Honours Project Report

The Development of a Multi-threaded Game Engine

Matthew Slade
Supervised by
Ken MacGregor
and
Edwin Blake

Department of Computer Science
University of Cape Town
2008

Abstract

In this report we discuss the development of a game engine middleware for the purpose of a teaching aid in games. A Multi-Threaded approach for the design and development of this game engine is adopted in order to cater to current and future multi core personal computing. Physics engine libraries and techniques for the implementation of a multi-threaded game architecture are investigated for use within the engine. The iterative design and development process of this multi-threaded game engine is given and design challenges faced when using existing libraries and a multi-threaded architecture are discussed. A demo state using the produced game engine is then implemented and an experiment is designed and run to test the performance of a multi-threaded game architecture on multi core hardware and contrast it against the performance of a single threaded game architecture. The multi-threaded game architecture is shown to perform significantly better overall than the single threaded game architecture on the multi cored hardware. These results are discussed and a conclusion that the constructed multi-threaded system successfully outperforms single threaded implementations is drawn.

Categories and Subject Descriptors
D.1.3[Programming Techniques]: Concurrent Programming - Parallel programming
K.3.2: [Computers and Education]: Computer and Information Science Education - *Computer science education;*
K.8.0 [Personal Computing]: General - G*ames*;

Keywords

Multi-threaded, Game Architecture, Simulation, Synchronization

Table of Contents

List of Figures:

Figure 14: Observed estimate of processor load. The usage is in percent on the y axis.100% means that both cores were being fully used. The x axis is the box count.

Figure 15: This graph shows the ratio of real simulation steps to physics engine update method calls. If the ratio is less than one it means the system is doing interpolations. If the number is greater than one the system is performing that many simulation steps per update call.

1. Introduction

Games development is a complex and multifaceted process which requires a new approach to conventional software engineering and development methods. This large scope and complexity coupled with the closed door competitive nature of established professional proprietary games developers and producers makes the teaching of games development a difficult and unique problem to any education institution.

Many issues and pertinent questions quickly become apparent when evaluating this problem. What role does the computer scientist play in development and design? What should actually be taught in games courses? How largely scoped should tutorials be? What systems and tools are needed to teach development?

The development of a game to rival current professional production standards is simply impossible to accomplish from scratch as practical work in a games development course since it takes large teams of experienced programmers working with even larger teams of content developers many years to develop a piece of software that merges cutting edge graphics and physics simulation technology with a vast amount of creative content that accounts for the richness of environment and fleshes out the body on top of the underlying computation process.

However these issues of scope can and have been partially mitigated in the professional industry and amongst independent enthusiasts. Professional development in the games industry is tending towards a development model where content is separated from computation. The generic, content free, middle ware that drives modern games is known as a games engine[1]. The development and reuse of game engines to reduce production costs is already an established technique in the games industry.

Proprietary engines are typically licensed out and reused with different scripting, logic and creative content to create many different games. Popular examples of this are the Unreal engines and Valve's Source engine. However these engines aren't suited for teaching purposes since the technology is mostly proprietary, closed source and often provides a steep learning curve for the user. Older engines, such as the Quake 3 engine, have been open sourced, however the code base is still huge, has little documentation and is obfuscated through optimizations. Further more if a stable existing engine designed for existing computer architectures 1-2 years ago was to be used as a tool to teach game development it would lack relevancy with today's hardware trends.

The hardware industry is tending towards the manufacture of products that are no longer faster but instead more parallel. Processors are no longer being clocked at higher and higher frequencies but are instead being outfitted with more and more cores. Entry level desktop machines now come with a separate graphics acceleration card as standard to remove even more of the computational load for 3D rendering from the numerous cores now found on CPUs.

These graphics cards in turn have been exploiting the parallel nature of the 3D rendering process for many years. The concept of numerous pipelines on GPUs is by no means new and recently, the ability to make use of these pipelines for computation other than rendering has become easier to achieve.

Computer games are arguably the most computationally intensive and performance

demanding subset of software run on the average desktop computer. The games industry is driven by delivering an increasing level of realism, immersion and game play enjoyment. This realism is achieved through more accurate simulations of real world physical behaviour and more realistically looking 3D graphics. Therefore it is absolutely crucial that any middle ware adopted in the teaching of games development should make full use of modern parallel architectures and be built from the ground up for execution on many cores with a multi-threaded structuring in mind.

Therefore we can propose a suitable solution to the unique situation discussed would be the development of a multi-threaded games engine as a foundational system for use in teaching games development in the future. Since the development of an industry level games engine takes teams of skilled game developers years to accomplish, the scope of this project is to develop an initial system using an iterative development paradigm and object oriented methodology with the intention of creating a base middle ware package that could be extended in future with the purpose of a teaching tool for games development in mind. This means that the main aim of this system will be to investigate the effectiveness of a multi-threaded game architecture on modern parallel hardware through measuring and comparing the performance between a single threaded implementation of the system and a multi-threaded implementation. The suitability of open source libraries in small scale development projects will also be investigated.

In order to achieve this goal we'll first investigate the scarce existing research on game design and particularly multi-threaded game design. A number of existing physics libraries will also be investigated for suitability to this particular project. The design and implementation details of the constructed system will then be laid out. However since an iterative development paradigm was chosen for the project there is no clear separation between the design and implementation of the system in the report. The subsystems of the game engine implemented in this project are :

- Physics engine
- Character control
- Resource loading and management
- Interface for game logic
- Graphics rendering and scene management
- Interface for AI
- Sound
- Multi-threaded game engine architecture

This report however will only be investigating the following components in detail within the design and implementation section.

- Physics engine
- Character control
- Resource loading and management
- Interface for game logic
- Multi-threaded game engine architecture

The experimentation procedure for the completed system will then be given in detail and the results of the experiment discussed in chapter 4.

2. Background

Background and previous work into the specific field of games development is often lacking. Most of the work one in the field is accomplished behind the closed doors of large game developers who consider any advance they make to the field a closely guarded trade secret which can be used to gain an edge on their competitors in the market. Therefore not many aspects of the game engine can be sufficiently investigated with regards to previous work. However there does exist some research into physics simulation and multi-threaded design which apply to games engine development.

This background chapter will focus mainly on investigating multi-threaded architecture and physics engine with suitability for games engine construction in mind.

2.1 Multi-threaded Architecture

All of the numerous sub systems of a game engine can't act independent of each other completely and there must be an overarching architecture that controls and manages communication and interaction between the subsystems. Traditionally in a single threaded engine, this would be the main module holding the common data structures shared between subsystems. It would execute a main game loop continuously for the duration of the game runtime. Inside the loop the subsystems would each get an opportunity to execute their operations in a sequential fashion one after the other in a single thread. Unfortunately a parallelized multi-threaded architecture can not be so easily defined and there are many different ways to implement the parallelism.

The two main characteristics to investigate when designing a multi-threaded architecture are essentially level of *granularity* and *synchronization*. Granularity refers to the level at which tasks are divided up for threading. Dividing different subsystems onto different threads is *coarse grained* whereas dividing tasks within subsystems onto different threads is *fine grained.* Synchronization refers to the areas in code where threads must execute in a serial fashion and wait for each other when reading or writing a resource.

2.1.1 Granularity

Granularity of the threading could be taken down to a very fine level and implemented with an API such as OpenMP for various data separable tasks in for loops[9] such as farming off collision pairs in the broad phase of the physics subsystem to multiple threads. However when parallelizing one cannot simply split the work to the smallest granularity possible and expect to see linear speed up. The most obvious reason for this is that desktop computers only currently have two to four cores on the CPU and by creating more than four or five threads would create contention between threads for the cores. Additionally, there is overhead created with thread creation, destruction and synchronization which might offset the speed-up of the parallelized resulting in a negligible net speed-up.

Another possible implementation is to implement the granularity at a coarser level with each subsystem (or groups of subsystems) running in its own thread implemented with more specific thread control such as with the Pthread Library. This approach of a more coarse level of threading is believed to be a more optimal solution since a lower number

of threads doing a larger amount of work each is more likely to offset the overhead incurred through creation[10].

2.1.2 Synchronization

Synchronization of the engine can be done per game object, leaving the different subsystems to run in an asynchronous manner [10] with the rendering occurring at a different frequency to physics and object updating etc. A mutex for each object can be created and when one of the subsystems attempt to access it, the associated mutex can then be locked to provide synchronization. This method ideally allows the threads to run with very little synchronization at all since with a world with a high count of game objects, two threads will rarely want to work on the same object at the same time. This also allows the graphics subsystem to operate at a higher frequency than slower systems such as the physics subsystem. However there is the possibility that the high number of mutex lock/unlock actions might cause a significant amount of overhead

The other possible alternative for synchronization is that it can be done on a per game loop or frame basis. The subsystems can do their appropriate jobs for a time step in different threads and then synchronize their data at the end of each time step. This system is potentially quite easy to implement since the parallelization only happens in an explicit portion of the game loop allowing communication between subsystems to be done during the serial portion of the game loop without fear of deadlock or other non-deterministic problems that threading risks. However, this method of implementation limits parallelization opportunity since all threads have to synchronize every time step. Further more, the frame rate becomes implicitly tied to the frequency of the physics thread thus slowing the frame rate dramatically and creating unsmooth rendering.

2.2 Physics Engine

Movement of objects within games engines is no longer a trivial task. Fully equipped physics engines that simulate real world physical responses have now become the norm within modern games. These systems in themselves are constructed by teams of highly skilled developers with much more than just a small background in the field of physics and mathematics. Therefore to recreate a subsystem to compete with the accuracy and features of these existing systems would be well beyond this research scope in itself. Instead, an existing library will need to be used in order to fulfil the requirements in this subsystem.

In Hecker [2] we see the functionality required by a physical simulation can be subdivided into the four phases, namely, contact detection, contact resolution, force computation and state integration. A more complete and technical definition of this subdivision is offered by Boeing et al [3] though. Boeing suggests that the six essential aspects of a physics engine are:

- Simulator paradigm: This refers to what physical aspects can be simulated[3]. If the simulator paradigm is constrained to only a few simple aspects of physical simulation, the realism of the simulation will be lower and the range of possibilities of game scenarios would be constrained.
- The integrator: This is the method used in the simulation to compute the next

time step of the simulation. Methods range in accuracy and computational expense and a sufficient compromise between these two competing factors is essential for a game since it needs to look accurate enough to convince the user, but at the same be computationally feasible to run acceptably in real time.

- Object representation: This refers to the bounding shapes handled by the simulation. If the simulation only handles basic shapes such as spheres and cubes, collisions will seem to happen too soon and appear incongruent to the user.
- Contact detection: This refers to how collisions between bounding shapes are detected. An efficient method that uses cheap detection at a broad phase level with a bounding hierarchy[4] such as an octtree or BSP-tree is a necessity to ensure that it can occur in real time.
- Constraint implementation: A constraint is a restriction placed on an axis of rotation or movement[3]. The constraint could be relative to another object and they are used to mimic the construction and behaviour of real world complex objects such as human joints, door hinges, swings etc. The greater the variety and type of constraints allowed within the simulation, the more complex the behaviour of composite objects within the game world can be.
- Material properties: This refers to what physical objects can be simulated e.g. Frictional methods

The library chosen not only has to have the functionality mentioned above but must also be open source to allow for customization if needed. It should also have a sufficiently large community following and support to ensure that the library is constantly being improved, maintained, supported and kept relevant. Havok, True Axis, Newton and Ageia PhysX, although all free to use are not open source and therefore fail to meet this criteria. Bullet Physics Library[5], Tokamak Physics[6], Jiglib[7] and Open Dynamics Engine[8] are all open source physics engines worthy of further investigation

A comprehensive comparison of the performance and features of these four libraries (and others) was made by Boeing[3]. The numerical integrators of theses four engines were tested by accumulating multiple steps and measuring the relative error. Bullet Physics was found to have the lowest in positional error. A testing of the material properties again showed Bullet to have the closest behaviour to the ideal case. However ODE showed the lowest error in constraint calculations and Tokamak had the lowest constraint solving time. Unfortunately both ODE and Tokamak fail the collision system benchmark which is one of the most important aspects required in game world simulation. Overall Bullet is mentioned by Boeing[3] as performing the best out of the open source libraries tested.

Further investigation was consequently done into the suitability of using the Bullet Physics library since it had superior performance over the other physics engines investigated. It also had a robust community support and was under constant development[5]. The documentation, source code and user forums of the library were investigated to see if the following set of criteria were fulfilled by the library:

- Modularity, customization and object orientated design: The library is coded in

c++ in an object orientated fashion. An object within the physics world is represented by an instance of the btRigidBody. Different steps in the physics simulation process are performed by different classes which are all modular and replaceable with customized inherited classes which override the base class methods.

- Interfacing with game logic: There per simulation step callbacks and collision callbacks which server as clean entry points for game logic and modification of the simulation. Rigid bodies also specifically contain a user pointer which is not used by bullet for the purpose of linking back to game objects independent of the physics engine during callbacks allowing for a low level of coupling between the simulation and the game logic. The rigid bodies contained built in collision bit masks which would simplify logic in collision callbacks. Finally the dynamics world allowed for the easy removal and insertion of rigid bodies on the fly as long as the operation occurred in a per simulation step callback.
- Threading compatibility: The ability to easily implement synchronization in the form of mutex locks on objects within the physics environment was a crucial factor in whether the physics library could be used. Since bullet is a very robust and complex library, the modification for thread safeness needed to consist of very shallow changes that did not interfere with the complex internals of the library. Bullet provides this ability through an abstract MotionState class. Each rigid body has a MotionState object associated with it. When the object is moved, bullet calls a virtual method defined in the MotionState class to change the transformation matrix stored within the MotionState. In order to implement synchronization the MotionState class merely needed to be extended and the appropriate set and get methods of the contained transformation matrix guarded with a mutex. This allows for the renderer, which only observes the states of objects, to lock the mutex and observe the current state of the transformation matrix safely, as well as for the physics engine to lock the transformation matrix and modify it safely.

2.3 Summary

Based on the investigation presented in this chapter, the Bullet Physics library built into a coarse granularity per object synchronization framework scheme seemed like the best combination of characteristics and technologies for a multi-threaded game architecture since it minimizes synchronization overhead and thread creation overhead and allows rendering to be done almost completely asynchronous from the world update allowing for higher frame rates and smoother graphics.

3. Iterative Design and Implementation

Game engine design is best suited to an iterative development process since functionality can be separated allowing certain components to be designed and added as needed with additional modules or code being added to the current working version. A casual development process based on the scrum paradigm suited the project since it is lightweight and removes the requirement for extraneous amounts of design documentation and focuses instead on delivering a functional implementation at all stages of development. The process also scales well and is suitable for the development team of two. As a result of this approach, design and implementation was interleaved

3.1 Prototype

The first system that needed to be designed and developed was a low fidelity prototype to test and validate the main underlying multi-threaded architecture and the physics library which were both chosen in the previous chapter.

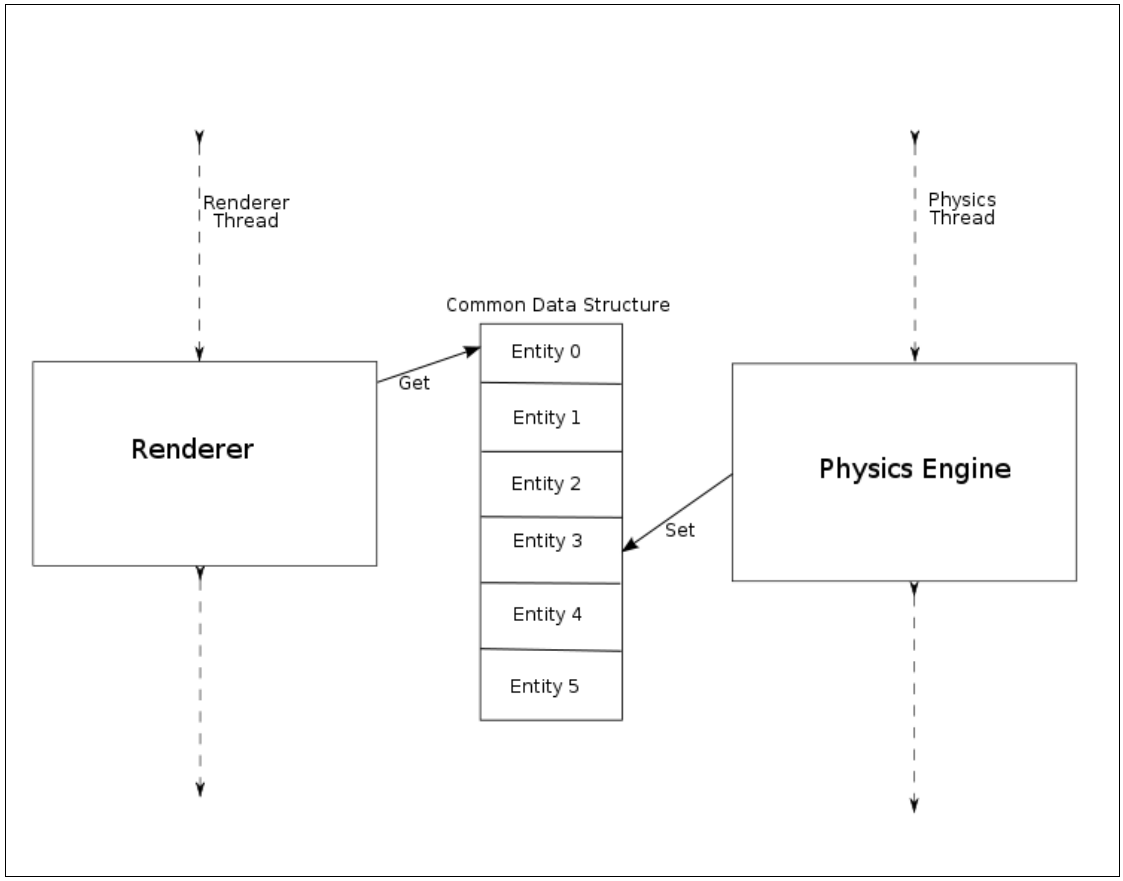

Figure 1: A diagram showing the basic shared data structure and multi-threaded architecture being investigated in prototype construction. The dotted lines represent the two threads executing in the multi-threaded engine with the data structure in the middle containing all the objects (entities) being shared between these threads.

The two most important components that would be needed to test the architecture choice of a per object synchronization scheme with a coarse level of threading are the physics

engine and the renderer. Since this prototype system would also be investigating the practical suitability of the bullet physics library, this library would be used in the prototype as well as the actual implementation for the physics engine.

A high level overview of the two main threads and the basic shared data structure can be seen in Figure 1. For the Renderer component of the system we chose to use GLUT, a simple primitive platform independent API for OpenGL, since it was relatively low fidelity and was quick to set up. Threading was implemented using the Pthread library by spawning a thread to run the physics simulation before initializing the renderer.

3.1.1 Synchronization using Motionstates

The data shared between these two threads within this initial prototype and the in the final implementation would need to thread safe since the data being shared was positional and rotational data of each object. This is encoded in the form of a 4x4 homogeneous matrix transformation within the bullet physics library. The physics thread would be responsible for updating the values and the rendering thread would be reading the values. When there is a only one thread performing write operations synchronization isn't necessary if all the data can be written atomically. However since a transformation matrix consists of multiple float values, it is likely that the operation would not be atomic and the renderer might sample the data in a state where some of the values are updated and some are still in the previous state. This would lead to a completely new non-deterministic representation of the object which is neither the old transformation nor the new transformation or any interpolation in-between resulting in a jump in location and or a spasm in rotation seen when the object is rendered.

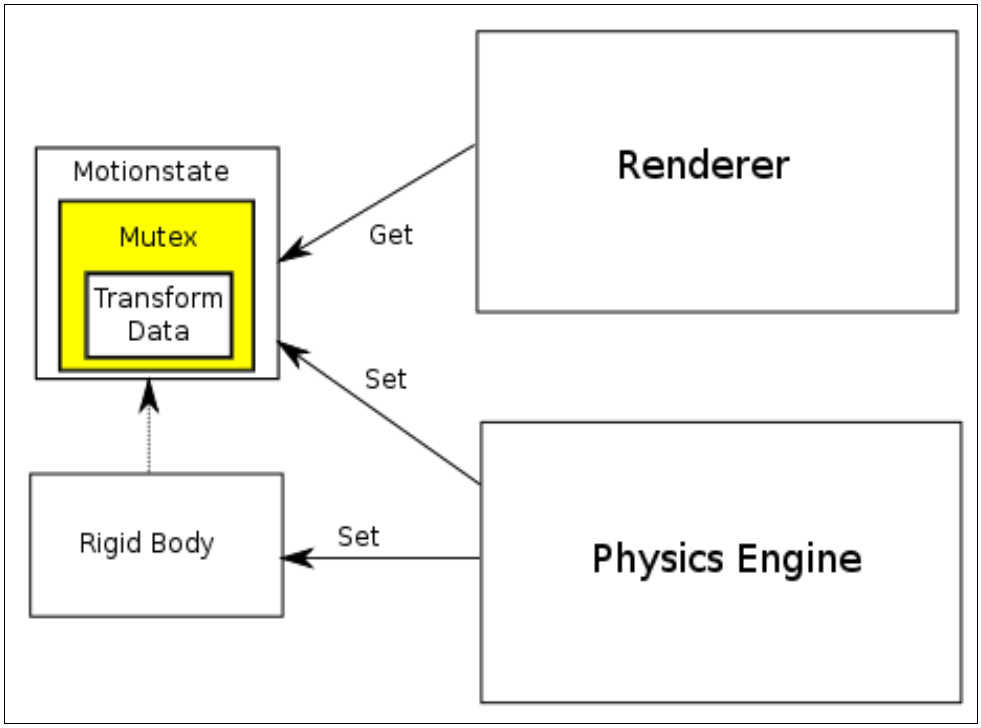

Figure 2: A diagram showing the synchronization scheme between the physics engine and the renderer using a copy of the transform in the Motionstate class

Since this behaviour was clearly undesirable a mutex lock to provide synchronization was implemented to protect this data structure when it was being accessed. In order to do this without tampering with the bullet physics internals we extended the motionstate class which holds a copy of the orientation of the object it is associated with. When bullet updates the real object it calls a virtual set method in the motionstate which was overrode to include a Pthread mutex lock. Similarly a get method was created in the motionstate which locked the same mutex to access the transformation matrix (Figure 2).

3.1.2 Outcomes of Prototype

The prototype was successful in demonstrating that the Bullet Physics Library could be coupled with a renderer in a coarse threaded per object synchronization architecture that made use of a customized motionstate class that could be taken forward into the final implementation to provide sufficient synchronization.

3.2 Initial Design

The purpose of this games engine was not only to demonstrate the performance and feasibility of a multi-threaded games engine, but also to develop a usable middle ware that could be adapted with ease to implement most 3D game design concepts. Subsequently the engine was designed in a generic modular fashion with loosely coupled modules that could easily be extended or replaced. The following modules were planned in the original high level design plan of the game engine:

- Physics Engine: This module serves as a wrapper for the instantiation of the btDiscreteDynamicsWorld which is essentially the existing Bullet Physics engine that we would be using for the game engine.
- Mutex Motionstate: this would be a thread safe derivation of the abstract motionstate class that holds a copy of a transform for one particular rigid body within the dynamics world.
- Entity: This module is a basic game object which would hold all data and methods and represent a direct one to one relationship with an object within the game world. It would be derived from the bullet rigid body and would be entered into the dynamics world just as a standard rigid body would be. Implicitly it would also keep a pointer to a unique mutex motion state object which held a copy of it's transform
- Entity Manager: This module would provide the functionality to manage and store efficiently all entities within the engine
- Character: The Character module would be one of the extended and customized object that inherits from the existing generic entity class. It requires pre built functionality over the generic object such as input management, other properties such as health/ammunition/weapons and of course be able to move turn and jump and be active within the world.
- Camera: The camera module would be responsible for following a character or keeping track of the arbitrary free form camera position in space. It would also

have to poll the cursor position to determine rotation.

- Renderer: This module's functionality would be to access the list of entities currently active in the physics engine and create a visual 3D representation of the entities on the screen. It should also contain a pointer to an instance of the camera module in order to determine the position of the camera at the beginning of a rendering step.
- Map Loader: This module's functionality is simply to load a map and create a congruent representation of the map in not only the physics engine, but in the renderer as well.
- State: The state module would serve as an abstract prototype for different types of stages or states in the game which require direct implementation of instances of the entity manager and all entities needed within the current state. It would also have its own unique map loader to load a map if needed and would set what control method is allowed within the state (I.e. menu state as opposed to a game state). With custom states being contained within a derivation of this component of the engine it allows for different states to be loaded into and out of the active current state.
- Game Engine: The Engine module would keep a pointer to both the renderer module as well as the physics engine module and pass a pointer of these two low level systems to the current active state. It would also would provide state management functionality.

From the above components laid out in our initial high level design plan, the physics engine, entity management, entity and renderer were decided on for initial development since they embody the core components of the system and would need to be in place before additional functionality could be implemented. Figure 3 gives an overview of how these modules would interact with each other to form a working initial system. The rest of this chapter will be dedicated to a more detailed description of the design and implementation of some of these components.

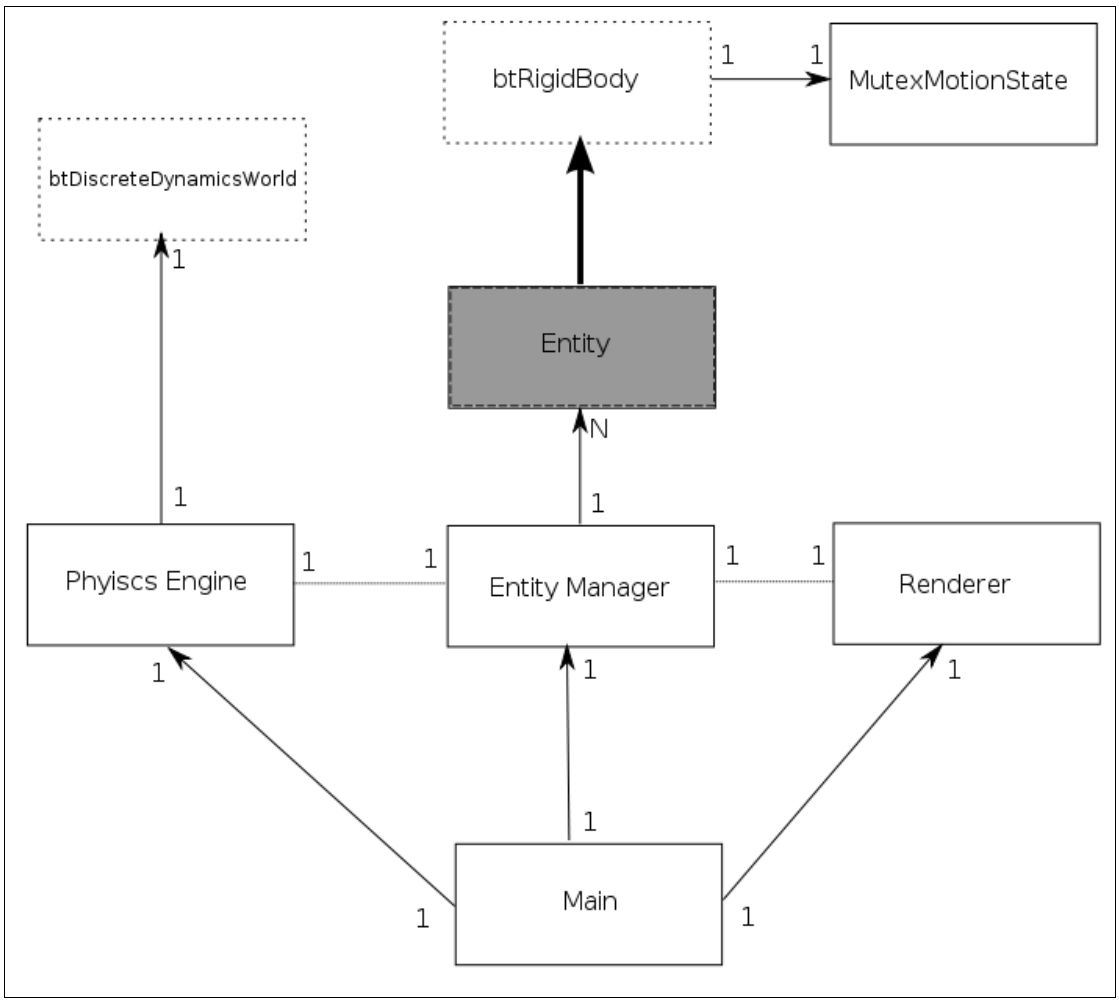

Figure 3: An initial high level design showing the interaction between the core components of the engine. Dark grey modules are generic and can be overridden for game specific functionality. Dotted lines represent modules that have pointers to each other for intercommunication and method calls. Bold lines represent inheritance. Normal solid lines represents an implementation relationship where the module pointed to is constructed by the module pointed from.

3.3 Base Entity

When approaching game development in an object orientated manner, a logical place to start development is with the base unit of the system. However in the bullet physics engine exists just such a base unit known as the btRigidBody. The rigid body stores all physics related data for an object in the dynamics world such as the position, orientation, velocities and forces applied to the body. Unfortunately just using a rigid body as our object class isn't suitable since many other attributes and methods are required for our game objects that do not relate to physics properties. Therefore when constructing the entity class which would represent every dynamic game object in our world a choice would have to be made to either extend the existing rigid body class bullet provides or create a pointer to a rigid body in the physics world that represents this particular object.

Both of these solutions have their advantages and disadvantages and during the early stages of development using a pointer instead of extending the rigid body class was opted for. The benefit of this would be that physics would be neatly separated from game related data and functionality within the entity class. However the drawback incurred from this is that entities and rigid bodies would require separate management systems and be paired together during the construction of a new object in the world. This would have lead to the development of seemingly overly complex management modules later on to perform this task. Another drawback would be that using a pointer instead of extending the rigid body class breaks from the object oriented paradigm of inheritance when a class is logically the extension of another class, which in this case is true, a game object is the logical extension to the functionality of a simple rigid body just reacting in the physics world.

Unfortunately the repercussions of this design decision extended beyond the entity class. When the physics engine registers a collision between two rigid bodies a callback method with a pointer to both rigid bodies is called. If this rigid body was associated with the entity object with only a pointer, an expensive matching process with O(n) complexity would have to be done to find the entity associated with the bodies involved in collision. This would be incredibly inefficient but fortunately this problem isn't a new one to the developers of the bullet physics library. A user defined void pointer which is unused by the bullet library was inserted into the rigid body to allow for a quick cast and dereference to reach the associated entity object. Inherently this problem would be avoided if the entity class was indeed extended instead since rigid bodies could be upgraded through a cast to an entity and the appropriate method called. However this would introduce a new problem of a plain rigid body being inserted into the dynamics world, receiving a collision and then being unsafely upgraded through a casting into an entity. An example of this would be when loading in a map of static rigid bodies you'd want to avoid assigning the extra memory for an entity since map objects are numerous and their interaction and functionality is suitably represented by a basic rigid body. A solution to this problem would be to use the standard C++ type info library to determine if the object is safe to cast up but this would require the creation of a dummy object to compare against which isn't a very elegant way to solve the problem. A much more acceptable solution that we finally opted for was to set the user pointer on plain rigid bodies that get inserted into the dynamics world to a non null value and just check the value of the user pointer in the callback to see if casting up is safe.

Finally one of the major reasons inheritance of the rigid body object was chosen over just using a pointer and a one to one relationship was because the re usability of generic rigid bodies was limited to its collision shape assigned at construction time. All rigid bodies have a pointer to a collision shape which represents the shape of that object in the world. Unfortunately the rigid body class had no method for modifying the collision shape after construction publicly. It would have been possible to insert the method and modify the bullet library since it's open source but the decision was taken to rather use the library as is so that future upgrades of the library wouldn't have to be modified as well. Therefore extending the rigid body class gave us access to the protected collision shape pointer allowing us to assign a shape or mesh to any object on the fly and reuse rigid bodies instead of discarding them.

Having decided that the best way to associate an entity with a rigid body was to inherit from the rigid body, we added some basic attributes that we imagined would be needed

by all derived classes from this entity class such as object type and health as well as some pointers to structures required for rendering.

3.4 Entity Management

After developing a base version of the entity class the next component of the engine that required development was the entity manager. Managing resources in games is an important aspect of game development. If the game were to assign and delete memory dynamically whenever it was needed the game would experience drops or jumps in frame rate or performance as the memory gets allocated. It is for this very reason that object pools, with totals more generous than what will be used in the game, should be used to pre-emptively load all resources into memory for a particular state before execution occurs. This might result in a bit of a loading period before the state starts but it will ensure smooth game play afterwards since little to no memory allocation will need to be done.

However the task of entity management becomes slightly more complex when we have to additionally work with dynamics world offered to us by bullet. Presumably, not allocated objects can be present in the physics environment since some should only enter in at later stages based on the game. These unused objects can't just be left somewhere in the dynamics world since they'd require additional processing power to calculate their physical reactions even though they weren't needed and if they weren't left far away positionally from where the game was being played they would also interfere with game play. Bullet's dynamics world allows for objects to be added and removed during simulation with the condition that it only occurs at specific times when it is deemed safe by the dynamics world to do so and will not destabilize the simulation.

Bullet's physics simulation runs on an internal step frequency of 60hz, this means that if bullets update method is called slower or faster than this step frequency the simulation either more accurately simulates and takes multiple steps or it interpolates or extrapolates the current step which then gets thrown away on the next real update since the next real step of the simulation is always calculated on the previous real step. Therefore changes to the simulation environment can only be done at the end of a step. If this rule is not adhered to the undesired and unpredictable behaviour in the simulation might occur from instability. To allow access to this important stage where changes can be performed, the simulation has a callback method that gets called immediately after a real simulation step completes. Therefore our entity manager would have to be built with these unique constraints in mind.

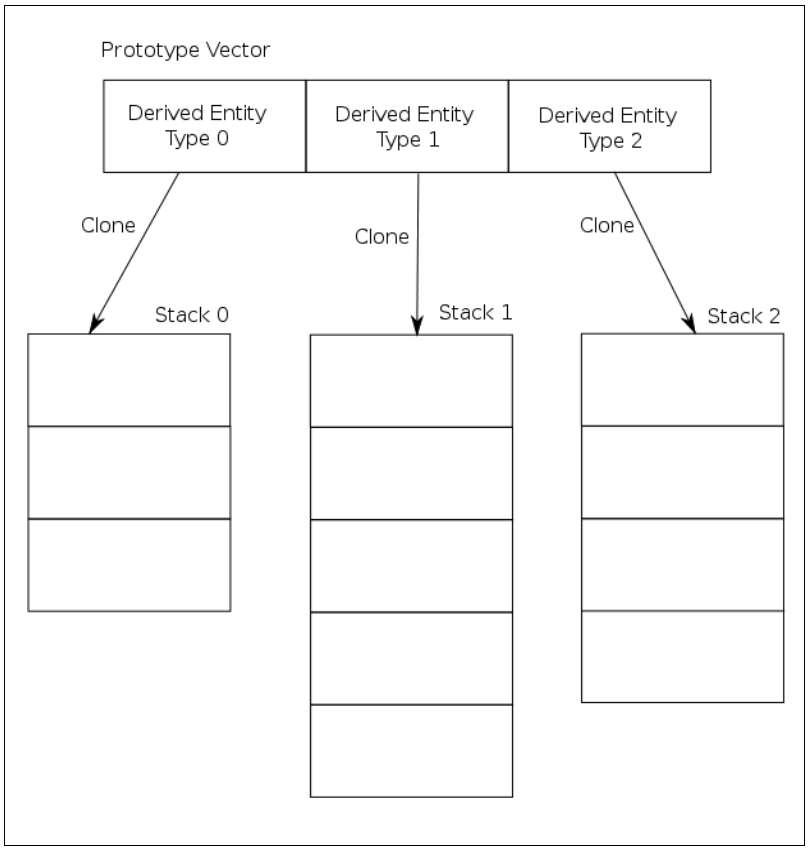

Figure 4: Diagram showing the factory template process using a generic clone method on a vector of prototype entities

3.4.1 Factory Templating

Another important design challenge our entity management scheme would have to overcome is staying generic in implementation. In other words, the code within the entity class would have to allow for different types of derived objects that are yet to be created and used in a particular game. It seems impossible to allow for the preconstruction and allocation for derived classes that don't even exists yet, however there is a rather elegant solution to the problem known as a factory template.

To implement a factory template solution the base entity class requires a virtual method called clone. The clone method creates a clone of the current entity and returns a pointer to this entity. When creating a derived class of entity, the clone method should be overridden and replaced with a suitable clone method that creates a clone of the derived class instead and returns a base entity class pointer to this derived entity. Then the constructor of the entity class can take a vector of entity pointers in along with a total for each corresponding entity and clone each entity type to that specific total. This allows for the code to remain generic within the entity manager and remain unique but brief in the state. In other words to make use of the entity manager one would first create a master prototype for each type of derived entity class, put all these prototypes into a vector of base entity pointers and pass this structure in when constructing the entity management.

To allow for these different types of entities a vector of stacks is created with a stack for each type of derived class. Each stack is filled with “dead” objects that are currently not being used. In order to be able to determine what type of object belongs where when returning a dead object onto the stack, we assign each object an integer type which was declared in the base class, This type value corresponds to the index of the stack to which that object type belongs. A vector of stacks were used for storage since all objects within the stack are for all intensive purposes identical and when needing a new object any object will do therefore removing and adding to the top of a stack of dead objects seems like the most efficient method.

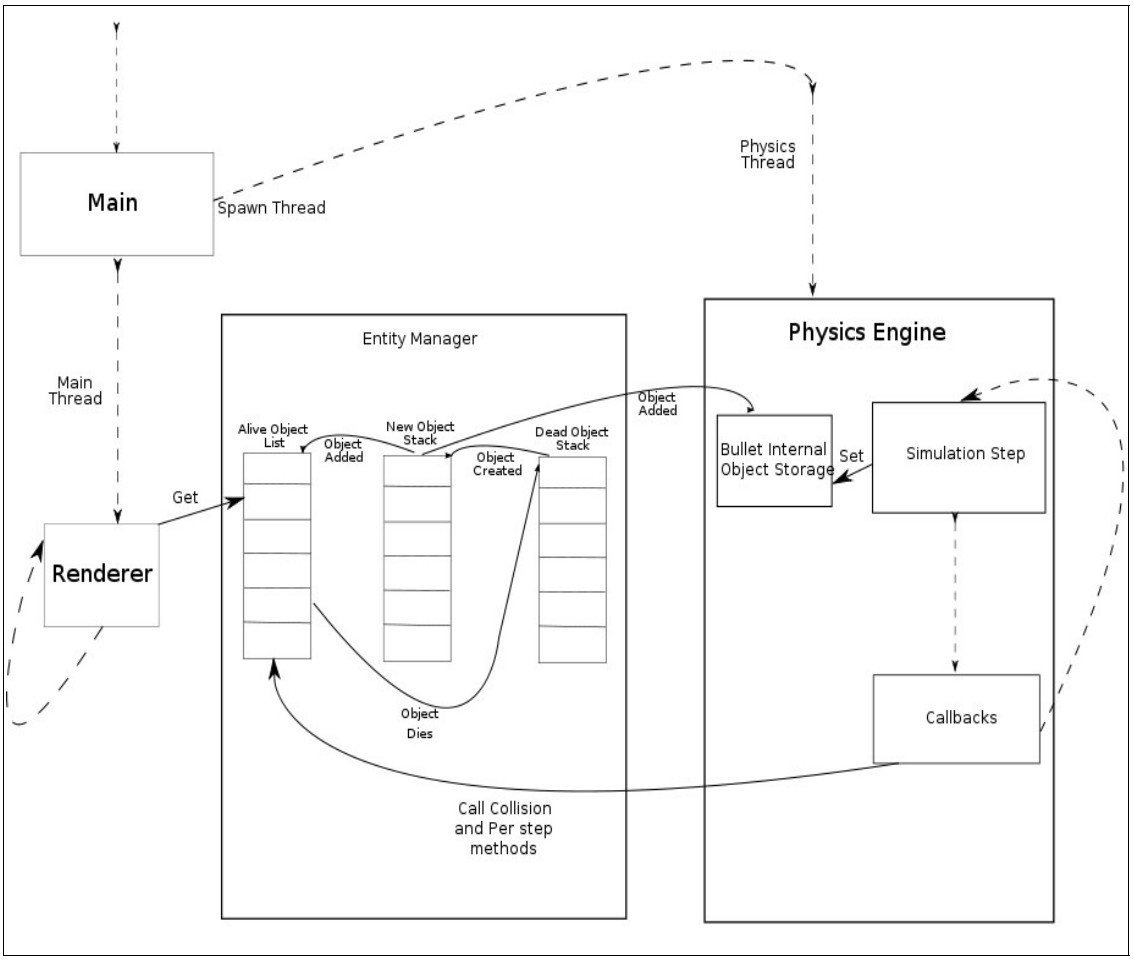

Figure 5: A diagram showing a simplified overview of the Entity Manager with its internal Object management and interfacing with the Physics Engine and the Renderer

3.4.2 Entity Life Cycle within the Management System

Besides a vector of stacks of dead objects, two other data structures were needed for entity management and can be seen within the Entity Manager module in Figure 5. Firstly, another stack for new objects to be inserted into the world is needed. In addition to this a list of alive objects is needed. In order to conform to the constraints placed upon insertion and removal of rigid bodies into the dynamics world, the method to request a new object of a particular type doesn't directly insert the object into the dynamics world but instead puts it into the new objects stack and returns a pointer to it. Objects then wait in the new object stack until the callback from the dynamics world

occurs. In this callback method the new object stack is emptied and every object on the stack is inserted into the dynamics world since it is safe to do so on the callback, the objects are also all pushed onto the list of alive objects. This list of alive objects is now iterated through and the virtual callback method for after a simulation step is called for every object in the list that is currently alive, immediately after that, the object is checked to see if it is still alive (it might kill itself in it's own callback). If the object is dead, then it is removed from the dynamics world since it's still currently safe to do so and the object is then removed from the list structure (a linear time removal since it is a list and not a vector) and the dead object is then sent back to it's appropriate object pool where it stays until it is needed again. Since the alive object are stored in an internal data structure within the dynamic world the purpose of the alive object list is for the renderer to iterate through to perform the updates on the position and orientation on the objects.

3.5 Interfacing Game Logic

Once the entity management and base entity class were in place and integrated correctly with bullet's dynamics world contained within the physics engine module, entities could be safely inserted and removed and correctly rendered and simulated. However the issue of interfacing with the bullet simulation and detecting collisions still needed to be addressed in order to allow game logic to correctly function and influence the simulation in an interactive way as one would expect in game play. The concept of collision callbacks and per step callbacks have been mentioned previously but only so far as to establish their relationship with other components.

3.5.1 Per Step Callback

The per step callback, which occurs after a real simulation step within the dynamics world has been completed, is the safe opportunity to modify add or remove rigid body objects from the dynamics world. The reason why a callback is used instead of just performing these modifications after the update method is called on the dynamic world is because more than one simulation step may occur during that update. In order to make full use of this callback all entities inherit a virtual method which is called once during this callback for each entity. This allows an entity to self regulate and perform game logic and checks after each simulation step.

3.5.2 Collision Callbacks

The collision callback occurs when the bounding shapes of two rigid bodies within the dynamics world collide. As mentioned previously a scheme has been developed to safely promote the pointers of rigid bodies from the arguments in this callback method to entities at which point the virtual method for collisions (passing the other entity as an argument) can be called. This method can also be overridden to execute unique game logic based on what the purpose of the object is.

3.5.3 Collision Masks

However collisions cannot simply be naively defined as always occurring between certain objects, depending on the game logic certain objects might collide with some objects but not others. Bullet handles this by allowing for custom collision masks and

types for objects to be specified. When a rigid body is added to the dynamics world the mask and type of the body can be specified. Each bit in the mask variable represents a particular collision type so when evaluating a type one simply needs to do a bitwise AND between the mask of the object with the type of the object that it is colliding with, the result of this operation is zero if there is no collision and non zero if there is a collision. Bullets collision dispatcher that is assigned to the dynamics world does this operation automatically.

Unfortunately with the built in collision masks, every collision that generates a callback must generate a physical response in the environment. This behaviour is limiting and undesirable in certain circumstances within games. For example a player activating an invisible trigger in the game or collecting a power up shouldn't be obstructed or affected by the trigger or power up that it collides with. However a collision callback should still be registered and game code should still be executed upon collision. This type of collision shall be referred to as a *soft collision*. A standard collision that generates a physics response as well as a collision callback to game logic shall be referred to as a *hard collision*.

Since the default collision dispatcher module used by bullet doesn't allow for both hard and soft collision detection, a new collision dispatcher module would have to be created. Instead of replacing the existing default dispatcher the methods that required modification were virtual in the collision dispatcher's base class and a derived class could be created. The derived class overrode the needs response method which is called on every colliding pair of bodies. The arguments of the method are pointers to the two bodies colliding. These rigid bodies were tested using our earlier devised test mentioned earlier to see if their pointers could be casted up safely to entities. If one of the bodies was a plain rigid body then a hard collision is definitely registered in the physics engine and a collision on the entity is called. If both of the rigid bodies can safely be casted up to entities then the soft collision mask of entity one and the collision type of entity two are ANDed together. If the result is non negative the collision method of entity one is called and the method returns without requesting any physical response. If a soft collision is not detected the same bit check is done with the hard collision mask of entity one. If this results in a non negative result the same collision method of entity one is called and a physical response is requested.

From the above logic it's clear that any hard mask collision would be overridden by a soft mask collision of the same type since the method returns upon discovering a soft collision. Therefore when specifying masks it's best to ensure that the masks remain mutually exclusive bitwise (ie a bitwise OR should always result in a zero result) .

3.6 Character Control

Character control is an important issue within game engine design. The character module serves the specialized purpose of being controlled and affected by outside interaction through either an AI or human controller. In our particular case it serves as one of the only child class to the generic Entity class built into the core middle ware of the game engine (other extensions to the Entity class would be game specific). The design issues and challenges faced when constructing a character that can interact with

the physics world are important and relevant since interfacing with the bullet physics library successfully is integral to the success of the engine as a whole.

3.6.1 Representation

The issue of character representation poses many problems. One could argue that based on the limited input received from the user and the complexity of controlling a character correctly, the task of accurately simulating a controlled humanoid creature within an accurately simulated physics environment has no feasible perfect solution. A system which made use of individual weighted rigid bodies constrained together through the use of physics joint constraints and motors to generate forces is well beyond the scope of this paper and would introduce a whole assortment of balancing issues and render synchronization issues. Manually setting the positions of rigid bodies that make up the parts of the character is also not possible since repositioning of a body may destabilize the physics engine if it's manually forced into a space already occupied by a body. Therefore the only realistic solution for this scope is a scheme that compromises realism for less computation and small programming scope.

The popular technique for representing a character is through the use of a single rigid body in the shape of a capsule. A capsule can allow for smooth movement up small stairs and ramps and is a good fit for most humanoid characters. The obvious drawback of using such as scheme is that rendering of limbs and animation is independent of the physics representation and accurate simulation of rag doll effects and extreme movements cant be represented accurately. The capsule will also produce false positives or false negatives when it comes to finer collision detection with rays for example, depending on the chosen fit of the capsule. Despite the noticeable downfalls of this capsule shape scheme, it was decided that accurate character simulation physically and visually was well beyond the scope of this work and this compromise would have to be made.

Ideally a character should be as interactive as possible with the environment. It should be able to influence other dynamic objects in the world and be influenced itself by dynamic objects, however it cannot be completely at the whim of the physics engine since if a capsule were to be placed into the dynamics world, when the simulation started, the first thing that would happen would be that the capsule would simply fall over due to the effects of gravity. This of course would be a completely undesired effect even though it would be physically accurate behaviour for a capsule. To prevent this two solutions were explored for suitability.

The solution attempted was to use a bullet generic six degrees of freedom constraint to a dead body. The constraint was added to the capsule and defined so that pitch and roll were restricted. This method succeeded in holding the capsule upright but when further functionality such as movement of the capsule was attempted, the capsule didn't move and then the constraint would suddenly explode making the capsule spasm violently meaning that this solution was clearly unacceptable. Finally after investigating the forums and the documentation a method to restrict all angular velocity was discovered. This method kept the capsule from rotating at all from any external force within the environment but still allowed it to move and be moved within the simulation.

3.6.2 Movement

Movement within a strict physics engine poses it's own set of design issues and challenges. Since we know that all modifications to a rigid body within the simulation can only be done during the callback after a simulation step has been taken, all logic to perform movement would have to be placed in that overridden callback method. This meant that when a key was pressed the event of movement would have to be deferred until the callback occurred. In order to achieve this, booleans for the different key presses were created within the character class. When a key was pressed the event manager which is on a different thread would switch these booleans either on or off depending on the state of the keys. No synchronization on these booleans is required since there is only one thread that changes the values and one thread that reads the values and since the change of a boolean is an atomic operation, the process could proceed without any race condition or non-deterministic behaviour.

The movement expected by a player for a character should be responsive. Acceleration and deceleration of the body should be instantaneous when a key is pressed and released. In order to achieve this the velocity of the body can be set when moving a character. However this setting of velocity is only setting the resultant velocity. If the body were in the process of being knocked back and we set the velocity in the direction of the move which might be completely different, the velocity currently being experienced previously will be removed. Instead we need to get the current velocity and add our desired move velocity to this through vector addition. However if this process were to continue with us adding velocity every step that the key is down, the body will quickly accelerate beyond our desired move speed since the velocities would compound together. To deal with this problem we need to remove the previous velocity that we added and then add in our new velocity. This unfortunately can't be done through a simple subtraction since the velocity we experienced in the previous step has now been reduced in magnitude by friction. Therefore if we were to simply remove the original velocity we added, we would not be returning the body to it's previous resultant velocity (see Figure 6). To solve this we use the vector projection of our resultant velocity onto the unit vector representing the direction of the previous velocity that we added. This calculation gives us the component vector in the direction of our previous added velocity which we added. It can then be removed and a new velocity for this step added or if there is no new velocity, the removal of the residual velocity from the previous step allows us to stop the character abruptly as opposed to it continuing for a bit as if it was moving on a slippery surface (see Figure 7)**.**

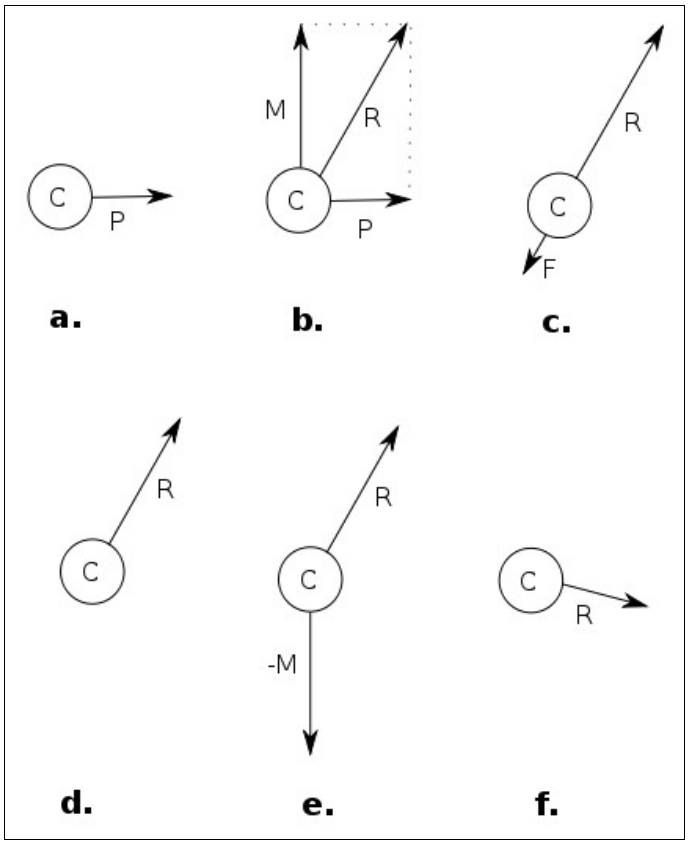

Figure 6: a) The initial state of the character C with previous existing velocity P. b) The movement vector M is added to give us resultant R. c) friction from the step is added on. d) The new resultant R after friction. e) The subtraction of the original move vector M. f) The result R of this subtraction is different to our original P velocity which is unacceptable.

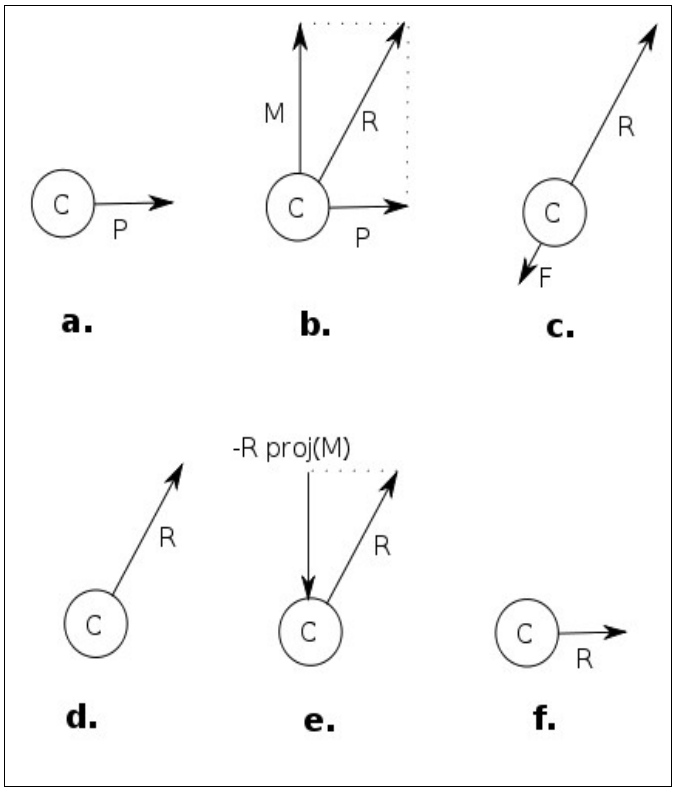

Figure 7: a) The initial state of the character C with previous existing velocity P. b) The movement vector M is added to give us resultant R. c) friction from the step is added on. d) The new resultant R after friction. e) The subtraction of the projection of the current R vector onto M. f) The result R of this subtraction is the same as our original previous existing velocity P

3.6.3 Jumping

This scheme for movement works well and give responsive character control instead of the feeling that the player is accelerating and decelerating like a space craft. However control is not only restricted to movement along a surface. Jumping must also be implemented to allow the player to navigate over obstacles and up onto higher levels in the environment. The problem introduced with jumping though is determining when jumping is determining when jumping is allowed. In order to do this we must attempt to detect when the character is on the ground.

Initially a system was implemented where the characters current vertical velocity was checked to see if it was less than a certain threshold in order to determine that the character was not falling or ascending in a jump. This had some major short comings since it would detect a false positive at the apex of a jump allowing a player to jump again in mid air and it would detect false negatives when a player was travelling up a ramp. To solve this issue a ray casting method was instead implemented where a ray was cast from the bottom of the capsule to test if the character was on top of anything, be it ground or another object. If a collision from the ray cast returned true the player was assumed to be on the ground and a jump would be allowed. The ray method does give false negatives if the ray length is too short and the character is on a steep slope (Figure 8b), however this could be mitigated by extending the length of the ray. This will in turn

cause a false positive when the character is slightly above the ground in the air (Figure 8a). The false negative with the shorter ray is probably a more realistic trade-off in this circumstance.

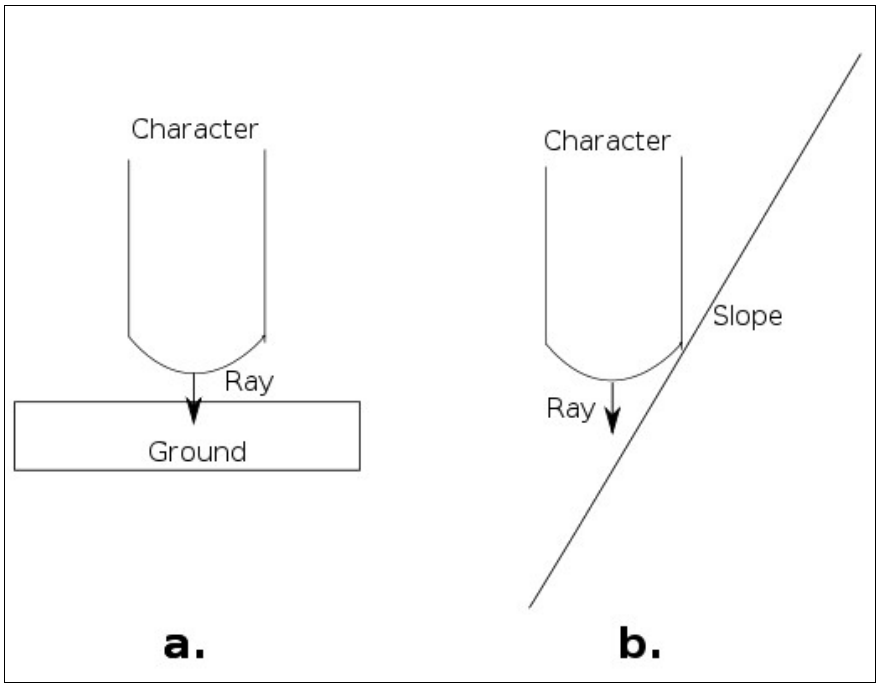

Figure 8: a) The ray registers a false positive before the character is actually on the ground. b) The ray registers a false negative even though the character is on a slope

When a character is in the air, the procedure to remove the last movement component is ignored since a body in flight should continue on it's path of motion. Similarly the amount of movement control in the air must be drastically reduced since technically the character is not running against anything and shouldn't be able to alter its movement path too much.

3.6.4 Rotation

A major issue when interfacing with the physics engine was manual rotation of a rigid body. A character should always move forward in the direction it is viewing. It should also shoot all it's projectiles in the direction it is viewing. Unfortunately the capsule representation in our implementation can't actually receive an elevation since it wouldn't make sense for the player to lean back like it was doing the limbo when looking up or lean forward with its face on the ground when looking down. Instead the elevation needed to be done separate from the orientation of the physical representation and not actually affect anything other than the direction projectiles are fired and the direction the camera is facing.

Therefore our initial implementation of viewing elevation independent to the capsule was to store an up and view vector in the character and project the view vector onto the xz plane and set the capsule rotation to this direction which would in turn orientate the model being displayed by the renderer. This would require some synchronization and the initial implementation attempted to modify the motionstate transform of the character appropriately. However this failed completely since modification of a motion state by

anything other than the physics engine is overwritten by the physics engine immediately in the next step since it only copies values to a motion state and doesn't actually read them. The safe alternative to this was to apply angular velocities to the character, but this would have resulted in a lagged control mechanism like a joystick which is less accurate and not conducive to the type of control you'd expect from mouse input. To solve this problem a simple solution was implemented where the capsules orientation is never actually used and all rotation and elevation is stored separately and accessed in place of the characters actual orientation for when rendering is required.

3.7 Overview of Achieved Implementation

The resulting system achieved from the above work was fully functional for the construction of a short game. A FPS style demo state similar to Quake 3 was created.

Characters and entities were loaded in with the entity manager and behaved correctly according to their hard and soft collision masks. The character was controllable and responsive as expected and the physics remained mostly stable producing realistic physical simulation that could rival the ability of current games on the market (Figure 9). The rendering content and maps were taken from Quake 3 resource files, so the graphical quality of the system was on a par with Quake 3 and had the exact same look as Quake 3 (Figure 10).

Overall a functional multi-threaded game could be built with system developed in this project. However, knowledge of the project and C++ would be essential to do so.

Figure 9: 1000 boxes falling all over the map and off onto the invisible bounding box at the bottom of the map in a physically realistic manner in the demo state.

Figure 10: View from the centre of one of the loaded Quake 3 maps in the demo state.

4. Experimentation

In order to investigate and test the effectiveness of our solution as to whether the engine we created performed better in a multi-threaded mode than a single threaded mode experimentation using the implementation would need to be done. In this chapter we design a suitable experiment to examine the performance and profile the thread execution. An estimate of the system resources that are used during the experiment is also noted to see if the multi-threaded hardware is really being utilized.

4.1 Experimental Design

The demo state discussed in the final section of the previous chapter incorporates all modules of the engine that had been designed and suitably includes their functionality or presence within the demo state to the capacity that one would expect in a game. The map, two characters and all the power ups on the map are present within this demo state making it a fairly good representation of the average load and diversity of objects in the game.

4.1.1 Measurement and Quantification

We now have a system with which to design an experiment with which to test the effectiveness of our system. However this brings us to the issue of how this can be measured and quantified. Traditionally the effectiveness of a multi-threaded or distributed approach to computation is measured with speed up where speed up is defined as T_n/T_1 , with T_n being the time taken to perform the computation with n cores, and T_1 being the time taken to perform the computation using 1 core. This traditional approach is ill suited to the system we're testing since there are two completely different operations being run on the two threads. Assuming we attempted to use this as a measurement we'd have to assign a fixed work load and time the execution in a single thread and in the multi thread state. Since the execution of the threads is non-deterministic, our rendering goal might be reached before our physics goal. Once this happens rendering will continue to happen which would obviously change the size of the work load which is undesirable since we're trying to fix all variables in the experiment except the dependent variable.

An alternative to the speed up measurement is to fix the time interval and measure the work load accomplished. This would mean that the performance of the two main sub systems, rendering and physics simulation, can be measured independently. The number of frames rendered by the renderer and the number of simulation steps taken by the physics engine could be counted in this fixed time period and the frequency of the updates in these subsystems calculated in hertz by dividing the number of updates (frames or steps) by the fixed time interval. This amount of updates accomplished in the time frame can be considered our dependent variable that we wish to measure in our experiment.

4.1.2 Independent Variables

The aim of the experiment being constructed is to investigate the effectiveness of our

multi-threaded game architecture over a traditional single threaded game architecture and see whether the multi-threaded game architecture performs better on multi cored hardware and makes more efficient use of all of the hardware. The way in which we can vary the variables to investigate these aims is to test a multi-threaded version on a multi-threaded machine and then again on single threaded machine as a control. This way we'll see if the multi-threaded architecture performs better on multi cored hardware than single cored hardware. However this introduces an important external variable that needs to be controlled which is the specifications of the two machines used. Anything other than number of cores within the machine that differs could potentially invalidate the results since the performance difference could be attributed to different cache sizes or clock speeds or any other hardware dependent factor or specific combination or bottleneck.

Unfortunately machines that are identical in every aspect except for numbers of cores were not available and a less desirable variation would have to do. Instead of varying the hardware as the independent variable, a slightly modified version to the system was made which executed in a single thread only in a serial loop fashion found in conventional single threaded game design. This means that only the performance of a multi-threaded game architecture would be compared to the performance of a single threaded version of the game architecture could be done with the game architecture being our independent variable. Unfortunately this also meant an improvement of performance from single core hardware to multi core hardware could not be measured.

In order to get a more complete view of how the performance varies in these two systems based on game load, a way to increase the load on both of the threads would need to be investigated and varied as a second independent variable. The main computation in the physics thread revolves around collision detection between objects and calculating the new rotational and angular velocity of the object. Therefore a high load on physics computation would be created if many objects were placed in close proximity where many collisions would occur and many collisions would have to be checked since the hierarchical bounding data structure within the physics engine would not be able to eliminate many of the collision checks. A large amount of objects again in view on the rendered that moved would seem to similarly cause the most computation on the renderer as well since the objects would all have to be updated constantly before being sent down the rendering pipeline. Therefore a second independent variable of the number of boxes in a cube of boxes positioned in the air for the demo to increase load on both main subsystems would be required.

4.1.3 Controlling Extraneous Variables

To ensure that only these two independent variables were varied and everything else in the environment remained constant, the following measures were taken in the experiment:

- No user input which could affect the deterministic behaviour of the tests
- No other unnecessary applications would be run in the background which could take up system resources.
- The simulation would run for exactly sixty seconds
- The character would be placed to observe as many of the falling boxes as

possible in order to ensure that sufficient rendering computation is required. This positioning of the character and camera would be kept constant to ensure consistency between runs (Figure 9).

- The boxes would be closely and uniformly spaced in order to exact heavy computation per step on the physics thread.
- Breaks would be taken between tests to ensure the operating temperature of the test equipment at the beginning of the test was the same for all runs. (The temperature was checked through checking the thermal zone readings from the terminal in /proc/acpi)
- The same machine and OS were used for all tests. The specifications of the machine used were:

    Acer Travelmate 4280 Laptop

    Intel Core 2 Duo CPU T5500 1.66GHz 2MB

    1GB DDR2 533MHz Samsung laptop RAM

    nVidia GeForce Go 7600 128MB PCI-E 16X

4.1.4 Preparation and Experimental Procedure

To measure the frames rendered by the renderer, a counter was incremented every time after calling the render method. To measure the physics engine, two variables were required since an actual physics step may not be done every update loop. Similarly more than one actual update could be done in an update loop. Therefore the number of times the update loop was executed as well as the number of times the per step callback was executed was counted. In addition to these exact measurements a rough estimate of processor usages throughout each benchmark was made. Ten benchmarks were made with the above specifications. 125, 250, 500, 750 and 1000 boxes using a multi-threaded game architecture and a single threaded game architecture.

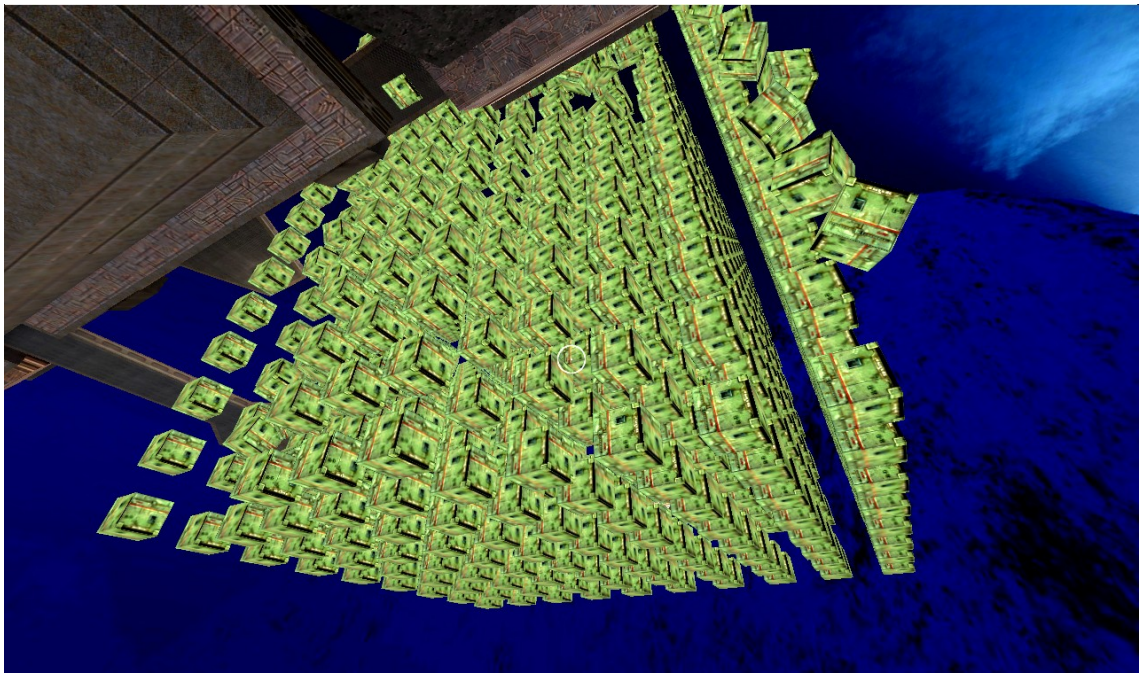

Figure 11: Screenshot of the view in the 1000 box test a few seconds after the test starts. Each of the green square like objects is a box. The boxes were spaced out and as the simulation starts they drop to the bottom of the bounding box of the map where the character is standing.

4.2 Results

The following 3 tables show the raw results for each test run:

Box Count	Actual Steps	Update Calls	Steps Per Update	Frames	Steps Per Second	Frames Per Second	Processor Load
125	3586	187971	0.02	3515	59.77	58.58	80.00%
250	3595	75110	0.05	3440	59.92	57.33	90.00%
500	3232	7743	0.42	3398	53.87	56.63	100.00%
750	2342	4477	0.52	2733	39.03	45.55	100.00%
1000	1781	2892	0.62	2284	29.68	38.07	100.00%

Table 1: Results of Benchmarks of the Multi-threaded Demo State

Box Count	Actual Steps	Update Calls	Steps Per Update	Frames	Steps Per Second	Frames Per Second	Processor Load
125	3600	3472	1.04	3472	60	57.87	50.00%
250	3599	2822	1.28	2822	59.98	47.03	50.00%
500	3330	693	4.81	693	55.5	11.55	50.00%
750	2387	419	5.7	419	39.78	6.98	50.00%
1000	1836	310	5.92	310	30.6	5.17	50.00%

Table 2: Results of Benchmarks of the Single Threaded Demo State

Box Count	Actual Steps	Update Calls	Steps Per Update	Frames	Steps Per Second	Frames Per Second	Processor Load
125	3600	4282	0.84	3589	60.00	59.82	50.00%
250	3599	4235	0.85	3557	59.98	59.28	65.00%
500	3413	1266	2.70	3548	56.88	59.13	100.00%
750	2460	708	3.47	2878	41.00	47.97	100.00%
1000	1795	620	2.90	2346	29.92	39.10	100.00%

Table 3: Results of Benchmarks of the Improved Multi-threaded Demo State

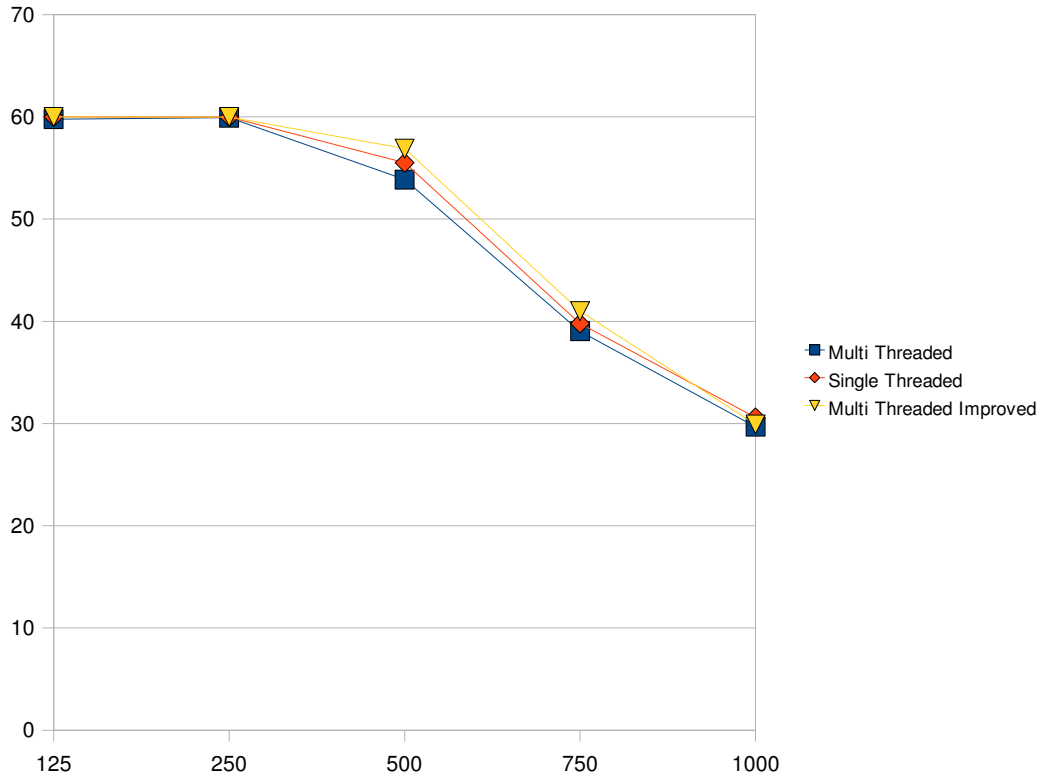

Figure 12: Physics engine performance. This graph compares the physics engine performance of the 3 systems seen over all 5 test runs. Number of boxes within the system is depicted on the x axis and simulation steps per second is depicted on the y axis

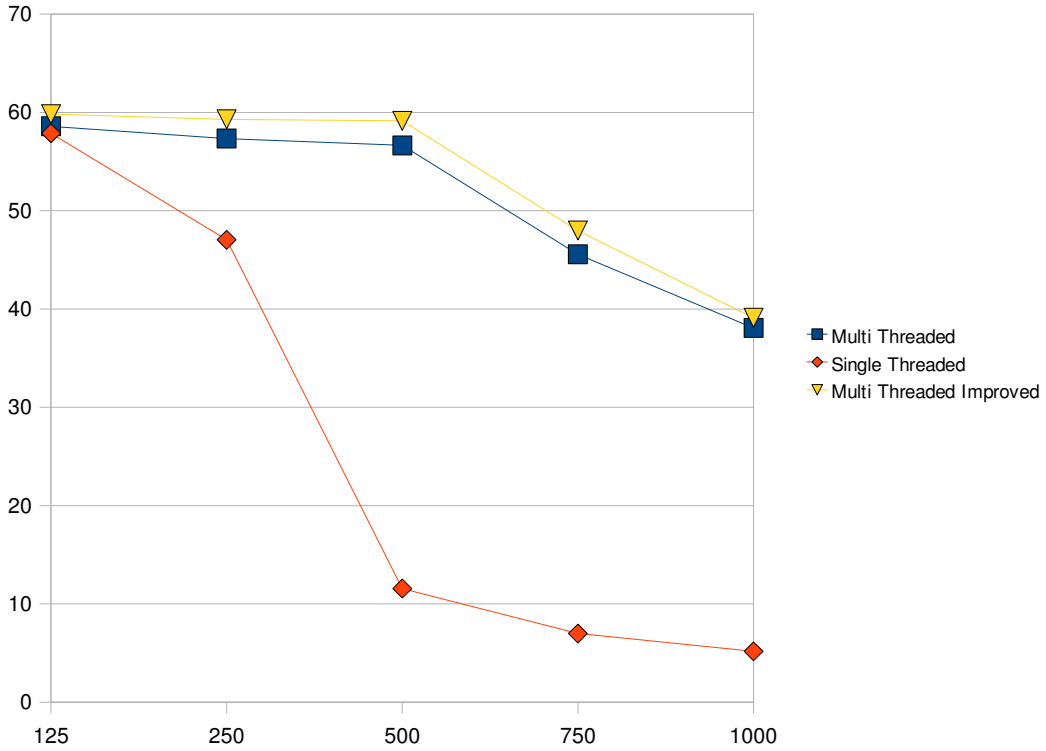

Figure 13: Renderer performance. This graph compares the frame rate of the 3 systems seen over all 5 test runs. Number of boxes within the system is depicted on the x axis and frames per second is depicted on the y axis

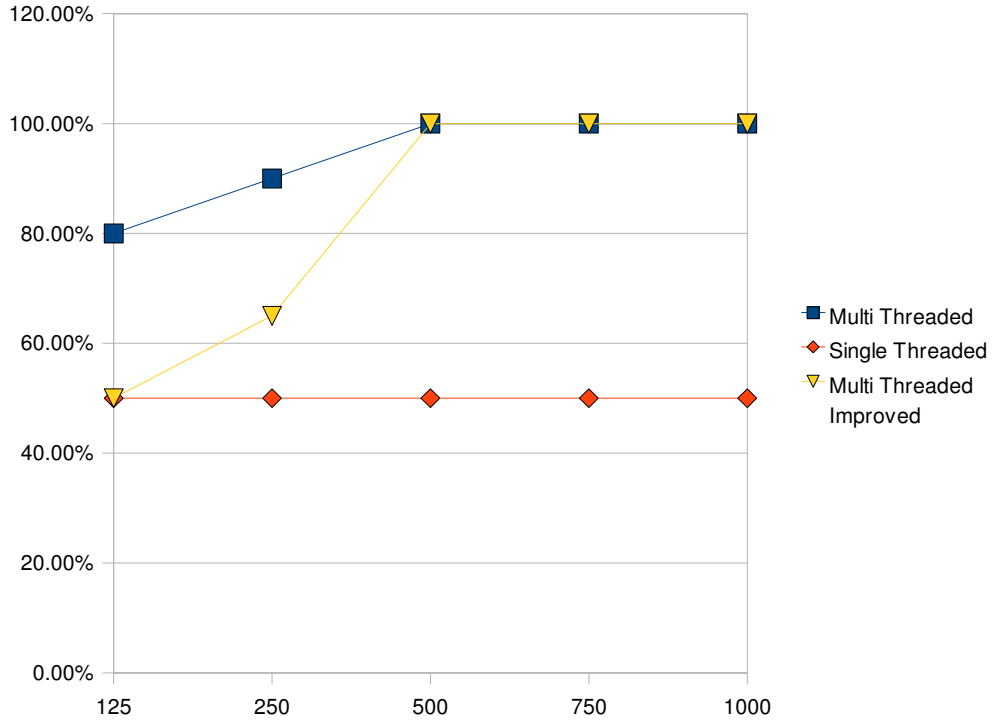

Figure 14: Observed estimate of processor load. The usage is in percent on the y axis.100% means that both cores were being fully used. The x axis is the box count

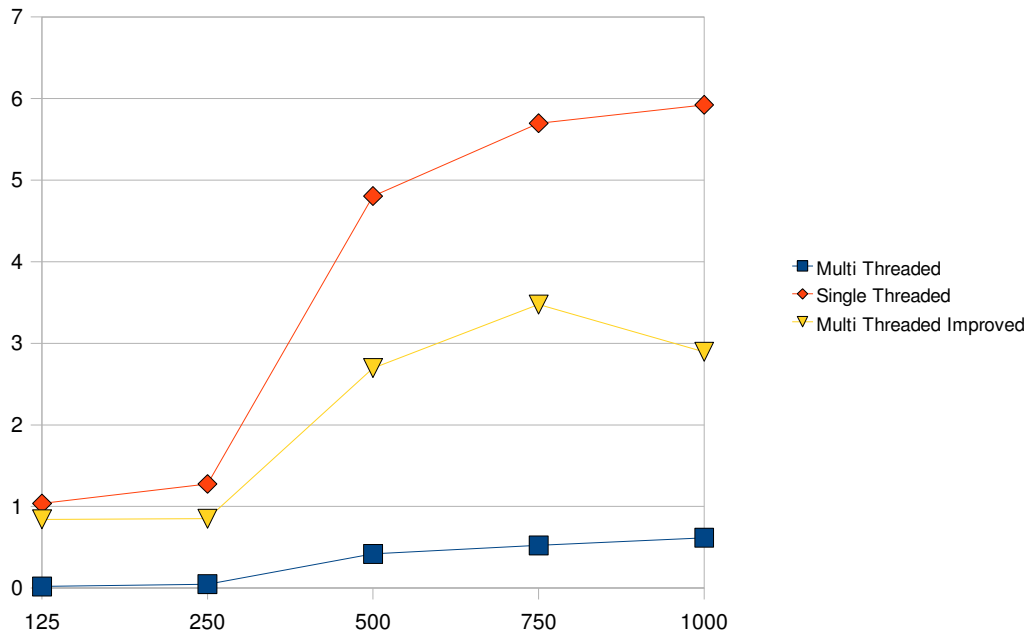

Figure 15: This graph shows the ratio of real simulation steps to physics engine update method calls. If the ratio is less than one it means the system is doing interpolations. If the number is greater than one the system is performing that many simulation steps per update call.

4.3 Discussion of Results

The initial test results of the system were very promising showing results of over 50 physics updates and frames per second for all test cases up to and including 500 boxes in the environment for the initial multi-threaded implementation (One can assume that performance of a frequency greater than 50hz is playable without noticing any lag or stutters in the game). For our smallest case of 125 boxes we see the frequency sitting at almost the theoretical maximum imposed on the system which is 60hz for both the physics simulation and the renderer (Figures 12 and 13). Given that most games would ever use more than 250 dynamic objects, let alone 500 dynamic objects in such close proximity to each other, the performance seen here is very promising. Processor load is seen to range between 80 and 100% for all test cases meaning that more than one thread is definitely executing for most of the time and the mutex locks do not restrict the thread execution to serial since the threads require low synchronization.

The single threaded implementation which was used as a control to contrast the performance gain of the multi-threaded implementation, performed about the same as the initial multi-threaded version for simulation steps (Figure 12). In some cases a minor amount better. However the number of simulation steps being done per update call escalated quickly to well more than 4 for 500 boxes and above (Figure 15). Since only one frame was being rendered per update, the computation required to step through an

excess of 4 full simulation steps begins to dominate the computation time on the single threaded implementation resulting in massive frame rate drops below 50hz on all test cases greater than 125 boxes (Figure 13). We also see processor usage strictly never deviate from exactly 50% during all single threaded tests (Figure 14) meaning that a single core was always constantly occupied by the test.

4.3.1 Improved Multi-threaded Implementation

These results seemed correct and correlate with what one would expect if one had to perform a theoretically analysis to predict how the two different implementations would run. However there was one concerning result which was the high number of update calls for the initial multi-threaded implementation in comparison to the actual number of simulation steps done. In the lower box count runs we see numbers as high as 180 000 updates in a 60 second interval (Table 1). Knowing that bullet keeps an internal clock at a constant frequency of 60hz and will only ever do a maximum of 3600 real simulation steps in a period of 60 seconds, we can assume that whenever the update method too soon an interpolation is calculated and given. Since we can only really view an object updating at 60hz since the renderer can only render at 60hz since that is it's maximum frame rate, this means that up to 150 000 unneeded interpolations are done in the multi-threaded version when the simulation calculation time is low because of few objects in the environment and the update method is called frequently.

In order to solve this problem once it was discovered through the analysis of the above results, an algorithm in the physics thread to sleep the thread until the next time a real update is needed was implemented. Of course if the calculation time reached or exceeded $1/60^{th}$ of a second the thread would not sleep.

This system was then tested using the exact same test conditions as the original testing. The results were as expected much better in this improved version of the multi-threaded implementation. The improved version outperformed both previous versions in both physics and rendering(Figure 12 and 13). This implementation also used considerably less processing resources (Figure 14) when the box count was low and simulation steps were quick to calculate since the thread now slept when it wasn't needed. However it still reached 100% during the higher box count tests where all resources were needed.

Finally the only negative side effect was that the simulation steps per update rose to greater than one for test runs of 500 boxes or more. Since this implementation had a separate rendering thread and all simulation steps still update the positions of the objects after each individual step, this doesn't actually affect physics or rendering performance and it is seen to perform the same and in most circumstance better than the original multi-threaded version while even using less system resources in lighter load cases.

4.4 Summary

The experiment designed and executed in this chapter successfully proves that the multi-threaded game architecture performs better on multi cored hardware than the single threaded game architecture. The results from the initial multi-threaded system also showed that a significant amount of computation was being wasted on extraneous numbers of extra interpolations between real physics simulation steps that don't contribute to smoother movement at all since the renderer doesn't render fast enough to

see these interpolations. The thread execution logic was then slightly modified and the results in this improved multi-threaded version were even better than the original multi-threaded results with better performance and greater efficiency.

5. Conclusion

The multi-threaded game architecture tested well against its single threaded counterpart and successfully made more efficient use of the multi-core hardware it was tested on and achieved more playable performance during periods of heavier loads than the single threaded game architecture did. However we discovered that letting the physics thread execute as fast as it could was wasteful in computation and that logic to sleep the thread when it was executing too fast on small loads was necessary in order to maintain efficiency within the engine.

Using existing libraries for common complex lower level tasks such as rendering and physics simulation we found saved on development time and made the constructed system more functional and technologically closer to current game offerings since re-inventing these systems would have been pointless and could have never be done to the standard of a professional library. The draw back of using an existing library though was that the library being implemented in the system needs to be well documented and understood in order to be used properly and to its full potential. Interfacing the bullet physics library with game logic and rendering components within a multi-threaded environment without compromising the stability of the internal simulation was difficult, but since the software was open source, it was well documented and had good community support in the form of user produced guides, tutorials and forums which made learning the library much easier than one could expect from a closed source system. A number of technological blocks that were reached when using bullet were solved through the help of other developers who used the library.

In conclusion the system designed and developed in this work successfully forms the base for what can be greatly further extended to possibly fulfil the needs of a middle ware application for the teaching or the development of games at a level that is near the level of the current commercial market using technologies that are all open source.

6. Future work

Game engine design and construction is very broadly scoped topic and what is covered in this paper is only really only an initial foundational system into what could be taken a lot further in respects to development and experimentation.

6.1 Physically Realistic Modelling

A difficult but possibly necessary extension to this work would be designing and implementing a more accurate representation of a humanoid character within the physics environment. The current Character class could be replaced with a much more complex and accurately simulated represented by multiple rigid bodies and joint constraints, with movement being implemented through physics motors instead of just velocities set on a single capsule.

In addition to this system, the renderer would have to accommodate for the rendering of each component of the body separately and might even need to use a skinning technique to produce accurate and aesthetic results for the characters.

6.2 Scripting Engine

The system presented in this paper can be used to construct a game, however a knowledge of the system and C++ would be essential in order for this to be done given the current system. The infrastructure to implement a scripting engine that held pointers to all entities including invisible trigger entities does exists. A module that received notifications from the collision callback methods of the entities and which had a method that was executed during the per step callback of the physics engine could easily be added. This module could parse a scripting language with a formalized grammar and syntax to remove the need for a user of the system to have experience with the internal operation of the system or C++. Additionally any sort of scripting interface developed would have to undergo user testing to see if it was effective in bridging the gap creative content creation and underlying complexity.

6.3 Finer Threading Within the Physics Engine

We've seen in the results and discussion of results that the physics engine comprises most of the computation within the system. In order to achieve better performance and scale to architectures greater than 3-4 cores, the feasibility of a finer grain of threading within the physics engine could be explored. The Bullet Physics Library is very modular and there are many modules that could easily be replaced by modules that make use of a multi-threaded architecture to achieve their goal in a more efficient and faster way.

7.References

[1] Tulip, J., Bekkema, J., and Nesbitt, K. 2006. Multi-threaded game engine design. In *Proceedings of the 3rd Australasian Conference on interactive Entertainment* (Perth, Australia, December 04 - 06, 2006). ACM International Conference Proceeding Series, vol. 207. Murdoch University, Murdoch University, Australia, 9-14.

[2] Hecker, C. 2000. Physics in computer games (title only). *Commun. ACM* 43, 7 (Jul. 2000), 34-39.

[3] Boeing, A. and Bräunl, T. 2007. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th international Conference on Computer Graphics and interactive Techniques in Australia and Southeast Asia* (Perth, Australia, December 01 - 04, 2007). GRAPHITE '07. ACM, New York, NY, 281-288.

[4] Luque, R. G., Comba, J. L., and Freitas, C. M. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the 2005 Symposium on interactive 3D Graphics and Games* (Washington, District of Columbia, April 03 - 06, 2005). I3D '05. ACM, New York, NY, 179-186.

[5] http://www.bulletphysics.com/Bullet/wordpress/ .Last Accessed 21 July 2008.

[6] http://www.tokamakphysics.com/index.htm . Last Accessed 21 July 2008.

[7] http://www.rowlhouse.co.uk/jiglib/ . Last Accessed 21 July 2008.

[8] http://www.ode.org/ . Last Accessed 21 July 2008.

[9] Marowka, A. 2007. Parallel computing on any desktop. Commun. ACM 50, 9 (Sep. 2007), 74-78.

[10] http://www.gamasutra.com/features/20051117/gabb_01.shtml . Last Accessed 21 July 2008.