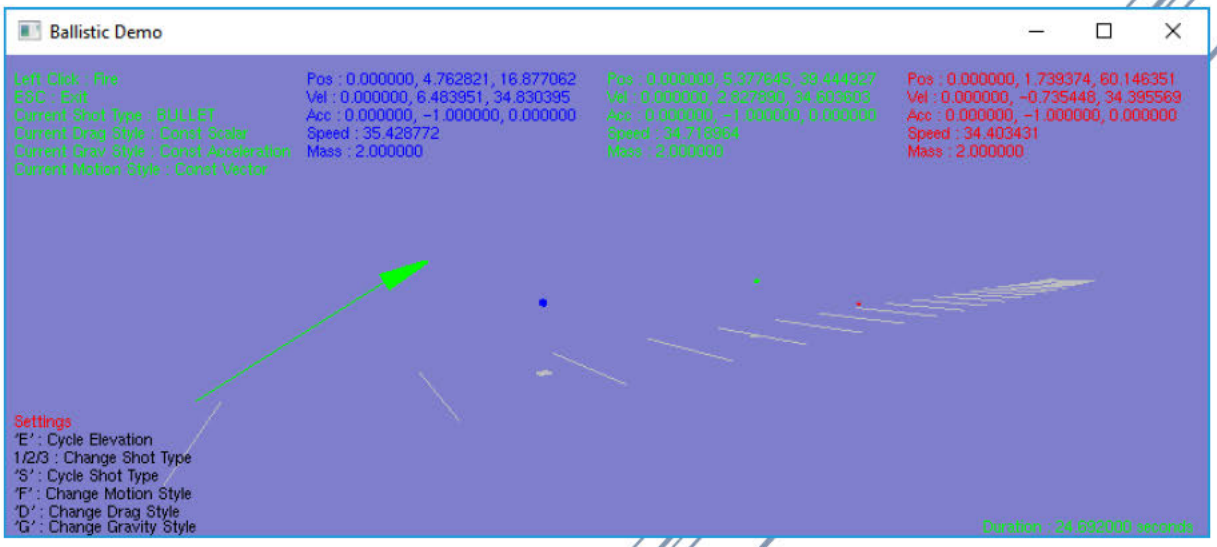
PHYSIM : PROJECTILE PHYSICS ENGINE

[redacted]

Supervisor: Silvester Czanner



Computer Games Technology

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work.

Abstract

The purpose of this project it to identify strategies used in physics engines at the two primary levels of accuracy. Game Engines which opt for speed of execution over accuracy and Simulation which opt for absolute accuracy to the best of its ability. Then, after analysing the different approach of the two, create a library focusing on a small aspect of a physics engine and implement dynamic accuracy execution of the methods it contains.

The goal is to produce an extensible product capable of quick calculation for game systems and more accurate calculations for simulation within a single library and accessible through configuration criteria.

The primary focus of the project will be the handling of a particle system and projectile motion. Exploration into how these calculations are carried out and managed by a system, will be explored and where appropriate, a selection will be incorporated into this system.

This hopes to bridge the differential gap between game engines and simulation engines for a projectile. Firstly, by identifying different methods for mapping physical attributes of a particle. Secondly by successfully implementing a 3D capable system which can recognise particles and add a projectile motion behaviour to them through the desired method chosen.

The outcome of the project did successfully implement a projectile motion system capable of upscaling its accuracy whist being designed in a way so that more “variable accuracy methods” can be implemented smoothly. An attempt to create simulation grade physics was unsuccessful as the relative physics and maths involved became too complex to implement into the system without time and project scope issues. Although it is still very possible to implement more systems and more accuracy functions in the future.

Insight and understanding as to why game engines use heuristic algorithms and so called short cuts is understandable when having to manage complex scenes of potentially thousands of different objects interacting with each other. And some concepts in physics are still approximated even for a simulation, meaning realism is increasingly closer to achievable but not fully at this moment in time.

Table of Contents

List of Figures

List of Tables

List of Equations

1 Introduction

1.1 Project Background

The demand for graphical modeling systems has seen a massive increase over the last few decades. With the advancement in technology and system/hardware architecture, this task is becoming increasingly desired and easier to replicate amongst a variety of devices. These advancements "*has also included the emergence of open standard application program interfaces ("APIs) for rendering, such as OpenGL from Silicon Graphics and more recently Direct3D from Microsoft.*" (Georges, et al., 2000). Thus, the technology for creating a GUI application for operating the proposed library is readily available.

The concept of using computers to render intense or complex graphical models is no new feat. However, one would struggled to find a system which is capable of accurately carrying out both realistic and alternative realism in terms of motion. There are many examples of systems and libraries modeling projectile motion, and also many graphic API's to represent them such as DirectX, OpenGL/OSC, NXA or more "heavy weight" game engines such as Unreal, Unity, Frostbite, Cry Engine and more.

What is noticeable here, is that larger graphics engines that may have such functionality included may not consider all true physical aspects of the motion. It could be argued that this is since large Graphics engines have a lot to model in a small amount of time, and optimization must be introduced to achieve this. On the other hand, lighter weight Graphic API's do not have a specific catered system to carry out these tasks and it is up to the developer at the time to devise such functionality.

As motion is a vast subject, this projects' primary goal is to explore projectile motion. However, that is not to say that provided this system is completed and reliable, it may be amended over time to include more physical characteristics.

1.2 Aim

To create a library, designed specifically for the rendering and mapping of projectile motion given a specified 3D plane and custom physical parameters.

This library will be capable of modeling motion simulations within a designed environment, or used to map kinetics of a fabricated object in a non-realistic game environment such as an arcade style space ship game.

1.3 Objectives

Objectives throughout the development cycle have been split into some primary categories, Analysis, Design, Development, Evaluation. Below is a representation of the key objectives within each of these categories to be carried out.

1.3.1 Analysis

- Analyze any pre-existing projectile modelling strategies currently used in the computing industry.
- Identify (if any) comparable limitations between projectile modeling in different platforms.

1.3.2 Design

- Decide on what language(s)/Frameworks will be used to develop this system.
- Design a library which improves on any of the weaknesses identified in the evaluation of previous concepts.
- Produces a list of dynamic constraints that can change the running mechanics of the library.
- Plan a simple, configurable GUI to demonstrate the libraries functionality.
- Suggest other possible uses of the library and provide a brief synopsis of their relevance.

1.3.3 Development

- Produce a draft structure of the proposed solution, thinking about naming conventions, namespaces and (if plausible) inheritance.
- Decide an input/output structure of displacement and velocity data of the projectile, this should be reusable and possibly recursive.
- Implement basic “SUVAT” (or similar alternatives) functionality to the library.
- Create a basic GUI to run testing on the developing library.
- Further develop the library to include ballistic and rotational characteristics for projectiles traveling at high speeds/distances.
- Further develop the GUI by adding a constraints menu to alter the parameters of the libraries modeling (per projection).

1.3.4 Evaluation

- Discuss the realistic feasibility of the produced system.
- Evaluate the possible uses of the library and compare them to the intended areas.
- Highlight the strengths and weaknesses of the library and any potential improvements.
- Compare the final product to the original aim and conclude its effectiveness.

2 Literature Review

2.1 What is a Physics Engine?

A Physics Engine is the term used to describe a piece of software whose sole purpose is to model physical phenomena in response to interactive stimulus. A physics engine should be well generalized, and usually refers to a system capable of coping with and modelling a generic and wide range of scenarios without zeroing in on a specific physical effect.

This means a physics engine would deal with projectile motion, collision detection and impulse, particle simulation and most other physical events to a close, realistic standard.
The use of a physics engine in games can inspire powerful immersion with players due to the realistic feel of the game. Its common knowledge that a game with powerful realism would dominate a game with less realism in terms of popularity, if the game was designed to be portrayed as a real-world scenario.

2.2 Why use a Physics Engine?

A good example for this is Half-life. "*Remember the movable crates in Half-Life 1? They formed the basis of only one or two puzzles in the game. When it came to Half-Life 2, crate*

physics was replaced by a full physics engine. This opens all kinds of new opportunities. The pieces of a shattered crate float on water; objects can be stacked, used as movable shields, and so on." (Millington, 2010)

In earlier years, the equivalent of a physics engine was a system developed to model the physical properties of a more specific action. For instance, this may at a time have been for a cross bow shooting game. This example would require developers to accurately map the effects over time of a cross bow bolt. Whilst this system would have worked well, it was still bespoke. This approach would have been used for both game development and simulation (physical studies) developments.

As time and technology progress, developers have managed to create "Physics Engines". The difference here is that a physics engine is slightly more imperative than older bespoke systems. *"The physics engine is basically a big calculator: it does the mathematics needed to simulate physics. But it doesn't know what needs to be simulated."* (Millington, 2010). Without clear (and detailed) direction on what the current scenario is and the characteristics of any objects involved, a modern-day physics engine would simply do nothing. A physics engine is a powerful tool however it may or may not be relevant depending on the complexity and scope of the game/simulation in question.

2.3 Programming Language Selection

There are many languages to choose from when programming a game or simulation. Years ago, variations of these languages existed but were not desirable for game development. Newer

and better machines combined with intelligent IDE's, code metrics and a good quality compiler has somewhat nullified this constraint to an extent.

Languages like the ones in this section and other like python and MATLAB all fall into one of two categories:

- Languages that can reduce the time spent coding by the programmer
- Languages that can reduce the time spent by the computer whilst running

These categories are sometimes referred to as Declarative and Imperative languages (respectively). With modern computers having plenty of power, most programmers would usually opt for a Declarative coding language. However, where game and simulation systems are involved, speed is of the essence. The faster a computer can carry out (sometimes complex) calculations, the smoother, more realistic and more believable the simulation in progress is. It is for this reason that some less appealing languages are used for such a task.

In extreme circumstances the most complex mathematical calculations of a system would be coded directly into assembly code, this saves time by producing faster code closer to the kernel than a compiler could produce. This is still the case but much less so since in recent years with the further optimization of compilers and the speedy architecture of modern computers.

2.3.1 C/C++

Putting modern day compilers aside. C and C++ are two of the fastest languages a computer can handle. This is essentially because they are "barebones" (C more than C++), meaning the languages don't possess the means to natively import heavy, computationally expensive

libraries. Whilst both syntaxes are similar, C++ is an update/variation spawned from the original C language. Coding in C can indeed speed up computational tasks considerably. This is partially because the language is closer to machine code than most. Unfortunately, friendly coding concepts like Object Orientation paradigms, Generic paradigms, Inheritance, Exception handling and others are not available in C. Some of these concepts have been regarded today as fundamental in the realm of good program architecture and can make C++ the more desirable language to some. A good depth of knowledge of C++ along with some well-structured/optimized code and a good compiler can make C++ systems at the least, as fast as C. That is not to say that C++ is a whole lot easier to use in comparison to some languages.

2.3.2 C#

C# is another variation of C, separate from C++. C# is known to be a powerful choice when dealing with WinForms and WPF Applications, it is (slightly) less known for developing simulations and games but with a good compiler, this is still possible. Large Game engines that have most of the complex and hidden mathematics will use C# for development scripting within the engine. It is one of the most developed languages in the computing industry. It supports most programming concepts. Combining C# with the Visual Studio IDE allows for clever garbage collection, error handling, Real time Code Diagnostics (VS 2015) and nested interface implementation. However, interfaces are just syntax sugar and a way to allow multiple inheritance an issue that is not present in C and C++. Whilst syntax between C, C++ and C# is similar, C# is the easiest of the three to code and read. Although mathematical calculations are very possible, they are not as efficient as C and C++ and can require extra libraries to carry out these tasks, making the system more computationally expensive. And obviously requiring more memory to load larger libraries of functions.

2.3.3 VB.Net

Visual Basic is Microsoft's independently developed language with very similar capabilities to C#. However, VB has attempted to become a more declarative language by changing a lot of its syntax to real world English. Due to this, dispute has long persisted as to the syntax readability. Some would argue that the codes literal readability is good but other opinions state that replacing coding standard formulae for "words" makes code subjective and harder to overall interpretation. Either way, VB is a well-known language though less popular than its competitor, C#.

2.3.4 Java/J-script

Like VB.NET, Java is another language derived from C and C++ with C style syntax. Java is a long-standing language used by many. However, claiming it to be the most popular would be highly speculative. Java is a competitor to C# and VB, though for a long time it has been the underdog between the three due to its lack of framework development. Simply put, it just didn't have as many importable libraries of C# and VB, and so it was hindered in terms of capabilities.

In recent times this is less of the case, as well as showing strong support for form based applications, Java now supports its own graphical framework, LWJGL (Light Weight Java Graphics Library) a system designed to invoke the use of graphical frameworks such as Open GL and Open AL. Though Java in terms of raw capabilities is (arguably) seemingly on the back foot for system development, Java-Script, a variation of Java is one of the most sought after skills in the development world. Java-Script is highly popular amongst web developers. In summary, Java is capable of being the language of choice for simpler and more "light weight"

games/simulations. However, its efficiency may be expected to drop as game complexity increases.

2.4 Physics Engine Fundamentals

2.4.1 What is a Particle System?

Each particle has a predetermined *lifetime*, during which it can undergo various configurable changes. It begins its life when it is generated (*emitted* by its parent system). A system emits particles at random positions within a region of space and can take various simple forms/shapes. The particle is displayed until its predetermined *lifetime* is up, at which point it is destroyed.

Individual particle properties can change over time in addition to emission rate and life span. Each has a *velocity* vector which can update the position of the particle per frame as it changes. The velocity can be changed by *forces* and *gravity* applied by the system itself or dynamically due to differentials within the scene. *Rotation* is also a key trait of a particle (especially in 3D space, this can change over time due to varying factors. To avoid non-realistic immersion, particles must be created and destroyed smoothly, therefore their colour characteristic includes an alpha component, so a particle can be made to fade gradually in and out of existence when such an effect is necessary.

Used in combination, particle dynamics can be used to simulate many kinds of fluid effects convincingly. This can be represented as many particles accumulated to form a large body such as water, snow, fire/smoke all of which will have slightly different characteristics. Or larger objects such as bullets. A good particle system will be able to react to varying conditions or interference from other particles, events, or objects.

2.4.2 What are Rigid Body Physics?

A Rigid Body is the physical representation of a Body or object, impervious to physical distortion. Meaning any two points inside the object will always have the same constant distance apart. Rigid body physics is used to map properties and state of a body more accurately (opposed to dynamic randomization in a particle system) by considering as many physical aspects as possible. A well-structured Rigid Body system allows for most fundamental operations to be handled such as: translation, rotation, transformation, positioning, and linear and angular acceleration.

In games *RBP* (Rigid Body Physics) is a critical part of realism and immersion, it works in turn with collision detection, impulses, and momentum to create a realistic state of events on an object under some form of physical manipulation. *RBP* is a vast area of knowledge, with a very large spread of physical phenomena to consider for realism. It is in these such places; techniques are sometimes developed to estimate or summarize certain motions to save on technical debt for complex situations. *RBP* is essentially an extension of a realistic particle system to account for orientation, rotation, or inertia.

2.4.3 What is Collision Detection?

"When several objects are moved about by computer animation, there is the chance that they will interpenetrate. This is often an undesired state, particularly if the animation is seeking to model a realistic world…" this "*is fundamentally a kinematic problem, involving the positional relationship of objects in the world."* (Moore & Wilhelms, 1988). The purpose of collision detection is to determine when these events will arise and react accordingly. It can be a hard

concept to conceive because computers do not have any concept of spatial awareness and therefore cannot possible, independently determine if two objects are to collide and/or interact.

Unfortunately, there is no single optimal function to determine collision between objects and this complexity increases in 3D space. Different algorithms are used to represent different scenarios of collisions and in most cases, some heuristics or assumptions are considered to ease the technical debt of the procedure.

2.5 Existing Game Engines

2.5.1 Unity

Unity is a more recent multiplatform game engine developed by Unity Technologies. In the past, Unity was deemed as an entry level game development suite with good compatibility and free to use, making it ideal for game developers to tinker with for personal and low-mid tier commercial use. Recently Unity has up scaled to become one of the most well-known game engines in the global market. "*Unity plays an important part in a booming global games market. The Unity engine is far and away the dominant global game development software. More games are made with Unity than with any other game technology. More players play games made with Unity, and more developers rely on our tools and services to drive their business.*" (Unity, 2016). Unity now boasts over 5 Billion downloads.

2.5.1.1 Particle System

The Unity engine (for the most part) recognizes particles as part of a larger swarm or system which has generated or emitted them. This mean that particles all have similar characteristics with slightly varying behavior. This particle system includes many possible characteristics and attributes. To attempt to logically relay these features to developers, Unity Technologies have divided these characteristics into "Modules". These modules (taken from the Unity documentation) are as follows:

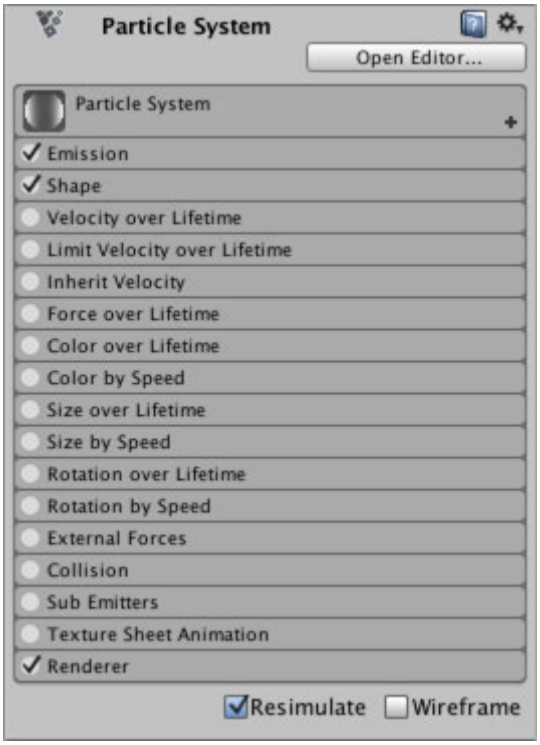


Figure 2-1 Unity Particle System component list

(Unity, 2016)

In the above picture, the particle systems characteristics are conveniently arranged into these modules. Most of the system attributes are dependent on time differentiation, collision being the most obvious exception collisions will be discussed separately as they are a fundamental system on their own with high complexity. Some other differentials like distance are also present, distance can affect emission rates from an instigating point in such systems as tire smoke, fire smoke, water spray/mist. These examples will require different densities or emission rates of particles as distance from a point increases.

It is worth noting here that the particle system on a whole contains much more characteristics than desired by this projects functionality. The Unity particle system has been refactored many times over to include these features. However negating swarm logic and concentrating on a single particle would be comparable to a desirable particle/projectile design.

Unity also includes many global particle settings; these settings relate to every particle within a single system. They include such values as Looping (Boolean), Prewarm (Boolean), Duration (Float). A full list can be seen below with their respective descriptions, provided by Unity Technologies.

Property:	***Function:***
Duration	The length of time the system will run.
Looping	If enabled, the system will start again at the end of its *duration* time and continue to repeat the cycle.
Prewarm	If enabled, the system will be initialized as though it had already completed a full cycle (only works if *Looping* is also enabled).
Start Delay	Delay in seconds before the system starts emitting once enabled.
Start Lifetime	The initial lifetime for particles.
Start Speed	The initial speed of each particle in the appropriate direction.
3D Start Size	Enable this if you want to control the size of each axis separately.
Start Size	The initial size of each particle.

Property:	*Function:*
3D Start Rotation	Enable this if you want to control the rotation of each axis separately.
Start Rotation	The initial rotation angle of each particle.
Randomize Rotation Direction	Causes some particles to spin in the opposite direction.
Start Colour	The initial colour of each particle.
Gravity Modifier	Scales the gravity value set in the physics manager. A value of zero will switch gravity off.
Simulation Space	Toggles whether particles are animated in the parent object's local space (therefore moving with the parent object) or in the world space.
Scaling Mode	Use the scale from the transform. Set to Hierarchy, Local or Shape. Local applies only the particle system transform scale. Shape mode applies only the scale to the start position of the particles.
Play on Awake	If enabled, the particle system starts automatically when the object is created.
Max Particles	The maximum number of particles in the system at once. Older particles will be removed when the limit is reached.

Table 2-1 Unity Particle System module list

(Unity, 2016)

These are the characteristics that make up a particle system in Unity. It is a useful fundamental for real immersion, and with so many modular properties, they can be configured to do many effects.

A good strength of the Unity Particle Engine is its use of randomization techniques. In an explosion, you may want some small particles to fly away from the explosion quickly, whilst larger fiery objects may be heavier and thus have less kinetic energy or not move at all. Unity allows many properties to be set to randomize between two points. This is good to avoid particle behavior repetition as in real world physics, the chances of two particles behaving the same are exponentially improbable. To expand on this feature, Unity allows these constraints to be set be defining a minimum and maximum value, this maximum and minimum value also have the option of being a curve representing the magnitude of a property against time. Essentially altering the minimum and maximum values to randomize between. This is a great technique to demonstrate dissipation of particles.

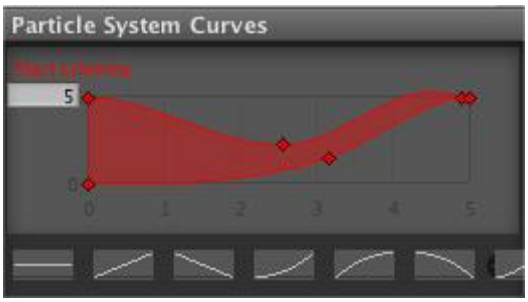


Figure 2-2 Unity Particle System Curve Editor

(Unity, 2016)

Above is an example of a start lifetime property minimum and maximum constraint changing over time. This shows how as time moves forwards the randomize range changes for particle emission.

2.5.1.2 Rigid Body Physics

Like the Particle System, Unity have conveniently modulated RBP system into the following categories:

Property:	*Function:*
Mass	The mass of the object (in kilograms by default).
Drag	How much air resistance affects the object when moving from forces. 0 means no air resistance, and infinity makes the object stop moving immediately.
Angular Drag	How much air resistance affects the object when rotating from torque. 0 means no air resistance. Note that you cannot make the object stop rotating just by setting its Angular Drag to infinity.
Use Gravity	If enabled, the object is affected by gravity.
Is Kinematic	If enabled, the object will not be driven by the physics engine, and can only be manipulated by its **Transform**. This is useful for moving platforms or if you want to animate a Rigidbody that has a **HingeJoint**attached.
Interpolate	Try one of the options only if you are seeing jerkiness in your Rigidbody's movement.
- None	No Interpolation is applied.
- Interpolate	Transform is smoothed based on the Transform of the previous frame.
- Extrapolate	Transform is smoothed based on the estimated Transform of the next frame.
Constraints	Restrictions on the Rigidbody's motion:-
- Freeze Position	Stops the Rigidbody moving in the world X, Y and Z axes selectively.
- Freeze Rotation	Stops the Rigidbody rotating around the local X, Y and Z axes selectively.

[UYR 2016]

Several properties have been removed from this table as they concerned collisions, these will be addressed separately. This system is well designed and handles the task of smooth immersion admirably. Rigid body physics is used by the Unity Engine when objects are to behave in an unpredictable manner.

Some interesting aspects of the RBP are:

Kinematics

Unity has acknowledged that the possible issues of manual interference with some objects characteristics may promote undesirable activity through the physics engine (regarding that game object). The introduction of the "isKinematic" property can disable physics engine interference; this is a good tool for level design. Unity uses a moving platform as an example of when scripting may be needed to manually transform rigid body objects.

Sleep

Unity allows rigid bodies to be put to sleep, this is an optimization technique to help deal with large counts of rigid bodies in a single scene. An object which is sleeping no longer requires the CPU to recalculate its physics until sleep is deactivated. This will be used on objects that are not actively being manipulated by the physics engine. Sleep can be automatically disabled when a force or collision is exerted in on a sleeping object, which will bring the object back into scope of the physics engines update routines. However, it has been noted that non-rigid body objects (static colliders) interacting with a sleeping rigid body, may not auto wake the

object. At this point script, can be used to overcome this issue and explicitly wake the object up.

2.5.1.3 Collision Detection

Collisions in unity are part of the rigid body physics model and are in turn handled by the physics engine rather than manually. There is lots of documentation on optimization techniques concerning collision detection and it is up to the developer to monitor the CPU profilers, watching for any warnings or excess time spent calculating physics.

This is a broad area to document and consists of many complex mathematical functions. Though the point to portray here is that realistic collisions with accurate meshes can become so expensive that they are not worth using. To combat this dilemma, Unity have split collision detection into several categories:

- Box Collision
- Sphere Collision
- Capsule Collision
- Mesh Collision
- Wheel Collision
- Terrain Collision
- Compound Collision

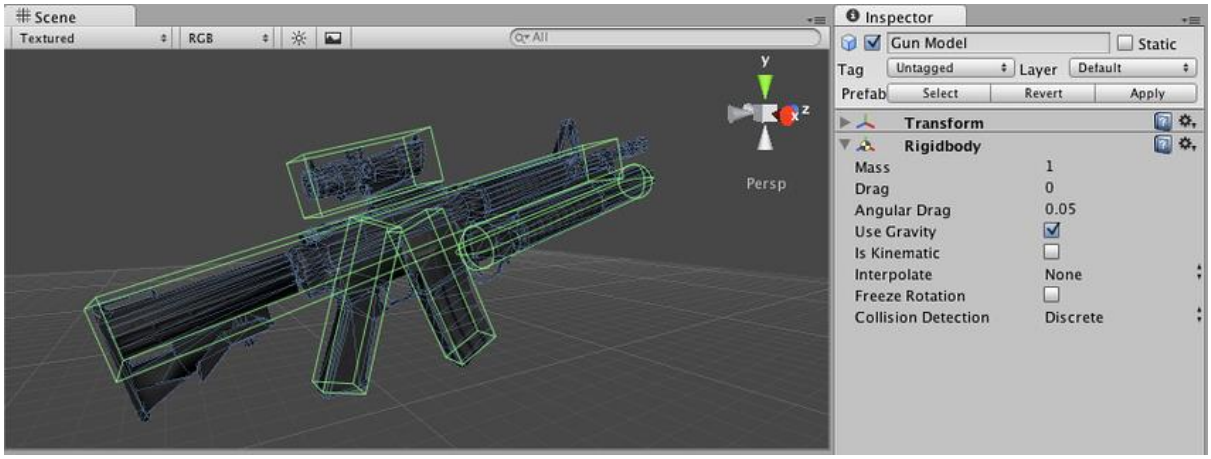
It's clear that each collision body that would attach to the game object here are of different shapes. The reasoning behind this is a simple one. Collision detection is so expensive, its more

efficient to sacrifice minute accuracies for greater computation speeds. The fundamental idea of programming collision detection is that it's easy to manage collisions between convex shapes; whereas concave shape collision detection is incredibly more complex.

By developing different collision objects, we can box off objects with these collision objects to cover a rougher, simpler volume of the game object. Each different collision object (box, sphere, capsule) requires different mathematics to account for its dimensional characteristics, spheres having a uniform radius being the easiest.

Terrain and mesh collision becomes slightly more complicated as both surfaces can be fragmented and contains complex vertex triangulation. Separate techniques can be implemented to account for these instances.

To cover more complex and irregular shaped polygons for example a gun, Unity offers compound collision functionality. This means that a developer can connect an accumulation of simple collision objects to cover the volume of a larger more complex game object, see below:



[UYR 2016]

As you can see, the body of this gun has been encapsulated with four different box colliders and a capsule collider.

The collision system also includes “Continuous Collision Detection” this can be an expensive technique that will require more computation per update from the physics engine. In short, this is used on box, sphere, and capsule colliders to manage scenarios where colliders may have passed each other between frames. Meaning a pair of colliders begin in one arrangement and due to a factor, such as high velocity, they may pass each other before the next frame is rendered. “*Note that continuous collision detection is intended as a safety net to catch collisions in cases where objects would otherwise pass through each other, but will not deliver physically accurate collision results”*. [UYR 2016]

2.6 Simulation Engines

The term Simulation Engine implies that said engine will have absolute accuracy. Whilst they will be much more inclined towards true figure, it’s still quite possible that they do not represent a 100% truth. Fortunately, this projects functionality (particle projectile motion) can be represented quite accurately with today’s mathematics. It is also much harder to find relevant data in term of simulation engines as each one is usually catered towards a specific mapping of physical behavior. They are not designed to handle large scenes or generic functionality on a whole.

2.6.1 Cyclone

One Engine of interest here is the segmented Cyclone engine (open source). Whilst this engine is designed to run as a simple game engine. Its accompanying documentation justifies a large amount of its chosen functionality. Meaning some concepts could be imitated and modified to

increase accuracy. The Cyclone engine can perform many more tasks than this project will aim to achieve. However good design of code should allow for extensibility in future developments. The more interesting attributes of Cyclone are its Vector, Particle, and Force Generator classes.

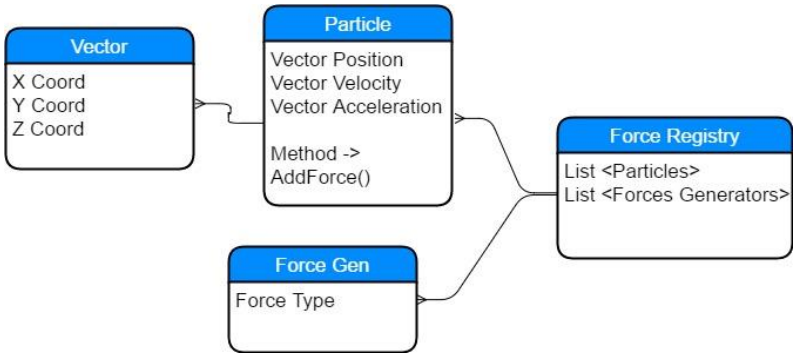


Figure 2-3 Cyclone basic entity relationship diagram

Above is low level entity relationship diagram of the cyclone particle design structure. Each particle holds an acceleration attribute which is computed from its resultant forces. The resultant forces are for each particle are updates each from through a physics update procedure in the Force Registry class. The base Vector class is instantiated within the Particle class multiple times to represent any non-scalar attributes. These particles are them kept inside of a separate world register.

"We could add this into each particle, with a data structure such as a linked list or a growable array of generators. This would be a valid approach, but it has performance implications: either each particle needs to have lots of wasted storage (using a growable array) or new registrations will cause lots of memory operations (creating elements in linked lists). For performance and modularity, I think it is better to decouple the design and have a central registry of particles and force generators" (Millington, 2010)

The possibility of adding different types of forces or directly setting acceleration, velocity and position can demonstrate how changing a scale of accuracy would set a different value of the particle which would need to be handled by the engine. More of this will be explained in the core attributes section.

2.7 Core Attributes

All the engines mentioned possess core mathematical functionality of a single particle and then extend to its required behavior. Meaning either the inheritance and addition of extra attribute to create rigid bodies or a grouping and randomization system for particle swarm effects such as smoke and sparks. Unfortunately, it was hard to find any specifics through the documentation of the heavy weight game engines. Although naming conventions and execution pipelines may vary, it is certain that they will all implement their own Vector and Particle/Actor classes.

Vector Calculus is a law of mathematics; therefore, this area of math's is going to be mostly consistent across all the highlighted engines. This includes such variables as (but not limited to):

- Vector addition/subtraction
- Dot Product
- Cross Product
- Vector Multiplication
- Vector Scaling

As an unwavering rule, each update of physics to an object in any game engine is effectively setting its position in world space. The difference here is how the position is derived. Depending on which attribute of the objects physical properties is altered.

To differentiate a value is to measure its rate of change. The rate of change of an objects position is its velocity. The rate of change of an objects velocity is its accelerating. This can be reversed through Euler's Integration.

If we think in terms of velocity, a position at a certain time can be call *p*. It's fair to say that if an object is moving then its position *p* will change over time *t*. Therefore, a rate of change of position with respect to time equals some vector value. We call this velocity.

$$v = \frac{dp}{dt}$$

Equation 2-1 First derivative of position with respect to time

Velocity can also have a rate of change, this is sometimes described as the derivative of velocity or the second derivative of position. In the same way as before, a change in velocity *v* over time *t* is equal to another vector. This is defined as acceleration.

$$a = \frac{dv}{dt} = \frac{d^2p}{dt^2}$$

Equation 2-2 Second derivative of position with respect to time

Differentiation can be added repeatedly to calculate the rate of change of each derivative with respect to time. The next derivative of acceleration is called Jerk which would equate to.

$$j = \frac{da}{dt} = \frac{d^2v}{dt^2} = \frac{d^3p}{dt^3}$$

Equation 2-3 Third derivative of position with respect to time

However, Newtons second law proves that Acceleration of a given object is dependent on the force applied to it and the mass of the object.

$$\vec{F} = m\vec{a}$$

Equation 2-4 Newtons 2nd law

Using Euler's integration, this process can be reversed from acceleration all the way to position. The acceleration can be calculated using a rearrangement of Newton's Second Law.

$$\frac{\vec{F}}{m} = \vec{a}$$

Equation 2-5 Newtons Second law rearranged

Engines such as Cyclone and Unity will/can cut corners and improve execution time by explicitly setting acceleration or velocity and then deriving a position through integration. Depending on the scenario this can cut a lot of mathematics out of a scene update loop, especially if the object count is high.

D'Alembert's Principle is a variation/extension of the Newtonian second law. D'Alembert stated that **"***The principle that the resultant of the external forces F and the kinetic reaction*

acting on a body equals zero. The kinetic reaction is defined as the negative of the product of the mass m and the acceleration ***a****"* (Schmidt, 2014).

$$F - ma = 0$$

Equation 2-6 D'Alemberts Principle (where -ma is the kinetic reaction)

This means that the sum of all forces acting on an object can be replaced by a single force. Calculating a resultant force acting on an object is a key fundamental.

2.8 GUI

Because This project is primarily aimed at creating an extensible library, and the GUI will have the purpose of demonstrating said library. There is little need to compare several frameworks in detail.

One of the longest standing versions of a graphical library is Open GL. Open GL works natively with C++, it has been well expanded and stabilized over many years. Their website holds plenty of information and documentation about using Open GL; mentioning some of its strengths on the homepage:

*"**Developer-Driven Advantages***

- ***Industry standard***
  An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.
- ***Stable***
  OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes. Backward compatibility requirements ensure that existing applications do not become obsolete.
- ***Reliable and portable***
  All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system.
- ***Evolving***
  Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application

developers and hardware vendors incorporate new features into their normal product release cycles.

- ***Scalable***
  OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that the developer chooses to target.
- ***Easy to use***
  OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.
- ***Well-documented***
  Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain." (Open GL, 2017)

Open GL also includes a main loop function to update any functionality or display. This is ideal to process physics per frame and then push any changes to the visual side of the system.

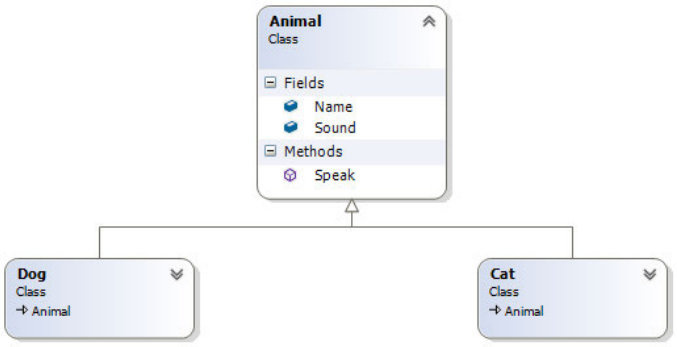
2.9 Optimization and Efficiency

Optimizations is a broad aspect of software and system improvement. There are many factors which can influence runtime efficiency of a program. However, there is a tradeoff, making a system run at peak optimization runs a high risk of sacrificing code readability and ease of navigation.

2.9.1 Object Orientated Programming

OOP is a dead certain for development in any software that intends to employ efficient, maintainable, and readable code. It uses a root and child tree structure to share attributes of

child nodes/classes which represent similar architecture. One of the more well-known examples of this is an Animal, Dog, Cat structure.



2-4 Diagram of Animal Class Inheritance

(Atlantic.net Community, 2017)

2.9.2 Mathematical Analysis

Function execution speeds is the primary point of interest when measuring code optimization. A measurement technique for this is using Big 'O' notation. *"Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm*". It is measured by predicting the growth in the number of iterations a function may need to carry out in proportion to the growth of its input data. Obviously a linear (direct proportionality) would be the most desired outcome of a function, thought this is impossible for many circumstances. Without clever management such as Sleep, isKinematic and other triggers used in game engines that contain many hundreds of objects. Update iterations could increase exponentially and cripple the system. This may be less apparent in simulation engines as they do not usually need to watch so many different types of objects and behaviors simultaneously.

Another metric in the way of mathematical analysis is execution timing. This can be on a large or small scale depending on the scale of optimization a programmer would try to achieve. On a larger scale this may be as simple as reducing lines of unnecessary code. Two examples are shown below:

```
bool SomeFunction(int Foo, int Bar)
{
    int temp;
    temp = Bar * 10 + 50;
    if (Foo < temp)
        {return true;}
    else
        {return false;}
}
```

2-5 Example function of unnecessary execution

The exact same functionality can be written like this:

```
bool SomeFunction(int Foo, int Bar)
{
    return (Foo < (Bar * 10 + 50));
}
```

Figure 2-6 Efficiency example 1

Saving several processor cycles. This may seem like a negligible difference but in large applications, an accumulation of these code refactors could start to make a noticeable difference.

Another much more specific example which may only be applicable in certain scenarios can remove the need for an else statement all together:

```
//Before:
    if( Condition ) {
        //DO A;
    } else {
        //DO B;
    }
```

```
//After:
    //DO B
    if( Condition ) {
        //Undo (DO B)
        //DO A;
    }
```

Figure 2-7 Efficiency example 2

*"Clearly this only works if **Undo Case B;** is possible. However, if it is, this technique has the advantage that the jump taken case can be optimized according to the Condition probability and **Undo Case B; Case A;** might be merged together to be more optimal than executing each separately." ... "Also since this optimization is dependent on sacrificing performance of one set of circumstances for another, you will need to time it to see if it is really worth it."* (Hsieh, 2016)

A technique that may be more prominent to physics engines would be to avoid complex mathematics when possible. Storing the square value of a number and not square rooting the value saves an expensive calculation. Even trying to reduce a multiply to add or subtract where possible is more efficient. Nearly all engines take this square root avoidance technique into account and usually in the same way with different naming conventions. OSG (Open Scene Graph) includes such functionality:

value_type	length () const
	Length of the vector = sqrt(vec . More...
value_type	length2 () const
	Length squared of the vector = vec . More...

2-8 OSG Documentation of vector length functions

(Open Scene Graph, 2016)

A noteworthy efficiency technique is also highlighted in the above figure. "value_type" Means the mentioned functions are not restricted to a single return object type (integer, float, double, etc). In fact, the recognized value type of the OSG API is configurable by a user, meaning (if well designed) the accuracy of calculation can be changed from integer to float in

a single line change rather than refactoring huge segments of code. This is a clever trick that can be used in several ways, though does depend on the program language.

There are many more optimization and efficiency techniques that are used in every day programing; so many that it would be impractical to mention them all. However, most of these techniques are the common standard of programming amongst experienced developers, and have been adopted as fundamental approaches to their respective solutions.

2.9.3 Memory Management

A core fundamental component of good code is the ability to manage memory well during runtime. A lot of modern languages and compilers can support garbage collection and error assertion such as (VB.Net, C#, Java). This allows a developer to have a more relaxed approach to coding and worry less about things that the compiler can take care of independently (if supported). A possible suspicion here is that a garbage collector managed by the compiler, may not do as good a job as a well experienced developer. And in truth a developer could identify when objects could be destroyed or reallocated before said object becomes out of scope, thus freeing memory up sooner.

More imperative languages (as opposed to declarative) such as C++ give the developer a lot of freedom in the way of garbage collection/object destruction. So much so that, it is very possible to cause looped referencing and memory leaks if coding is not well planned/designed.
Pointers are another useful but potentially dangerous tool available.

"C++ gives you the power to manipulate the data in the computer's memory directly. You can assign and de-assign any space in the memory as you wish. This is done using Pointer variables.

Pointers variables are variables that points to a specific address in the memory pointed by another variable."

This essentially means that data can be altered through functions that have received a pointer to an object, opposed to a full copy of an object. Obviously reducing memory consumption.

3 Design

3.1 Product Design

3.1.1 Physics Library

3.1.1.1 Language selection

Although generally more complicated than others, C++ is the chosen language to undergo this task. The use of pointer representation of objects could prove to be a string advantage in terms of execution efficiency. It is of course important to avoid creating as much technical debt in the system as possible. Negating the need to make a solid copy of an object every time it is passed through to functions, could save both the number of processor cycles needed to initialize the method signature; and reduce memory consumption. C++ also works seamlessly with Open GL without the need for any extension libraries to overload native Open GL functions.

3.1.1.2 Core Functions

The Physics Library named “PhySim” will begin by mirroring the same core two classes used in the Cyclone engine. A Vector class and a Particle class. And then modified to suit the needs of this system.

The Vector Class will need to include a zeroed constructor to avoid and null errors where vector mathematics are involved. Then an attribute to represent the value type (int, float, etc) of each of the three axis’ in 3D space. In pseudo code, it could look something like this:

```
class Vector
{
    Vector() {x = 0, y = 0, z = 0}
    Vector(a, b, c) {x = a, y = b, z = c}

    value x;

    value y;

    value z;

}
```

Figure 3-1 Vector Pseudo Code

The next task would be to implement vector calculus into the class to account for standard vector mathematic (which a compiler would not natively know). This means added operator overloads for standard mathematical functions such as “==”, “+=”, “-=” and more. The pseudocode for these functions are as follows:

```
invert()
{
    x = -x
    y = -y
    z = -z
}


Length()
{
    return square root(x*x + y*y + z*z)
}


LengthSquared()
{
    return (x*x + y*y + z*z)
}


func *=(value)
{
    x *= value
    y *= value
    z *= value
}

Normalise()
{
    magnitude = Length()
    this *= 1 / mag
}


func +=(vector)
{
    x += vector.x
    y += vector.y
    z += vector.z
}


func -=(vector)
{
    x -= vector.x
    y -= vector.y
    z -= vector.z
}
```

Figure 3-2 Vector Pseudocode 1

```
func + (vector)
{
    return vector{ x + vector.x, y + vector.y, z + vector.z }
}


func - (vector)
{
    return vector{ x - vector.x, y - vector.y, z - vector.z }
}

func AddScaledVector(vector, scale)
{
    x += (vector.x * scale)
    y += (vector.y * scale)
    z += (vector.z * scale)
}


func dot product (vector)
{
    return x*vector.x + y*vector.y + z*vector.z
}


func Cross Product(vector)
{
    return vector(y*vector.z - z*vector.y, z*vector.x - x*vector.z, x*vector.y - y*vector.x)
}


func Clear()
{
    x = 0
    y = 0
    z = 0
}
```

Figure 3-3 Vector Pseudocode 2

This covers all the basic math's involved with 3D vectors and how they might interact with other vector or scalar values.

Next, would be to implement some form of object or particle that would include instances of the vector class to represent its world position, velocity, acceleration, and resultant force. Each particle would also need an update function to carry out its position update over time. Again, the Cyclone engine provides a good start point for the implementation of this class (this can be found in the (Millington 2010) reference). Then modifications to the update procedure by using accuracy configurations will provide flexibility in terms of the update accuracy. The particle class will hold vector instances for the attribute mentioned earlier. Pseudo code for the update procedure is shown below:

```
func update(time)
{
    position += velocity * time

    velocity += acceleration * time

    if gravity is off then
        do nothing
    else if gravityIsConstant then
        add gravity acceleration
    else if gravityIsForce then
        do nothing
    end if

    if drag is off then
        do nothing
    else if dragIsConstant then
        add drag acceleration
    else if dragIsForce then
        do nothing
    end if

    if motion is Velocity then
        do nothing
    else if motion is force then
        acceleration = resultantForce * 1/mass
    end if

    ClearForce();
}
```

Figure 3-4 Particle Update pseudocode

There are several points to mention in the above example code. Firstly, the code is designed to calculate the next frames particle attribute values and store them to be applied at the beginning of the iteration of the next frame update loop (where position is set). Secondly the clear force function is added to remove any forces acting on the particle. This is done to avoid and forces being applied over a period; a constant force applied to an object will mean that its rate of change of acceleration will also increase, meaning the particle will fly off into the distance extremely fast.

The three if statements in the update loop are used to check how the integral of the particles position is calculated. In terms of accuracy, there will be three variations of settings for each of the following:

- Gravity
- Drag

These being:

- Not applied to object
- A constant scalar value
- A force applied to object

Also, a constant of force setting for the motion of the object. It wouldn't make sense to have a "non" setting for the motion of the object as this would effectively mean that the position wouldn't update. It is also noteworthy that, for force applications to act on the particle from either the gravity or drag; the motion of the particle would also need to be set to force orientated, or the system, will simply calculate a resultant force and not use it to update the position.

Finally, the do-nothing statements are present if a physical effect should not be included or if it is force based. If it is force based, depending on the type of force, an acceleration should be added to the resultant force using newtons second law. Then in turn, the acceleration will filter down into the position.

Force manipulation is done through a separate class. This is a good example of where efficiency counteracts readability. It would be very possible to include different types of force functionality into the particle class. However, if many forces are acting on some particles and less on others, holding a type of list or array of these forces per particle could cause a lot of unnecessary memory use. To counter this, a world control class will manage the engine in

terms of updating forces to particles and then telling each particle to update its statistics. For forces, the world class will contain a collection of force instances which will be a structure of a type of force and a reference to the particle in the scene it should act on.

There is an extra advantage of making the system extensible in doing this, Forces could be viewed as a separate area of the engine in which their own accuracy settings could be implemented, which would change the mathematical formulae used to resolve a force on an object from its (if required) input values.

This force class will consist of three types of forces to be added to the particle where prompted. These are, a standard linear force, a drag force using provided drag coefficients of the developer, added to a formula provided by Cyclone. (Millington, 2010)

$$f_{\text{drag}} = -\hat{\dot{p}}(k_1|\hat{\dot{p}}| + k_2|\hat{\dot{p}}|^2)$$

Equation 3-1 Drag Formula

"where k1 and k2 are two constants that characterize how strong the drag force is. They are usually called the "drag coefficients," and they depend on both the object and the type of drag being simulated. The formula looks complex but is simple in practice. It says that the force acts in the opposite direction to the velocity of the object (this is the $-\hat{\dot{p}}$ *part of the equation:* $\hat{\dot{p}}$ *is the normalized velocity of the particle), with a strength that depends on both the speed of the object and the square of the speed. Drag that has a k2 value will grow faster at higher speeds. This is the case with the aerodynamic drag that keeps a car from accelerating indefinitely. At slow speeds the car feels almost no drag from the air"*
(Millington, 2010)

This is a simplified formula for drag as real-time calculation of drag using fluid density and density of the projectile object would require some form of huge matrix for density of materials, and slow down update execution. Accuracy of drag applied can still be changeable by this system when added directly as a user set scalar or as a fraction of the kinetic reaction of the projectile which would increase with speed of the projectile. A special variation of linear force, used for gravitational force will also be available. These will be added inside a force generator style class. The force generate will be an instance of one of these three forces, structured to a particle and saved to a collection ready to be applied during a physics update. Value (declared variable type) accuracy will play a large part as to the smoothness of the projectiles during run time. Whist using float variables is the chosen variable type for this system. Using a global type definition to set the internal accuracy of all variables would be a nice tough. Using a type definition for this means changing accuracy would only require a single line change rather than a full code refactor. The code in C++ to do this would be:

```
/** Hold an object orientated Accuracy Value Type */
typedef float avt;
```

Figure 3-5 C++ Type Definition example

This would be declared in a public class using a global namespace, therefore all internal classes od (PhySim) can have access to it.

3.1.2 Demonstration GUI

The GUI will require four main specifications.

1. The ability to show/read-back attribute values of each spawned particle and display them on the screen.
2. To display any controls for changing settings and the current settings in use.
3. To create a basic 3D space with some concept of representable distance for a projectile to move along.
4. Create projectiles and fire them whilst binding the current settings to the projectile.

To show all the written information on the screen, a generic function could be implemented. Take input of an orthographic x, y coordinate for position in the running window, the text to display, a font/color/style.

The user interface is drafted below:

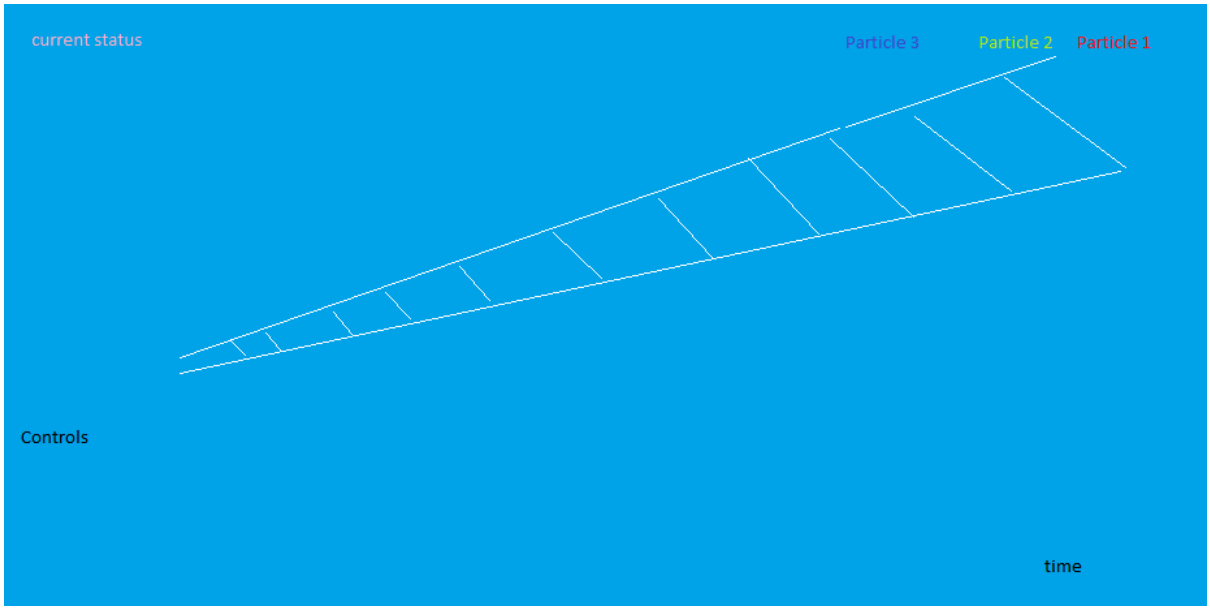


Figure 3-6 Draft User Interface

The labels included in the image above intuitively highlight the area of the window that this information will be displayed. The controls and current settings will be shown along the right-hand side of the screen. Each particle statistics information will be dynamically generated and color coded to match the particle in flight. This will help with recognition of statistics to relevant particle if more than one particle is in flight at any time. Time will show the time elapsed after the latest shot has been fired.

3.2 Evaluation Design

3.2.1 Relevance of Evaluation

“Evaluation is the collection of, analysis and interpretation of information about any aspect of a programme of education or training as part of a recognised process of judging its effectiveness, its efficiency and any other outcomes it may have.” (Ellington, et al., 1988)

It can be carried out either at the end of the development cycle to judge success, failure, or potential improvements. Or it can be used in parallel with development to assist design heuristics recalibrate goals.

In this circumstance, there is little room for variance in terms of usability and interface as the heart of the project is a compiled library of functions. To this end, users will not need to have in depth knowledge of its inner functions but rather how to collate data in and out of the library. Therefore, most of the projects evaluative works will be considered post development.

3.2.2 Evaluation Techniques

Though the list of possible evaluation techniques is not in any way broad, they can be tailored or manipulated to suit the task at hand. Some chosen evaluation techniques that could be considered are listed below with a brief justification.

3.3 Testing Plan

3.3.1 Available Methods

3.3.1.1 Questionnaire

Questionnaires are a powerful tool for collection data, they possess convincing advantages and disadvantages. For this reason, they must be carefully designed for the task at hand to maximise effectiveness.

3.3.1.1.1 Advantages

- Can cover broad areas and a lot of information in a concise and informative format.
- By allowing anonymity, honest and intuitive responses can be expected.
- Can be easy to analysis and collate data (requires a well-planned questionnaire).

3.3.1.1.2 Disadvantages

- Having too many open-ended questions may make it difficult to categories answers/data.
- Successive and non-successive events happening chronologically may mask previous events unfairly during the final questionnaire.
- Speculate answers will require more investigation face to face.

3.3.1.2 Black Box Testing

Black box testing is especially useful in these circumstances as the finished product will not reveal any internals of the code library. This will also be useful to monitor the efficiency of control functions managing function allocation depending on system accuracy settings at runtime.

3.3.1.2.1 Advantages

- Requires no knowledge of internals.
- Efficient for larger code segments/modules.
- Doesn't require developer interaction and therefore separates developer and tester.

3.3.1.2.2 Disadvantages

- Code coverage is limited as there is no decision on methodical navigation.
- Tester rather than developer carrying out the testing may prove a failure due to depth of knowledge of code.

3.3.1.3 White Box Testing

The complexity and efficiency of some of the control classes/mechanisms may prove to add too much time to execution if a user is aiming for speed and accuracy. Using white box testing may help to locate these strenuous areas for refactoring. Allowing direct access to mathematical algorithms by bypassing control classes may also be feasible pending the outcome of these tests. White box testing is a generic name given to such methods as unit tests, it may be plausible to include some of these in an accompanying project file.

3.3.1.3.1 Advantages

- More efficient at accurately locating errors and problems
- The required knowledge of source code may be beneficial towards output data interpretation.
- Effectively debugging code whilst testing may make optimisations or lack of, more apparent.
- More in depth testing

3.3.1.3.2 Disadvantages

- Tester requires in depth knowledge of internal function.
- Requires access to full, non-compiled project files.

As White box and Black box testing have strong advantages and disadvantages, it would be wise to use a combination of the two to get the best efficiency evaluation of the project possible.

3.3.1.4 Usability Study

Usability studies/engineering is worth mentioning as this is one of the most powerful versions of evaluation. A usability study must be designed and carried out in a bespoke manor on every project as it consists of modular protocols and measuring techniques. A usability study is a good tool for evaluating modules of a larger system. As this product is only focusing on projectile motion physics, usability engineering may not yet be needed in its full form. With the intention of good code being designed in a modular and extensible manor, a usability study may become more relevant with later additions to the project.

3.3.2 Chosen Test Strategy

Although a usability study could hold great value to wards gathering feedback for new systems. It would be uneconomical to use such techniques in this circumstance. A usability study would require a separate investigation and design the process of testing usability in this project; this would take a massive amount of time and usually requires more than a single invigilator. Another issue would be the target audience on which to carry out the test. Although shallow in terms of depth, this system is built for developers to use in conjunction

with a graphics framework to produce another system/game. This means an experienced developer would be required to carry out any tests and give feedback. These developers would be hard to find on a large scale, especially when they would also need to be willing to take the time to develop a system using the physics engine.

The same issues as above also stand for a questionnaire. The questionnaire would need to be issued to developers that have had a large amount of time to use the engine and provide any feedback. The time scale here could be in the form of weeks even after enough developers have been accumulated.

3.3.2.1 Black Box

Black box testing is more relevant for testing this sort of system. Because it is not intended for a developer to need to alter any of the internal functionality if the physics system. More use its functions, evaluate their efficiency, and compare the output to what would be expected of it. This is something that would essentially evaluate the success or failure of the systems primary aim (To see if is a good idea to try and implement different levels of accuracy for a physics engine).

By using different configurations in the system, the attributes could be fired one at a time and paused incrementally. The projectiles world attributes could be recorded and graphed for comparison across different settings configurations.

3.3.2.2 White Box

White Box testing would be a good solution to track or locate any potential problems in code. The issues to look for will consist of the following:

- Execution time hotspots

- Unexpected behavior
- Bugs
- Possible extensions/improvements

It will be useful to highlight any risks to the execution iteration multiplier (Big ‘O’). Meaning, predicting any parts of the system that may begin to run slow or struggle with a higher count of projectiles spawning in.

4 Implementation

4.1 Introduction

The following chapter will provide an extension of the design of the system. Covering explanations of the front-end (demo) and back-end (PhySim). The chapter focuses on key functionality in a higher level of detail of both systems and how they will communicate between each other.

Because there are a high number of functions and methods in this system. The core functionality will include snippets from the system. However, the full source code will be available in the appendices.

4.2 PhySim (Back End)

4.2.1 Primary Attributes

The back-end system is a library capable of mapping the position over time of a particle acting under specific physical stimulus.

As previously explained in the design section. The core classes of this system are the Vector and Particle classes. Below show the header files for both classes.

```
#pragma once
#include "stdafx.h"
#include <math.h>
#include "Precision.h"
namespace PhySim
{
	class Vec3
	{
	public:
		/*default 0 constructor*/
		Vec3() : x((avt)0), y((avt)0), z((avt)0) {}

		/*standard 3 dimensional vector constructor*/
		Vec3(avt x, avt y, avt z)
		{
			this->x = x;
			this->y = y;
			this->z = z;
		}

		~Vec3();

		/*Holds the 3D Vector Coordinate vale in the x axis*/
		avt x;

		/*Holds the 3D Vector Coordinate vale in the y axis*/
		avt y;

		/*Holds the 3D Vector Coordinate vale in the z axis*/
		avt z;

		/*Reverse the direction of each axis*/
		void invert()
		{
			x = -x;
			y = -y;
			z = -z;
		}

		/*Returns the Magnitude of a Vector3*/
		avt Length()
		{
			return sqrt(x*x + y*y + z*z);
		}

		/*Returns the Square Magnitude of a Vector3 (faster at run time)*/
		avt LengthSquared()
		{
			return (x*x + y*y + z*z);
		}

		/*Converts a non 0 Vector3 into its respective Unit Vector*/
		void Normalise()
		{
			avt mag = Length();
```

```
	if (mag > 0)
		(*this) *= ((avt)1 / mag);
}


/*Multiply a Vector3 by a scalar value*/
void operator *=(avt value)
{
	x *= value;
	y *= value;
	z *= value;
}

/*Return a new instance of the given Vector3 multiplied by a scalar
value*/
Vec3 operator * (avt value)
{
	return Vec3{ x * value ,y * value ,z * value };
}


/*Add a Vector3 to a given Vector3*/
void operator +=(Vec3& value)
{
	x += value.x;
	y += value.y;
	z += value.z;
}

/*Subtract a Vector3 by a given Vector3*/
void operator -=(Vec3& value)
{
	x -= value.x;
	y -= value.y;
	z -= value.z;
}

/*Return a new instance of a Vector3 added to another Vector3*/
Vec3 operator + (Vec3& value)
{
	return Vec3{ x + value.x, y + value.y, z + value.z };
}

/*Return a new instance of a Vector3 added to another Vector3*/
Vec3 operator - (Vec3& value)
{
	return Vec3{ x - value.x, y - value.y, z - value.z };
}

void AddScaledVector(Vec3 vector, avt scale)
{
	x += (vector.x * scale);
	y += (vector.y * scale);
	z += (vector.z * scale);
}

/*Return the resultant vector of each component multiplied by its
reletive vector component*/
Vec3 ComponentProduct(Vec3& vector)
{
	return  Vec3(x * vector.x, y * vector.y, z * vector.z);
}
```

```
		/*Update the current vector multiplied by each respective vector
		component*/
		void  ComponentProductUpdate(Vec3& vector)
		{
			x *= vector.x;
			y *= vector.y;
			z *= vector.z;
		}

		/*Dot Product*/
		avt operator *(Vec3 &vector)
		{
			return x*vector.x + y*vector.y + z*vector.z;
		}

		/*Cross Product*/
		Vec3 CrossProduct(Vec3&vector)
		{
			return Vec3(y*vector.z - z*vector.y, z*vector.x - x*vector.z,
			x*vector.y - y*vector.x);
		}

		/*Cross Product Update*/
		void operator %=(Vec3 vector)
		{
			*this = CrossProduct(vector);
		}

		/*Cross Product operator varient*/
		Vec3 operator %(Vec3 vector)
		{
			return CrossProduct(vector);
		}


		void Clear()
		{
			x = (avt)0;
			y = (avt)0;
			z = (avt)0;
		}

	};
}
```

Figure 4-1 Vec3 Snippet

As previously stated, the *avt* type definition is used throughout the class for accuracy variation if required. The Vector class (Vec3) includes all required 3D vector calculus to carry out its tasks in the engine.

Interestingly, it is a commonly shared ideal to store a particles mass as its inverse. This is to stop physics instability when trying to set immovable objects. To set an object as immovable,

one would set the mass to 0. Using a high number would not make the object truly immovable, it would just require a very hard force (impulse). However, having a mass of 0 in some calculation may cancel the result of said calculation, therefore running the risk of inaccuracy and instability. Using an inverse mass would mean that setting a mass of 0 would invert to infinity, thus creating a truly immovable object.

The vector class is repeatedly implemented throughout both front-end and back-end systems. As this is one of the primary type of variables used to represent values in three-dimensional space.

The Particle class implements several vectors for different purposes. A snippet of the particle header class is listed below.

```
#pragma once
#include "stdafx.h"
#include "Precision.h"
#include "Vec3.h"
#include <tuple>
namespace PhySim
{
	class Particle
	{
	protected:

		//Holds the current 3 dimensional position in world space
		Vec3 position;

		//Holds the current 3 acceleration position in world space
		Vec3 acceleration;

		//Holds the current 3 rotation position in world space
		Vec3 rotation;

		//Holds the current 3 velocity position in world space
		Vec3 velocity;

		//The scale of implicit drag applied to any forces acting on the
		particle
		avt    damping;

		/*Holds inverse of the mass of the projectile to ensure physics
		stability*/
		avt    InverseMass;

		/*resulting force in acordance with D'alemberts principle*/
		Vec3 resultantForce;


		Vec3 GravityAcceleration = { 0.0,0.0,0.0 };
```

```
public:

	Accuracy_Motion set_motion = Accuracy_Motion::VELOCITY;

	Accuracy_Gravity set_gravity = Accuracy_Gravity::CONSTANT_ACCELERATION;

	Accuracy_Drag set_drag = Accuracy_Drag::CONSTANT;

	void AddForce(Vec3 force);

	/*reset resulting force to (0,0,0)*/
	void ClearForce() { resultantForce.Clear(); }

	void Intergrate(avt dt);

	Vec3 GetPosition() { return position; }

	void SetPosition(Vec3 &pos)
	{
		position = Vec3();
		position = pos;
	}

	Vec3 GetAcceleration() { return acceleration; }

	void SetAcceleration(Vec3 a) { acceleration = a; }

	void SetGravityAcc(Vec3 a) { GravityAcceleration = a; }

	void SetRotation(Vec3 rot) { rotation = rot; }

	Vec3 GetRotation() { return rotation; }

	Vec3 GetVelocity() { return velocity; }

	void Setvelocity(Vec3 v) { velocity = v; }

	avt    GetDamping() { return damping; }

	void SetDamping(avt d) { damping = d; }

	avt GetIMass() { return InverseMass; }

	avt GetMass() { return (avt)1 / InverseMass; }

	void SetMass(avt m) { InverseMass = (avt)1 / m; }

	Vec3 GetResultantForce() { return resultantForce; }

	void SetResultantForce(Vec3 f) { resultantForce = f; }

	bool operator == (const Particle &pi)
	{
		return &pi == &(*this);
	}


private:

	bool GravityAccelerationAdded = false;
```

```
};

}
```

Figure 4-2 Particle Snippet Header

The other, more important aspect of the particle class is the functionality of the Integrate/Update procedure (snippet below):

```
void Particle::Integrate(avt dt)
{
	//acceleration in terms of updating position using a frame rate dt is
	finite and negligible therefore position is:
	//p' = p + (Δp/Δt)t
	//instead of p' = p + 1/2*(Δp/Δt) t^2
	if (dt <= 0.0f) return;
	if (InverseMass <= 0.0f) return;


	position.AddScaledVector(velocity, dt);

	//acceleration += GravityAcceleration;
	Vec3 resultingAcceleration = acceleration;
	//velocity is : Δ(p')/Δt = (Δp/Δt)d + (Δ^2p/((Δt)^2))t where d is
	dampening of velocity over a constant time.
	velocity.AddScaledVector(resultingAcceleration, dt);


	switch (set_gravity)
	{
	case PhySim::Accuracy_Gravity::NON_OFF:
		GravityAccelerationAdded = false;
		break;
	case PhySim::Accuracy_Gravity::CONSTANT_ACCELERATION:
		if (!GravityAccelerationAdded)
		{
			GravityAccelerationAdded = true;
			acceleration += GravityAcceleration;
		}
		break;
	case PhySim::Accuracy_Gravity::FORCE_BASED:
		GravityAccelerationAdded = false;
		//Do nothing here as resultant force has already been calculation
		from the front-end input of a force generator and world force
		register physic update
		//IMPORTANT: THIS REQUIRES FORCED BASED MOTION TO BE ENABLED
		break;
	default:
		break;
	}

	switch (set_drag)
	{
	case PhySim::Accuracy_Drag::NON_OFF:
		break;
```

```
        case PhySim::Accuracy_Drag::CONSTANT:
            velocity *= avt_pow(damping, dt);
            break;
        case PhySim::Accuracy_Drag::FORCE_BASED:
            //Do nothing here as resultant force has already been calculation
            from the front-end input of a force generator and world force
            register physic update
            //IMPORTANT: THIS REQUIRES FORCED BASED MOTION TO BE ENABLED
            break;
        default:
            break;
        }

        switch (set_motion)
        {
        case PhySim::Accuracy_Motion::VELOCITY:
            //do nothing the current velocity will update position next
            frame;
            break;
        case PhySim::Accuracy_Motion::FORCE_BASED:
            acceleration.AddScaledVector(resultantForce, InverseMass);
            break;
        default:
            break;
        }

        //Clears force in per frame to allow for non-constant forces to objects
        (do not want increasing acceleration)
        //IN THIS CURRENT BUILD: to add forces through time (impulse) handle
        this by adding a type of force generator to the world registry each
        frame or adding a duration to a force generator instance.
        ClearForce();
}
```

Figure 4-3 Particle Update Snippet

This is a modified version of the basic update functionality used in Cyclone (Millington, 2010)

The position is determined from the previously calculated velocity, acceleration, and force (if set to use forces). After the position is set, the velocity is then updated via acceleration and a change in time (frame rate) ready for the next update.

The three switch statements are representative of the three variable settings implemented into the instance of each particle. Using a force based update system does not require any work on forces here as the resultant force is already held in the particle attribute "*resultantForce*" the

resultant force is added to acceleration which is eventually integrated into a change in position. The force system is mentioned below.

The resultant force is cleared at the end of the first iteration of updating the particle so as not to add a constant force to the particle. This is acceptable here in terms of expected outcome as the resultantForce has already influenced a resultant acceleration. This would case the particle to accelerate off the screen rapidly. A separate implementation for force overtime using a custom timing data system would be implemented to deal with such occasions.

4.2.2 Force System

The force system is implemented by holding a registry of all forces acting on all particles in the scene. During an update sequence, a RunPhysics function is called which tells the registry to add these forces to each particle, then clear said force from the registry so as not to add a constant force to the particle (unwanted incremental acceleration).

The structure to hold the in the registry is as follows:

```
struct ForceInstance
{
	Particle *p;
	ParticleForce *f;

	bool operator == (const ForceInstance &fi)
	{
		return std::tie(p, f) == std::tie(fi.p, fi.f);
	}

};
```

Figure 4-4 Force Instance Structure

As you can see the force instance structure hold pointers to the particle and force so duplicate objects are not wastefully stored in memory. This is one of the proactive reasoning's for choosing C++ as the system language.

A particle force is an overridable interface containing an update procedure which is overridden in each form of force generator (below). The compiler can cast to the instance of the force generator which implements the Particle Force interface during update physics.

A force Generator is the name given to a force type which implements particle force:

```
#pragma once
#include "ParticleForce.h"
using namespace PhySim;
class StandardForce : public ParticleForce
{
public:
	Vec3 acceleration;
	void UpdateForce(Particle *p, avt duration) override;
	StandardForce(Vec3 &vector) : acceleration(vector) {}
	~StandardForce();
};

class DragForce : public ParticleForce
{
public:
	DragForce(avt k_1, avt k_2) : k1(k_1), k2(k_2) {}

	//drag coifficient 1
	avt k1;

	//drag coifficient 2
	avt k2;
	void UpdateForce(Particle *p, avt duration) override;

};

class GravityForce : public ParticleForce //this is a special variation of a Standard
Force designed to act in only 1 direction
{
public :
	Vec3 gravForce = { 0.0,5.0,0.0 }; // default acceleration value due to gravity

	GravityForce(const Vec3 & force);

	void UpdateForce(Particle *p, avt duration) override;
};
```

Figure 4-5 Force Generators

As mentioned prior, there are three available force to be added to a particle. Each update force calls the overridden variation of the particle interface update force procedure; this eventually addresses the particle included with that force and calls the add scaled vector function in Vec3 to the resultant force of said particle. Hence creating a new resultant force

vector which can be used in the particle update function. A force can be instantiated in the front-end system and added to the world controller class which holds in instance of a world force register collection.

An interesting point when using force based mechanics as opposed to constant scalars and vectors is the gravitational acceleration. The acceleration is a constant throughout the scene/world. Gravitational forces are calculated per particle in direct proportion to their mass; this also means that code management for global gravity strength requires the alteration of a single value. Whereas changing gravity when using constants would require the developer to manually change the Y value of gravitational acceleration for every particle in that could be spawned in the scene.

4.2.3 Settings Configuration

The settings configuration system utilizes the use of enums to differentiate states of each setting. These are declared in the global precision class and instantiated in each particle instance so they can vary per particle in the world. An alternative to this would be to bind a structure and particle collection like the force registry. However, this is less necessary as the settings will have a predetermined count of enum values and each be instantiated once in a particle; causing less memory wastage. The settings are as follows and are intuitively named.

```
enum class Accuracy_Motion
{
	VELOCITY,
	FORCE_BASED,
};
```

Figure 4-6 Setting for Motion Accuracy

```
enum class Accuracy_Gravity
{
	NON_OFF,
	CONSTANT_ACCELERATION,
	FORCE_BASED,
```

```
};
```

Figure 4-7 Setting for Gravity Accuracy

```
enum class Accuracy_Drag
{
	NON_OFF,
	CONSTANT,
	FORCE_BASED
};
```

Figure 4-8 Setting for Drag Accuracy

The developer will be able to set these values for each particle when they are spawned effecting the order of calculations used to update the particle. It must be stressed again that there is no reason behind using force based drag and gravity settings on a particle that does not employ the force based motion setting. The resultant force of the particle only updates its acceleration when the motion force setting is active.

4.2.4 World Manager

The world manager class acts as a liaison between the PhySim physics system and the front-end system. The world class (once instantiated) implements the world force register, and includes the run physics procedure which updates all forces to their relevant particles. Then updates the list of particles in the world.

The world class is shown below:

```
#pragma once
#include <vector>
#include "ParticleForce.h"
#include "WorldForceRegister.h"
#include "ForceGenerator.h"

namespace PhySim
{
	class World
	{
	public:

		struct Shot
		{

			Status status;
			Particle round;
```

```
            ObjectType ot;
            Vec3 Colour;
        };

        typedef std::vector<Particle*> Particles;

        typedef std::vector<Shot*> ShotsList;

        ShotsList Shots;

        ShotsList& GetShotList();


        void AddShotToList(Shot *s);

        void AddParticleToList(Particle *p);

        bool Stop = false;

        void RunPhysics(avt duration);

        Particles& GetParticles();

        World();
        ~World();

        WorldForceRegister forceRegistry;


        Particles allParticles;

        void ClearParticle(Particle p);


    };
}
```

Figure 4-9 World Control Class

The world class holds collections referencing all particles and forces active in the world. And possesses functionality to manipulate these collections.

This class also includes a Shot structure. This is the primary access for the front end to implement a particle for projectile motion, and in effect the physics pipeline starts there.

The Shot structure holds an instance of a particle, its chosen color (using a Vec3 for its RGB values), an object type and status.

An object type is an enum which highlights some simple predetermined vales of a particle for ease of demonstration. The attributes of that particle can still be altered from the set values in the object type enum (mainly mass).

Finally, the status of a particle is a enum to show the state of a current particle, if it is fired or dead. Once a particle has been fired it is up to the developer to set constraints to de-spawn that particle. To de-spawn a particle simply set its status to dead and all its attributes will be cleared.

4.3 SimClient (Demo Layer)

The demonstration project was written in C++ and OpenGL, whilst using the PhySim::Vec3 object for vector mathematics.

The three core functions involved in the SimClient demonstration project is Fire(), Update() and Display().

The Fire Function is shown below:

```
void fire()
{
	World::Shot *newShot;
	int index = 0;

	startTime = clock();

	for (newShot = shotList; newShot < shotList + maxAmmoCount; newShot++)
	{
		if (newShot->status == Status::DEAD) break;
		else index++;
	}
	if (newShot == NULL) return;
	switch (index)
	{
	case 0:
		newShot->Colour = Vec3{ 1.0,0.0,0.0 };
		break;
	case 1:
		newShot->Colour = Vec3{ 0.0,1.0,0.0 };
		break;
	case 2:
		newShot->Colour = Vec3{ 0.0,0.0,1.0 };
		break;
	default:
		break;
```

```
	}
	if (newShot >= shotList + maxAmmoCount) return;


	newShot->status = Status::FIRED;
	newShot->round.SetPosition(Vec3{ 0.0,1.5,0.0 });

	//start shooting projectile.
	switch (currentShotType)
	{
	case PhySim::ObjectType::ARTILLERY:
		newShot->round.SetMass(200.0f);
		if (set_motion == Accuracy_Motion::VELOCITY) newShot-
>round.Setvelocity(0.0f, 0.0f + (float)Config.elevation * 2, 40.0f);
		else
		{
			LinearForce.acceleration = Vec3{ 0.0f,(float)Config.elevation,
10.f };
			world.forceRegistry.Add(&newShot->round, &LinearForce);
		}

		switch (set_gravity)
		{
		case PhySim::Accuracy_Gravity::NON_OFF:
			break;
		case PhySim::Accuracy_Gravity::CONSTANT_ACCELERATION:
			newShot->round.SetGravityAcc(0.0f, -20.0f, 0.0f);
			break;
		case PhySim::Accuracy_Gravity::FORCE_BASED:
			GravitationalForce.gravForce = { (avt)0.0f, (avt)-
worldGravityConst,(avt)0.0 };
			world.forceRegistry.Add(&newShot->round, &GravitationalForce);
			break;
		default:
			break;
		}

		switch (set_drag)
		{
		case PhySim::Accuracy_Drag::NON_OFF:
			break;
		case PhySim::Accuracy_Drag::CONSTANT:
			newShot->round.SetDamping(0.9f);
			break;
		case PhySim::Accuracy_Drag::FORCE_BASED:
			world.forceRegistry.Add(&newShot->round, &ResistanceForce);
			break;
		default:
			break;
		}

		break;
	case PhySim::ObjectType::BULLET:
		newShot->round.SetMass(2.0f);
		if (set_motion == Accuracy_Motion::VELOCITY)newShot-
>round.Setvelocity(0.0f, 0.0f + (float)Config.elevation, 35.0f);
		else
		{
			LinearForce.acceleration = Vec3{ 0.0f,(float)Config.elevation,
10.f };
			world.forceRegistry.Add(&newShot->round, &LinearForce);
		}
```

```
		switch (set_gravity)
		{
		case PhySim::Accuracy_Gravity::NON_OFF:
			break;
		case PhySim::Accuracy_Gravity::CONSTANT_ACCELERATION:
			newShot->round.SetGravityAcc(0.0f, -2.0f, 0.0f);
			break;
		case PhySim::Accuracy_Gravity::FORCE_BASED:
			GravitationalForce.gravForce = { (avt)0.0f, (avt)-
worldGravityConst,(avt)0.0 };
			world.forceRegistry.Add(&newShot->round, &GravitationalForce);
			break;
		default:
			break;
		}
		switch (set_drag)
		{
		case PhySim::Accuracy_Drag::NON_OFF:
			break;
		case PhySim::Accuracy_Drag::CONSTANT:
			newShot->round.SetDamping(0.90f);
			break;
		case PhySim::Accuracy_Drag::FORCE_BASED:
			world.forceRegistry.Add(&newShot->round, &ResistanceForce);
			break;
		default:
			break;
		}

		break;
	case PhySim::ObjectType::LASER:
		newShot->round.SetMass(1.0f);
		if (set_motion == Accuracy_Motion::VELOCITY) newShot-
>round.Setvelocity(0.0f, 0.0f + (float)Config.elevation, 35.0f);
		else
		{
			LinearForce.acceleration = Vec3{ 0.0f,(float)Config.elevation,
10.f };
			world.forceRegistry.Add(&newShot->round, &LinearForce);
		}
		switch (set_gravity)
		{
		case PhySim::Accuracy_Gravity::NON_OFF:
			break;
		case PhySim::Accuracy_Gravity::CONSTANT_ACCELERATION:
			newShot->round.SetGravityAcc(0.0f, 0.0f, 0.0f);
			break;
		case PhySim::Accuracy_Gravity::FORCE_BASED:
			GravitationalForce.gravForce = { (avt)0.0f, (avt)-
worldGravityConst,(avt)0.0 };
			world.forceRegistry.Add(&newShot->round, &GravitationalForce);
			break;
		default:
			break;
		}
		switch (set_drag)
		{
		case PhySim::Accuracy_Drag::NON_OFF:
			break;
		case PhySim::Accuracy_Drag::CONSTANT:
			newShot->round.SetDamping(1.0f);
			break;
```

```
                case PhySim::Accuracy_Drag::FORCE_BASED:
                        world.forceRegistry.Add(&newShot->round, &ResistanceForce);
                        break;
                default:
                        break;
                }

                break;
        case PhySim::ObjectType::END:
                break;
        default:
                break;
        }
        newShot->ot = currentShotType;
        SetParticleSettings(newShot->round);
}
```

Figure 4-10 SimClient Shoot function

The pipeline of functionality for this method is to:

1. initialize a Shot structure
2. Compare current settings to set/initialize the required attributes.
3. Add these attributes to the particle and force registry
4. Store the shot to a collection
5. Store the current setting to the particle for update configuration.

Practically speaking this would not be as big for a system that was not demonstrating the combinational use of different settings in the PhySim engine. It would be expected that setting would be initialized once in the project and added to each particle fired.

The Update function essentially instructs the world control class of PhySim to execute all pending force generators on spawned particles and then update any particles that are currently active in the world checking their status, see below:

```
void Update()
{

        if (!paused)
        {
```

```
		world.RunPhysics((avt)1 / 60);

		for (World::Shot *shot = shotList; shot < shotList + maxAmmoCount;
shot++)
		{
			if (shot->status == PhySim::Status::FIRED)
			{


				// Run the physics
				shot->round.Intergrate((avt)1 / 60);

				// Check particle removal constraints
				if (shot->round.GetPosition().y < -5.0f || (shot-
>round.GetPosition() - Vec3{ 0.0,1.5,0.0 }).LengthSquared() > 10000)
				{
					// We set the status to dead, so the memory it
					occupies can be reused by the next shot.
					shot->status = PhySim::Status::DEAD;
				}
			}
			else
			{
				world.ClearParticle(shot->round);
				shot->round.Clear();
			}
		}

		glutPostRedisplay();
	}
}
```

Figure 4-11 Front end Update snippet

A pause Boolean is added to interrupt updating during the demonstration of the engine to read back any data on currently active particles. The end of the update function calls glutPostRedisplay(), this informs the compiler to execute the predefined display function for the Open GL GUI (display(), below).

Finally, the display function again iterates though the list of shots, accesses the round of each shot structure and reads the required attributes into values that can then be rendered as text in orthographic view.

```
void display()
{
	if (!paused)
	{
		// Clear the viewport and set the camera direction
		glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

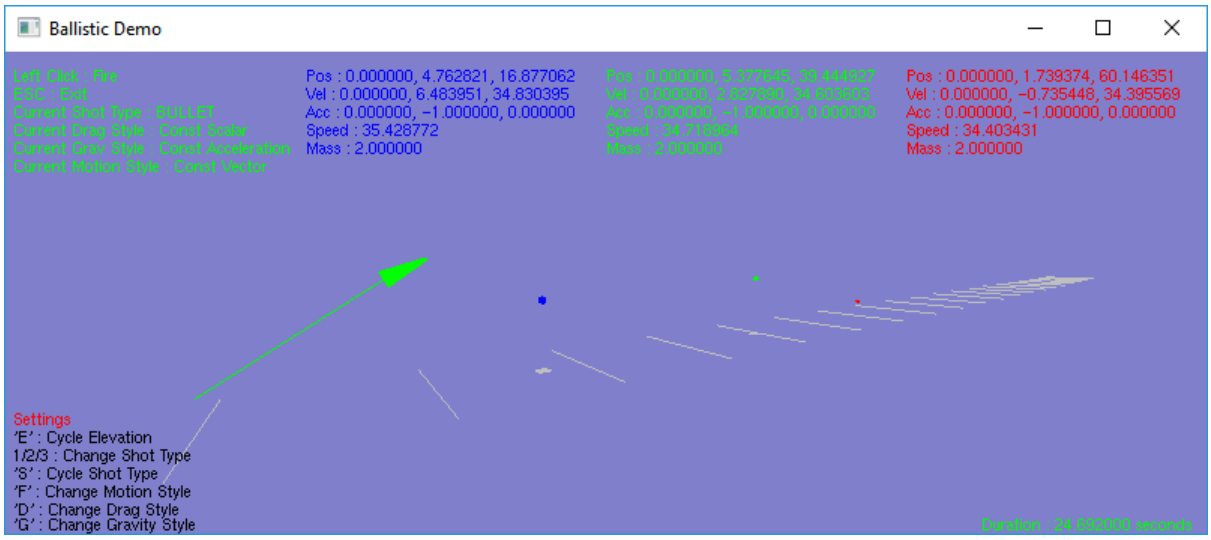
```
		glLoadIdentity();
		gluLookAt(-25.0, 8.0, 5.0, 0.0, 5.0, 22.0, 0.0, 1.0, 0.0);

		// Draw some scale lines
		glColor3f(0.75f, 0.75f, 0.75f);
		glBegin(GL_LINES);
		for (unsigned i = 0; i < 200; i += 10)
		{
			glVertex3f(-5.0f, 0.0f, i);
			glVertex3f(5.0f, 0.0f, i);
		}
		glEnd();

		DrawShootingLine(Vec3(0.0, 1.5, 0.0));


		//Show delta time since start
		duration = (std::clock() - startTime) / (double)CLOCKS_PER_SEC;

		RenderText(650.0f, 2.0f, ("Duration : " + std::to_string(duration) + "
seconds").c_str());

		RenderText(650.0f, 12.0f, ("Elevation : " +
std::to_string(Config.elevation)).c_str());

		//	// Render the descriptions

		glColor3f(0.0f, 1.0f, 0.0f);
		RenderText(5.0f, 300.0f, "Left Click : Fire");
		RenderText(5.0f, 288.0f, "ESC : Exit");

		ShowShotType();



		glColor3f(1.0f, 0.0f, 0.0f);
		RenderText(5.0f, 72.0f, "Settings");
		glColor3f(0.0f, 0.0f, 0.0f);
		RenderText(5.0f, 60.0f, "'E' : Cycle Elevation");
		RenderText(5.0f, 48.0f, "1/2/3 : Change Shot Type");
		RenderText(5.0f, 36.0f, "'S' : Cycle Shot Type");
		RenderText(5.0f, 24.0f, "'F' : Change Motion Style");
		RenderText(5.0f, 12.0f, "'D' : Change Drag Style");
		RenderText(5.0f, 2.0f, "'G' : Change Gravity Style");

		float x_coord = 600;
		for each (World::Shot proj in shotList)
		{
			if (proj.status == PhySim::Status::FIRED)
			{
				//show the current particle on screen
				renderParticle(&proj.round, &proj.Colour);

				//show said particles statistics
				ShotShotStats(&proj.round, x_coord, &proj.Colour);
				x_coord -= 200;
			}
		}

		glutSwapBuffers();
	}
```

}

The display function starts by telling the perspective camera to be placed in 3D space, then point the cameras field of view in a determined direction. Then, draw some horizontal lines along the “ground” for depth perception awareness in the direction of the pending projectile launch. After, the function renders text for all non-physics dependant information to show to the screen, this includes, controls, time elapsed, current settings.

Finally, the collection of particles is retrieved from the world class and iterated through. Each particle is physically rendered on the screen, and its accompanying statistics rendered afterwards.

A screen shot of the finished Demonstration GUI is below:



Equation 4-1 GUI Demo Image

5 Evaluation

5.1 Introduction

"The testing process is used to ensure that the program faithfully realizes the function. The essential components of a program test are the program in executable form, a description of the expected behavior, a way of observing program behavior, a description of the functional domain, and a method of determining whether the observed behavior conforms with the expected behavior." (Richards, et al., 1982)

Unfortunately, due to the circumstances of this project, a usability study would be un-economical to perform due to the required specific target audience. However, black, and white box testing can be carried out thoroughly. It is virtually impossible to account for every test possible when evaluating the effectiveness of a system. Through there are several main aspects of this system that can be tested independently.

The black box testing will essentially be executed by giving an input to the system, recording its output and comparing it to what would be expected. Then repeating this for various configurations. Varying configurations whilst testing will also passively test the stability/robustness of the current PhySim build. Also, some general heuristic style tests can be carried out with the user interface to make sure that its behavior and controls work as expected.

Finally, the white box testing will come in the form of a core functions walkthrough, highlighting any potential problems or risks with the code and where improvements could be made.

5.2 Black Box Tests

The definition for black box testing is as follows; "*Black box testing is based on the analysis of the specifications of a piece of software without reference to its internal working. The goal*

is to test how well the component conforms to the published requirement for the component." (Mohd, 2010). This sort of testing can be carried out on both sides of the developed system. Firstly, the demonstration GUI will be tested for general stability, functionality, and control/input handling. Then a set of scenarios will be devised to configure different inputs passed through the front end to the physics engine, and output recorded to analyze.

5.2.1 Front End

The front-end demonstration project is not designed with graphical complexity, rather control variation. Therefore, there will be a finite number of tests that can be carried out on the project. The following table shows the tests and results carried out on the SimClient project.

Test Name	Execution	Expectation	Result	Comments
Start	Run the program	The program will start and a window (rendering the scene will appear)	As expected	The program initializes itself fine, fast, and as expected.
Cycle Shot	Using the 'S' key, cycle through the three predefined shot types	The value showing current shot type will change after each cycle/toggle	As expected, though will not cycle if caps lock is on.	The shot type can be change with the 's' key, the input handling for keyboards is set to only recognize lower case letters
Specify Shot	Using numbers 1,2,3. Set a specific shot type to use.	Shots will change depending on the number inputted. These numbers will always bind to the same shot type.	As expected.	No case comparison with numbers, so they work fine.
Cycle motion	Use the 'f' key to cycle the different motion types.	Motion will cycle through the available selections in a loop.	As expected, though caps problem is present again.	The motions can be cycled though only when caps lock is disabled.

Cycle drag	Use the 'd' key to cycle the different motion types.	Drag will cycle through the available selections in a loop.	As expected, though caps problem is present again.	The drag can be cycled though only when caps lock is disabled.
Cycle gravity	Use the 'g' key to cycle the different motion types.	Gravity will cycle through the available selections in a loop.	As expected, though caps problem is present again.	The gravity can be cycled though only when caps lock is disabled.
Cycle elevation	Use the 'e' key to increase elevation	Elevation will increase to its maximum then loops back to minimum. And shown on lower right of screen.	As expected, though caps problem is present again.	The elevation can be cycled though only when caps lock is disabled.
Fire	Left click to fire a projectile	A projectile will be spawned on screen and fired in the direction the arrow is facing.	As expected	
Arrow path	Cycle elevation and monitor arrow movement	Arrow will increase and decrease appropriately with elevation changes	As expected.	Though angle of arrow head looks slightly off in a single instance of elevation (3).
Pause	Fire a particle and pause the scene.	All calculations will be paused including timing and projectile stats	As expected, though unpausing sees the duration time jump ahead to where it would be if not paused.	Some sort of issue with duration still incrementing in the background, though the pause does hold a set duration on the screen when firing.
Particle stats	Fire a particle and check that statistics for the particle whilst it is spawned is shown on screen.	Particles stats are shown in the top right of screen, same color as particle and are updated every frame.	As expected	
Multiple particles	Fire multiple particles and pause, make sure the	Particle stats are listed in the top right with different colors	As expected.	

	rendered stats are correctly color bound to the projectiles fired in order.	relating to the different colors of particles spawned.		
Exit	Close the application using 'Esc' key, the window X and the mouse right click.	Application will exit	As expected.	

Table 5-1 Front end basic tests

All functionality works as expected for the front-end system except for two main points which could be identified as bugs. Firstly, the keyboard input handler does not account for capital letters. To fix this the program could add extra handling to account for each capital letter, or write a generic procedure to evaluate the capital version of each letter input also. Secondly, the timing of the duration does not stop internally as proven when the system is un-paused. However, by design the duration does restart with each new projectile fire (to map the newer statistics as the projectiles will be traveling on the same path). Upon investigation, this is due to the project using the system clock for time stamping the start time of a projectile, then calculation the duration by subtracting the start time. As the system clock cannot be stopped. A new timing system implementation would be the preferable course of action to fix this, possibly running in parallel on a separate thread.

5.2.2 Back End

To test the back-end system, a predefined collection of setting combinations was identified. The engine was then run, firing a projectile with each separate set of settings. The statistics were then recorded for each projectile over several points in time. These points were then plotted onto graphs to examine the different curvature of the properties of the projectile over time. It was a logical decision to separate statistics via each relevant axis. The projectiles

were fired along the Y and Z axis with no change to X, therefore X was not recorded. Each graph of an axis (if relevant to the graph) includes the corresponding, acceleration, velocity, and displacement in that axis direction.

For readability, each test case will list configurations used in the scenario, then a table of values, then any graphs generated from said values and finally a description of the results.

5.2.2.1 Test 1

5.2.2.1.1 Results

Shot	Bullet
Drag	Constant Scalar
Gravity	Constant Acceleration
Motion	Constant Velocity
Elevation	2
Scalar constant for drag	0.9

Figure 5-1 Test 1 Settings

Time	Pos (x)	Pos (y)	Pos (z)	Vel (x)	Vel (y)	Vel (z)	Acc (x)	Acc (y)	Acc (z)	Speed	Mass
0	0	1.5	0	0	2	35	0	-2	0	35	2
0.449	0	2.290903	15.395968	0	1.484171	33.37929500	0	-2	0	33.412273	2
0.683	0	2.628527	23.639082	0	1.199069	32.511562	0	-2	0	32.533665	2
1.267	0	3.145672	42.0492860	0	0.562321	30.573565	0	-2	0	30.578735	2
2.051	0	3.276828	65.0570980	0	-0.23345	28.151579	0	-2	0	28.152548	2
2.668	0	2.956363	81.8800740	0	-0.8153	26.380665	0	-2	0	26.393261	2

Table 5-2 Test 1 Results

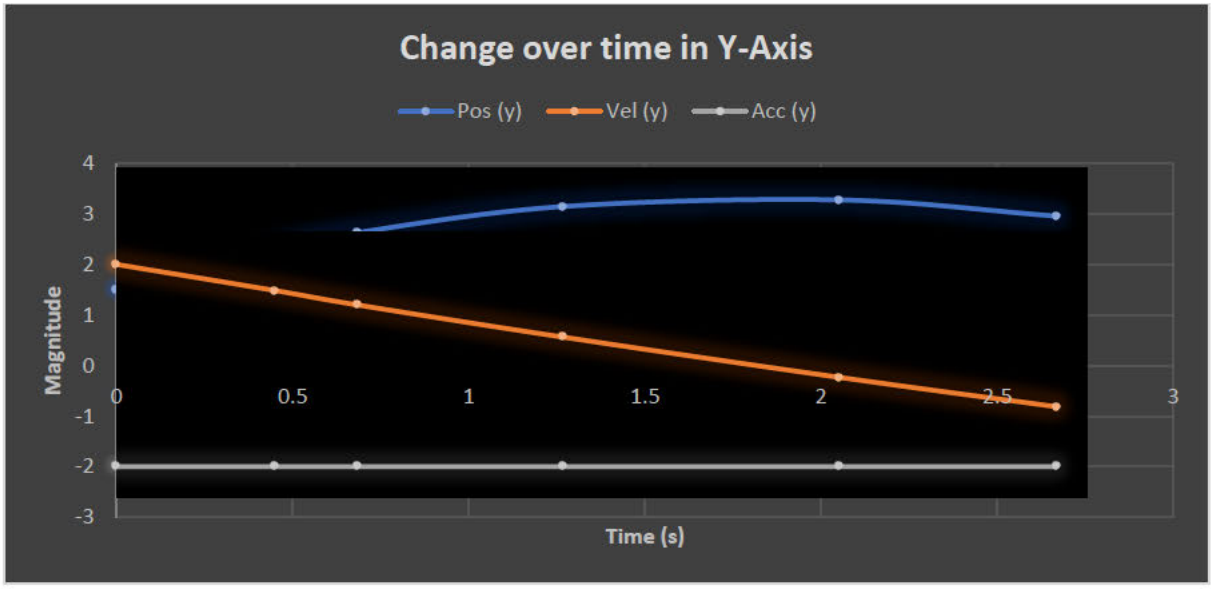


Figure 5-2 Test 1 Change in magnitude over time for Y axis

As shown, there is an attractive curve highlighting the perfect displacement in the Y axis of any projectile acting under gravity.

The velocity of Y also drops gradually because the scene is set to use a constant dampening scalar with respect to a constant time interval, which is effecting the objects velocity.

Acceleration was set as a constant value and so the result would be a horizontal straight line as expected.

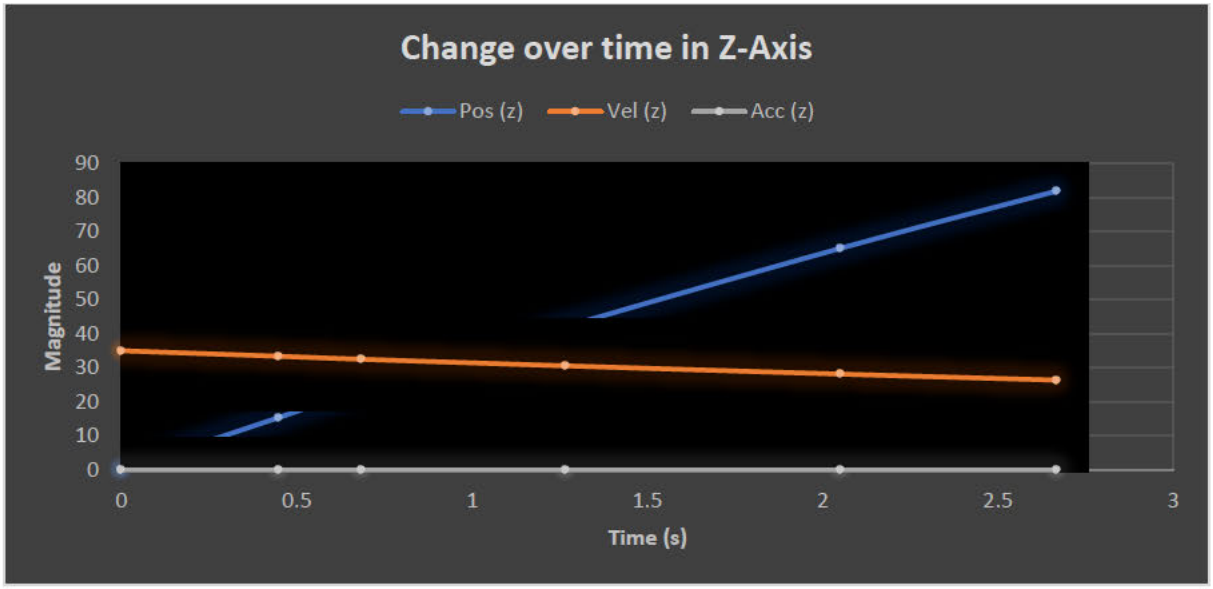


Figure 5-3 Test 1 change in magnitude over time Z axis

Again, position Z should have a near linear increase in position as this is the horizontal component of motion. There seems to be a slight arc which would be down to the rate of change of position (velocity) being reduced over time due to constant dampening. This is also visible from the velocity line. Acceleration is a constant 0 because Velocity of the object is directly injected.

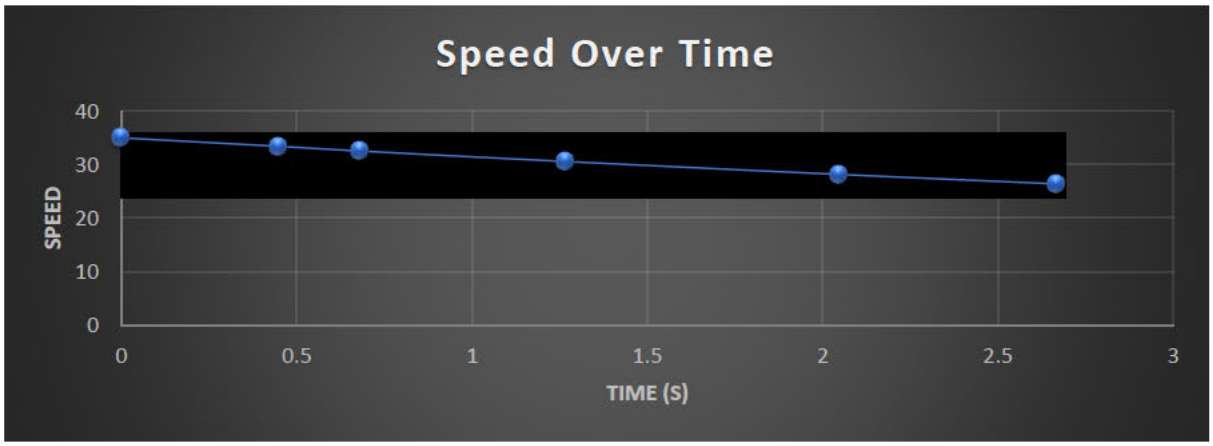


Figure 5-4 Test 1 change in speed over time

Speed over time falls linearly due to constant velocity dampening, this is a good expected result.

5.2.2.1.2 Summary

So far, all the results of a constant scalar velocity test for a bullet with small mass have executed as predicted with some attractive and fairly accurate graph plots. It is worth noticing the speed of the projectile at time = 0; This is one major difference of using direct velocity injections from a game engine perspective. The particle is spawned with its velocity.

5.2.2.2 Test 2

5.2.2.2.1 Results

Shot	Bullet
Drag	Constant Scalar
Gravity	Constant Acceleration
Motion	Constant Velocity
Elevation	0
Scalar constant for drag	0.9

Figure 5-5 Test 2 settings

Time	Pos (x)	Pos (y)	Pos (z)	Vel (x)	Vel (y)	Vel (z)	Acc (x)	Acc (y)	Acc (z)	Speed	Mass
0	0	1.5	0	0	0	35	0	-1	0	35	2
0.1	0	1.4585	4.061904	0	-0.099388	34.57242107	0	-1	0	34.572552	2
1.301	0	0.7031	43.0675130	0	-1.213833	30.466377	0	-1	0	30.490547	2
1.984	0	-0.32	63.1720730	0	-1.788248	28.350014	0	-1	0	28.406357	2
2.067	0	-0.471	65.5262910	0	-1.855512	28.102188	0	-1	0	28.16338	2
2.435	0	-1.203	75.6428220	0	-2.144555	27.037243	0	-1	0	27.122162	2

Table 5-3 Test 2 Results

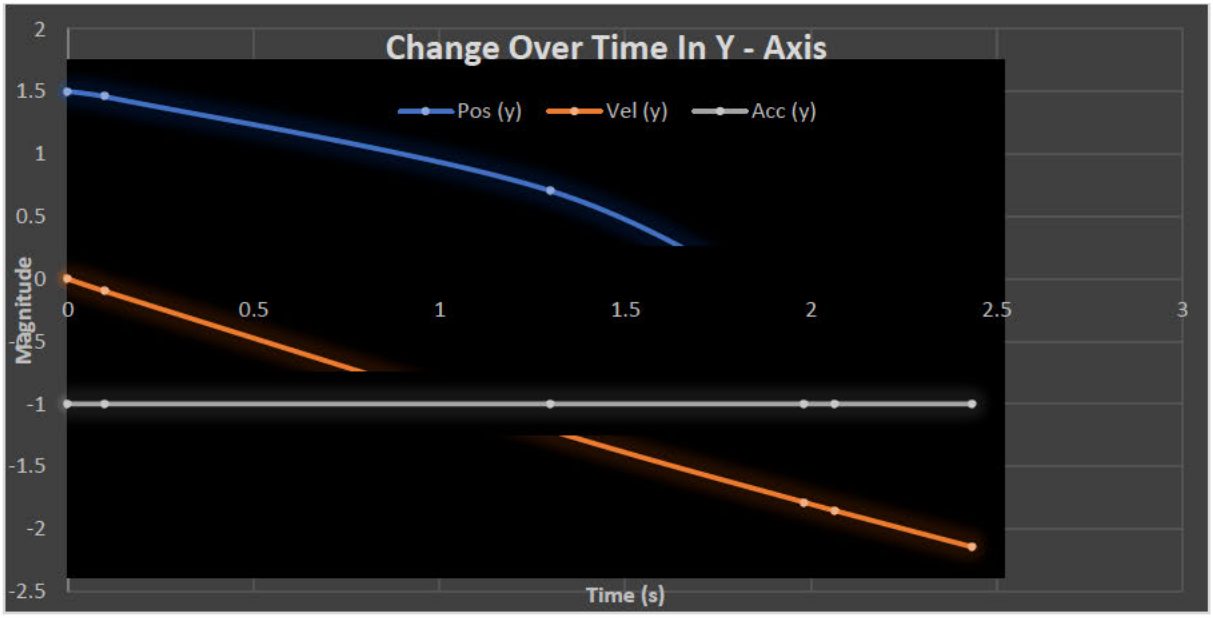


Figure 5-6 Test 2 magnitude over time Y axis

Again, results appear nicely as expected, a consistent drop from the starting Y position due to 0 shot elevation. The position curve looks to pinch inwards due to the gradual reduction in velocity. Acceleration is once again. A constant acceleration downwards.

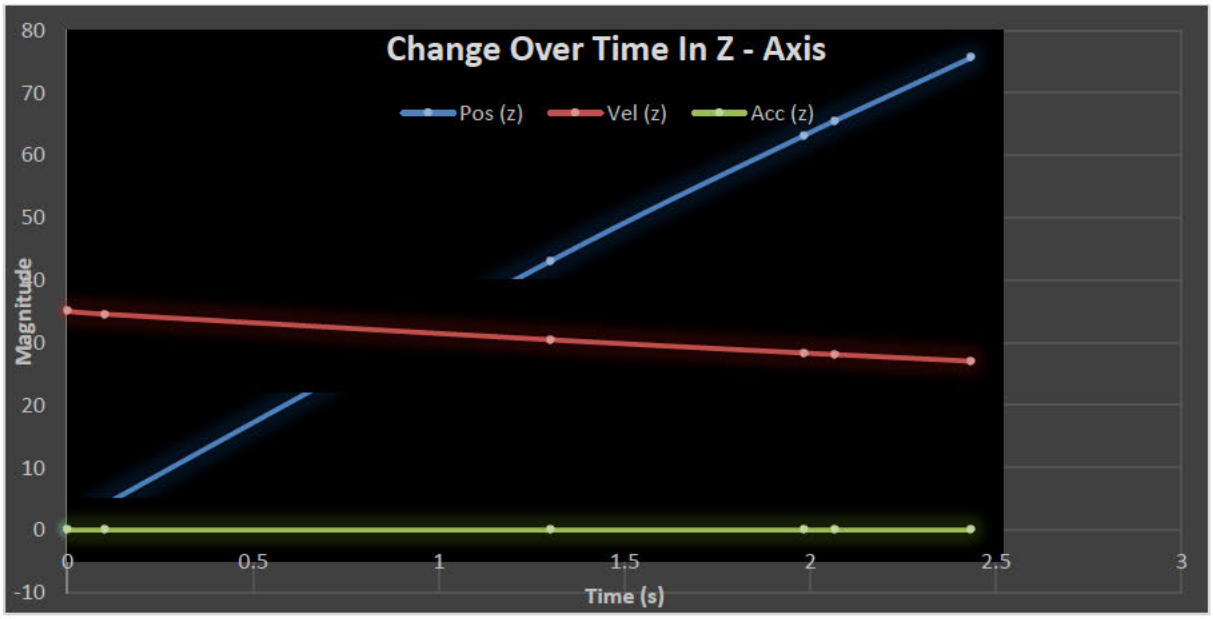


Figure 5-7 Test 2 magnitude over time Z axis

Again, results are as expected. A slight curve in the rate of change of position originates from the gradually decaying velocity.

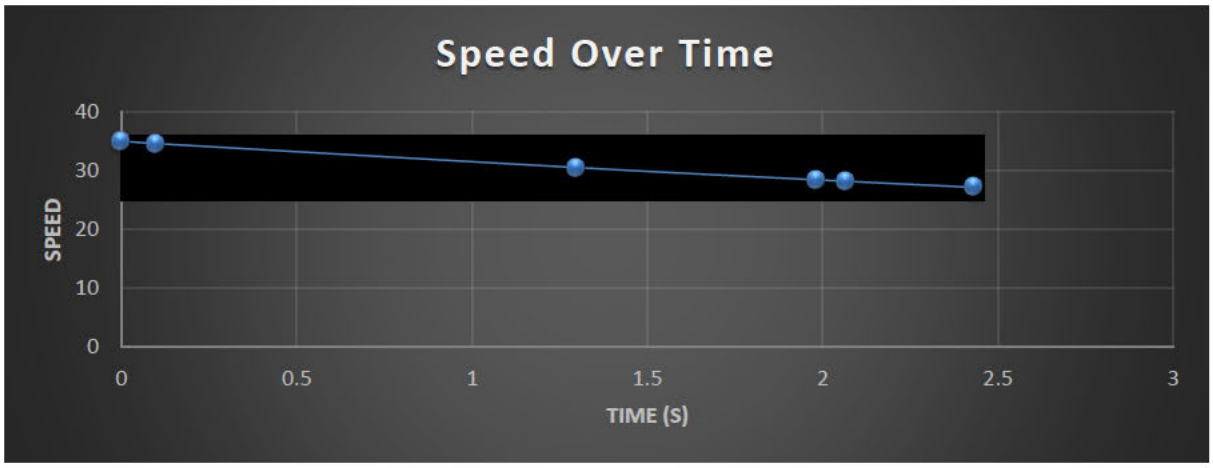


Figure 5-8 Test 2 change in Speed over time

As mentioned above a decaying velocity will equate to a decaying speed.

5.2.2.2.2 Summary

All results for the constant setting variation of the system using a preset bullet mass have displayed exactly what would be expected of them. So far, the engine seems to stably calculate this accuracy with relative ease.

5.2.2.3 Test 3

5.2.2.3.1 Results

Shot	Bullet
Drag	Vector Force
Gravity	Vector Force
Motion	Vector Force
Elevation	0

Figure 5-9 Test 3 Settings

Time	Pos (x)	Pos (y)	Pos (z)	Vel (x)	Vel (y)	Vel (z)	Acc (x)	Acc (y)	Acc (z)	Speed	Mass
0	0	1.5	0	0	0	0	0	-1	0	0	2
0.3	0	1.3	0.425000	0	-1.2	3	0	-4	10	3.231099	2
0.717	0	0.4966666	2 508333	0	-2.8666667	7.166664	0	-4	10	7.718734	2
1.176	0	-1.1833333	6.7083330	0	-4.666665	11.666670	0	-4	10	12.565387	2
1.534	0	-3.15111	11.6277800	0	-6.133330	15.333344	0	-4	10	16.514513	2
1.751	0	-4.566665	15.1666710	0	-6.999996	17.5000006	0	-4	10	18.848082	2

Table 5-4 Test 3 Results

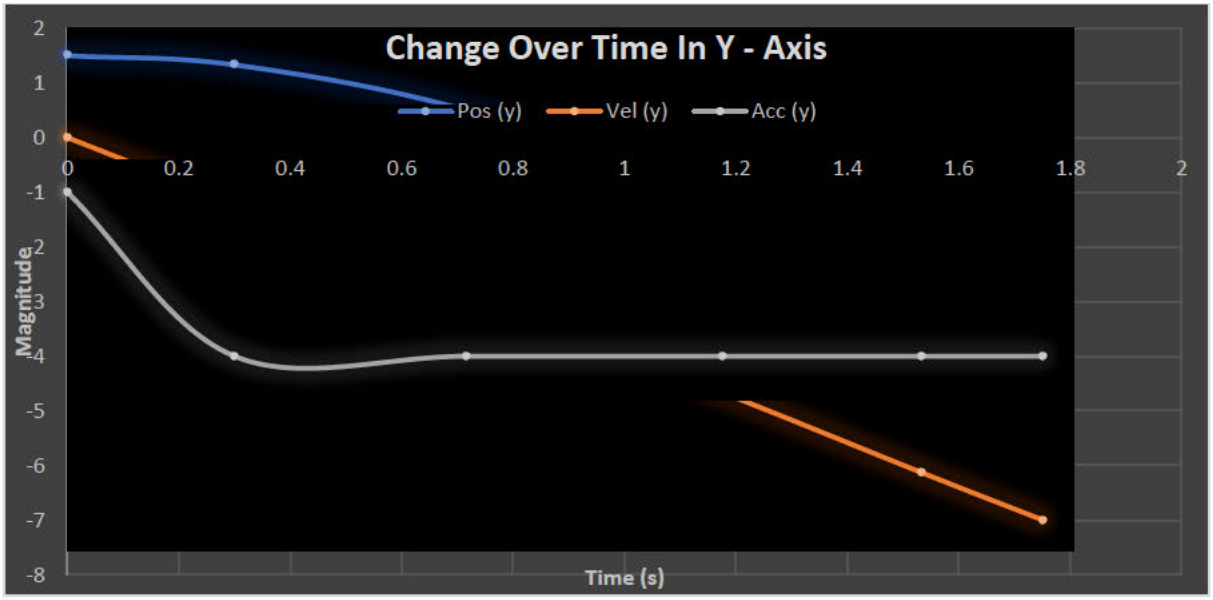


Figure 5-10 Test 3 change in magnitude over time Y axis

The position change and velocity decay both complement each other and work as expected, the velocity does not show as increasing here, nor should it. This is a single component of the dimensional vector (non-scaler). Acceleration shows some expected behavior but seems to be much sharper than anticipated. The world gravity acceleration is acting on the particle as soon as it spawns. However, the local acceleration of the particle should increase in magnitude (become more negative) much more gradually. Until it flattens out at which point terminal velocity is attained. This may require some investigation and will be summarized when the root cause has been detected.

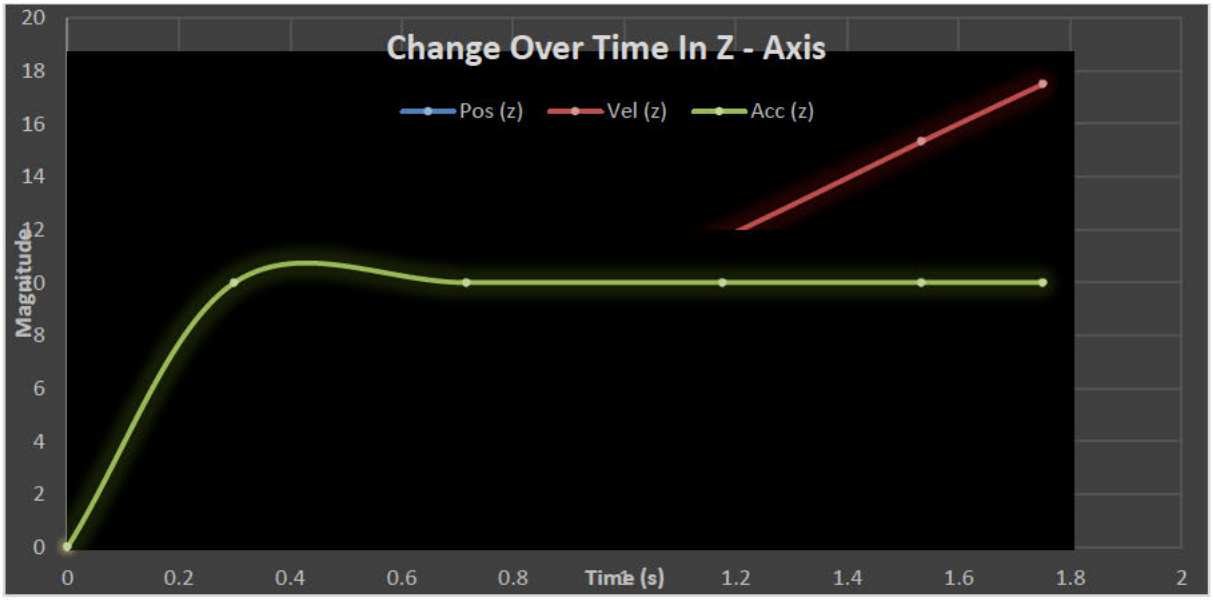


Figure 5-11 Test 3 change in magnitude over time Z axis

The first point of interest here is all points begin at 0 as appose to an injected velocity. This is a good representation of how force variant motion is calculated from adding resultant forces to a projectile which begins at rest. The velocity and position curves here complement each well once again. However, it seems there is strange behavior with acceleration. The curve itself is a slightly bad fit as it shows a bump upwards which does not represent the flow of acceleration over time. As you can see the acceleration quickly increases to a maximum value and then stays there. It is possible that this is due to scaling issues from the drag coefficients used when running the force registry of physics. As this behavior would be expected in the Z axis but on a more gradual scale. Also, the curvature of acceleration in the first 0.4 seconds should cause a steeper gradient in the velocity curve during the same time frame. This would lead us to believe that the acceleration values do not directly affect the particle. However, this would be impossible because without an acceleration/force, there would be no velocity within these settings. This may require some further investigation.

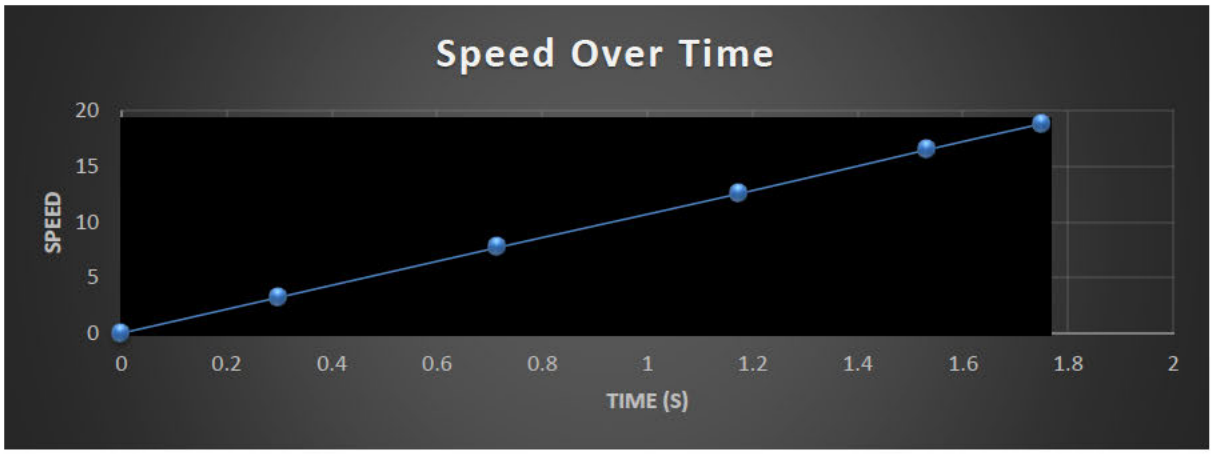


Figure 5-12 Test 3 change in speed over time

The speed curve again only partially acts as expected. Due to a constant acceleration toward s the ground, it would be expected that the speed would gradually increase in a slightly curved manor, before leveling out at terminal velocity and then dipping below its highest point slightly due to horizontal drag forces (kinetic reaction). This is suspected related to the strange acceleration behavior noticed in the two prior graphs.

5.2.2.3.2 Summary

The particle seems to fall nicely under vector force constraints as a projectile would. However, it has been identified that there is a potential problem with that the acceleration change over time. It does start from 0 and grown in magnitude as expected for more realism. Although the rate of change of acceleration seemed anomalous. After some internal investigation, there would be to reasons combining to create this sort of behavior.

Firstly, there is the possibility of the two default drag coefficients being completely inaccurate. It would be up to the developer to investigate the interaction between the objects drag and drag instigating components constants for accurate force calculation.

Secondly would be the issue of force applied over time. Unfortunately, an issue with the C++ Vector class and custom object iteration meant that it is not yet possible to remove single forces from the world force register. This means that all resultant forces are applied to particles and then all forces in the register are removed. This is sadly not the real case in terms of forces as the force of gravity should be applied every frame, and drag applied every frame in proportion to velocity. Now, the engine can apply each frame any forces added in the registry and then remove them by clearing the registry. A work around to fix this issue could be achieved by re-adding the gravitational and drag forces to a particle each frame by re-adding them to the world force register. This is an undesirable and messy technique which would leave it up to the user to know about the internal constraint. The preferable way to combat this would be to re-0investigate the <vector> class in C++ and understand its internal issue with custom object, iteration removal. From here, find a better way to use said vector class or implement a custom collection system all together.

Unfortunately, both methods have an unpredictable estimated time to completion and so would be a point of interest for future works in the project. The positive point to make here is that the issue was within the world controller class and not the physics engine specifically. Meaning resultant force vectoring and update methods do work. Also, though there is a slight problem, it could be said that the vector force method still shows more realism than the constant setting method. Because acceleration, speed and displacement start at 0.

5.2.2.4 Test 4

The test in this section is simply a proof on concept/capability for the engine. It will not be analyzed in depth past this brief explanation as the result are intuitive to understand. The results here are exactly what would be expected of such a scenario. What follows here is an example of a laser projectile. A laser projectile would be an object of finite mass. Though the

mass here has been set to 1 because a mass of 0 as mentioned earlier, would represent an immovable object of infinite mass. A laser (having finite mass) would not experience any drag or gravity forces, therefore position change is linear and velocity is constant.

Shot	Laser
Drag	Constant Scalar
Gravity	Constant Acceleration
Motion	Constant Vector
Elevation	1
Drag	1

Figure 5-13 Test 4 Settings

Time	Pos (x)	Pos (y)	Pos (z)	Vel (x)	Vel (y)	Vel (z)	Acc (x)	Acc (y)	Acc (z)	Speed	Mass
0	0	1.5	0	0	1	35	0	0	0	35.014282	1
0.2	0	1.7	7.583335	0	1	35	0	0	0	35.014282	1
0.567	0	2.0833333	20.4166680	0	1	35	0	0	0	35.014282	1
1.8	0	3.316665	63.583279	0	1	35	0	0	0	35.014282	1
2.034	0	3.549998	71.7499770	0	1	35	0	0	0	35.014282	1
2.335	0	3.849998	82.2500230	0	1	35	0	0	0	35.014282	1

Table 5-5 Test 4 Results

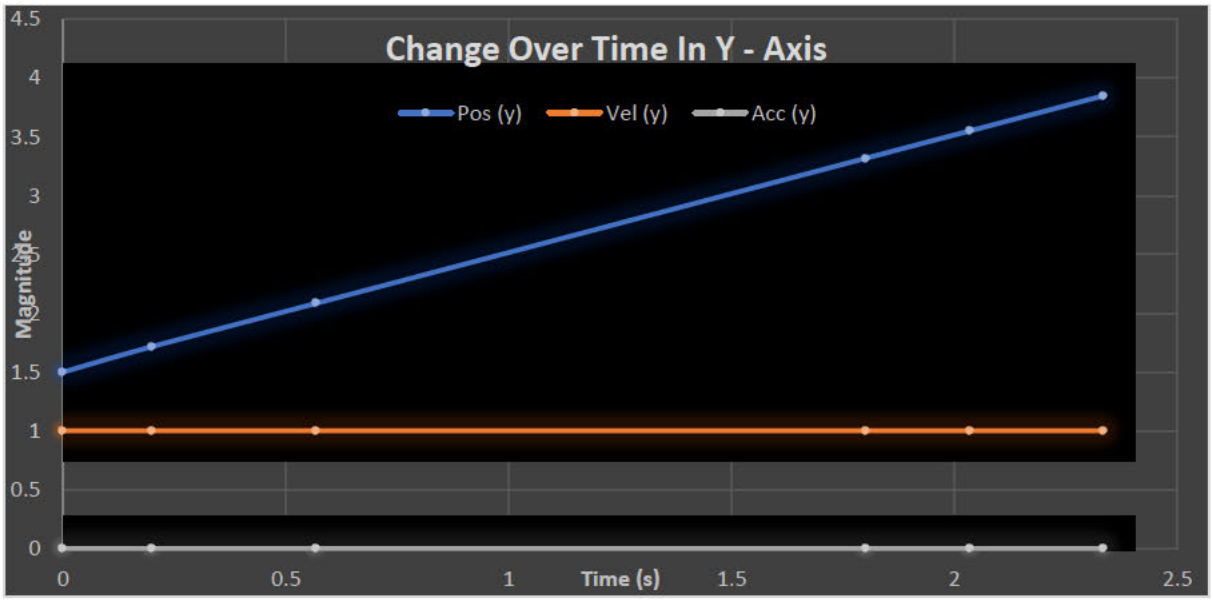


Figure 5-14 Test 4 Change in magnitude over time Y axis

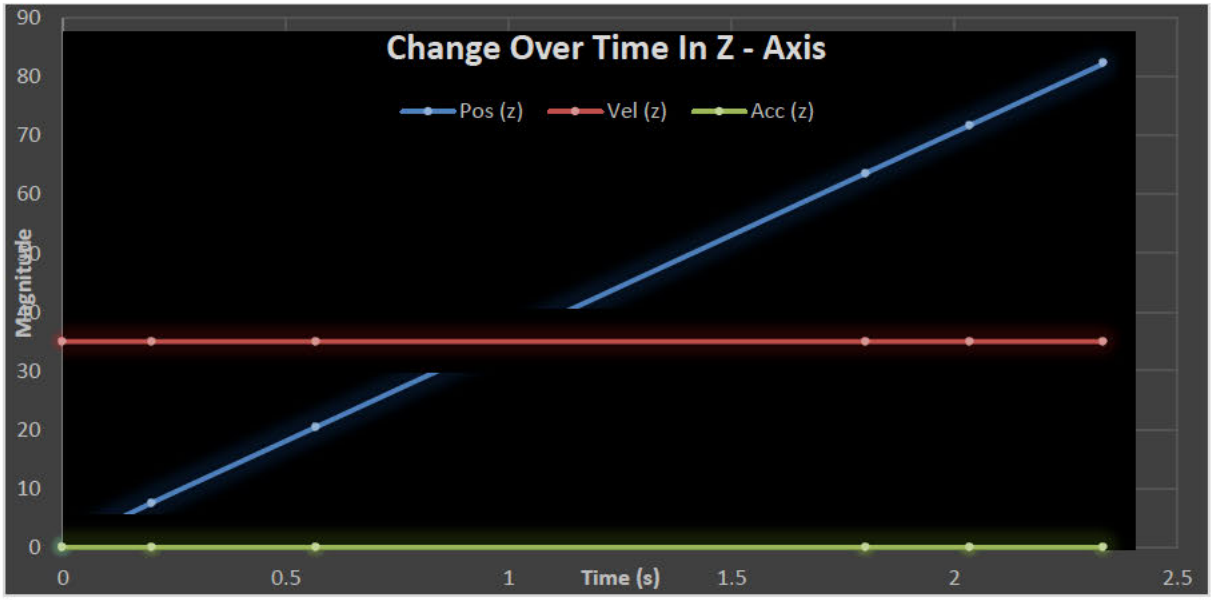


Figure 5-15 Test 4 change in magnitude over time Z axis

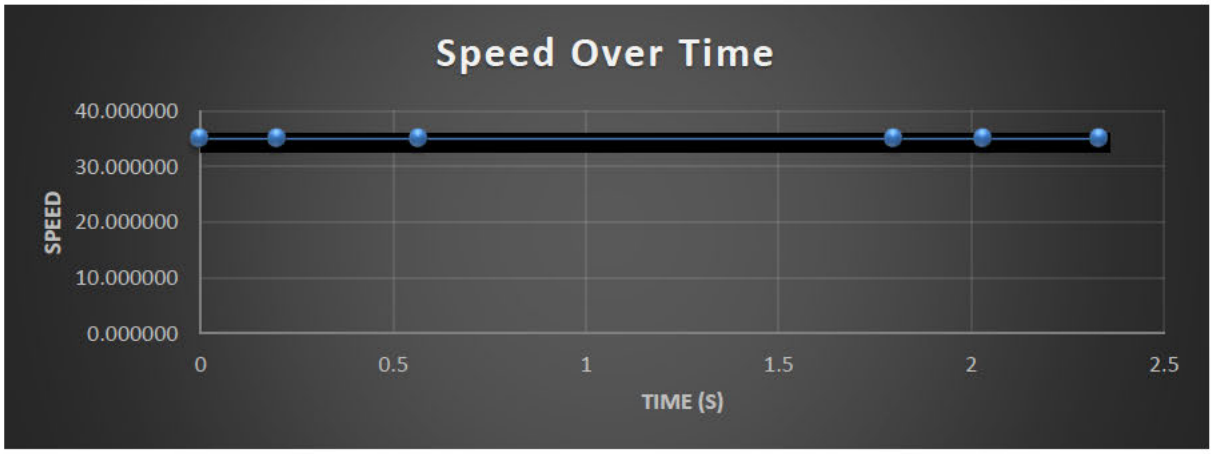


Figure 5-16 Test 4 change in speed over time

5.2.2.5 Test 5

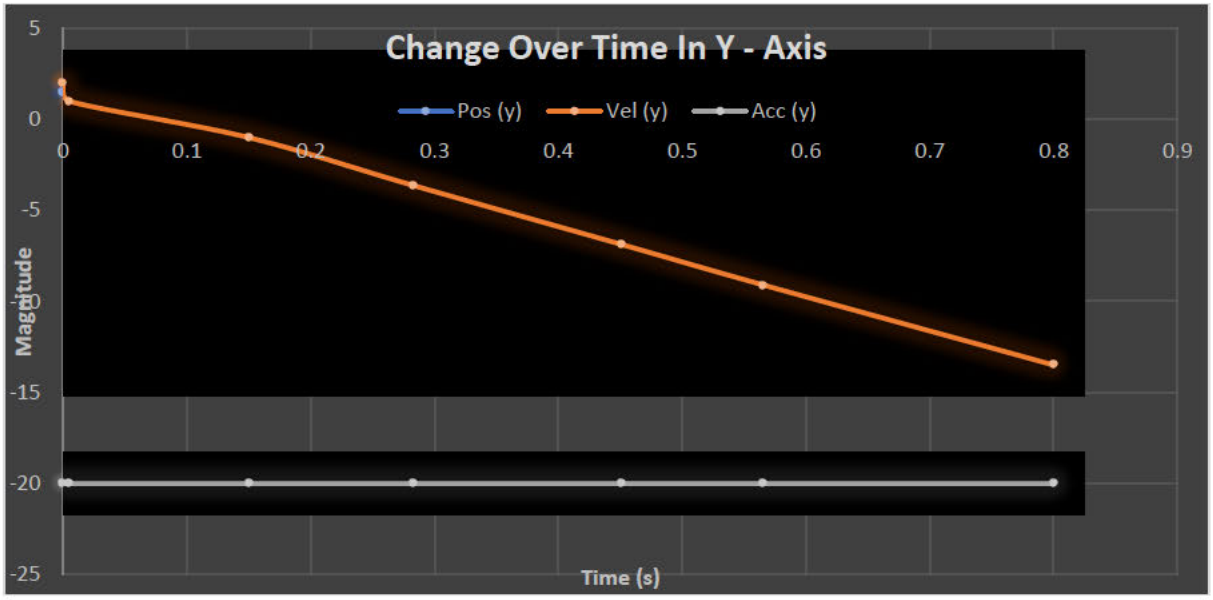
5.2.2.5.1 Results

Shot	Sphere
Drag	Constant Scalar
Gravity	Constant Acceleration
Motion	Constant Vector
Elevation	1
Drag	0.9

Figure 5-17 Test 5 Settings

Time	Pos (x)	Pos (y)	Pos (z)	Vel (x)	Vel (y)	Vel (z)	Acc (x)	Acc (y)	Acc (z)	Speed	Mass
0	0	1.5	0	0	2	40	0	-20	0	40.04997	200
0.005	0	1.616355	2.659657	0	0.989506	39.720024	0	-20	0	39.73235	200
0.15	0	1.63188	6.614278	0	-1.008620	39.30373	0	-20	0	39.316673	200
0.283	0	1.343481	11.8227090	0	-3.640248	38.755451	0	-20	0	38.926037	200
0.451	0	0.493143	18.2311920	0	-6.878222	38	0	-20	0	38.697033	200
0.566	0	-0.421227	22.6506400	0	-9.111207	38	0	-20	0	38.703342	200
0.8	0	-3.024244	31.3282240	0	-13.495668	37	0	-20	0	39.104736	200

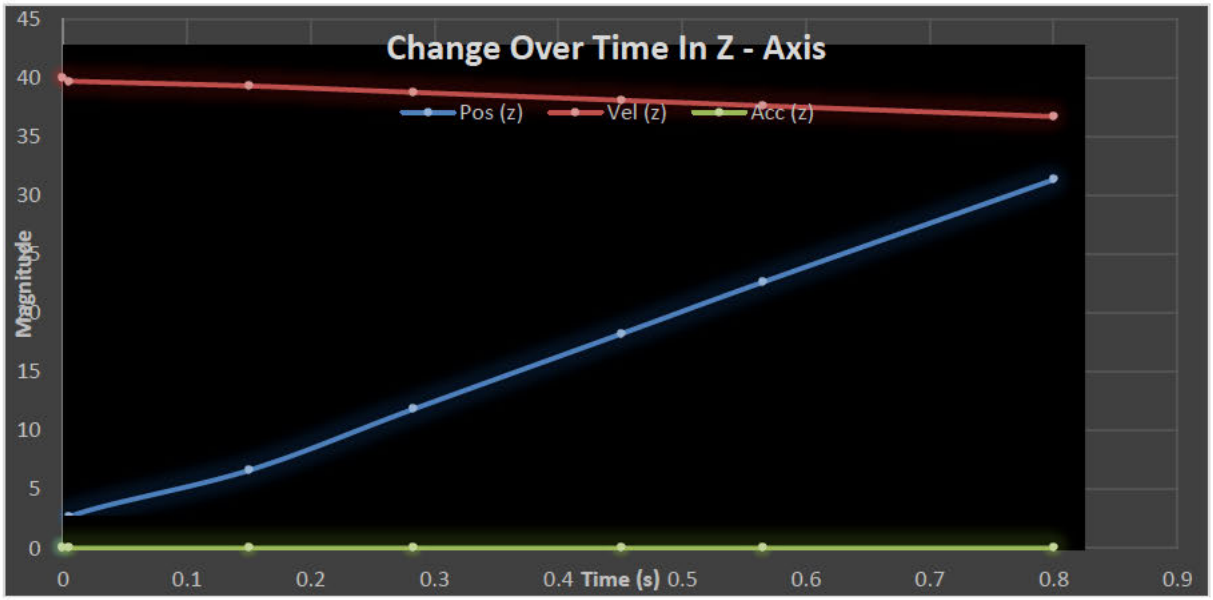
Table 5-6 Test 5 Results



Equation 5-1 Test 5 change in magnitude over time Y axis

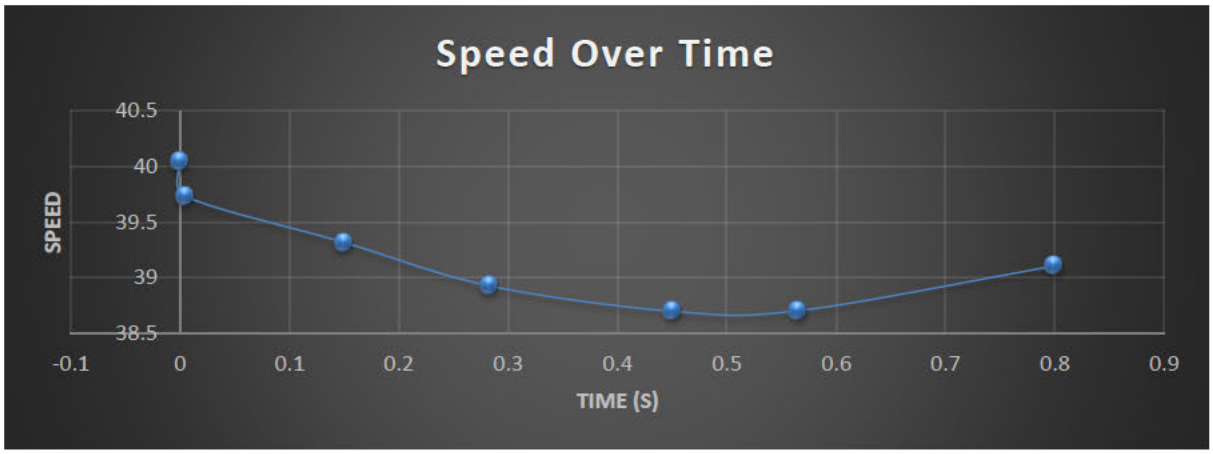
With the smallest increase in elevation, it is visible that there is a very slight increase in the Y position before an increasing decline over time. An increase in magnitude of velocity is obvious due to the very high negative acceleration in the Y axis. There appears to be some form of anomalous data in the at the beginning instant of velocity mapping. I would expect

this to be human error when recording the data because the velocity, position and acceleration trend lines work cohesively.



Equation 5-2 Test 5 change in magnitude over time Z axis

The decay in velocity here mirrors through to the position nicely. If the velocity were constant, the gradient of position over time would be of the order f(X)=Y. The gradient of the position over time trend line is slightly lower than X=Y which is to be expected with velocity decay. As this is a constant configuration, acceleration is constant again in each direction. Finally, there seems to be another possibly anomalous data point in the position over time data. At time = 0.15, the position in Z is slightly lower than as to be expected, this is obvious as the position trend line should be completely linear. Though the data is still very close to its predicted point and within acceptable error margins.



Equation 5-3 Test 5 Change in speed over time

The speed over time graph has the correct shape of curvature as expected of the particle over its lifetime. However, the curve is far from smooth and continuous. It is possible at this point to contemplate this is due to floating point inaccuracy between PhySim computations and graph recording to plotting. The general behavior of the curve is still identifiable and as expected, though it may be worth passing over this test again when updating the engine further.

5.2.2.5.2 Summary

The fundamental nature of the projectile in under these settings is relatable to what would be expected. However, there did seem to be several accuracy instabilities. Though these instabilities were not out of range of acceptable data deviation.

5.2.2.6 Test 6

5.2.2.6.1 Results

Shot	Sphere
Drag	Vector Force
Gravity	Vector Force
Motion	Vector Force
Elevation	4

Figure 5-18 Test 6 Settings

Time	Pos (x)	Pos (y)	Pos (z)	Vel (x)	Vel (y)	Vel (z)	Acc (x)	Acc (y)	Acc (z)	Speed	Mass
0	0	1.5	0	0	0	1.666667	0	0	10	0	200
0.6	0	1.5	1.750000	0	0	5.999999	0	0	10	5.999999	200

1.751	0	1.5	15.1666710	0	0	17.500006	0	0	10	17.500006	200
2.101	0	1.5	21.875004	0	0	20.99999	0	0	10	20.99999	200
2.685	0	1.5	35.7777710	0	0	26.8333303	0	0	10	26.833030	200
3.3668	0	1.5	56.3916320	0	0	33.66663	0	0	10	33.66663	200

Table 5-7 Test 6 Results

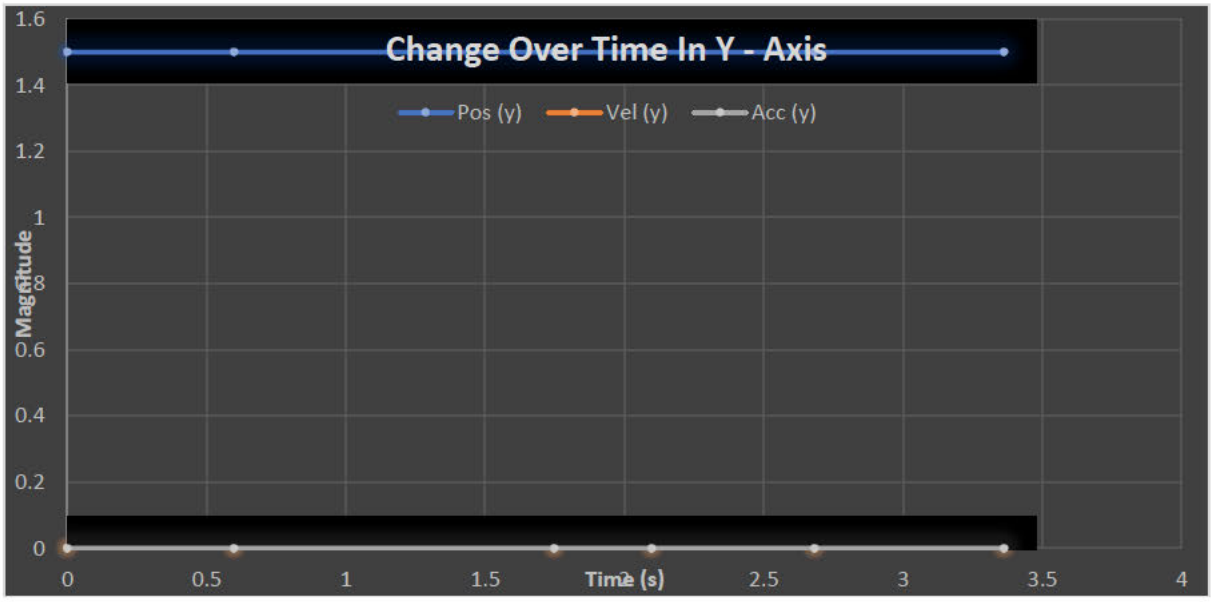


Figure 5-19 Test 6 change in magnitude over time Y axis

This graph shows a failure in the functionality of the PhySim system. The Position in the Y axis is constant, meaning no falling under gravity is happening to the object. There is 0 velocity and 0 acceleration in the Y direction.

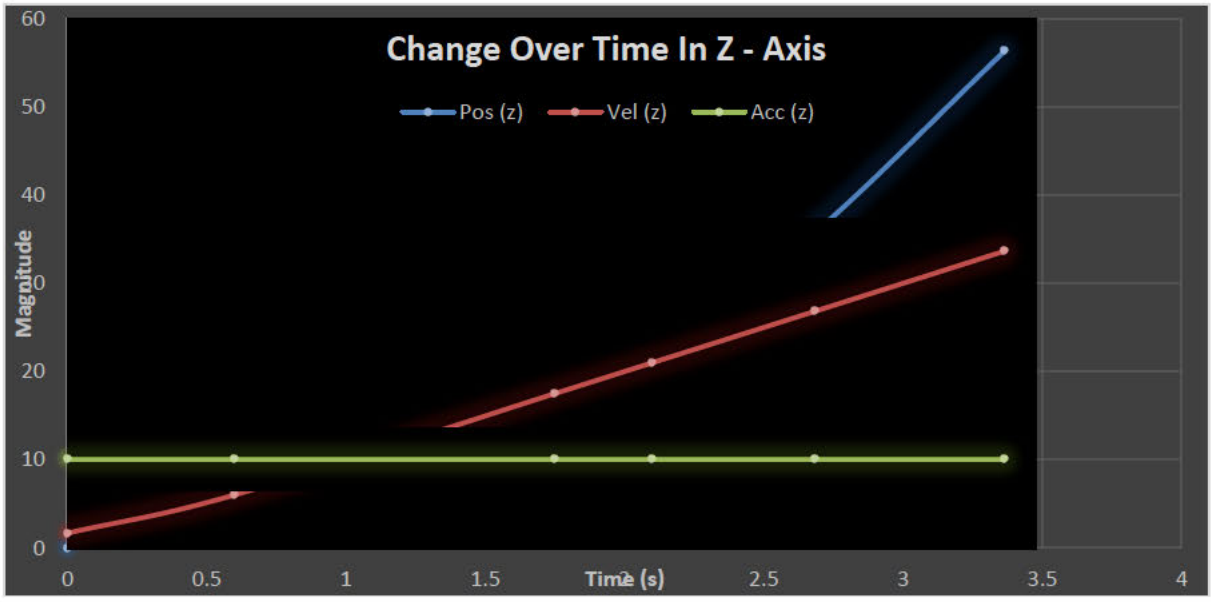


Figure 5-20 Test 6 change in magnitude over time Z axis

The Z axis show similar behavior to what should happen with constant velocity settings, except the start velocity is much smaller. A constant acceleration in the Z axis is met by no drag, resulting in no velocity decay. This means that the projectile with speed up at a constant rate until destroyed.

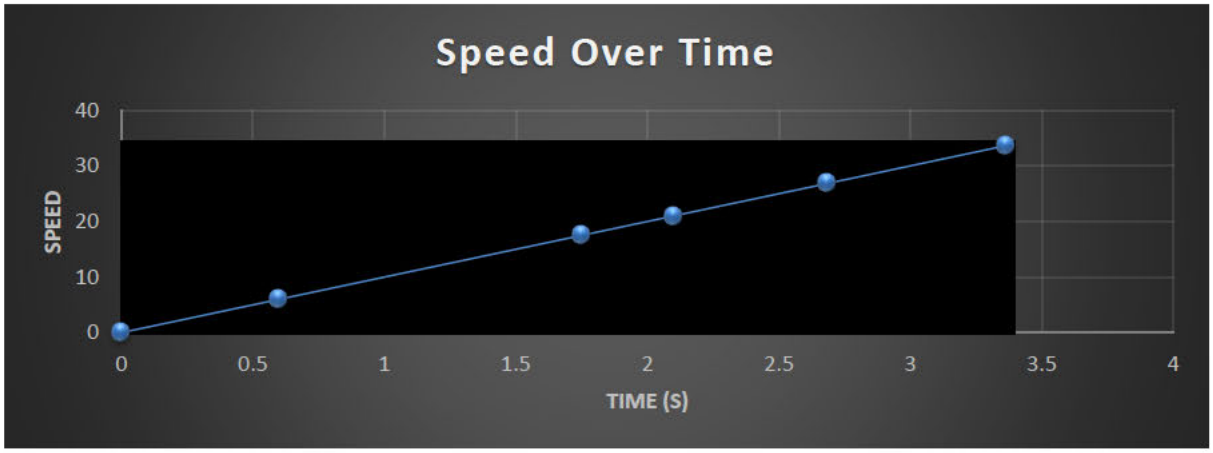


Figure 5-21 Test 6 change in speed over time

As mentioned previously, this circumstance shows the speed incrementing at a constant rate from 0 to infinity or death of the particle.

5.2.2.6.2 Summary

This test represents the first complete failure when comparing the expected result to actual results. The suspected reasoning behind this is the same as the reason for issues in Test 3, There is no capability to add a constant force over time. The force was applied at frame 1, but because the mass is so much bigger, the resultant direction change was negligible. Again, to combat this issue properly, an update to the engine to manage a world force register over time would need to be implemented. Drag force should be proportionate to the increasing speed of the projectile, and thus the acceleration should decrease over time. Unfortunately, to fully implement this fix would require an investigation into the C++ <vector> collection class, with no estimated period. It is an issue to address when improving the system in the future.

5.3 White Box

5.3.1 Front End

The front-end system proved to be robust and stable. There was not a lot of depth in the aspect of variable functionality, but it was designed well do demonstrate the capabilities of the built system.

Conducting a walkthrough within the code showed that most the SimClient application was design well, with no major efficiency issue. There are two potential improvements that become apparent from a glance, however these are strictly to improve code readability.

Firstly, the code openly mixed PhySim library manipulation with the Open GL library. It may be argued that the code could be better maintained if these areas were separated. Possibly by loading all Open GL functions into a secondary class of which the SimClient.cpp inherits.

Secondly. An independent timing class could be developed to maintain force overtime metrics and projectile duration measurements. This would help negate both issues found through black box testing (timing duration issue, extended force application issue).

5.3.2 Back End

Conducting a walkthrough of the PhySim application showed less need for improvement of core mathematical functionality. The primary issue here was the manipulation of the

<vector> class which is used for the force register collection. The vector class was necessary as the collection it implemented was dynamic in size. The was paramount for a physic engine. Further investigation is needed to acquire more insight as to why there were internal errors within the vector library.

Another possibility for refactoring would be to privatize more of the engine, a lot of classes were public that were not particularly required to be. Some developers may find it useful to have these classes removed from their view when using the engine. Although this is purely speculative and other may prefer the exact opposite.

6 Conclusion

6.1 Chapter Summaries

The following sections contain a brief conclusion of the Literature Review, Design, Implementation, and Evaluation chapters.

6.1.1 Literature Review Conclusion

The literature review proved very successful in gaining knowledge of key attributes of a particle system and what its capabilities are in a swarm scenario. Whilst swarm particles is several updates ahead of this system. The key attributes and underlying math’s appeared to be similar.

6.1.2 Design Conclusion

The design showed a brief insight into the intentions of the project. The GUI design proved reliable and informative of all relevant data. However, a timing data design may have been preferable, had that issue been foreseen beforehand.

6.1.3 Implementation Conclusion

Implementation ran smoothly in terms of mathematic functionality. It would have been desirable to add functionality to increase accuracy one step further, possibly by implementing newtons full form gravitational law. Also, the complete force drag equation. Though because enums were used as the basis for these configurations, these extras could be later developed and added with ease. This would not pose a risk to any already implemented effects. A similar example would be the testing of this project. The faster computing constant setting variation worked seamlessly, which the force engine had a few teething issues. The code has been built in a relatively extensible way; however, there may be opportunities for optimization further by refactoring the use of pointers in the engine. This would require a stronger experience with using the C++ language.

6.1.4 Evaluation Conclusion

As mentioned, the outcome of testing showed that the engine managed to handle basic projectile motion very well. There were some undesired effects when accuracy was scaled up, though these issues have been identified and solutions considered.

The GUI project performed perfectly as desired, and did not present a single real issue. Code improvements were considered and could be implemented in later versions. However, as this part of the project was not the primary concern, nor would it be used every time the

physics engine is used. These changes to the GUI will most likely not be implemented. The PhySim engine has now provided a good basis for the start of a complete, modular physics/game engine. The project has proved that it can implement different executional pipelines to augment scale of realism in world physics. However, it did not manage to implement enough accuracy to be truly tagged as simulation grade. This is not to say that this couldn't be added in the future.

It may take a different angle of investigation (once the engine has been developed further) to compare its performance with existing game engines and simulations engines. Whilst the basis of PhySims architecture shows that it is possible to implement several accuracy ranges of functionality in a single engine. It is yet unknown if its performance would be hindered in comparison to other engines for and major use.

7 References

Atlantic.net Community, 2017. *Object-Oriented Programming: Constructors And Inheritance.* [Online]
Available at: https://www.atlantic.net/community/howto/object-oriented-programming-constructors-and-inheritance/
[Accessed 12 March 2017].

Ellington, H., Percival, F., Race, P. & Thorpe, M., 1988. *Handbook of Education Technology.* Virginia: Kogan Page.

Georges, G. G. et al., 2000. *Apparatuses, methods, computer programming, and propagated signals for modeling motion in computer applications.* United States of America, Patent No. US6714201 B1.

Hsieh, P., 2016. *Programming Optimization.* [Online]
Available at: http://www.azillionmonkeys.com/qed/optimize.html
[Accessed 20 March 2017].

Millington, I., 2010. *Game physics engine development.* 2nd ed. Amsterdam: Morgan Kaufmann Publishers.

Mohd, E. K., 2010. Different Forms of Software Testing Techniques for Finding. *IJCSI International Journal of Computer Science Issues,* 7(3), pp. 11-16.

Moore, M. & Wilhelms, J., 1988. *Collision Detection and Response for Computer Animation.* New York: ACM.

Open GL, 2017. *Open GL Overview.* [Online]
Available at: https://www.opengl.org/about/
[Accessed 20 April 2017].

Open Scene Graph, 2016. *osg::Vec3f Class Reference.* [Online]
Available at:
http://trac.openscenegraph.org/documentation/OpenSceneGraphReferenceDocs/a01007.html#a6cac1a3c1eb0db5ead2f141f71104fde
[Accessed 20 March 2017].

Richards, A. W., Branstad, M. A. & Cherniavsky, J. C., 1982. Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys (CSUR),* 14(2), pp. 159-192.
Schmidt, P. W., 2014. *D'Alembert's principle.* [Online]
Available at: https://www.accessscience.com/content/d-alembert-s-principle/179920
[Accessed 20 04 2017].

Unity, 2016. *Unity3d Manual ParticleSystem.* [Online]
Available at: https://docs.unity3d.com/Manual/class-ParticleSystem.html
[Accessed 15 November 2016].

Unity, 2016. *Unity3d Manual ParticleSystem Curve Editor.* [Online]
Available at:
https://docs.unity3d.com/401/Documentation/Manual/ParticleSystemCurveEditor.html
[Accessed 15 November 2016].

Unity, 2016. *Unity3d Particle System Main Module.* [Online]
Available at: https://docs.unity3d.com/Manual/PartSysMainModule.html
[Accessed 15 November 2016].

Unity, 2016. *Unity3d Public Relations.* [Online]
Available at: https://unity3d.com/public-relations
[Accessed 15 November 2016].

8 Appendix

8.1 Terms Of Reference

8.1.1 Title

Development of a Specified Projectile Motion Library

8.1.2 Course Specific Learning Outcomes

- Implement systems and frameworks that demonstrate use of mathematical tools and techniques used in 3D computer graphics and in game physics.
- Apply my knowledge of mathematical techniques to (to the best of my ability) represent true projectile motion.
- Gain an in-depth knowledge into the use of graphics application interfaces (APIs) such as OpenGL, Open Scene Graph and the C++ Syntax.
- Develop an interactive graphics environment to demonstrate the functionality of the library.
- Understand the strengths and constraints of 3D modeling with Open GL and OSG and explore possibilities to overcome these issues.
- Discover (if any) relevant optimization techniques for run time rendering, real time object analysis. (displacement, velocity etc.)

8.1.3 Project Background

The demand for graphical modeling systems has seen a massive increase over the last few decades. With the advancement in technology and system/hardware architecture, this task is becoming increasingly desired and easier to replicate amongst a variety of devices. These advancements "*has also included the emergence of open standard application program interfaces ("APIs) for rendering, such as OpenGL from Silicon Graphics and more recently Direct3D from Microsoft.*" *(LLC, 2004).* Thus, the technology for creating a GUID application for operating the proposed library is readily available.

The concept of using computers to render intense or complex graphical models is no new feat. However, I have struggled to find a system which is capable of accurately carrying out both realistic and alternative realism in terms of motion. There are many examples of systems and libraries modeling projectile motion, and many graphic API's to represent them such as DirectX, OpenGL/OSC, NXA or more "heavy weight" game engines such as Unreal, Unity, Frostbite, Cry Engine and more.

What is noticeable here, is that larger graphics engines that may have such functionality included may not consider all true physical aspects of the motion. It could be argued that this is since large Graphics engines have a lot to model in a small amount of time, and optimization must be introduced to achieve this. On the other hand, lighter weight Graphic API's do not have a specific catered system to carry out these tasks and it is up to the developer at the time to devise such functionality.

As motion is a vast subject, this project primary goal is to explore projectile motion. However, that is not to say that provided this system is completed and reliable, it may be amended over time to include more physical characteristics.

8.1.4 Aim

To create a library, designed specifically for the rendering and mapping of projectile motion given a specified 2D or 3D plane and custom physical parameters.

This library will be capable of modeling motion simulations within a realistic environment, or used to map kinetics of a fabricated object in a non-realistic game environment.

8.1.5 Objectives

Objectives throughout the development cycle have been split into some primary categories, Analysis, Design, Development, Evaluation. Below is a representation of the key objectives within each of these categories to be carried out.

8.1.5.1 Analysis

- Analyze any pre-existing projectile modelling strategies currently used in the computing industry.
- Identify (if any) comparable limitations between projectile modeling in different platforms.

8.1.5.2 Design

- Decide on what language(s)/Frameworks will be used to develop this system.
- Design a library which improves on any of the weaknesses identified in the evaluation of previous concepts.
- Produces a list of dynamic constraints that can change the running mechanics of the library.
- Plan a simple, configurable GUI to demonstrate the libraries functionality.
- Suggest other possible uses of the library and provide a brief synopsis of their relevance.

8.1.5.3 Development

- Produce a draft structure of the proposed solution, thinking about naming conventions, namespaces and (if plausible) inheritance.
- Decide an input/output structure of displacement and velocity data of the projectile, this should be reusable and possibly recursive.
- Implement basic “SUVAT” functionality to the library.
- Create a basic GUI to run testing on the developing library.
- Further develop the library to include ballistic and rotational characteristics for projectiles traveling at high speeds/distances.
- Further develop the GUI by adding a constraints menu to alter the parameters of the libraries modeling (per projection).

8.1.5.4 Evaluation

- Discuss the realistic feasibility of the produced system.
- Evaluate the possible uses of the library and compare them to the intended areas.
- Highlight the strengths and weaknesses of the library and any potential improvements.
- Compare the final product to the original aim and conclude its effectiveness.

8.1.6 Problems

With modern mathematical calculations. Modeling projectile motion is entirely possible. However, the possibility of discovering new mathematical/physical phenomena may cause the system to grow increasingly inaccurate. This means the library will require continuous maintenance after initial development has been completed.

Newer releases to the selected libraries syntax, may warrant code refactoring with the intention of improving efficiency and optimization.

Some mathematical calculations may become severely complex and have an impact at run time. In the worst-case scenario, this could affect the user's feedback negatively in the form of latency and frame drops.

Issues on mathematical functionality may arise during the development phase due to complexity of modeling. If this happens, the resolutions may have to be to treat certain physical factors as negligible in order to simplify the methodology. If this is not possible, time schedules may become obscured and/or need refactoring to accommodate further mathematical investigation.

Issues may arise when producing the proposed GUI due to the graphic API capabilities. This could be overcome by revisiting the desired specifications and making required adjustments. If the development timeline is in its infant stage, it may be possible to explore options for another GUI framework that could prove more accommodating.

8.1.7 Required Resources

The Microsoft Visual Studio IDE can accommodate most (if not all) of the implementation of this project. It supports multiple languages and allows this inclusion of additional systems such as OpenGL and Open Scene Graph.

In terms of hardware, a good GPU and Fairly quick processor will be recommended. This is because the system will not only be carrying out intense mathematical formulae, but will be interoperating this data, per frame and building a graphical representation of the data.

The PC to be used for this project includes the following specifications.

Processor: i7 6700HQ 2.6GHz
Ram: 16GB DDR4 2000Mhz
Storage: 256GB M.2 PCIe SSD
Graphics: Nvidia GTX 1060 6GB VRAM

This computer possesses more than enough computational power for performing required calculations and rendering them into a visible scene.

8.1.8 Timetable and Deliveries

Below is a list of primary tasks and their respective submission dates. Some of these tasks have been divided into subsidiary tasks, each with their own completion date.

Primary Task	Primary Task Submission date	Sub-Task Submission Date	Sub Task
Terms of Reference submitted	17/10/2016	Date	Perform background analysis on feasibility of project and current alternatives.
		Date	Produce a brief plan to outline the derived system and its dependencies/constraints.
Ethics Form submitted	24/10/2016		Explore any potential for ethical infringement during the development cycle of this system.
Literature Survey submitted	21/11/2016	Date	Analyze current methods for modeling projectile motion both in game and simulative circumstances.
		Date	Outline any proposed improvements to current systems that will be explored and/or included into this system
		Date	Highlight potential difficulties and deficiency's that may be expected in the implementation of this system.
Product Design submitted	12/12/2016	Date	Create a design structure for the projectile motion library, include any necessary entity relationship explanations.
		Date	Design a basic GUI to operate the library, include any diagrams/sketches where needed.
Evaluation Design submitted	30/01/2017	Date	Produce a group of unit tests to operate the library and measure a predicted outcome.
		Date	Have a small number of test users carry out some specific, planned tasks through the GUI collect feedback in the form of set questions. Gather more feedback through conversation of test subjects and translate these findings into usability heuristics.
Report Outline submitted	20/02/2017	Date	Initialize a draft outline for the structure of the system report and evaluate this with other project members.
		Date	Refactor the draft report structure into a final report structure to be used.
Draft Slides submitted	06/03/2017		Produce a presentation to outline the implemented system.
Practice presentation held	13/03/2017		Perform a test run of the compiled presentation.
Report and Product submitted, Presentations held and final slides submitted	24/04/2017		Submit all produced work on the system including final product, plan, design, report and presentation.

8.1.9 References

Grinstein, G.G., Leger, J.R., Lee, J.P., MacPherson, B.E. and Southard, D.A., 3D Open Motion, LLC, 2004. Apparatuses, methods, computer programming, and propagated signals for modeling motion in computer applications. U.S. Patent 6,714,201. https://www.google.com/patents/US6714201

8.2 Ethics Declaration

ETHICS CHECKLIST

This checklist must be completed before commencement of any research project. This includes projects undertaken by staff and by students as part of a UG, PGT or PGR programme. Please attach a Risk Assessment

Please also refer to the University's Academic Ethics Procedures; Standard Operating Procedures and the University's Guidelines on Good Research Practice

Full name and title of applicant:		
University Telephone Number:		
University Email address:		
Status: All staff and students involved in research are strongly encouraged to complete the Research Integrity Training which is available via the Staff and Research Student Moodle areas	Undergraduate Student Postgraduate Student: Taught Postgraduate Student: Research Staff	☑ ☐ ☐ ☐
Department/School/Other Unit:	School of Computing, Maths & Digital Technology	
Programme of study (if applicable):	Computer Game Technology	
Name of DoS/Supervisor/Line manager:	Silvester Czanner	
Project Title:	Development of a specified Projectile motion library	
Start & End date (cannot be retrospective):	19/09/2016 - 28/04/2017	
Number of participants (if applicable):		
Funding Source:		

Brief description of research project activities (300 words max):

This project aims to explore the current standards for modeling and rendering projectile motion both in simulation and augmented reality, game play scenarios.
By identifying any constraints in current systems. A new library will be designed and implemented to address projectile motion on a more specific scale from some systems. This library will include dynamic constraint manipulation in order to render both simulation standard and augmented reality rendering of projectile motion. Finally a basic GUI will be developed to demonstrate this libraries functionality.

	YES	NO
Does the project involve NHS patients or resources? If 'yes' please note that your project may need NHS National Research Ethics Service (NRES) approval. Be aware that research carried out in a NHS trust also requires governance approval. Click here to find out if your research requires NRES approval Click here to visit the National Research Ethics Service website To find out more about Governance Approval in the NHS click here	☐	☑
Does the project require NRES approval?	☐	☑
If yes, has approval been granted by NRES? Attach copy of letter of approval. Approval cannot be granted without a copy of the letter.	☐	☐

Click here to read how participants need to be warned of pain or mild discomfort resulting from the study and what do about it.		
14. Will the study involve prolonged or repetitive testing or does it include a physical intervention?	☐	☑
Click here to discover what constitutes a physical intervention and here to read how any prolonged or repetitive testing needs to managed for participant wellbeing and safety		
15. Will participants to take part in the study without their knowledge and informed consent? If yes, please include a justification.	☐	☑
Click here to read about situations where research may be carried out without informed consent		
16. Will financial inducements (other than reasonable expenses and compensation for time) be offered to participants?	☐	☑
Click here to read guidance on payment for participants		
17. Is there an existing relationship between the researcher(s) and the participant(s) that needs to be considered? For instance, a lecturer researching his/her students, or a manager interviewing her/his staff?	☐	☑
Click here to read guidance on how existing power relationships need to be dealt with in research procedures		
18. Have you undertaken Risk Assessments for each of the procedures that you are undertaking?	☐	☑
19. Is any of the research activity taking place outside of the UK?	☐	☑
20. Does your research fit into any of the following security sensitive categories: • commissioned by the military • commissioned under an EU security call • involve the acquisition of security clearances • concerns terrorist or extreme groups If Yes, please complete a Security Sensitive Information Form	☐	☑

I understand that if granted, this approval will apply to the current project protocol and timeframe stated. If there are any changes I will be required to review the ethical consideration(s) and this will include completion of a 'Request for Amendment' form.

☐ have attached a Risk Assessment
☐ have attached an Insurance Checklist

If the applicant has answered **YES** to **ANY** of the questions **5a – 17** then they must complete the MMU Application f[redacted]

Signature of Applicant [redacted] Date: 25/10/2016 (DD/MM/YY)

Independent Approval for the above project is (please check the appropriate box):

Granted

☑ I confirm that there are no ethical issues requiring further consideration and the project can commence.

Not Granted

☐ I confirm that there are ethical issues requiring further consideration and will refer the project protocol to the Faculty Research Group Officer.

Signature: Silvester Czanner Date: 25/10/2016 (DD/MM/YY)

Print Name: Silvester Czanner Position: Supervisor

Approver: Independent Scrutiniser for UG and PG Taught/ PGRs RD1 Scrutiniser/ Faculty Head of Ethics for staff.