert  completed 803 Jobs
Logged Data for Thread: Albert
------------------------
ID:     257     Name:   Application_Update
Start Time:     0
Duration:       13
End Time:       13
ID:     258     Name:   Wait_Time_For_Frame
Start Time:     13
Duration:       3
End Time:       16
Logged Data End
------------------------
Curie   completed 593 Jobs
Logged Data for Thread: Curie
------------------------
ID:     772     Name:   Physics_Component_Update
Start Time:     7
Duration:       2
End Time:       10
ID:     1284    Name:   Render_Update
Start Time:     10
Duration:       2
End Time:       13
Logged Data End
------------------------
Newton  completed 187 Jobs
Logged Data for Thread: Newton
------------------------
ID:     772     Name:   Physics_Component_Update
Start Time:     7
Duration:       2
End Time:       10
Logged Data End
------------------------
Dennis  completed 187 Jobs
Logged Data for Thread: Dennis
------------------------
ID:     773     Name:   Collision_Detection
Start Time:     7
Duration:       0
End Time:       8
Logged Data End
------------------------
Dennis  completed 1 Jobs
```

Figure 12

We have two screen shots the prove this test. The first figure (Figure 11) is a screen shot of the Debugger for a single frame in the game. It has selected the job “Render Update” and in the title there are completion times for this specific job.

The next screenshot is the print outs for a number of jobs for another frame of the game. It has the same information, beginning, end and total times, as the debugger just in text form. With these two screen shots, I have concluded that this test has **passed**.

Test #2 – Jobs Finish within Time Limit

This test is to make sure that all jobs are completed within the time limit. The time limit that I am going for is 60 fps or 16.6667 milliseconds.

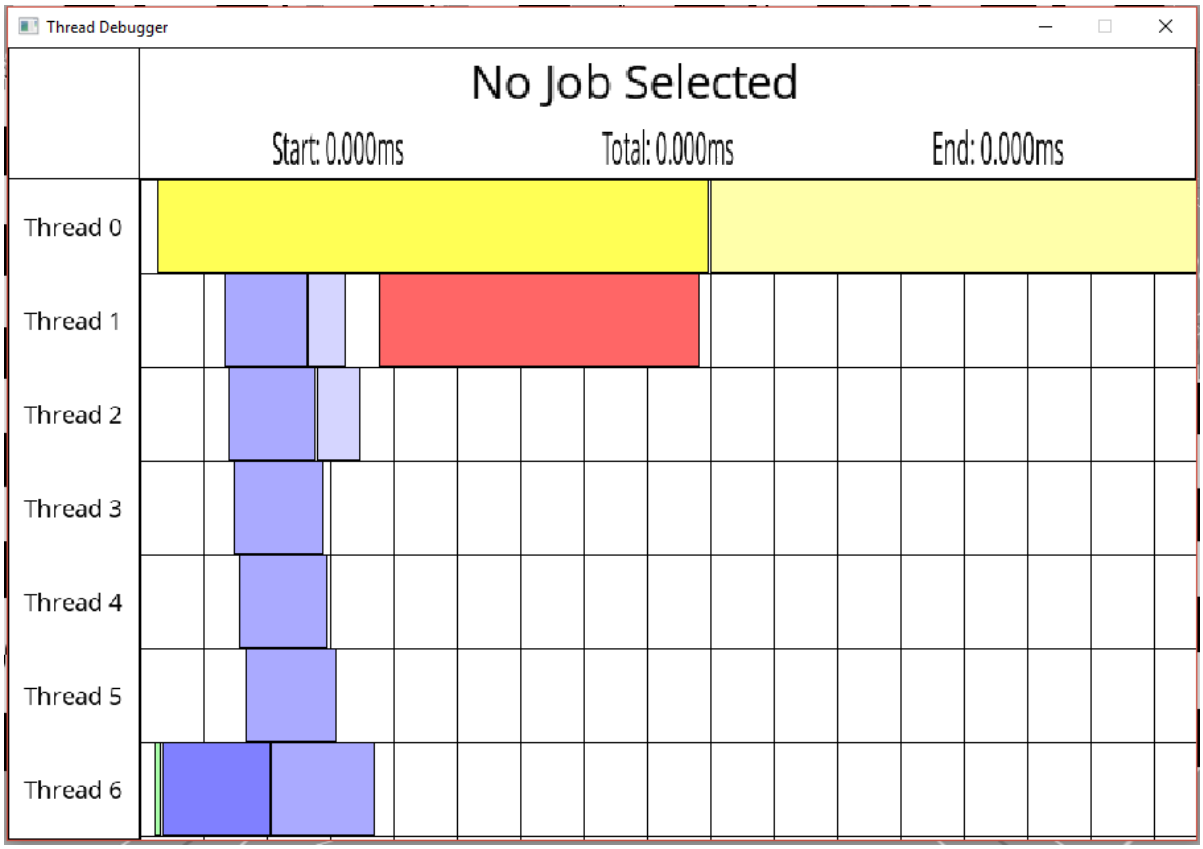


Figure 12

Here we have the debug logger again with the full list of jobs completed for this frame. The far-right side of the window is the time limit which is 16.6667 milliseconds. All jobs are completed before the time limit which mean that this test has **passed**.

Test #3 – Equal Number of Jobs

This test is to make sure that we have not lost any jobs in the middle of processing and distributing them.

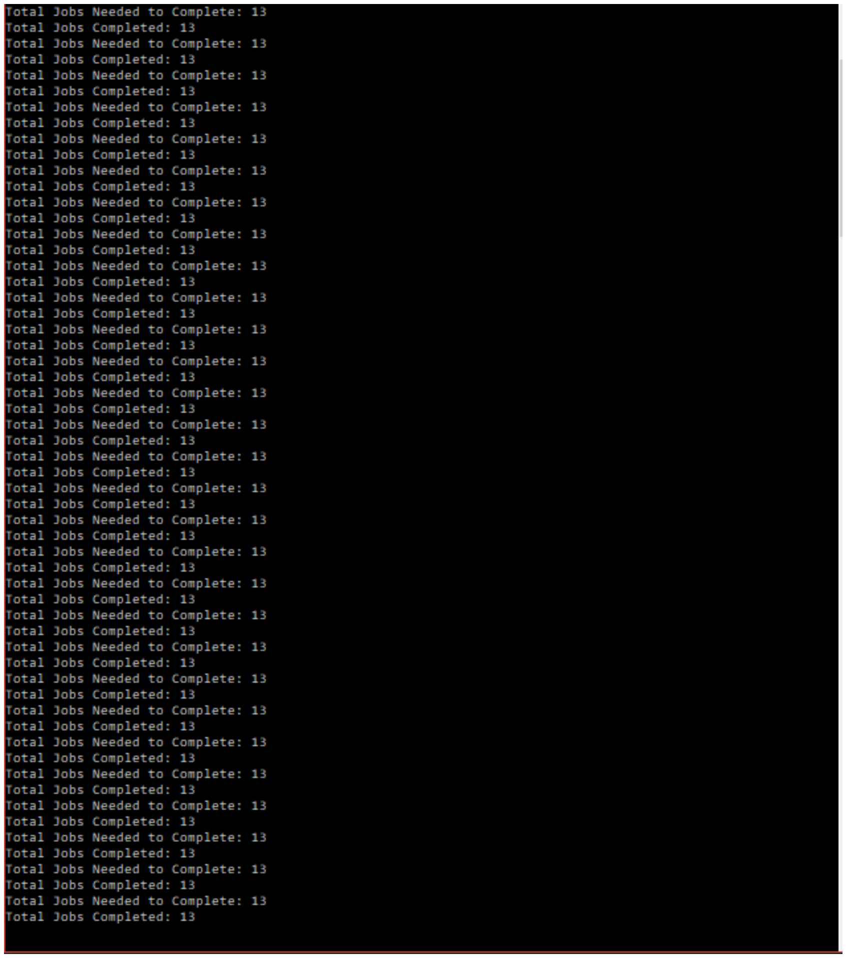


Figure 13

The figure above shows us two numbers that belong to the Thread Manager and the Task Manager. Both keep track of what jobs go in and what jobs go out. As we can see that the number of jobs are matching. Furthermore, if look back up to Figure 12 we can see that there are indeed 13 jobs completed and shown in the debugger. This means that this test has **passed**.

Test #4 – Jobs by Priority

This test is to make sure that jobs are taken in the right order and are sorted into the correct threads. This will ensure that everything in the engine is running smoothly and on time.

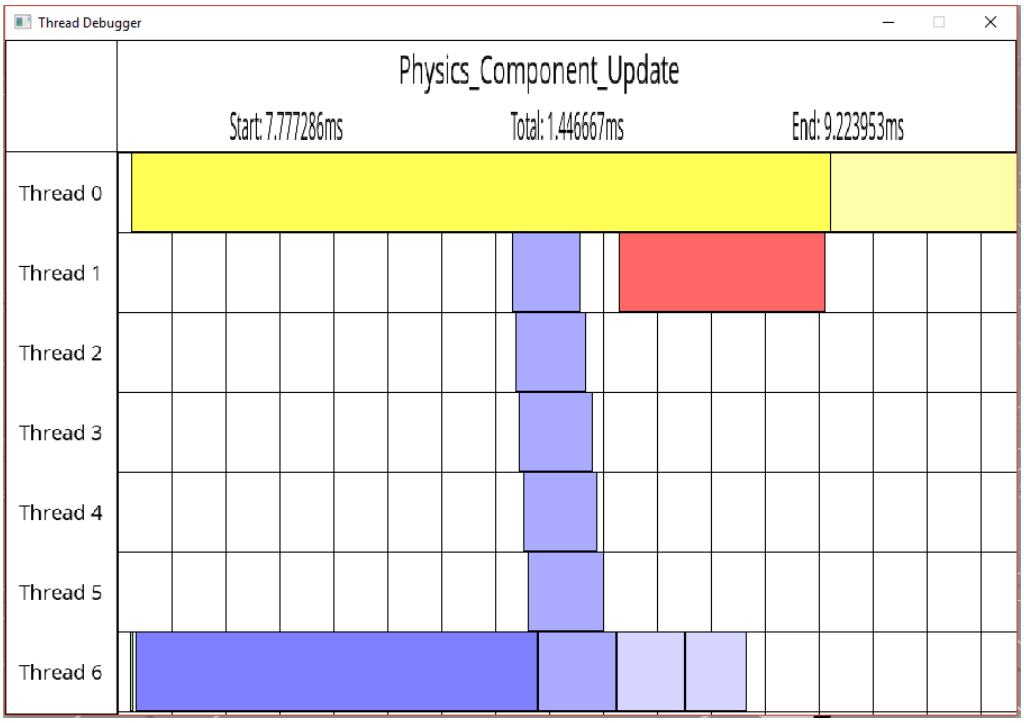


Figure 14

Here we have 13 jobs that are completed within the time limit of the frame. The most important job is the Input Update job first which is the tiny sliver in Thread 6 first. As we need to update the player object before working on the physics. Next, we have the Physics Step Simulation which is processed by the Bullet Library. Following the aforementioned, we have the Physics Component Update which updates all the transforms of all the objects in the game. These jobs can be done asynchronously but after the step simulation which is what is happening. Finally, there is the Collision Detection and the Render Update which are both read access functions which can both be done at the same time. Although the layout may not be perfect it does complete the jobs when they need to. So, this test is has **passed**.

Test #5 – Threads Stop and Pause

This test shows that threads pause when they are supposed to and don't process any jobs. There are a couple of changes that happen to go against the initial design of this plan. The very first thread that is created is constantly working even through the pauses of the game because I had to design around a different issue altogether (For details see Page 12, paragraph 3). Furthermore, even when a game is paused there are still some background processes still working such as rendering.

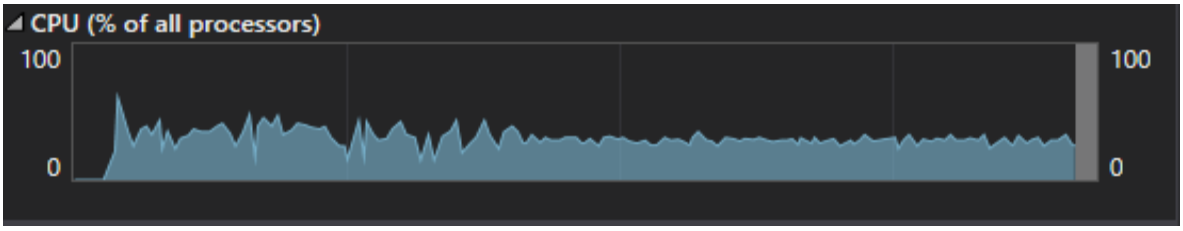


Figure 15

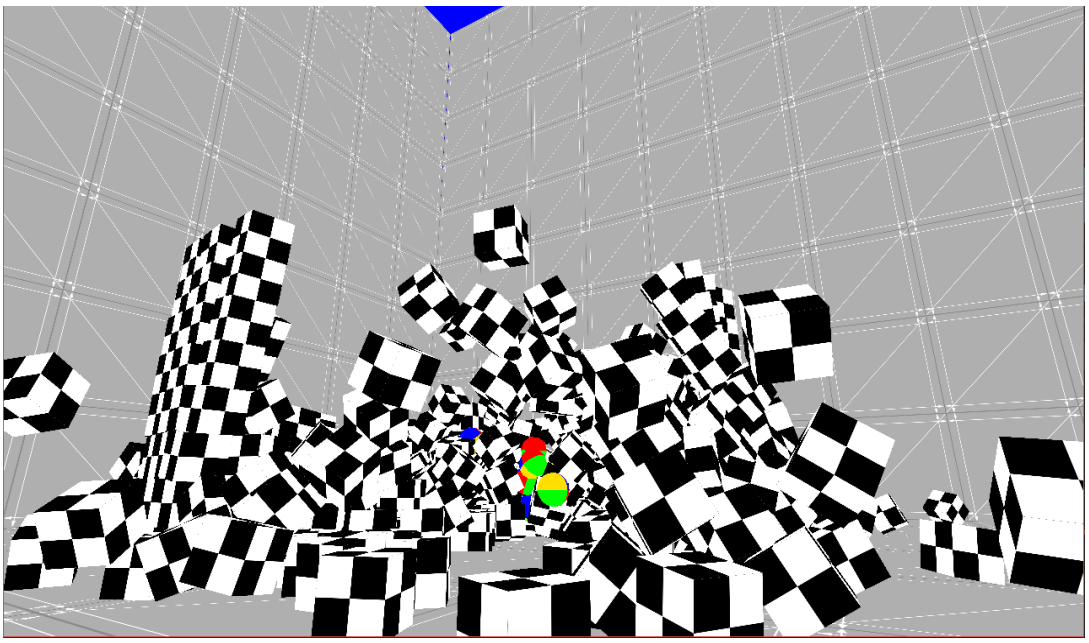
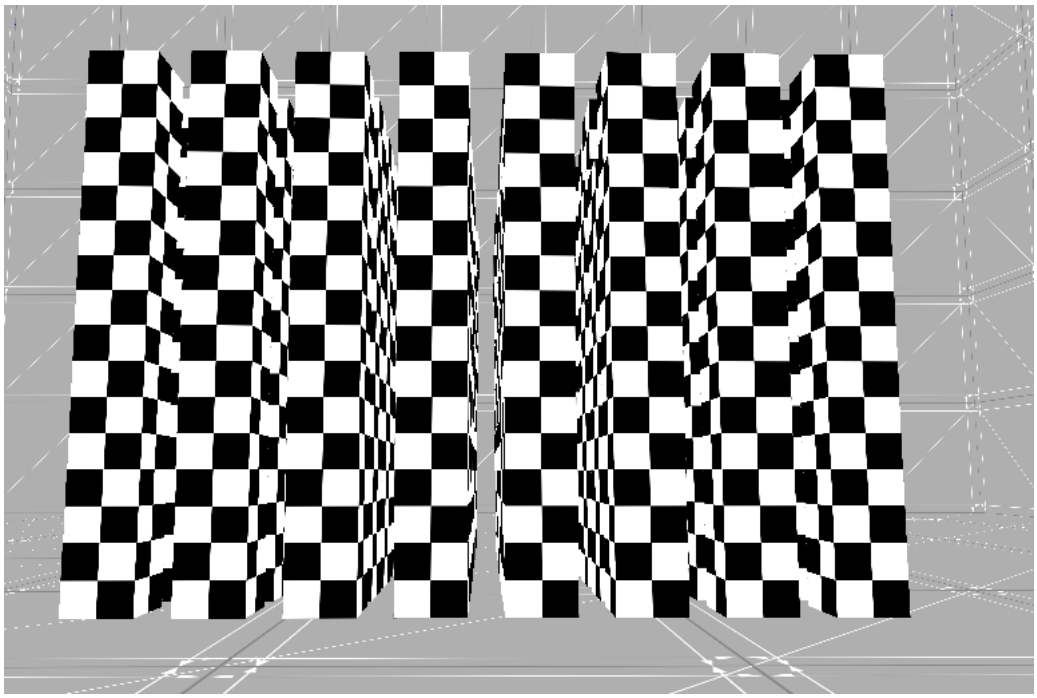


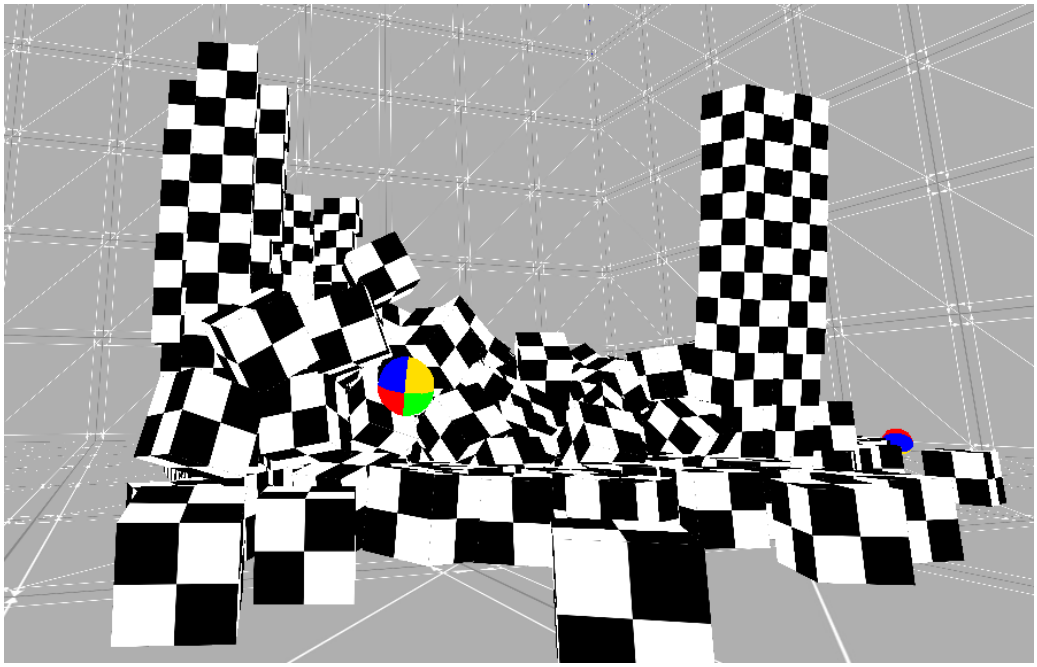
Figure 16

From the first screenshot above, we can see that the CPU usage starts to decrease around the center of the screenshot. That was about the time I hit the pause button meaning that some threads have stopped working and further down it starts to even out. There are at most 3 threads still working while the game is paused. The Main Thread that started the application, the secondary Main Thread that controls the Job allocation and the Render Thread that keeps rendering everything on screen. The second screenshot is of the game paused although it is hard to tell because it is a screenshot. You'll have to test it out yourself. For the most part this test is a **pass** but because of the changes I have made to the project it is a skeptical pass.

Test #6 – Physics Interactions

This test is to make sure that all the physics still interact with each other. There are a couple of objects in the game world that can interact. Such as the player, projectiles, the cubes, the walls, and the floor.





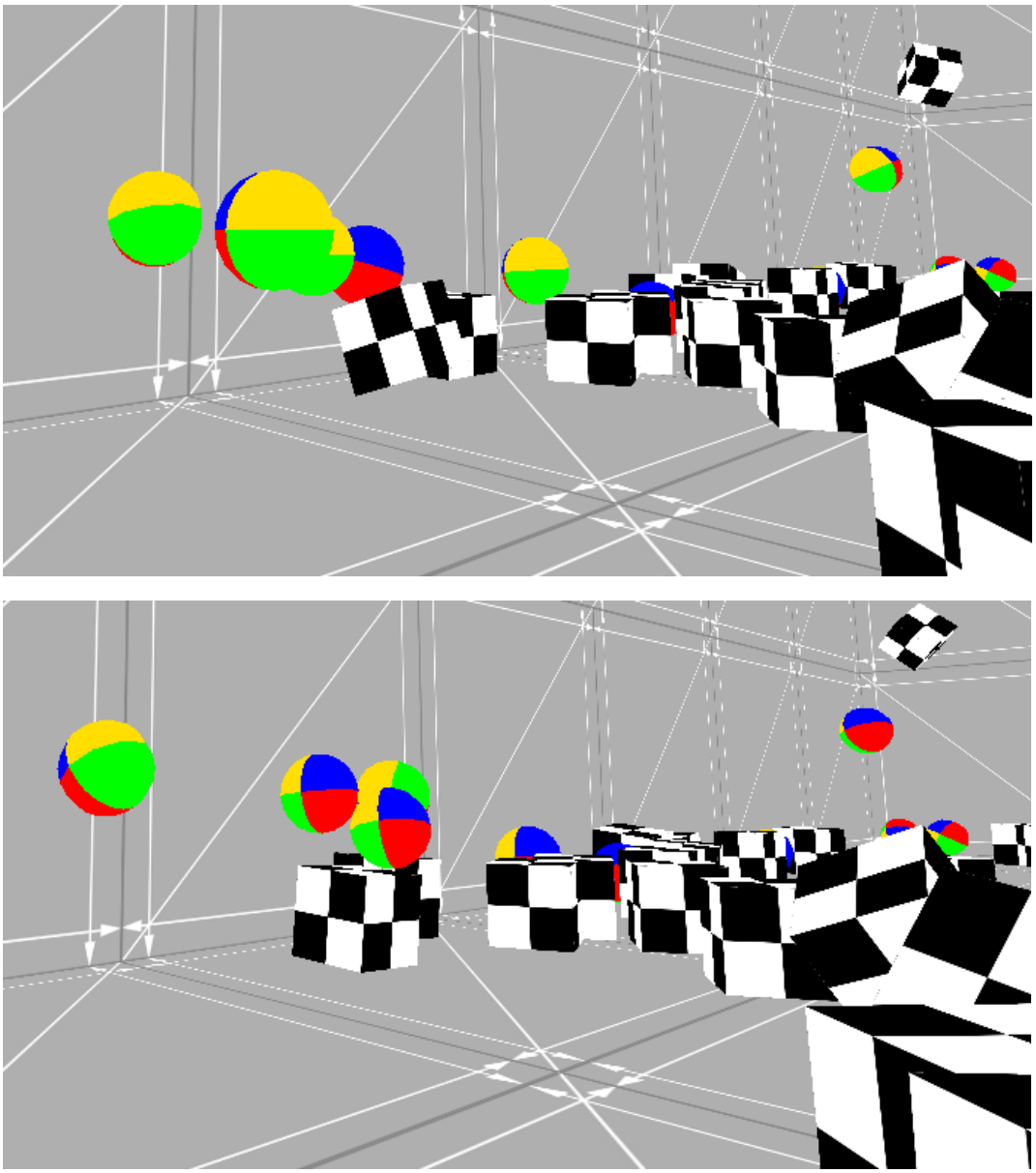
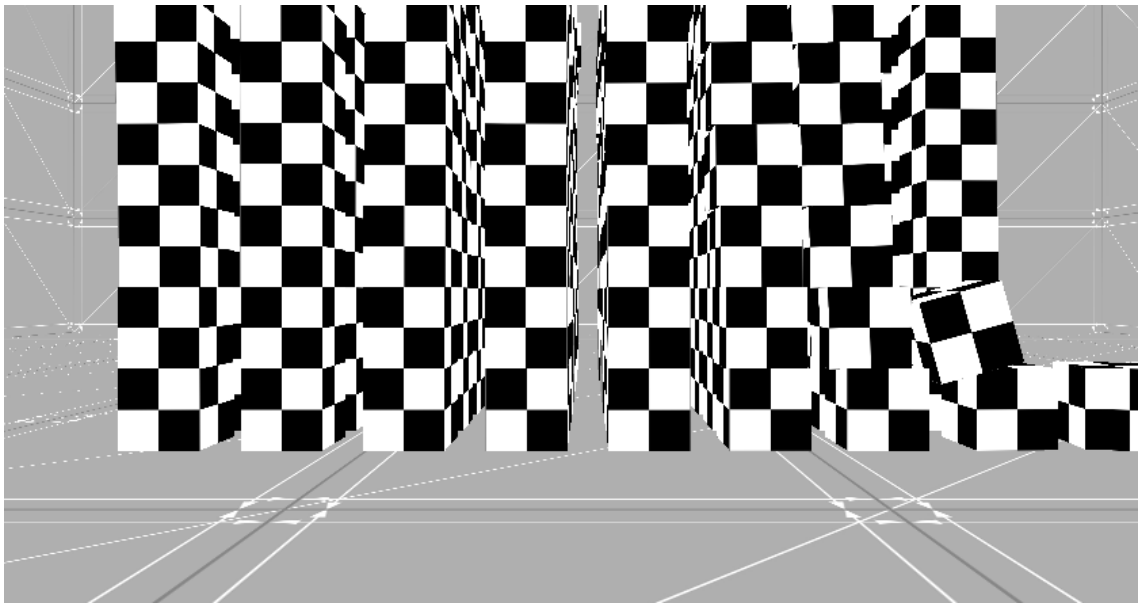


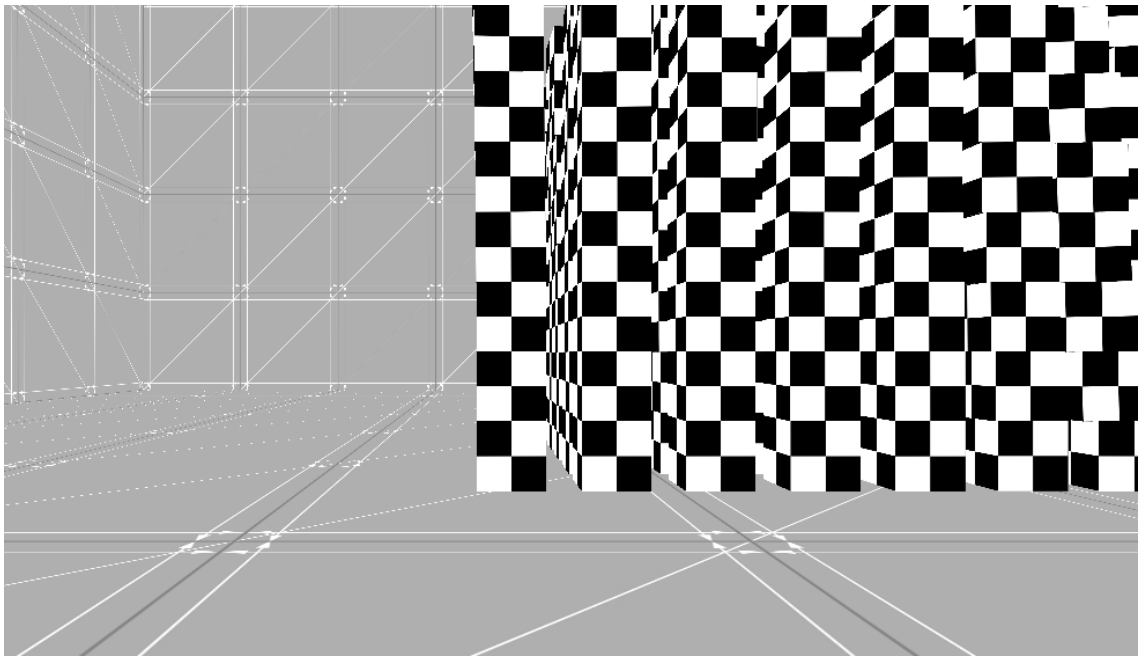
Figure 17 – 20

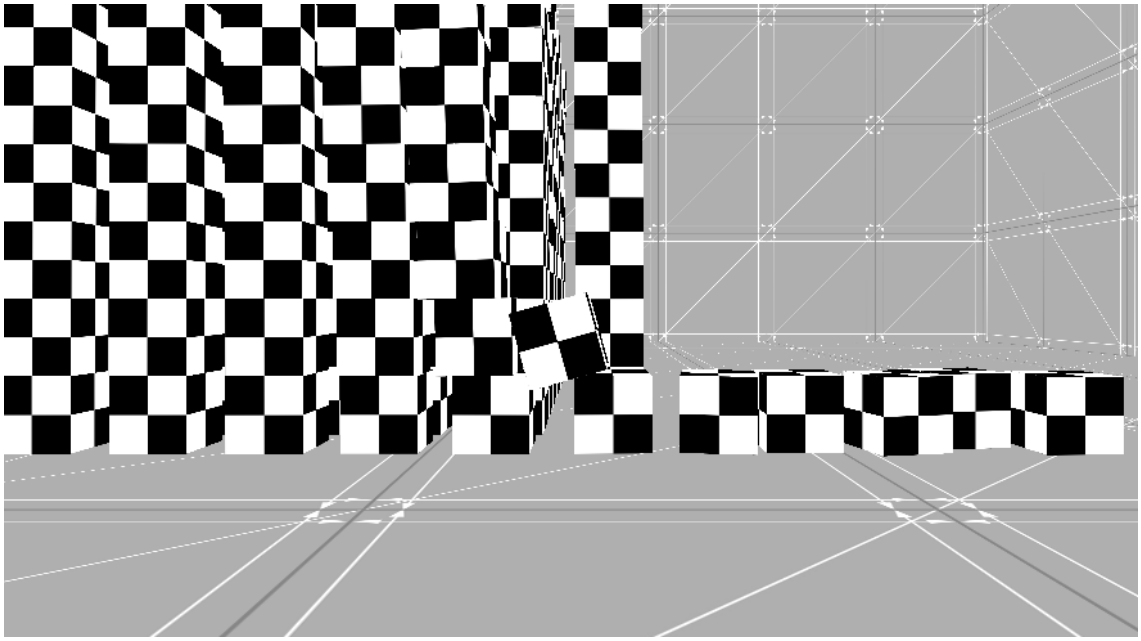
Here we have four screenshots of the physics interactions in the game. The first one is the start of the game where the objects have spawned in and are ready to be interacted with. This shows the physics is working between the cubes themselves and they are being stopped by the floor. Next, I threw a couple of projectiles into the columns of cubes to show the interactions. Most of the cubes have been knocked down and one projectile is on its way in. The last two screenshots are closely taken together to show object bouncing off the walls. A lot of the interactions are shown better in gameplay rather than in screenshots. However, all object interact with each other as predicted which means that this test has **passed**.

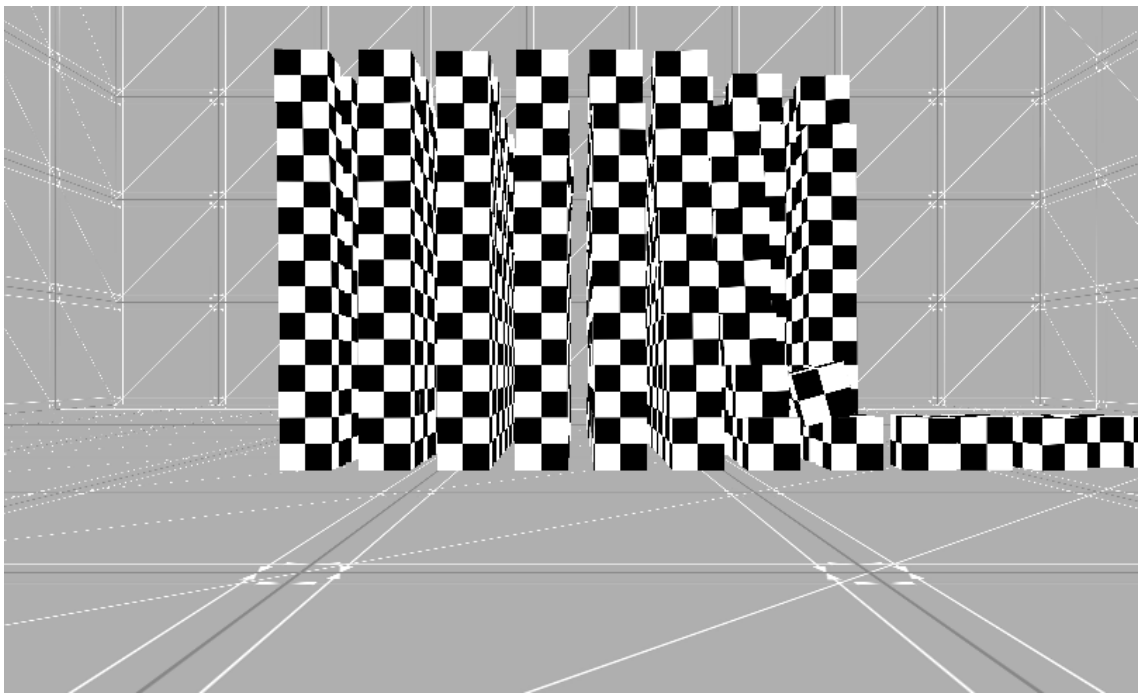
Test #7 – Player Movement

This test is to show that the player is able to interact with the world with basic movement using the keyboard keys. The keys WASD are going to be the default keys tested.









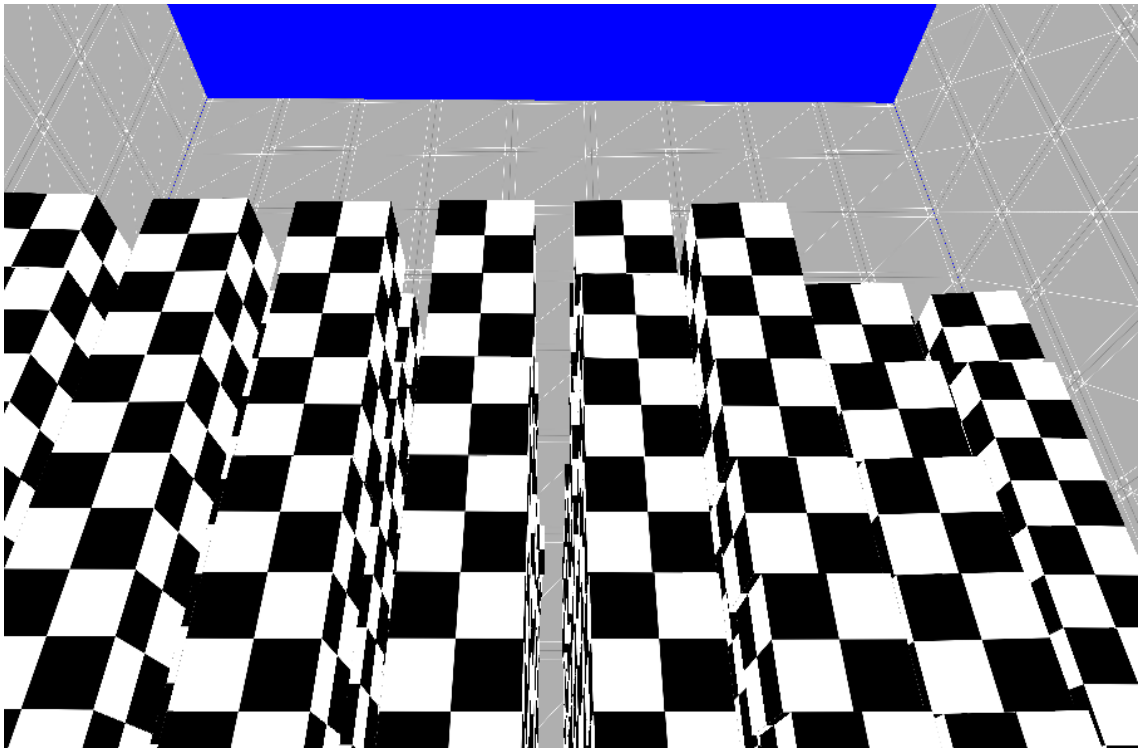


Figure 21 – 25

The screen shots provided show the directional capabilities of the player. They can move from left to right, forwards and backwards. They can also move the mouse around to look at the environment. Not much else to say about this test other than it has **passed.**

Test #8 – Game Displays

With all the screenshots that I have shown you so far, it is pretty clear that the game has no issue displaying incorrectly. The game itself runs at a smooth 60 fps and the graphics are top of the line. Everything in the game has the correct textures and models, all of the screenshots prove that. This had the failure condition of not displaying correctly such as misalignment and incorrect colours. However, the game easily **passes** this test.

Test #9 – Scheduling Jobs

This test is to verify the times of the jobs and to order them based on priority.

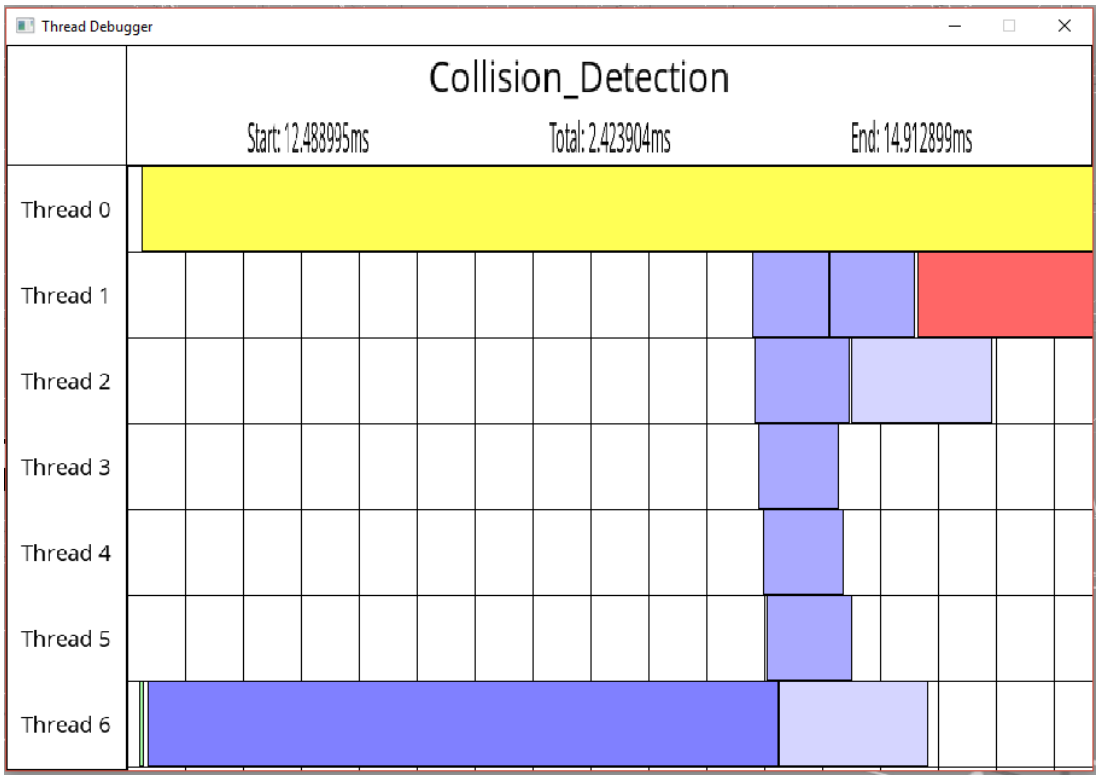


Figure 26

When Jobs are processed by the task manager, they are first taken to the scheduler to get an what the average job time would be for this new job. The scheduler looks up a map of values that point to the job id and give the total time it took to complete the task last time in nanoseconds. With this new information, the thread debugger uses the total time to allocate the jobs to an appropriate thread. The screenshot below (Figure 27) is the map for all jobs that currently exist in the game engine with data on the total time taken.

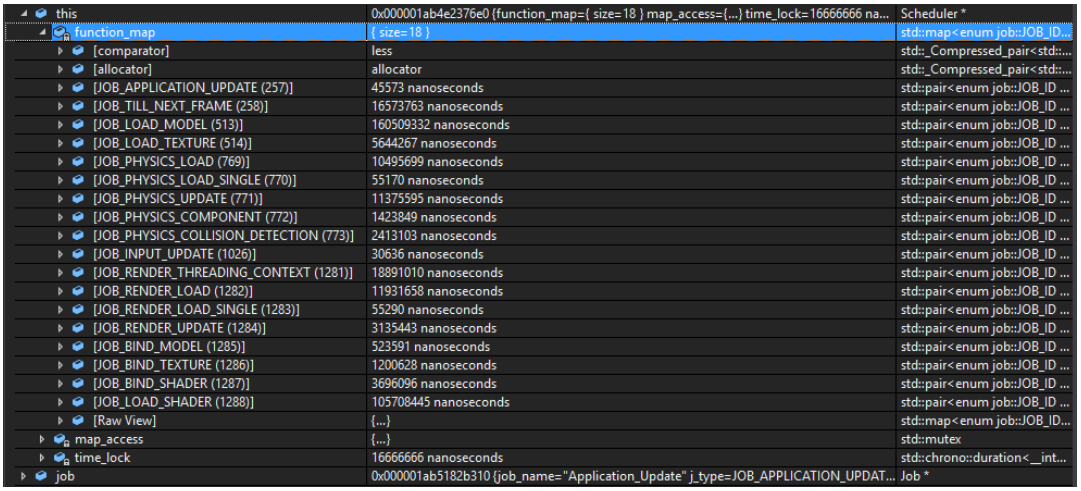


Figure 27

Here we can see which jobs take the longest and which take the shortest. The job with the lowest amount of time is the Application Update function with a time span of 0.045573 milliseconds. This job is so low because the game is paused to look at this data and when the game is paused nothing is updates. Otherwise, the time span would be greater. The job with the longest loading time is the Load Model function with a time span of 160.509332 milliseconds. Now that is a lot but there are a couple of things that do influence this data. The size of one of the models being loaded in is relatively large and will extend the time greatly, the screenshot was taken while visual studio is run during debug mode which does slow down some applications and finally poorly optimized loading in of the model asset from the File Loader. This would be a function that I would look back at and try to lower the time cost.

Nevertheless, the jobs that are taken to the thread manager are judge based on the current threads end times. It would then allocate the job to the thread that has the lowest end time. I have used a parent-child method to deal with jobs that happen one after the other so jobs wait until they are called in the Task Manager waiting queue. I conclude that this test has **passed**.

Test #10 – Many Objects in Game World

This test is to prove that my game engine can handle a substantial number of objects in the game world as well as provide a smooth frame rate. The frame rate that I will be testing for is 60 fps.

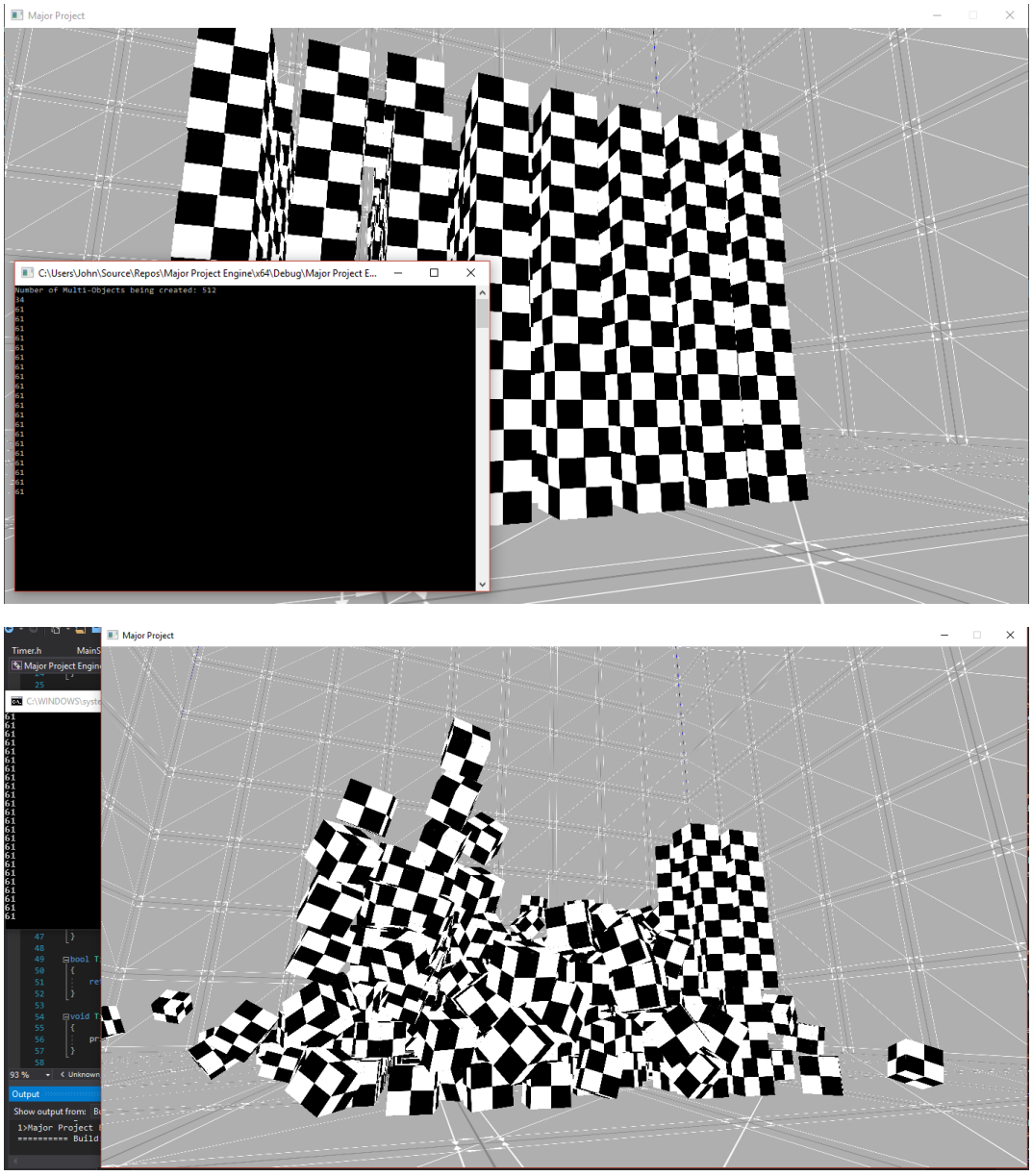


Figure 28 + 29

The above screenshots both have both 512 (8x8x8) cube objects, 4 walls and 1 floor in the game world. All these objects have physics that need to be calculated and the rendering has to be done too within a 16.667 millisecond time limit. As you can see in both screenshots, the black window to the right displays a constant framerate of 61 fps. The frame count is does not waver and the game runs smooth with little to no frame drops. Even with the inclusion of projectile objects being added to the game world dynamically.

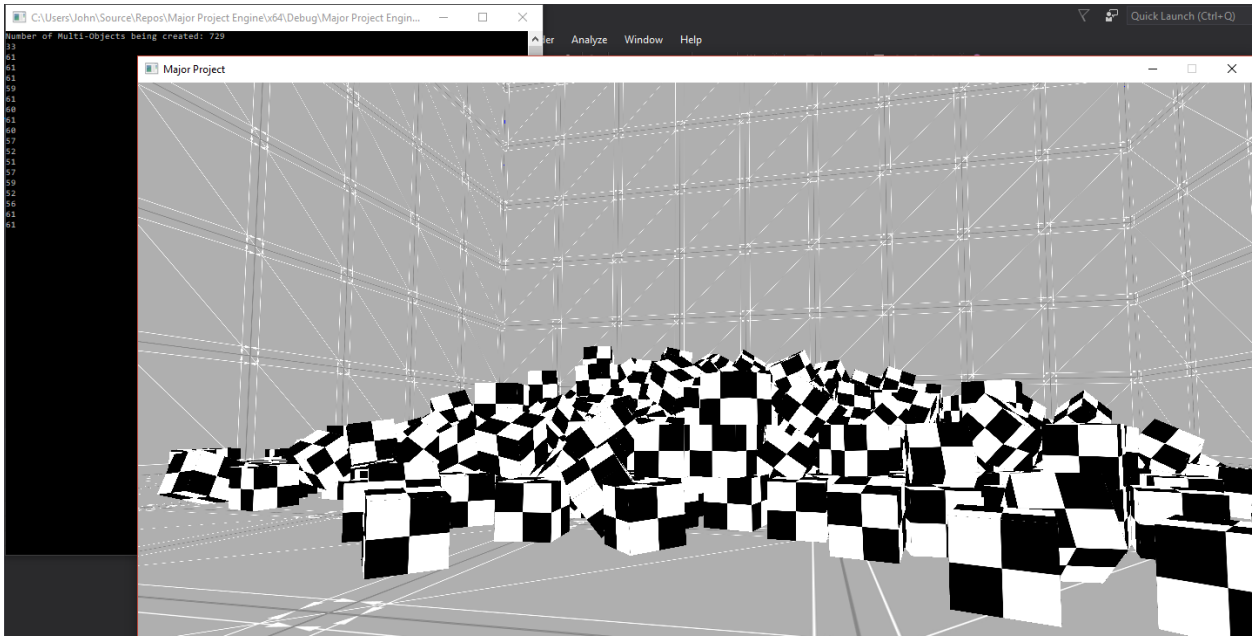


Figure 30

Here we have a screen shot with 729 (9x9x9) cube objects and we can see that the frame rate starts to decrease when interacted with the physics. So, it looks like this was too much for the computer to handle. Although 512 objects is still an impressive number. This test is a **pass**.

After doing some more testing, it looks like the Visual Studio debugging is taking up a lot more computing power than I had anticipated. Testing 729 objects on the release version still gave me a smooth 60 fps. However, I will be keeping the number of objects at 512 for the release version just in case there is issues with slower computers.

2.7 Technical Implications

Some of the implications of implementations are:

- Entity – Component – System design
  - All objects can have one of any type of component. Whereas a hierarchal design would have many different superclass and subclasses to fit the same mold. *(Functionality)*
  - Systems only need to check if the object has the correct combination of components to work on the entity. *(Functionality)*
  - Components can be kept with similar types of components so there is quick access for a system to work on them. *(Performance)*
  - All functionality is kept in the System classes where all data is kept in the Component classes. *(Functionality)*
  - Systems can be modular and don't have to be in every scene of the game. A main menu scene doesn't need physics to render. *(Functionality)*
- Worker – Job design
  - Worker threads continuously work on jobs until they run out. *(Performance)*
  - No need to keep on destroying and creating threads when not needed. They will be needed when the time comes. *(Performance)*
  - Greater use of the computer's CPU for calculations. *(Performance)*

2.8 Technical Challenges

Many technical challenges will be coming from the combination of a game engine with the base architecture of Thread-Pooling. They present as follows:

- **Performance + Stability**: When I get to the point where I can start making a display of the game on the computer, I will have to make sure that the game runs at a smooth frame rate and doesn't stutter. This includes the large number of obstacles that the proof-of-concept will be creating and calculating.

- **Timing**: I will need to figure out how to time all functions in a quick and non-intrusive way. Furthermore, threads will execute these functions within a limited amount of time such as the time it takes for a frame to finish. This is important because if a Job is going to take a long time to complete then they should have a higher priority earlier on in the program than if it was later in the program.

- **Sorting and Priority**: I will need to develop a system in order to assign every Job a level of importance relative to where the engine is at the frame. This will involve sorting through the queue of Jobs to prioritize the correct sequence of Jobs. I foresee this as a great learning opportunity.

- **Debugging Threads**: Debugging just one additional thread is a challenge in of itself. I will have to figure out a way to debug N number of threads where any of them could be anywhere at anytime.

- **Avoid using mutexes and semaphores**: As known in usual multi-threading courses mutexes and semaphores are a must when it comes to threading. However, I have learned that game makers should avoid whenever possible to using them because of the waiting that the other threads could be doing in the meanwhile.

2.9 Research used in Project

Some of the research that I have done for this project are:

- *Thread-Pool design pattern*: I have learned a lot about threads during my classes at BCIT but that was on the functionality of them. While looking through a textbook, I found a chapter on threading implementations in game engines. I also read up on some of the benefits and drawbacks of using this design pattern. However, it was a good learning opportunity to learn about the implementation of threads.

- *Scheduling*: Having a time limit within a program is always going to have to deal with some form of time constraints. Furthermore, we have to distribute the jobs optimally in order to beat the time limit. I had to look at some algorithms for organizing jobs in the time constraint. I use a greedy algorithm to sort the jobs into the threads. This means that the computer will take the thread that finishes the fastest and emplace the job there.

- *Bullet 3D*: We touched upon two physics engines during my classes at BCIT Bullet and Box2D. Though, we were told to use either of the engines with Apple's API and Objective-C. With this project, I had to learn how to import, implement and use Bullet 3D with Windows and C++.

2.10 Technologies

Some of the technologies and libraries that I will be using for my project are:

- C++
- OpenGL
- SDL2
- Bullet
- GitHub.
- SDL TrueType Fonts *New*
- DevIL *New*

C++ is widely used within the game industry and is a robust language that is able to use a ton of libraries. As for the graphics of my game engine, I will be using a combination of both OpenGL and SDL2. OpenGL has many funcitionalities and modern tools for game developers to use for their games. It can render both 2D and 3D graphics which makes it a good match for my project. SDL2 stands for Simple DirectMedia Player is another robust library that has any functionalities that a game developer needs such as audio, video, keyboard, controller and more. I will mostly be using it for the video and input options, but it is nice to know in the future that those options are there.

Bullet is a physics library that I will be using for my game to calculate all the physics that I need like collision detection and acceleration. Lastly, GitHub will serve as my version control in order to keep track of all the changes that I have made to the game engine along the way.

There were some additional libraries I used for this engine. SDL TTF (TrueType Font) is a free library allow developers to make text in their applications. DevIL (Developer's Image Library) is a free library that allows developers to import images into a texture format and use it. These two were what I needed in order to help the project look better. Furthermore, they do not help in the actual purpose of the project and are only included for additional features such as text and image loading.

2.11 Future Enhancements

Some of the future enhancements that I would like to make are:

- *My Own Physics Engine*: The Bullet 3D Physics is a good engine, however the main drawback is that it is essentially a black box with little to no wiggle room make your own calculations or multi-thread it. With my own physics engine, I could calculate everything on separate threads and make it as efficient as possible.

- *Artificial Intelligence*: The ability for in game objects to act as enemies in my game with actions like, pathfinding, follow player, hunt player, etc.

- *2D Rendering (Menus and Overlays)*: I tried to figure out how to render 2D objects with OpenGL, however it was something I had to out-of-scope and I had to settle with 2D rendering using SDL2. It would be nice to have an in-game menu system that actually functions.

- *2D Gameplay*: I would actually like to turn this into a 2D game because 3D gameplay is a lot more complex than what it seems. Furthermore, I can keep the threading and make the gameplay even run better.

- *Game Logic*: Further development on in-game logic would include winning conditions, losing conditions, scoring systems, lives, etc. I would also develop this into an actual game instead of just a physics demonstration.

- *Better Rendering*: Currently, the rendering is fairly basic where it is just textures, models and shaders. Adding more complexity to the rendering like bitmapping, shadows and lighting effects.

- *Animations*: Adding animations to 3D models or use 2D sprite maps would be an interesting next step. Although, I would need to figure out how to animate first and foremost.

- *Libraries*: A lot of what I have done with this game has been done with external libraries. I would try and implement them into the engine to see if there is any benefit to gain.

2.12 Timeline and Milestones

This was the estimated timeline and milestones for the project in the proposal. All of the milestones can be considered under the game engine category as that is what I have chosen what to focus on during this project. Nonetheless, the milestone names are just to keep it clear and concise for the reader. Times are also going to be adjustable as the project goes on as these are rough estimates. Certain Milestones could take longer, and others could take up less time.

Milestones	Component	Hours (Estimated)
Project Initialization	• New Project (Visual Studio) • GitHub Setup • Libraries imported (SDL2, Bullet)	5
Thread-Pool (1)	• Asynchronous Threads • Threads Run and Gets Single Job • Runs Function + Return Correct Results • Manager that keeps track of Threads	20
Game Engine (1)	• Rendering object in game world • Add Objects to the world • Input allows player to move around • Co-op with Thread-Pool Design • Vertical Sync	45
Timer	• Timer is Constant with Framerate • Consistent Times for Function Run-Times • Get Time of when frame is done	35
Thread-Pool (2)	• Multiple Jobs on Threads List and Sort by highest priority or FIFO • Threads done when Time-Limit is up (Frame ends) • Remove and/or Insert Jobs that have a Higher Priority	40
Game Engine (2)	• Add Physics to Engine • Co-op with Thread-Pool Design	50
Scheduler	• List for Threads will Account for Times of Functions • Organize Which Thread has Room for More Jobs Depending of Length of Time Left for Frame	75
Thread Visuals	• Get Information Based on What Threads are Currently Working on Quickly with Little Interruption	35

	• Read and Display Information in Graph Formation	
Proof-Of-Concept	• Adding all objects in project at start up • Ensuring Everything Runs Smoothly	15
Final Touches	• Bug Fixing • Finishing Touches	20
Testing	• Debugging Threads • Hands-on Testing	50
Documentation	• Function Descriptions • Class Descriptions • Milestones	25
Total		415

This is the current timeline and milestone for the project. Some of the milestones in the project I had to revisit in order to fix an issue when making it. If there are more than a single number listed, then it was a time when I had to go back and fix something in the milestone.

Milestones	Component	Hours (Actual)
Project Initialization	• New Project (Visual Studio) • GitHub Setup • Libraries imported (SDL2, Bullet)	Total 5
Thread-Pool (1)	• Asynchronous Threads • Threads Run and Gets Single Job • Runs Function + Return Correct Results • Manager that keeps track of Threads	Total 16
Game Engine (1)	• Rendering object in game world • Add Objects to the world • Input allows player to move around • Co-op with Thread-Pool Design • Vertical Sync • File Loading *New*	A - 42 B – 17 C – 16 Total 75
Timer	• Timer is Constant with Framerate • Consistent Times for Function Run-Times • Get Time of when frame is done	Total 15
Thread-Pool (2)	• Multiple Jobs on Threads List and Sort by highest priority or FIFO • Threads done when Time-Limit is up (Frame ends) • Remove and/or Insert Jobs that have a Higher Priority	A – 14 B – 14 C – 20 Total 48
Game Engine (2)	• Add Physics to Engine • Co-op with Thread-Pool Design	A – 15 B – 36 Total 51
Scheduler	• List for Threads will Account for Times of Functions • Organize Which Thread has Room for More Jobs Depending of Length of Time Left for Frame	A – 21 B – 40 Total 61
Thread Visuals	• Get Information Based on What Threads are Currently Working on Quickly with Little Interruption • Read and Display Information in Graph Formation	Total 30

Proof-Of-Concept	• Adding all objects in project at start up • Ensuring Everything Runs Smoothly	Total 15
Final Touches	• Bug Fixing • Finishing Touches	Total 15
Testing	• Debugging Threads • Hands-on Testing	Total 42
Documentation	• Function Descriptions • Class Descriptions • Milestones • Final Report *New*	Total 35
Total		408

There were a couple of Components that I had forgot to include when making the proposal that are pretty important. Just as predicted I had to revisit some of the milestones multiple times in order to improve them. In particular, Game Engine (1) was the biggest Milestone because this include everything in the engine except for the Physics. This means that anytime I had to go back and change a fundamental aspect of the game engine it was on that milestone. Furthermore, the early stages of the project was not my best code writing and I had to make lots of changes some milestones to better improve the engine.

Towards the later part of the project, I just started to work on the project with the milestones in mind and I forgot to log some of the times. With all of it together I certainly worked over the 400 hrs on this project.

3 Conclusion

3.1 Lessons Learned

Throughout this project I was able to learn and improve skills that I can certainly use in the future. Some of those skills are:

- Game Engine Building
- Multi-Threading designs
- Scheduling
- OpenGL
- C++
- ECS (Entity Component System) designs
- UML diagrams

3.2 Final Comments

This Project is what the Game Development Course has been building up to. Creating my own game engine that I can use and improve upon in the future. I know it is not perfect, but it is what I think is a pretty good engine that I can be proud of. I want to be a Game Developer and this project perfectly aligns with that goal. Even if I don't use this project in the future, I can still use the skills and knowledge I have gain from doing this project in order to excel in the working environment.

4 Change Log

To be filled when I get feedback back.

5 Appendix

To be filled at a later date.