BSc (Hons) Computer Games Technology

Implementing the GJK Distance Algorithm for 3D Collision Detection

Craig McCorrisken

B00317267

9/04/2020

Supervisor: Dr Marco Gilardi

Declaration

This dissertation is submitted in partial fulfilment of the requirements for the degree of BSc (Hons) Computer Games Technology in the University of the West of Scotland.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

Name: CRAIG MCCORRISKEN

Signature:

Date: 9/04/2020

Form to Accompany Dissertation

To be completed in full

Surname:	McCorrisken
First Name:	Craig
Initials:	CM
Borrower ID Number:	B00317267
Course Code:	COMPSCI
Course Description:	BSc (Hons) Computer Games Technology
Supervisor:	Dr Marco Gilardi
Dissertation Title:	Implementing the GJK Distance Algorithm for 3D Collision Detection
Session:	2019-2020

COMPUTING HONOURS PROJECT SPECIFICATION FORM
(Electronic copy available on Moodle Computing Hons Project Site)

Project Title: Implementing the GJK Algorithm for 3D Collision Detection

Student: Craig McCorrisken **Banner ID:** B00317267

Supervisor: Marco Gilardi

Moderator: Paul Keir

Outline of Project: *(a few brief paragraphs)*
This project will involve researching, designing and implementing a piece of software that contains the Gilbert-Johnson-Keerthi (GJK) distance algorithm. This software will use this algorithm to test for collisions between 2 or more colliding 3D objects. The main focus of this project will be researching, planning and implementing the GJK algorithm and getting it to a working stage to test for 3D collisions between complex polytopes. Alongside the collision detection, this project will also contain basic collision reactions to accompany the detections. This project will compare the GJK algorithm with another 3D collision algorithm and quantitative research will be used to gather and compare the data from the two algorithms.

The proposed Hons Project should include practical work of some sort using computing technology / IT

A Passable Project will:
- Research the GJK distance algorithm.
- Design and develop a prototype project which uses the GJK algorithm to test for collisions between 2 3D objects.
- Contain colliding objects which are all the same shape.

A First-Class Project will:
- Contain the GJK distance algorithm working between multiple different types of 3D objects (box, ball, pyramid, etc.)
- Contain basic physics for the collision responses.
- Compare GJK performance against another 3D collision detection algorithm.

Reading List:

Real-Time Collison Detection – Christer Ericson
Game Physics Engine Development – Ian Millington
Game Programming Patterns – Robert Nystrom
A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space – E. G. Gilbert, Fellow, IEEE, D.W. Johnson & S. S. Keerthi
Implementing the GJK – Casey Muratori
Resources Required: *(hardware/software/other)*

Software:	**Hardware:**	**Source Control:**	**Other:**
IDE – Visual Studio 2019	Custom build laptop	GitHub	One Drive
3rd Party Libraries - GLM, SLD2, RT3D	Custom build home computer	Git GUI	Google Drive
Microsoft Office – Word, PowerPoint	Razor Deathadder Chroma Gaming Mouse		
	Corsair Gaming k70 Mechanical Keyboard		
	1TB Seagate Hard-Drive		

Marking Scheme:

	Marks
Introduction	10
Literature Review	15
Design	20
Implementation	25
Results and discussion	15
Conclusion	10
Critical Self-Appraisal	5

Signed:

Student **Supervisor** **Moderator** **Programme Leader**

IMPORTANT:

(i) By signing this form all signatories are confirming that the proposed Hons Project will include the student undertaking practical work of some sort using computing technology / IT, most frequently achieved by the creation of an artefact as the focus for covering all or part of an implementation life-cycle.

(ii) By signing this form all signatories are confirming that any potential ethical issues have been considered and if human participants are involved in the proposed Hons Project then ethical approval will be sought through approved mechanisms of the School of CEPS Ethics Committee.

Maths Symbols

$Sf(d)$	Support Function in a given direction d
d	Direction
$S[\cdot]$	Simplex
P	Supporting Point
C	Convex Set
H	Convex Hull
$\boldsymbol{x}$	A point in Euclidean Space

Contents

Acknowledgements

Firstly, I would like to thank my supervisor for this project Dr. Marco Gilardi for his continued support and feedback throughout this year. Not only did he push me to choose a project which I viewed as out of my abilities, but he also passed on support and knowledge, which was valuable to completing the project. Without someone on board with his experience and understanding of this field, the project would not be what it is today. Dr. Gilardi always pushed for my best and ensured academic levels of work throughout, which helped my work reach a level of professionalism it has not before reached.

I would also like to give thanks to my moderator Dr. Paul Keir whose support proved invaluable. His feedback helped shape the project and documentation to what it is today. From small pointers for writing academically to advising on improvements and optimisations throughout the software, Dr. Keir's advice ensured the project was moving in the right direction.

Summary

This report discusses the techniques involved in the creation of the Gilbert-Johnson-Keerthi (GJK) distance algorithm. This report explores the fundamentals behind the mathematics of the GJK algorithm before following this up with how these principles are used within this algorithm. The literature surrounding different collision detection algorithms was studied to gain a better understanding of what collision detection is as well as, to gain an insight into how this was previously researched and implemented throughout past years. Alongside collision detection algorithms, methods for improving these algorithms are explored. The design and implementation of software which uses the GJK algorithm are then presented. This report concludes by analysing the implementation of the GJK.

The software produced is a primary system for showing collisions between objects using the GJK algorithm. Objects of differing convex shapes, comprised of a collision volume, a mesh and a movement class are tested for collision within this technical demo. A reaction takes place if the two objects test positive for a collision in terms of a change in texture. This new texture helps the user identify which objects have encountered a collision and which objects have not yet. The user controls a shape which they can move around the screen in all three axes and use this box to collide with all other shapes which are in the environment. The user can change shape to any of the four possible shapes implemented within the game at any desired time. Each time the user collides with an object, a collision detection test takes place in the form of the GJK algorithm. The user also controls a camera. This camera allows the user to move around the environment independently to the shape they control. This allows the user to view collisions between objects in more detail and explore the scene.

1 Introduction

Collision detection sometimes referred to as intersection detection, is one of the most important aspects of modern, state-of-the-art video games (Ericson, 2004). Collision detection allows interaction between objects in an environment and maintains the sense of a non-hollow world (Ericson, 2004). Since two objects within the real world cannot occupy the same point in space at the same time, it makes sense for the same to be true for two objects within a virtual environment when trying to immerse a user (van den Bergen, 2003). This is where collision detection comes into play, collision detection in video games stops two objects from intersecting and enforces the illusion of solidity within these objects. Without intersection detection, video games such as Shigeru Miyamoto's Super Mario Bro's could never have existed as colliding with objects within the game stands as the basis for the entire game. Jumping on platforms, enemies and collecting power-ups all involve Mario colliding with objects in some form.

Collision detection is the process of testing if two or more objects within a scene have intersected each other. An intersection is defined as at least two objects containing the same point in space (van den Bergen, 2003). Particularly, collision detection establishes three factors: if, when and where objects are colliding. The algorithm finds out if a collision has occurred, at what frame it occurred and at what location in space it occurred (Ericson, 2004).

Since interactive software such as video games contain user-defined movement, collision detection cannot be precalculated. This is due to the user's movement being unpredictable, meaning collision detection systems must work in real-time to handle this unpredictability. The collision detection must be a generic system, one which takes the objects types instead of each objects instance as this would be far too expensive to compute in real-time. A reliable collision detection system should be able to take any two objects within an environment as it is input and output whether they have intersected.

One way in which collision detection systems improve their timing is by testing for collisions between basic shapes which are placed on top of meshes. This speeds-up collision testing as fewer vertices are being tested for than the possible thousands contained with the mesh of an object (Vries, n.d.). One of the most common volumes used for collision detection algorithms is the *convex hull*, a tightly fitting shape that fully encloses the object's mesh. One such algorithm for collision testing which uses these convex hulls and is widely renowned for its accuracy and efficiency was proposed back in 1988 by Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi—known as the Gilbert-Johnson-Keerthi algorithm or GJK for short. The algorithm itself tests for the shortest Euclidean distance between two convex hulls (Gilbert et al., 1988) which makes it perfect for performing collision testing. Since one of the most intuitive methods for checking how close two objects are to each other is to calculate the Euclidean distance between them, i.e., the length of the shortest line segment which joins the two objects then the GJK works as an intuitive algorithm for collision testing (Gilbert et al., 1988).

The purpose of this report is to research collision detection as a field of study and more specifically learn about the Gilbert-Johnson-Keerthi (1988) algorithm itself. Following on from this research, a technical demo was developed which houses an implementation of the GJK algorithm used to detect collisions in a 3D environment. This demo allows user input to control different 3D shapes and move these shapes freely around a scene. The project contains multiple different shapes, each at different locations, in which the user can interact. The user can collide the shape they control with the other shapes in the environment, allowing the user to see how the collisions occur in real-time.

1.1 Source Code

The source code for this project includes original code based around the maths explored throughout this report alongside sample code which helped push the project forward. All sample code which was used contains comments giving citation of source. After cloning the project from GitHub, run as an x86 project. All libraries required are contained within the project.

The source code for this project can be retrieved from the following GitHub repository:

https://github.com/CraigMcCorrisken/Honours-Project

The following link accesses the development folder which contains all the third-party libraries used:

https://github.com/CraigMcCorrisken/Development-Folder

A backup for this code can be found from the following link:

One Drive

2 Background Information

The understanding of the GJK algorithm requires an understanding of the mathematics which underpins it. This chapter explores the meaning behind mathematical concepts and functions which are used throughout the remainder of the report and serves as the foundation for understanding the mathematics underpinning the GJK algorithm.

2.1 Euclidean Space

Euclidean Space is a vector space which can be denoted by $\mathbb{R}^n$. If the Cartesian product of ordered n-tuples of real numbers is used, then the notation for a point $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{n}}$ would be $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, where$(x_1, x_2, \ldots, x_n)$ are the coordinates of $\boldsymbol{x}$ with respect to the Cartesian reference frame.

In this dissertation $\mathbb{R}^n$ represents the n-dimensional Euclidean space denotes the dimensions for the coordinate system used, n represents the number of vectors that compose a base for the vector space (a.k.a. the dimension of the space). $\mathbb{R}^n$ also represents the set of tuples on the field or real numbers.

> ***Example 2.1 –*** Some examples of Euclidean spaces using different dimensions are:
>
> For *n=1,* $\mathbb{R}^1$ represents a 1-dimensional space (line). Points in this space will have a single coordinate, which is indicated with $\boldsymbol{x} = (x_1)$.
>
> For *n=2,* $\mathbb{R}^2$ represents a 2-dimensional space (plane). Points in this space will have two coordinates, indicated with $\boldsymbol{x} = (x_1, x_2)$.
>
> For *n=3,* $\mathbb{R}^3$ represents a 3-dimensional space. Points in this space will have three coordinates, indicated by $\boldsymbol{x} = (x_1, x_2, x_3)$.

2.2 Convex Set

A convex set is a subsection of Euclidean Space in which a straight line, drawn between any two points within the set, lies entirely within the set. Another way to understand this would be to say; if a straight line is drawn between any two points within a shape and this line leaves the shape at any point, then it is not convex. Otherwise, the shape is convex. See, Figure 2.2.1 and Figure 2.2.2 for examples of convex and non-convex shapes, respectively.

> ***Definition 2.2*** *– A subsection of space C is a convex set if for $p, q \in C$ the line segment $\overline{pq}$ is completely contained within C (Berg et al., 1997).*

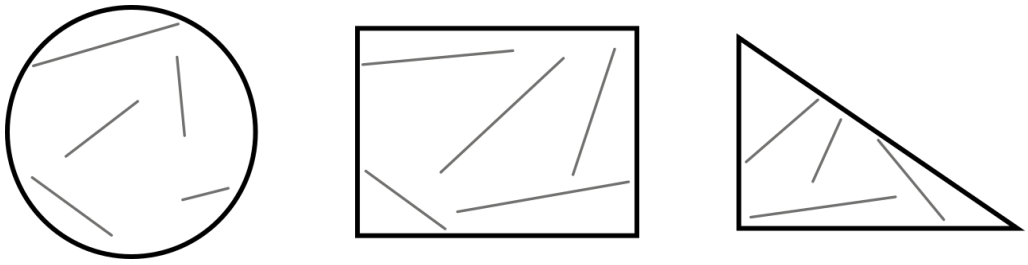

Figure 2.2.1 – Examples of Convex Sets.

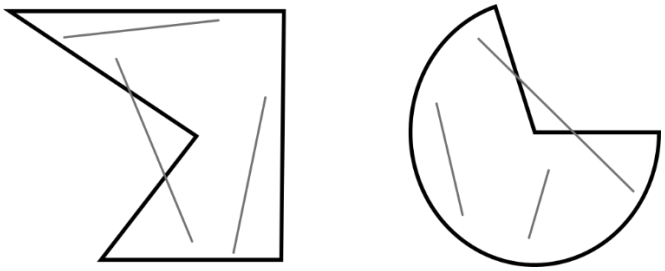

Figure 2.2.2 – Examples of non-convex sets.

2.3 Convex Hull

The convex hull of a set of points is the tightest possible wrap around them which contains all the points within the wrap (van den Bergen, 2003). A convex hull H is easy to imagine as a set of thumbtacks on a board, H would be the result of wrapping a string around the outside of those thumbtacks (Wang, n.d.). See Figure 2.3.1 for an example of a convex hull of a set of points in 2D space.

> ***Definition 2.3*** *For a group of points C, the convex hull H is the tightest possible convex shape that contains every point within C.*

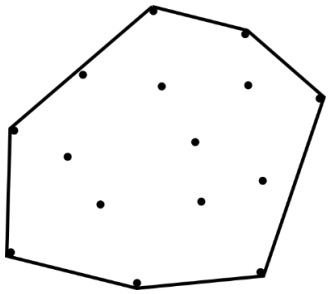

Figure 2.3.1 – A convex hull of a set of points.

Convex hulls are desirable as bounding volumes for collision testing as they are relatively cheap to compute, fit tightly around a more complex object and they make collision testing easier due to their simple shape (Ericson, 2004).

2.4 Simplex

A *simplex* is a list built up from a collection of vertices. Each additional vertex builds the simplex up from a 0-simplex point to a 3-simplex tetrahedron (Ericson, 2004). The simplex gets its name as it represents the *simplest* possible polytope for any given dimension (Weisstein, n.d.). The size of the simplex is determined by how many dimensions there are. For example; 2D space would only require a triangle/ 2-simplex; whereas in 3D space a tetrahedron/ 3-simplex, would be necessary. See Figure 2.4.1 for a list of these dimensionally based simplices. Throughout this report, $S[\,]$ will be used to represent the simplex list.

> ***Definition 2.4*** *- Where d denotes the number of dimensions of a simplex, a d-simplex is the convex hull H of its d + 1 vertices (Ericson, 2004).*

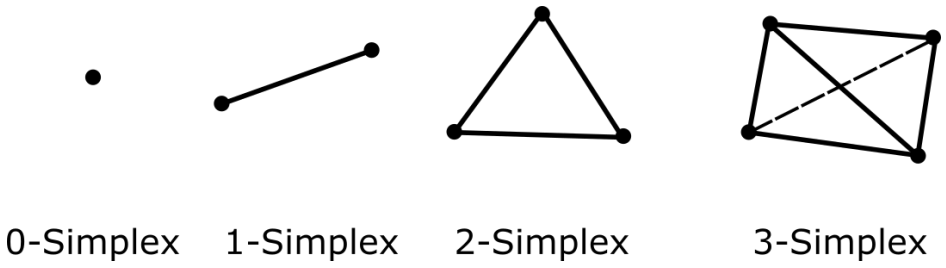

Figure 2.4.1 – Simplices (Image From: Ericson, 2004).

2.5 Polyhedron

A polyhedron is the three-dimensional version of a polygon (Ericson, 2004). The outer edge of a polyhedron is comprised of several flat, polygonal faces connected along their edges so that each edge is made up of exactly two polygonal faces (Ericson, 2004). Polyhedrons also contain vertices, which are points along the boundary where two or more edges meet. Figure 2.5.1 shows the elements which build up a polyhedron.

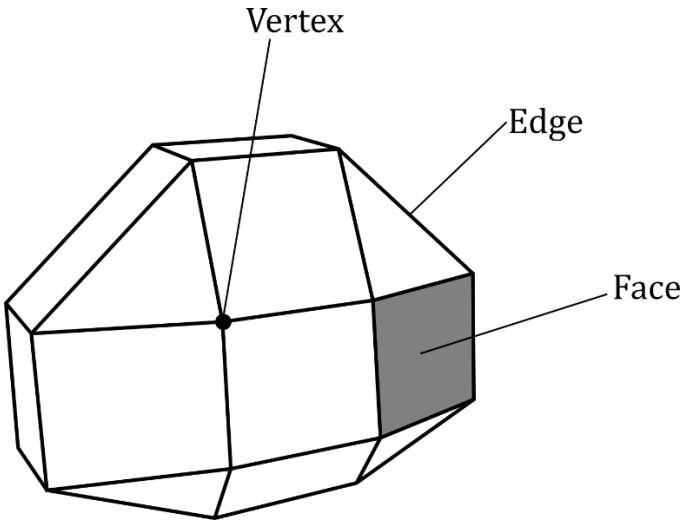

Figure 2.5.1 – Example of a convex polyhedron (Image From: Ericson, 2004).

Definition 2.5 – *A polyhedron is a three-dimensional shape built up by flat, polygonal faces.*

2.5.1 Polytope

A polytope is a generalisation of a polyhedron in any dimension (Weisstein, n.d.). It is a geometric object with flat sides. A convex polytope is the convex hull of a finite set of points (van den Bergen, 2003). A polytope is defined by its number of dimensions. For example, a 2D polytope is known as a polygon, and a 3D polytope is known as a polyhedron.

> ***Definition 2.5.1*** *- A polytope P is the convex hull H of finitely many points in* $\mathbb{R}^d$ *(Wang, n.d).*

2.6 The Minkowski Sum

The Minkowski sum is a basic, but important algebraic sum which allows for the addition of geometric shape. The Minkowski Sum is the summation of all the points which make up one shape with all the points which make up another shape (dyn4j, 2010).

Definition 2.6 *– Given two sets of points A and B, the Minkowski sum, denoted as* $A \oplus B$*, is defined as* $A \oplus B := \{a \oplus b : a \in A, b \in B\}$ *(Ericson, 2004).*

The Minkowski sum is a useful method for producing swept volumes from two generalised shapes (van den Bergen, 2003). This sum creates a resultant, which is a new shape made up of the two original shapes added together. The resulting shape would be like "sweeping" the first shape around the second shapes perimeter (Muratori, 2016). See Figure 2.6.1. The summation of two convex hulls results in a convex hull (van den Bergen, 2003).

Example 2.6 *– if* $Shape\ A \oplus Shape\ B = A_i \oplus B_j$*, where* A_i *and* B_j *are points of the corresponding shapes* A *and* B *then the Minkowski Sum would be* $AB_{ij} = A_i \oplus B_j$*.*

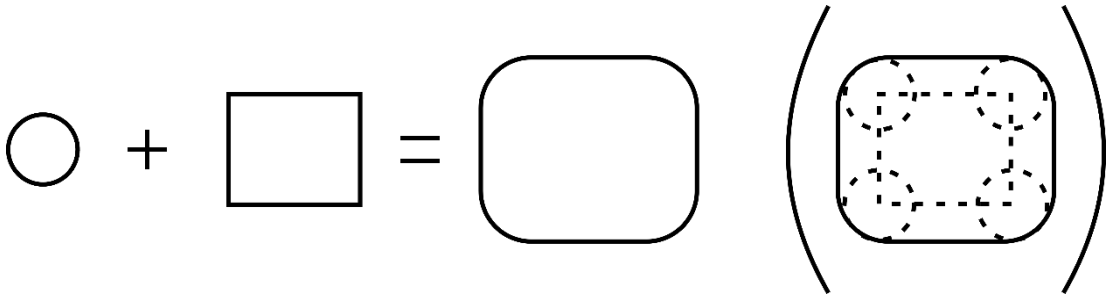

Figure 2.6.1 – Resultant of Minkowski Sum obtained by sweeping a sphere around the perimeter of a box (Image From: Muratori, 2006).

2.7 Support Function

The support function $Sc(d)$ is a function linked to a convex set C that maps the direction d into a supporting point P of C (Ericson, 2004). The support function is a function that finds the supporting point P, this being the farthest possible point on a shape in a given direction d. Figure 2.7.1 shows the supporting point on two separate shapes when the support function is given the same direction for both shapes.

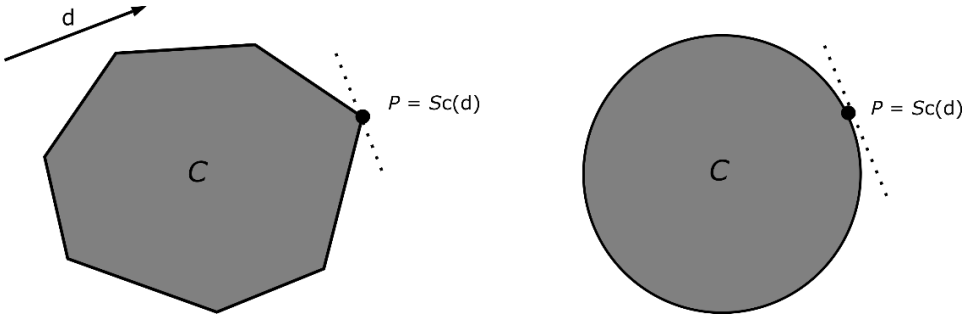

Figure 2.7.1 – Supporting Point (Image From: Ericson, 2004).

Definition 2.7 – *A point* $P \in C$ *is the supporting point of C for the direction* $\boldsymbol{d}$ *if* $P = S_C(\boldsymbol{d})$*.*

2.8 Conclusion

All mathematical concepts discussed above build the foundations for what will become the Gilbert-Johnson-Keerthi algorithm. Each concept is used within the algorithm to achieve the goal of detecting collisions between two objects. However, before delving into how this mathematics is used within the GJK algorithm, an understanding of collision detection systems in general and how they have previously been discussed is required.

3 Literature Review

This chapter focuses on the literature surrounding collision detection and how it has been studied in the past. Examples of different approaches to collision detection algorithms are explored alongside methods for improving these algorithms. The literature, along with the background, build up foundational knowledge for collision detection and understanding the Gilbert-Johnson-Keerthi algorithm.

3.1 Collision Detection

Collision detection is one of the most important aspects of interactive software (Klosowski et al., 1998). Even as computers become increasingly more capable of handling many thousands of calculations at once, collision detection and handling multiple collisions at once remains a crucial challenge for developers to balance alongside all other aspects of game development (Ericson, 2004). When video games first emerged, collision detection within these games began as a 2D problem, involving testing for an overlap on only two axes (x, y). However, as time has moved on and 3D games have taken over the market, the problem of collision detection has only increased in complexity (Ericson, 2004) as objects now contain possibly millions of vertices that need to be checked for collision.

To hold the illusion of objects being solid, computer games and interactive software must make use of some version of a collision detection algorithm, sometimes referred to as an intersection handling algorithm (Hubbard, 1997). Collision detection algorithms are usually broken down into two phases; the *detection* phase where the algorithm checks if a collision has taken place and the *response* phase where the algorithm reacts accordingly after a collision has been detected (Hubbard, 1997). Usually, the detection algorithm takes both static and dynamic objects which make up the environment as inputs (Klosowski et al., 1998). The aim of the detection algorithm is then to decide whether an intersection has already happened or is about to happen between these objects by repeatedly checking for contact (Gottschalk et al., 1998). The response phase aims to react accordingly if a detection has occurred and effect the colliding objects, for example, this could be to stop the player moving through a solid wall or allow a ball to bounce off the ground. For a collision detection algorithm to work accurately to represent solidity within objects, the algorithm must meet two key principles: it must be able to work in real-time without breaking the illusion of a solid world, and it must be able to handle the users input, as the motion of the player-controlled objects are unpredictable, and therefore collisions involving these objects cannot be pre-calculated (Hubbard, 1997).

3.2 Simplifying Collision Detection

Collision detection can be computationally expensive. This can cause the frame rate to slow down if the algorithms for detecting collisions are not optimised and certain approaches are not taken to make sure these algorithms can work at full potential. This section discusses methods that exist for optimising and improving collision detection systems.

3.2.1 Bounding Volumes

One way to improve collision detection algorithms as well as speed up the software is to reduce the number of vertices on the shapes which are being checked during the detection phase of the algorithm. To do this, a simplified version of the geometric shape is used for testing instead of the mesh itself. This is done by creating a *bounding volume*. (Ericson, 2004). This cuts the number of tests that are performed to logarithms of the number of objects (Choi & Sung, 2017). Wrapping an object in a bounding volume and testing for collisions between these volumes can noticeably improve the performance of the tests (Choi & Sung, 2017). Testing two simple polygons against each other for collision detection usually results in an $O(n^2)$ complexity where n is equal to the number of objects.

However, the overall workload can be reduced by up to 3/4ths of the number of polygons being tested was to be halved by placing the bounding volumes in a *spatial data structure* (Ericson, 2004).

3.2.2 Spatial Data Structures

Collision detection is a problem-solving exercise within computing where the developer must balance accuracy over computational speed (Vassilakopoulos, 2016). One way of improving the speed of the algorithm is to implement a *spatial data structure* (SDS) (Ericson, 2004). An SDS is a *divide et conquer* approach for diving up the data via a specific attribute. Spatial data structure's help with computational speeds as it allows fewer data to be checked at any given cycle. Instead of checking all the data for collision detection, also known as a brute force method, the system could, for example; check which objects are within a given range and only check if those objects are colliding (Vassilakopoulos, 2016). Spatial data structures allow unnecessary data to be ignored and in turn, stops redundant checks or computations. One such SDS which can be used is the *Region-Based QuadTree* which is a spatial data structure in which each section of 2D space has four child nodes, or in the case of 3D space, eight child nodes, while the root of the quadtree represents the whole region of space (Vassilakopoulos, 2016).

3.2.3 Region-Based Quadtrees and Octrees

As mentioned above, the region based quadtree (RBQT) takes the screen space and divides it down into smaller, more easily manageable subsections, each subsection being the parent node divided evenly into four separate blocks of space for 2D, known as *quadrants*. The RBQT is a data structure that stores information which can be accessed through a key, an example of a key could be the number assigned to the quadrant (Finkel, 1974). The region-based quadtree continues to break down the screen into smaller 2x2 sections until some stopping condition(s) are met (Samet, 2006). An example of a stopping condition would be if a pre-determined depth for the tree were reached (Ericson, 2004). Each quadrant within the tree holds a smaller number of objects that its parent node, separating the stored data up into smaller chunks. This removes checks for unnecessary collision testing as two objects which are in different quadrants cannot be colliding and therefore do not have to be checked for collision. Removing these checks saves time and removes redundant computations. Similarly, a quadtree exists which stores data for three-dimensional space known as the *Octree*. The octree has many similar features, however where the quadtree divides the spaces down recursively into four smaller sections each time, the octree divides the space into eight sections, known as *octants*, to account for the extra dimension (Ericson, 2004).

3.2.4 Broad and Narrow Phases

One method for reducing the number of collision detection calculations and in turn, speed up the detections is to split the collision handling into two separate phases, these being the *broad phase* and the *narrow phase* (Ericson, 2004). The broad phase, sometimes called *n-body processing,* registers the objects which are likely to be colliding and removes those which are not colliding. In contrast, the narrow phase, sometimes known as *pair processing,* takes the objects which could be colliding and tests each pair for intersection (Ericson, 2004). Without the implementation of a broad and narrow phase, there would be no reduction in the number of objects being tested which would occur in every object within a scene being tested against every other object within the scene. This would result in a calculation of $(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = O(n^2)$ pairwise tests where n is equal to the number of objects (Ericson, 2004).

3.3 Accuracy

Bounding volumes help reduce the number of redundant intersection tests by allowing for an "early-out" approach which attempts to exit the test early if no intersection is possible (Ericson, 2004).

However, this comes at a trade-off between how tight the bounding volume fits around the shape and the cost of the intersection tests (Ericson, 2004). Wider or less form-fitting volumes allow for quicker tests, whereas tighter volumes allow for more accurate collisions.

When talking about accuracy between volumes, it is best to start with the lower end of the accuracy scale and work towards more form-fitting volumes. Spheres are a good starting point as they are one of the easiest to compute and have the attribute of not needing to be rotated, only moved location (Ericson, 2004). The main drawback of the sphere is that unless the object is spherical in some form, the volume will have much empty space which could cause redundant collision detection. A middle ground volume that could be used would be the *object-oriented bounding box* (OOBB). This bounding volume is a box that is drawn around the mesh and oriented along a given axis (Ericson, 2004). However, the most important bounding volume when discussing the GJK algorithm would be the tightest fitting bounding volume which is the convex hull. A convex hull fits tightly around the object without allowing too much empty space, this makes the convex hull the best-suited bounding volume for accurate collision testing and allowing for an "early-out" approach. Figure 3.3.1 shows examples of the five most common bounding volumes. Most collision detection algorithms use convex hulls for their bounding volumes due to the many properties which they have, such as the existence of a separating plane. This does not mean non-convex shapes cannot be used within games, as this is fixed by dividing these non-convex shapes down into multiple sub-volumes comprised of convex hulls (Ericson, 2004).

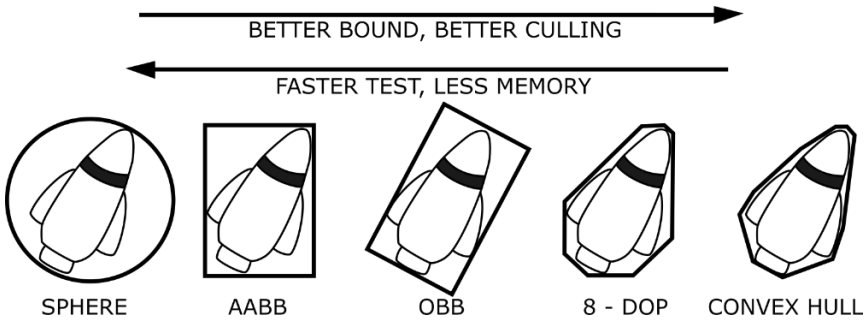

Figure 3.3.1 - Types of bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB), eight-direction discrete orientation polytope (8-DOP), and convex hull (Image From: Ericson, 2004).

3.3.1 Bounding Volume Hierarchies

One method for improving the collision testing speeds even further would be to introduce a *bounding volume hierarchy* (BVH) into a project. A BVH is a tree that contains all the bounding volumes in a scene. (Ericson, 2004). The root node of the tree would be the scene itself, the tree then branches downwards, and the larger bounding volumes are broken down into smaller volumes until each volume contains only one object from the scene or until a stopping condition has been met (Eberly, 2010). The leaf nodes of the tree contain the single meshes/ objects themselves (Ericson, 2004), the parent nodes of the tree contain the volumes which are recursively broken down into smaller volumes from the root to the leaves (Eberly, 2010). By introducing a bounding volume hierarchy into a project, the time complexity can be condensed down to the logarithmic in the number of tests that are being performed (Ericson, 2004). Figure 3.3.2 shows an example of a BVH for five separate shapes.

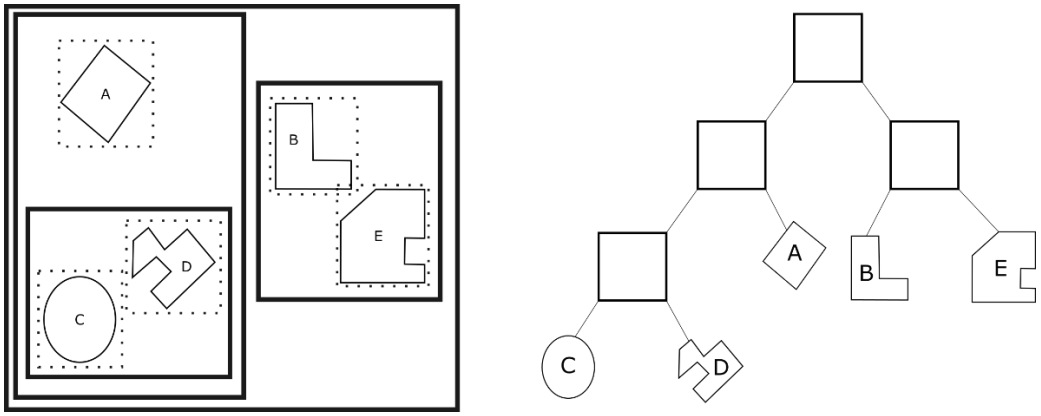

Figure 3.3.2 - A bounding volume hierarchy of five simple objects. Here the bounding volumes used are AABBs (Image From: Ericson, 2004).

3.4 Sweep and Prune

The *sweep and prune algorithm*, sometimes referred to as the *sort and sweep algorithm,* projects a 3D box onto each axis of the coordinate system, for 3D space (x, y, z). This would result in one projection for each axis, which creates *intervals* along each axis (Cohen et al., 1995). An interval being the start of the projection on that axis until the end of the projection, defined by $[b, e]$ where b represents the beginning of the interval and e represents the end. The sweep and prune algorithm states that two bounding boxes are only intersecting if their intervals overlap on all three axes (Cohen et al., 1995). Storing these intervals in a linked list allows for updates as the dynamic objects move around the scene (Ericson, 2004). Tracking an interval is possible by adding an active interval to a linked list whenever their b value is encountered and eliminating them when their e value is encountered (Ericson, 2004). Each axis is represented by a single linked list to hold the separate intervals. Sorting these linked lists shows which intervals overlap (Cohen et al., 1995). Sorting each list would cost $O(n\ log\ n)$ time, where n is the number of objects. However, it is possible to save time if the objects are not making large leaps of movement by carrying over the sorted list to the next frame and updating only the endpoints of each interval, this changes the complexity from $O(n\ log\ n)$ to $O(n)$ time (Cohen et al., 1995). See Figure 3.4.1 for examples of objects and their intervals projects onto an axis.

> ***Theorem 3.4 (Sweep and Prune) –*** *A pair of axis-aligned bounding boxes A and B can intersect if and only if their projections along each coordinate axis overlap on all three dimensions. The projections of an AABB on the coordinate axis are called* intervals*. (Cohen et al., 1995).*

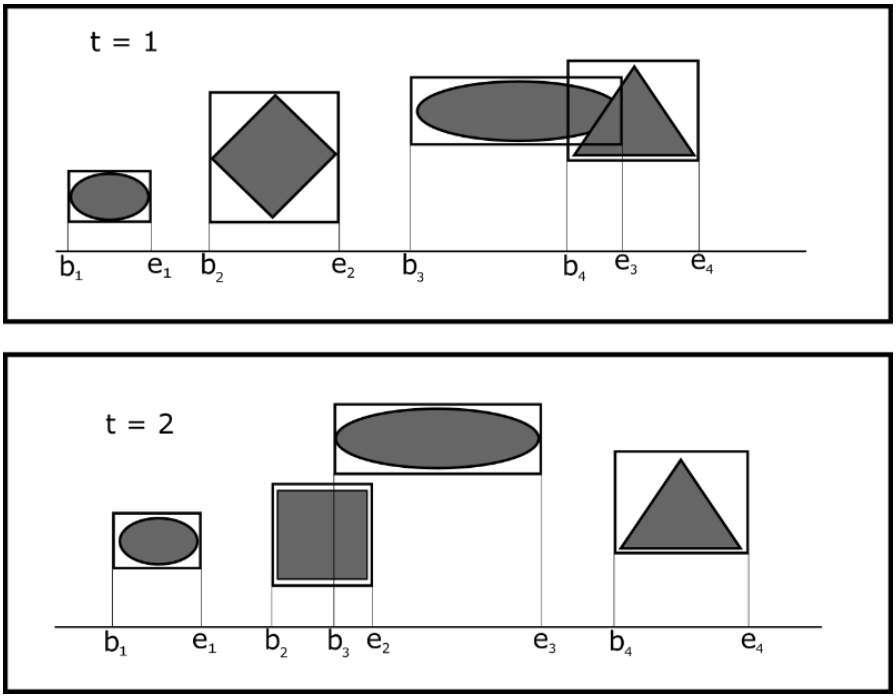

Figure 3.4.1 – Sweep and Prune at Two Separate Times (Image From: Cohen et al., 1995).

3.5 Separating Axis Theorem

The *Separating Axis Theorem* (SAT) states that collision between two objects does not exist if a line exists on which the projections of the object do not overlap. There also exists a line known as the *separating line*, which lies perpendicular to the separating axis and divides the two objects (Gottschalk, 1997). The two objects cannot be intersecting if a gap exists between their projections on the axis (Ericson, 2004).

The separating axis theorem, like many algorithms for collision detection, only works with convex shapes. However, it is possible to represent non-convex within smaller, groups on convex shapes, theoretically allowing the algorithm to work on non-convex shapes (Dyn4j, 2010). Figure 3.5.1 shows two 2D shapes and their projections.

> ***Theorem 3.5 (Separating Axis Theorem)*** *Two convex objects cannot be colliding if there exists a direction L, called the separating axis, along which the projections of the two objects do not overlap.*

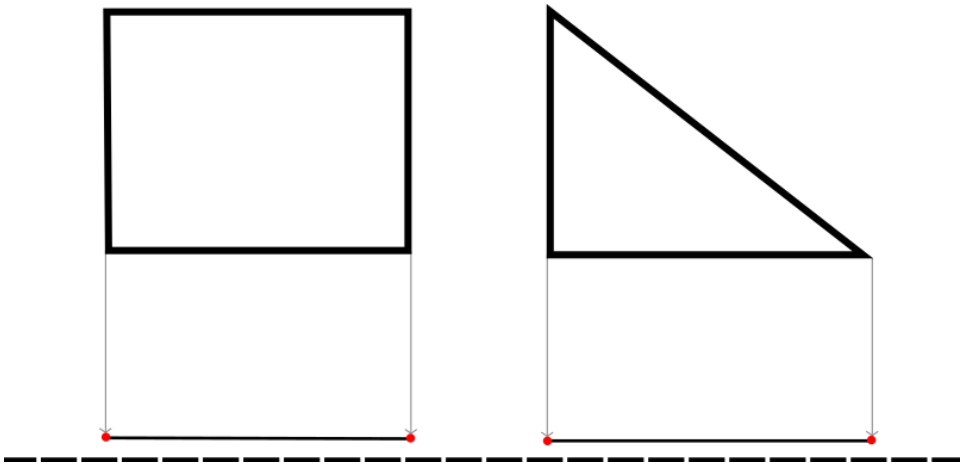

Figure 3.5.1 – Projecting Two Objects onto the Separating Axis, No Intersection.

One of the aspects of this theorem which makes it a great option for game development is the early exit strategy in which it implements. Due to the algorithm only requiring one gap between any of the projections, the algorithm can exit as soon as the first gap appears between two objects. This early exit strategy improves the calculation speeds and makes it a great option for implementation within a game engine since in most cases objects within a gaming environment are not colliding.

The SAT algorithm has three possible returns values. There can be no collision, where two objects are too far apart, and no points overlap. A collision between two edges or faces where the objects are touching against each other and no more or full collision, where two objects are colliding at two or more points across their convex hulls (Ericson, 2004).

The number of axes in 2D which are being tested against is equal to the number of edges on each shape. The axes themselves which the shapes are projected onto are the *normals* of the edges along each shape hulls, where the normals are defined as the perpendicular to the face of the object (dyn4j, n.d.). For 3D shapes the faces which comprise each shape are also added to how many axes are required as collision can occur along a 3D shape's edges and faces.

3.6 Discussion

Collision detection is a complex topic with many different algorithms existing all with the same basic purpose of testing objects within a system for intersection. However, some algorithms such as the GJK which is implemented within this project only detect collision and return a true or false. Whereas other algorithms can be used to find more specific information about the intersection, such as the SAT which can be used to detect how far two objects are intersecting. A basic system which returns true or false is great for a fast and reliable detection test in which the application does not require more information. However, an application which uses the range of intersection when performing a reaction would not get much use out of such a basic return value, and would instead require a system more like the SAT or an enhanced version of the GJK which returns how far the two objects are intersecting.

An example of an application which would require intersection data would be a mesh which bends when pushed against. The application would require the point where the mesh is intersected with and the distance in which the mesh is being pushed against to represent this change on the mesh appropriately. An application like this could not work if the intersection distance was not recorded during intersection.

Collision detection systems are built to be fast and not slow down the system which utilises them as to remain undetected by the systems user. However, if a system contains many objects which all require collision tests then the system eventually slows down to a halt if a brute force method is used, as discussed previously in this chapter. This means collision detection systems are not enough on their own if a system is complex and contains many objects, so different optimisation methods are required such as using collision volumes to test smaller numbers of vertices and spatial data structures to break down the number of collisions occurring at once. A robust system should take advantage of all the points discussed within this chapter and use them to create a smooth system which goes unnoticed by the user.

3.7 Conclusion

The information discussed in this chapter builds a foundation for understanding the topic of collision detection within video games. From collision detection systems to bounding volumes and optimisations, each aspect has been given a brief overview. Now that the fundamentals are in place, the GJK algorithm itself can be explored in greater detail.

4 Gilbert-Johnson-Keerthi Algorithm

This chapter will explore, in detail, the Gilbert-Johnson-Keerthi (GJK) algorithm. Specifically, this chapter will discuss how the mathematics from chapter two relates to the algorithm by exploring how each aspect is used within the algorithm itself. A brief example of the algorithm is shown and discussed.

4.1 Introduction

The Gilbert-Johnson-Keerthi algorithm is an advanced mathematical algorithm, created in 1988 by Elmer Gilbert, Daniel Johnson, and Sathiya Keerthi, which finds the minimum distance between two convex sets (Gilbert et al., 1988). This algorithm was originally developed as a method for finding the Euclidean distance between two objects which are defined by their mathematical models in 3D space, this Euclidean distance being the shortest possible line segment which separates the two objects (Gilbert et al., 1988). Finding the shortest distance between two objects allows the algorithm to be used for multiple different things; from collision detection across 2D or 3D space to pathfinding around obstacles (Gilbert et al., 1988). However, this algorithm is best used for collision detection as it is one of the most effective methods for determining collision between two polyhedra (Ericson, 2004). The GJK algorithm works by using a Simplex, $S[\cdot]$. Taking two sets of vertices and finding the Euclidean distance that separates them (Ericson, 2004). Instead of working directly on the two sets of vertices, the GJK is based on the The Minkowski Sum of two objects. The GJK takes the shape which is created using the Minkowski sum and iteratively builds a simplex inside this shape, this simplex attempt to enclose the origin, which results in a positive collision detection between the two shapes which created the Minkowski sum (Ericson, 2004).

4.2 The Mathematics of the GJK

This chapter relates to the mathematics which was previously discussed within Chapter 2 and how it correlates with and is used throughout the GJK algorithm.

4.2.1 The Minkowski Difference

The GJK algorithm relies heavily on the Minkowski sum (dyn4j, 2010). It is a basic mathematic equation that forms the theoretical basis for the overall algorithm. This being the distance which separates two polyhedrons corresponds to the distance between their Minkowski difference and the origin (Ericson, 2004).

When discussing the Minkowski sum in chapter 2.7, the addition of two shapes was used. However, as previously mentioned, for the GJK algorithm, we are only interested in the results from the difference, not the addition of two polyhedrons. Therefore, the GJK uses what is sometimes referred to as the *Minkowski Difference* (dyn4j, 2010).

> ***Definition 4.2.1*** *- The Minkowski difference, denoted as* $AB_{ij} = A_i \ominus B_j$*, is defined as* $A \ominus B := \{a \ominus b : a \in A, b \in B\}$ *(Ericson, 2004).*

The Minkowski difference is important for the GJK algorithm because if the difference between two shapes creates a shape which encloses the origin (0,0), then the two shapes must contain points which overlap. For example, if two shapes share one or more of the same points in Euclidean space, then the two shapes must be colliding. This means that the Minkowski difference of two convex hulls is equivalent to finding the difference between the AB_{ij} and the origin (Ericson, 2004). See Figure 4.2.1 for two examples of the Minkowski difference, the first showing no intersection between the two objects and the later showing intersection.

Example 4.2.1 *– Let A and B be two 2D shapes represented by* $A\big((2,3),(6,8)\big)$ *and* $B((1,4),(6,8))$. *The Minkowski difference between A and B is then* $A \ominus B = ((1,-1),(0,0))$,*since* $A \ominus B$ *contains* $(0,0)$ *the two shapes intersect.*

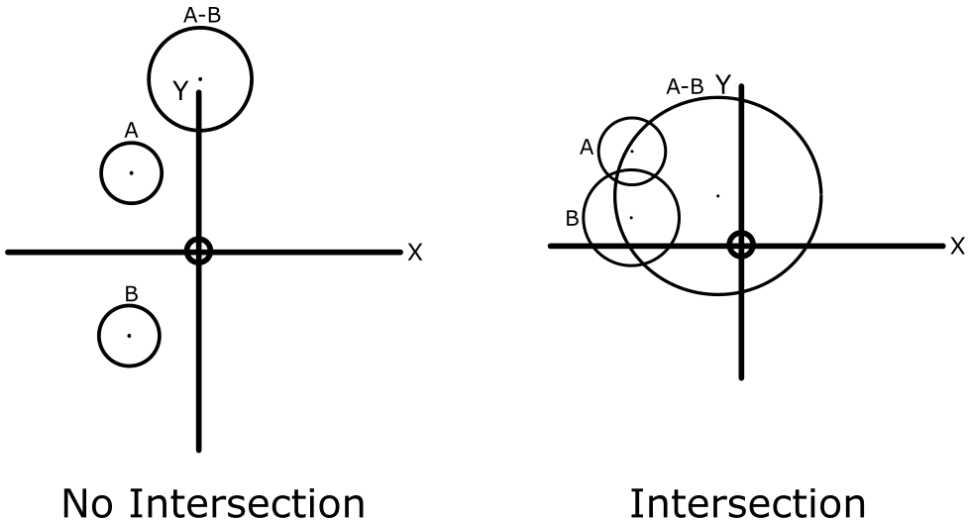

Figure 4.2.1 - Minkowski Sum Shown Geometrically (Image From: Muratori, 2006).

One way in which the Minkowski difference helps with reducing calculations is down to the fact that it is only interested in the points which lay on the convex hull of the shape, allowing all points which are contained inside the shape to be ignored.

Since the GJK algorithm works on the Minkowski difference between two objects and not the objects themselves, the problem of intersection testing is condensed down to simply finding the distance between the origin and a single convex set (Ericson, 2004).

A key point to note is that the Minkowski difference is implicit within the GJK algorithm; this is to say it stands as the theory as to why it works and is never explicitly calculated.

4.2.2 Simplex

As previously mentioned in chapter 2.4, the simplex $S[\cdot]$ is a collection of vertices. This collection of vertices is built up to create a polygon inside the Minkowski difference of two convex hulls. If this simplex contains the origin $(0,0)$, then the Minkowski difference must also contain the origin $(0,0)$, which means the algorithm has detected an intersection between the two convex hulls (dyn4j, 2010).

The number of vertices which are used to build a simplex depends on the number of dimensions within that space, for example, a 2D space would only require a 2-simplex which creates a triangle, however, in 3D space, a 3-simplex tetrahedron would be required as a simple triangle would not be enough to attempt to enclose the origin.

4.2.3 Support Function

The support function is the method that builds the simplex for the GJK algorithm, meaning the GJK relies heavily on this function to work. The support function finds the farthest point in any given direction known as the supporting point P which lies somewhere on the convex hull of the Minkowski difference. If the supporting function is given a different direction each time, then it will return a different value for P each time. This value is added to the simplex to build it from a point to a tetrahedron (dyn4j, 2010).

The advantage of using P is that choosing the farthest points along a distance allows for a large area to be covered. This large area increases the chance of exiting early (dyn4j, 2010). A method for selecting the largest area would be done by choosing an arbitrary direction and using this to find the first P by passing this direction into the supporting function. After the first point is found, the second direction which is passed into the supporting function would be in the exact opposite direction, by multiplying the original direction by a minus (dyn4j, 2010). This method creates a segment in which the two end vertices are as far apart on the convex hull as they can be. Next, a third P is found, and a 2-simplex triangle is formed, for 2D this would be enough to check for collision, however, this project was developed for 3D space which would require a tetrahedron to be created by adding a 4th P to the simplex. If this simplex contains the origin, then the collision detection would return true.

4.3 Pseudocode

This section will display the pseudocode for the Gilbert-Johnson-Keerthi algorithm to explain how the algorithm works within a coding environment. A step by step description is also shown to demonstrate the algorithm further.

Pseudocode:

```
Set search direction to random direction

Set simplex to 0

WHILE origin not found

        Get the first supporting point by running supportFunction(S[·],  search  direction)

        add this supporting point to the simplex

        SET search direction to - search direction

        GET the second supporting point by running supportFunction (S[·], - search direction)

        IF simplex contains the origin

                Intersection found

                Exit loop

        END IF

        ELSE

                Update the search direction

                Get a new supporting point by running supportFunction(S[·], search  direction)

        END ELSE

END WHILE
```

The following stepwise method talks through the generalised GJK algorithm and describes the start to finish of attempting to find a collision between two objects. Figure 4.3.1 shows how the simplex is built up to reach the origin within the Minkowski difference of two shapes which are not colliding.

Stepwise method:

1. Call the GJK algorithm and pass in two convex hulls.
2. Chose an arbitrary search direction.
3. Pass the direction and two hulls into a support function to return the supporting point.
4. Add this supporting point to the simplex. Simplex now contains one point.
5. Flip the direction to the negative of the current search direction
6. Pass this new direction and two hulls into the support function to return the supporting point.
7. Add this new supporting point to the simplex. Simplex now contains two points, a line.
8. Check if this new point allows the shape being built within the simplex to cross the origin.
9. If it cannot cross the origin, then collision is not possible, early exit.
10. Otherwise, begin loop
11. Find the new direction by finding the cross product of, for example, ABxAOxAB.
12. Pass this direction and two hulls into a support function to return the supporting point.
13. Add this new supporting point to the simplex. Simplex now contains three points, a triangle.
14. If the triangle contains the origin, then continue
15. Otherwise, early exit
16. Repeat from step 11 until a 4-point tetrahedron is built containing the origin.

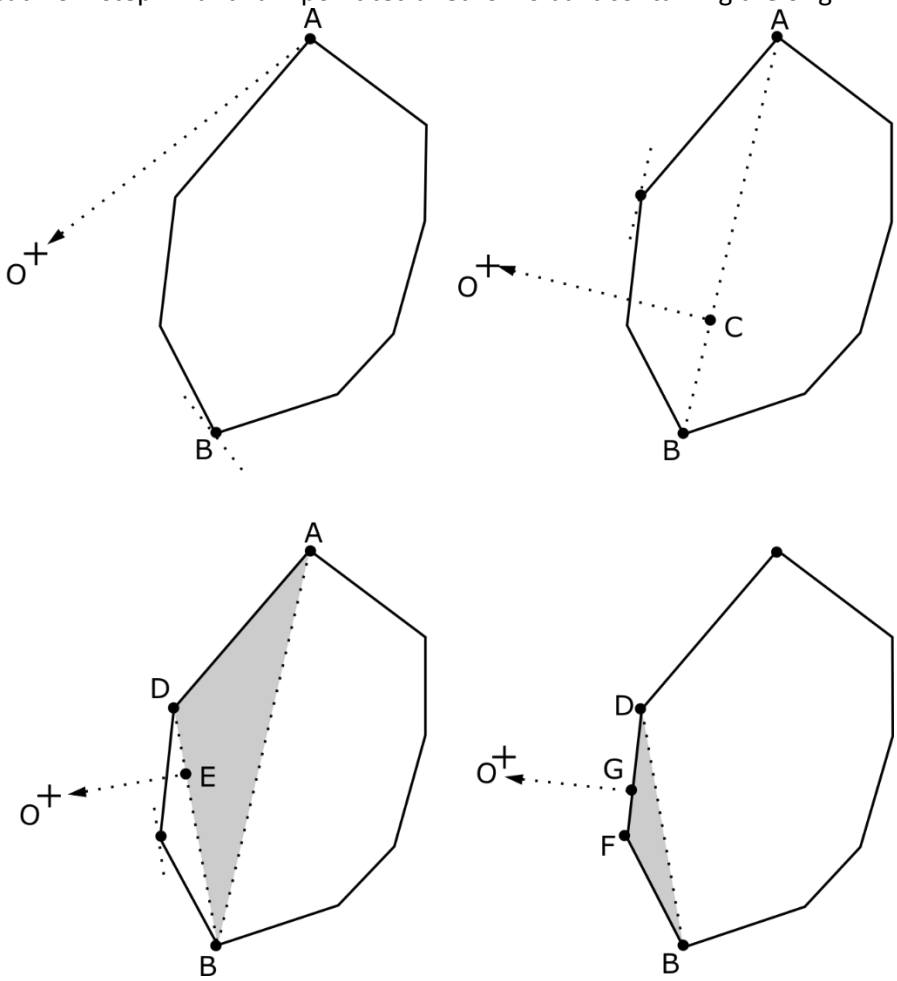

Figure 4.3.1 – GJK Finding the Point on A Polygon Closest to The Origin (Image from: Ericson, 2004).

4.4 Conclusion

Throughout this chapter, the mathematics, functions and information required for understanding how the GJK works are detailed. Now that the fundamentals for collision detection systems and more importantly, the GJK algorithm is in place, a look at how the system which implements these aspects was designed is shown in the following chapter.

5 Methodology and Design

This chapter examines the development plan, requirements and software design for this project. The user controls are presented along with graphical asset and audio lists.

5.1 Software Development Life Cycle

The most suitable Software Development Life Cycle (SDLC) for this type of project was one which allowed for changes and adaptations. Routine meetings and short development cycles also had to be taken into consideration. Considering these factors, the Agile method was chosen, which allowed the overall project to be broken down into smaller, easier to achieve, sections known as sprints (Team Linchpin, 2019). The project supervisor was kept in the loop throughout the full process and assessed progress along the way. Short two-week sprints were tracked through commits to an online GitHub repository.

5.2 Controls

Since the project is an interactive demo, controls for this project were established, which allows the user to move around the scene and interact with the objects within the scene. Table 5.2.1 shows these controls. Having control of the camera allows the user to move closer to colliding objects and view in greater detail the collisions.

Table 5.2.1 - Controls

Number	Key	Reaction
1.	‘W’	Move Camera forward
2.	‘A’	Move Camera left
3.	‘S’	Move Camera backwards
4.	‘D’	Move Camera right
5.	Mouse	Rotate/Pan Camera
6.	‘UP’	Moves Cube up on the Y-axis
7.	‘DOWN’	Moves Cube down on the Y-axis
8.	‘LEFT’	Moves Cube left on the X-axis
9.	‘RIGHT’	Moves Cube right on the X-axis
10.	‘Q’	Moves Cube forward on the Z-axis
11.	‘E’	Moves Cube backwards on the Z-axis
12.	‘1’	Updates player object to Cube
13.	‘2’	Updates player object to Sphere
14.	‘3’	Updates player object to Cone
15.	‘4’	Updates player object to Cylinder

5.3 Graphical Asset List

This project contains different 3D objects to show how the collision algorithm works with different shaped bounding volumes. The objects used are a sphere, box, pyramid, and a hexagon. Table 5.3.1 shows these assets along with the file format.

Table 5.3.1 - Table of Assets

Number	Asset Filename	Format
1.	“Sphere”	.obj
2.	“Cube”	.obj
3.	“Cylinder”	.obj
4.	“Cone”	.obj

5.4 Audio

Audio is implemented within this project to give another level of feedback for the user. An audio file is played when the user presses any of the number keys which are linked with changing the users shape. For example; if the player presses “1” on their keyboard a sound file is played to indicate that the system has detected this input. Table 5.4.1 shows this audio in table.

Table 5.4.1 - Audio Asset List

Number	Extension Name	Extension Format
1.	“click”	“.wav”

5.5 Software Design

The aim of this software is to show collision detection in action. The design of this software aims to make the collision detections the forefront of the project. This means the project could not include any aspects which could distract from the collisions or reactions. The interactive demo mirrors this design plan by only containing objects which the user can interact with via collisions. No object within the demo can be moved/phased through without a detection taking place followed by a response.

This software was designed for ease of use and portability into future projects. The GJK takes two point-clouds stored as vector containers as the inputs and outputs a boolean value to determine whether a collision has occurred between the two hulls. This means that any system which stores convex hulls as vector containers can implement this software's implementation of the GJK algorithm.

5.5.1 Game

The *Game* class within this project contains the calls to the GJK class and implements the previously designed system which uses collision detection testing. The *Game* class includes the engine side of the project, such as the *window* class which houses the SDL window creation code, the *textureLoader* class which uses the SDL Image library to load textures into the project and the *PlaySound* class which uses the Bass sound library to load and handle the audio within the project. As well as this the game also hosts the four shapes which are used for collision testing, the *GJK* class which is used to test collisions and the *player* class which is used to control the camera and move around the scene. The *Game* class itself contains the code for creating the shapes, adding them into the spatial data structure and calling the GJK class for collisions.

The collision detection code contains an abstract *collisionDetector* class which allows the project to grow and contain more collision detection algorithms in the future. The current child of this class is the *GJK* collision detection algorithm. The *GJK* class contains one supporting class; the *SupportFunction* class, which returns the supporting point for building the simplex. The class diagram for this can be seen in Figure 5.5.1.

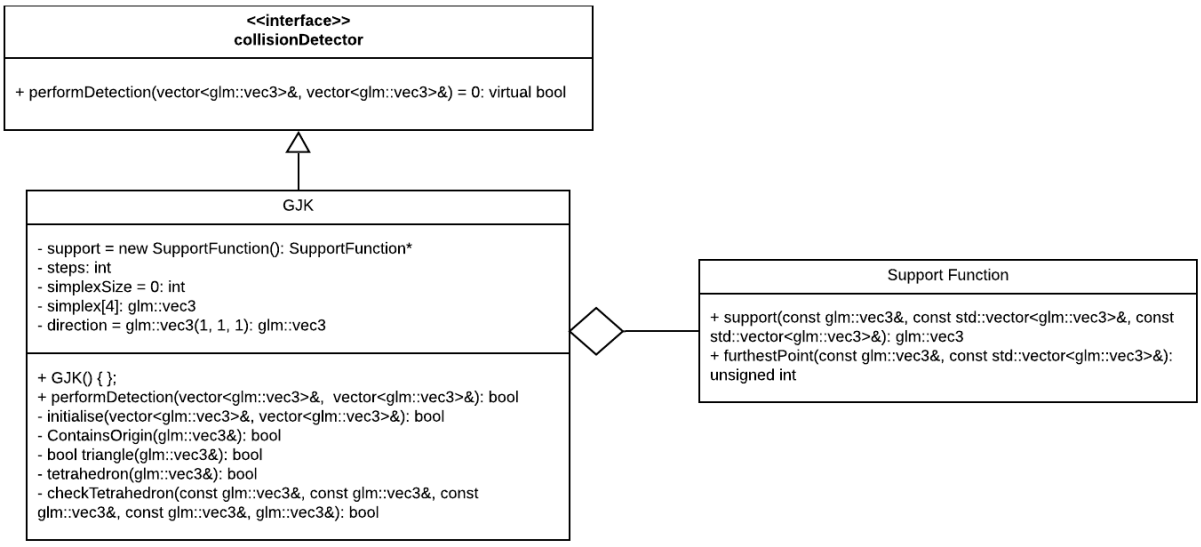

Figure 5.5.1 - Collision Detector Hierarchy (Figure 8.2.4)

To create a window for the project to be rendered onto, the SDL2 library was used. The code for loading and creating a window is held within the *window* class. This class takes the width, height and title for the window as inputs and creates a window using the SLD2 library. This class is a child of the abstract *abstractWindow* class. Figure 5.5.2 shows this class diagram.

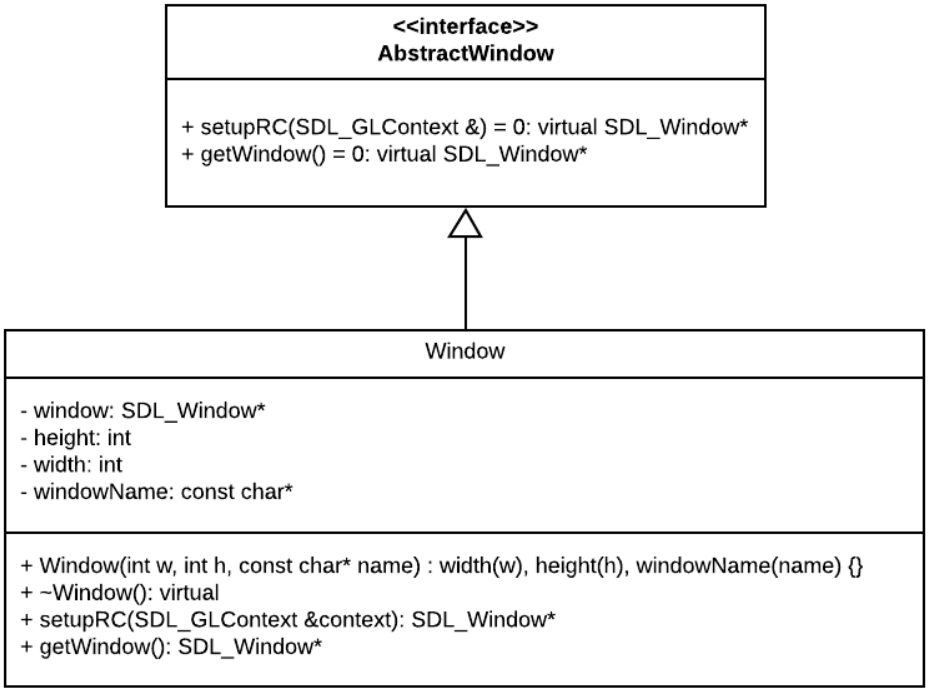

Figure 5.5.2 – Window Class (Figure 8.2.2)

The player object within the game is a child of the *entity* class. The player contains controls for the camera and therefore contains an association of the *camera* class. The player can change the shape

which they control. This requires the *player* class to contain the four shapes. The class diagram for the player can be seen in Figure 5.5.3.

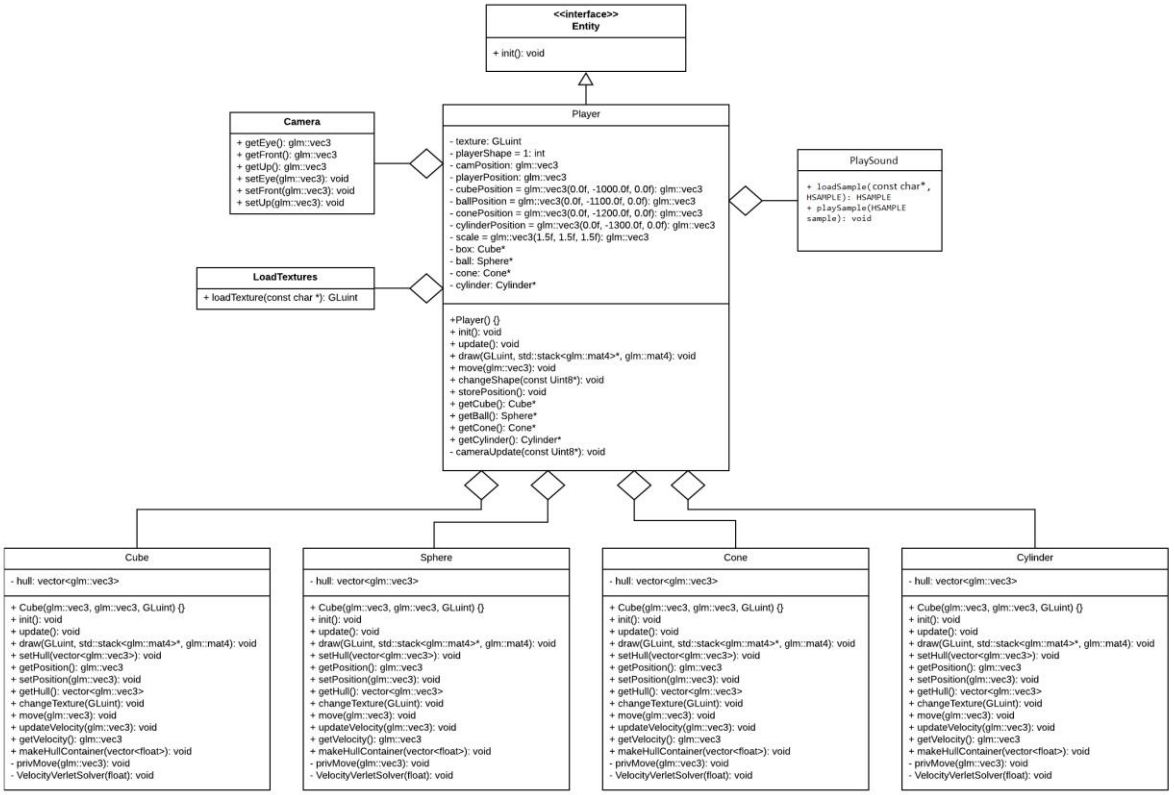

Figure 5.5.3 - Player (Figure 8.2.1)

To load a model into the project, the *RT3D* library is used. The *RT3D object loader* takes the RT3D library, which includes the *md2Model* class and uses the RT3D code to load a model from file. The *mesh* class then takes this code and calls it whenever a model is created within the project. Figure 5.5.4 shows the class diagram for the RT3D code.

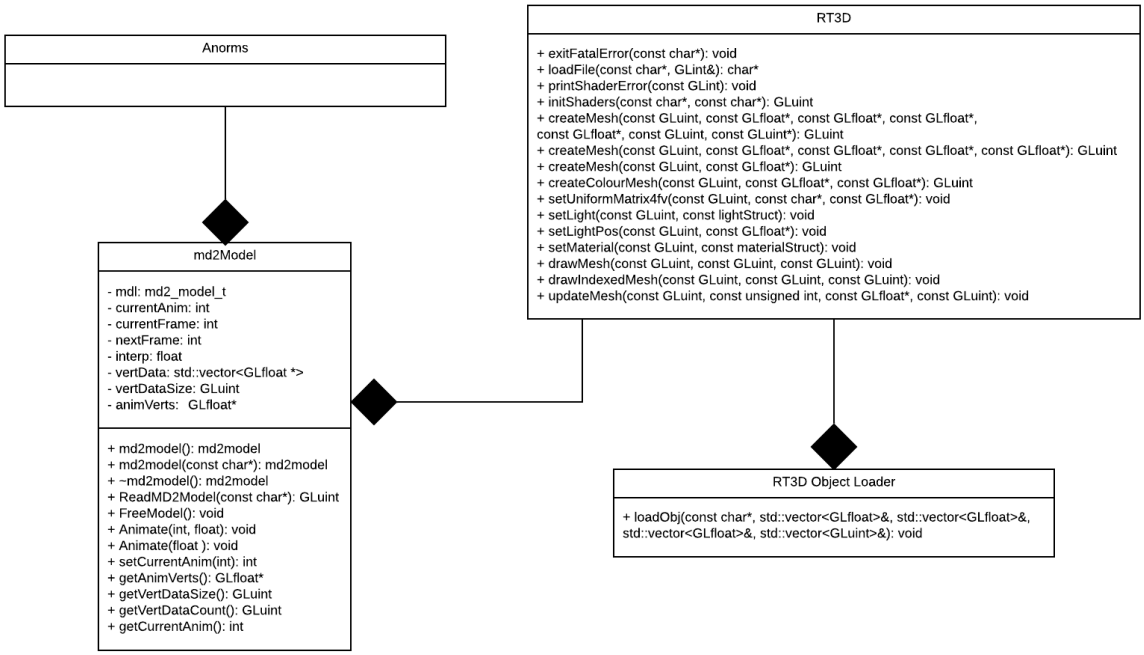

Figure 5.5.4 - Model Loading Using RT3D (Figure 8.2.3)

5.5.2 Architecture

The architecture for this application is based around reusability and ease of implementation. The main *Game* class is used to control the system while all the required classes are “plugged” into their respective areas. For example, since the game requires collision detection, the GJK class is added to the game class and since the game also requires textures, the *loadTextures* class is added in as well. Each class contains its own implementation which allows it to be easily added into the project with little to no additional work. Figure 5.5.5 shows the architecture for this project.

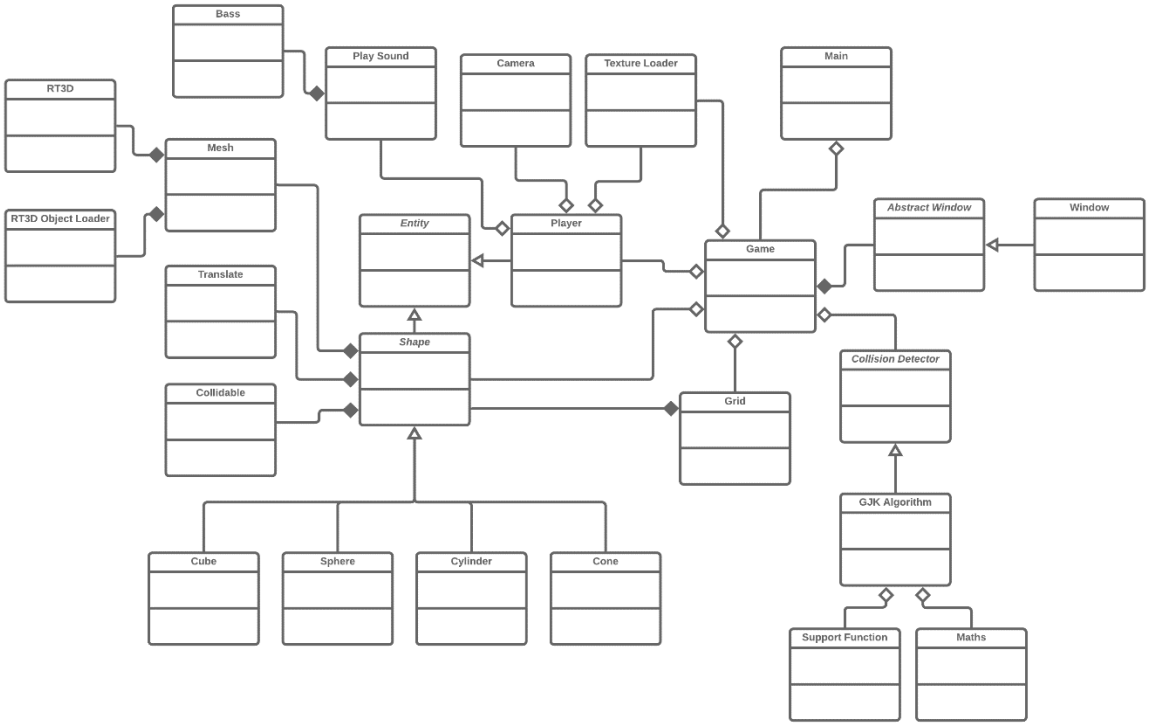

Figure 5.5.5 - Software Layout (Figure 8.2.6)

5.5.3 Object Creation

As mentioned above, the project was designed with the idea of "plugging" components together to create something new. This is best shown when creating a new *Shape*. Every object within the game is of type *Entity*, this is an interface which forces every child object to contain an initialise function. *Shape* is one child of the *Entity* interface. *Shape* takes inputs, a *transform* class which moves, rotates and scales using a model matrix, a *mesh* class which loads and draws a model from file and a *collidable* class which creates a convex hull which is used for collision detection. This allows every child of the *Shape* class to contain and use these classes. Using this system allows for objects to grow in functionality without growing in complexity as each class handles its own implementation, and only minor changes would be required within each shape class to implement a new class. For example, a *move* class could be plugged into the *shape* class giving all children of *shape* access to this new class's functionality. Figure 5.5.6 shows this.

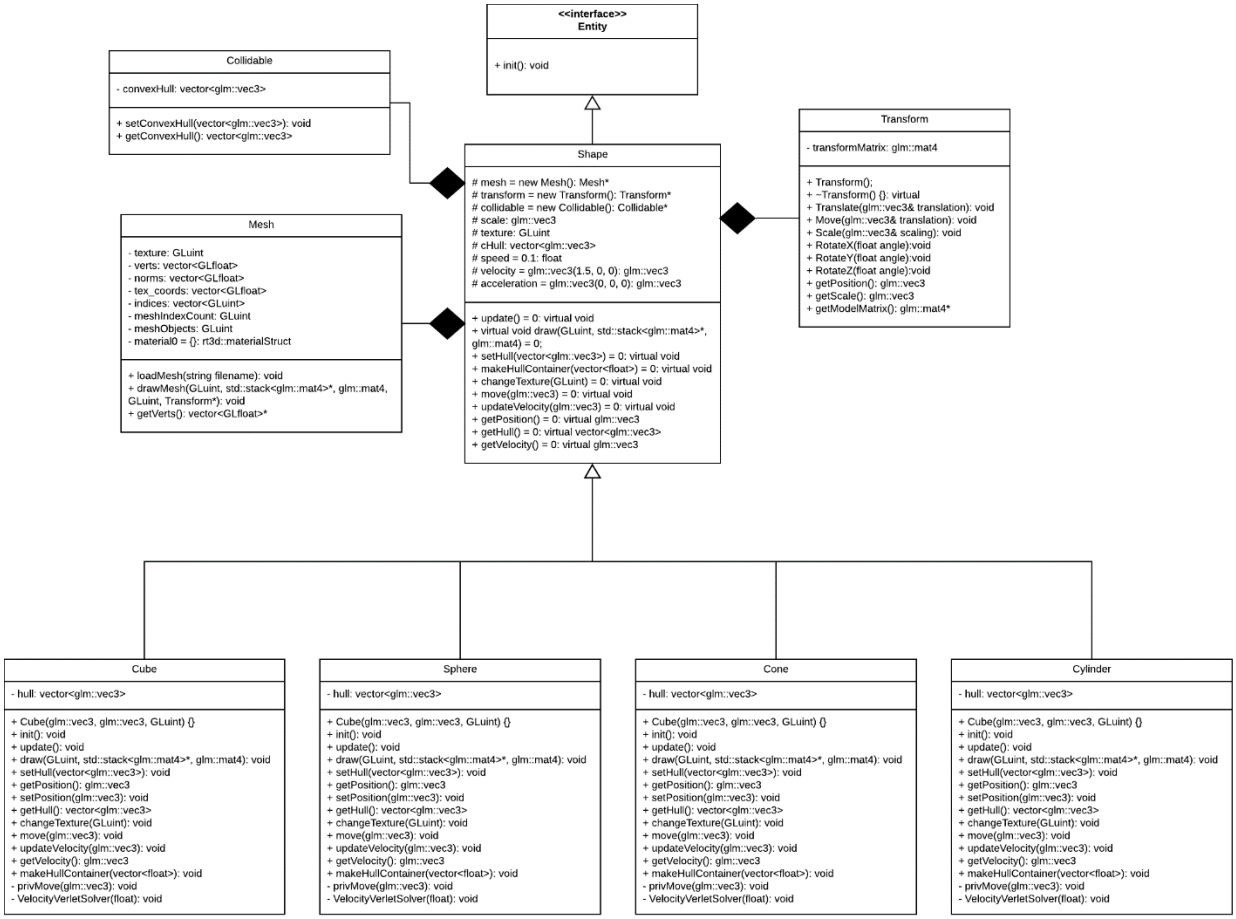

Figure 5.5.6 - Entity Hierarchy (Figure 8.2.5)

5.6 Optimisation

To improve the efficiency of the system, the GJK and support function classes take the address of data as parameters instead of the data itself. Passing in the memory address by using the '&' symbol improves the efficiency as the system is not passing all the data across to another class and all the data back again if required. Only the memory addressed is passed through improving the speed of the calculations.

As well as using the GJK for the narrow phase, a broad phase was implemented in the form of a spatial data structure. This spatial data structure is stored with the *grid* class and takes the width and height of the screen as inputs, along with the size of the quadrants. The *grid* class then uses this to divide the screen into quadrants to contain the elements on the screen. When a shape is created within the *game* class, it is pushed to the container of shapes called gameEntities, it is also added to the grid and its location stored for later use.

During the update, before the objects are tested for collisions, the objects are first tested to make sure they are in the same quadrant. If the two objects are within the same quadrant, then they can be tested for collisions, otherwise, this test is not required and skipped.

5.7 Technical Overview

The following will layout the software which was used to develop this project as well as the minimum requirements necessary to run the project.

5.7.1 Development Tools

This project was developed using the C++ programming language. The project was developed using Microsoft's Visual Studio 2019 Edition IDE. Local repositories were used for development and implementation, while GitHub was used for storing a cloud-based master edition of the code within an online repository. GitHub was used extensively throughout development for version control. While Git GUI was used for pushing commits to and pulling versions of the code from the online, master edition of the code. Backups of the code were made throughout development and stored on Microsoft's OneDrive.

This project used GLM for the maths library as building one from scratch appeared unnecessary. For creating windows and handling user input, SDL2 version 2.0.10 was used. When it comes to handling images, the SDL Image library was implemented and used within a texture handler class. For loading in models and 3D assets, the RT3D library was sufficient in performing this task. OpenGL is implemented within this project for rendering the 3D models and textures to the window. The sound files within this game are handled using the BASS sound library.

5.7.2 Constraints

This project was built to run on Windows devices only. This means only Windows devices can run this software. Since this project was designed to use the RT3D library for model loading and this library only loads ".obj" files, this system cannot load in 3D models which are not of type obj. The texture loader can only load bitmap files with the ".bmp" file extension. This is due to the texture loader not implementing the required library files for other image file types. This project is designed as a 32bit system and implements 32bit libraries and does not currently have the requirements to be built as a 64bit system.

5.7.3 Style

While writing this code there was a focus on ensure ng that it was kept clear and readable throughout. This was done by following basic coding standards. An example of this would be camel case for variable names. For example; camelCase, where the first word is entirely lowercase, and the first letter of each following word is capitalised. Another example would be constant variables being named thusly: MAX_COUNT to clearly distinguish constant variables. Keeping the code clear was the main priority, this involved intelligent use of comments, clearly labelled variables and functions as well as ensuring every class had a clear description within its header file.

5.7.4 System Requirements

This project was developed mainly using one computer. The specifications for this computer will stand as the minimum requirements for any other system to run this software. See Table 5.7.1 for these requirements.

Table 5.7.1 - System Specification

No.	Type	Specification
1.	Operating System	Windows 10
2.	OpenGL	3.3+
3.	C++ edition	17
4.	Processor	Intel Core i5
5.	Memory	8GB RAM
6.	System Type	64-bit

5.8 Risk Management

To help keep this project from straying too far from the original purpose, a MoSCoW (Must, Should, Could, Will not) analysis was completed to help decide which areas of the project were necessary and which areas were extras. This essentially broke down the project into smaller areas of focus. Anything under the title "Must" was a requirement to the project and must be implemented for a passable project, anything under the "Should" or "Could" titles were areas of the project which were less important and not required for a pass. These were still included in the project as they could help push it towards a higher mark. The "Won't" title is for anything which was unnecessary to the project passing and would not help the project develop. Table 5.8.1 clearly lays out each section of the MoSCoW analysis within a table.

Table 5.8.1 – MoSCoW Analysis

No.	Must	Should	Could	Won't
1.	Contain the GJK algorithm for collision detection	Contain different types of bounding volumes (Spheres, Cubes, and Pyramids).	Implement self-made models	Contain more than nine possible objects for the user to choose from
2.	Contain 3D objects of which at the least, all objects must have a collision volume of the same shape	Implement working physics for the responses to the collisions	Implement more accurate physics for the responses	Be developed for Linux
3.	Implement some form of response to the collisions	Have sound effects for the different objects	Have a background sound playing to give the demo some depth	Be developed for release
4.	Be implemented in a 3D environment.	Implement sound for feedback to user input	Use the physics engine from the Game Physics module from 2nd year as a basis for the collision responses	Contain copyright images, models or sound files
5.	Be developed using c++17.	Implement a second collision detection algorithm		Use unethical means to achieving goals, such as plagiarism or theft
6.	Build a convex hull from a set of given points	Compare the data gathered from the two algorithms		

5.9 Evaluation

A white box test was completed on the finished GJK code to determine how fast, after a collision has occurred, the GJK class performs a collision detection test from start to finish. White box testing involves testing the design and structure of the code. This requires the code to be visible to the tester with the goal being to improve the structure and security of the code (Guru99, n.d.). To keep the tests as fair and accurate as possible the software was compiled ten times each with the results stored in a table. An average of these results is calculated and stored in a separate table as well as the standard deviation of the results. Every object is tested for collision against all other objects within the project to help determine if the different vertex sizes have an impact on the time taken to perform a test.

The test involves taking two objects and forcing a collision between them. A breakpoint is added into the code which activates after the GJK has finished collision detection. When the GJK is called, the time is recorded and stored in a variable. When the GJK has found a positive result, confirming collision the time is recorded again and stored in a second variable. The start time is taken away from the end time resulting in the time taken for the GJK to determine collision has occurred. The time taken to perform this detection is written to the console window and recorded in the tables used throughout chapter 7: Results and Evaluation.

5.10 Conclusion

This chapter detailed the SDLC which was used throughout development, alongside the design of the system which implements the GJK algorithm. The different aspects of collision detection which were implemented into this project were discussed, such as the spatial data structure implemented for optimisation. Diagrams were presented which explored the systems classes in more detail. Finally, the testing which evaluates the software was discussed. Now that the design has been explored, how this design relates to the implementation within a project will be discussed. The following chapter shows how the design was implemented.

6 Implementation

This chapter discusses the implementation of the GJK collision detection algorithm, as previously mentioned in the Methodology and Design chapter. The implementation of the theories, key functions and aspects previously discussed are explored within this chapter using visual aids, such as snippets of code and UML diagrams.

6.1 Application - Game

As mentioned in the previous chapter, the application which houses this version of the GJK takes different 3D shapes and forces them to collide. This application uses the GJK algorithm to test for these collisions and then reacts accordingly, showing the user a collision has occurred with a collision response. The application which does this is the *Game* class within this project.

Each shape within the game is given a random position when initialised. The c++ standard *rand* function was used to find the randomised X, Y and Z coordinates. The random positions were kept within a short range as to improve the chances for collisions to occur. Then the shape is added to the container of game entities, called *gameEntities*. This allows for simplified methods for accessing the data, for example; instead of updating each shape individually, a single for loop can be used to update all shapes within the container. Finally, the shape is added to the grid and stored within the section of the grid based which corresponds to its current position. Since there are multiples of each shape, for loops are used to initialise each array of shapes as shown in Figure 6.1.1.

```
//randomly placing the boxes
srand(time(NULL));
for (auto i = 0; i < MAX_SHAPES; i++) {
    glm::vec3 position = glm::vec3(rand() % MAX_X, rand() % MAX_Y, rand() % MAX_Z);
    boxes[i] = new Cube(glm::vec3(1.0f, 1.0f, 1.0f),
        position, textures[0]);
    boxes[i]->init();
    gameEntities.push_back(boxes[i]);
    grid->registerObj(boxes[i]);
}
```

Figure 6.1.1 - initialise Shapes

The game class uses the *update* function to check for collisions and perform other updates which are required throughout the project. This function starts by setting empty variables for the X and Y position of the mouse on the screen. These variables are used to get the position of the mouse using the SDL library. This data is passed into the mouse call-back function which uses these positions to update where the camera is facing, allowing the user to pan around the screen by moving the mouse. Next, the grid is cleared; this removes the currently stored positions for each object, meaning if an object is moved, the new position for this object can be added into the grid and the previous position is not stored. If the grid was not cleared and the objects were moved positions, then false collisions would eventually occur as positions would be stored within the grid which are not accurate as an object may have moved out of that position. After this, each object within the container of shape objects is updated and added back into the grid. After the objects new positions are added to the grid the collision detection function is called, and the player object is updated to allow for updates to the camera and player object movements. The code for the game update function can be seen in Figure 6.1.2

```
void Game::update(SDL_Event sdlEvent) {
    int mouseX, mouseY;
    if (!SDL_GetGlobalMouseState(&mouseX, &mouseY)) //tracking mouse position/ input
        mouse_callback(mouseX, mouseY);

    //Clear grid here
    grid->clearGrid();
    for (vector<Shape*>::iterator it = gameEntities.begin(); it < gameEntities.end(); it++) {
        (*it)->update();
        grid->registerObj(*it); //Register object to grid here
    }
    checkCollisions();
    player->update();
}
```

Figure 6.1.2 - Game Update

Every time the update loop of the *Game* class is called a function is called which checks for collisions, the *checkCollisions* function. This function is shown in Figure 6.1.3. The *checkCollisions* function starts by iterating through each object stored within the *gameEntities* container. This is a container which holds all the shapes within the scene, as a new shape is added it is pushed to this container for storage. To check for collisions between two cube objects, firstly a cube is cast to a pointer then checked to ensure it is not null. Secondly, as the system uses a spatial data structure, the cube object is passed into the grid, and all objects within neighbouring grids are tested, during this, a check is done to make sure the object is not tested for collision with itself, then a second cube object is cast to a dynamic pointer and checked for null. Now that the system has two cube objects and an area of the game environment to check, the two objects are passed into the GJK system, and the GJK checks for collisions. If the GJK finds a collision, then a reaction takes place between the two tested objects. Otherwise, the system moves on to the next pair of objects. Figure 6.1.3 and Figure 6.1.4 show an example of this for two cube objects. This code is expandable for any pairs of objects, not only two of the same type.

```
void Game::checkCollisions()
{
    for (vector<Shape*>::iterator it = gameEntities.begin();
        it < gameEntities.end() - 1; ++it)
    {
        //dynamic cast first object here //
        Cube* cube1 = dynamic_cast<Cube*> (*it);
        if (cube1 != nullptr)
        {
            vector<Shape*> objs = grid->getNeighbours(cube1);
            for (auto it1 = objs.begin(); it1 != objs.end(); it1++)
            {
```

Figure 6.1.3 - CheckCollisions Function

```
vector<Shape*> objs = grid->getNeighbours(cube1);
for (auto it1 = objs.begin(); it1 != objs.end(); it1++)
{
    if (*it != *it1)
    {
        //dynamic cast second object here //
        Cube* cube2 = dynamic_cast<Cube*> (*it1);
        if (cube2 != nullptr)
        {
            if (gjk->performDetection(*&cube1->getHull(), *&cube2->getHull()))
            {
                //Collision response
                cube2->changeTexture(textures[1]);
            }
        }
```

Figure 6.1.4 - CheckCollisions Function (cont.)

6.2 GJK

This project contains an abstract class called *collisionDetector* which is a parent to the collision detection system implemented within the project, the GJK. This interface allows the software to expand and implement more collision detection systems in the future. One current child of this interface is the GJK algorithm class. This GJK class only requires one addition class; The support function class which returns the supporting point These includes can be found in Figure 6.2.1.

```
#pragma once
#include "collisionDetector.h"
#include "SupportFunction.h"
```

Figure 6.2.1 - GJK Includes

The GJK class requires the support function class and stores a pointer to this class within itself. Some other variables are created to help the GJK perform its operations. A step counter to stop the GJK loop from hitting an infinite loop and never exiting is implemented named *steps*. A variable is created which stores the size of the simplex. This is done as the size of the array, which is used for the simplex, must be defined at compile-time and cannot be resized during run time. Therefore, a variable is used to store how many points it currently contains, named *simlexSize.* Next, the simplex is created. Since this simplex is a container of 3D points for this version of the algorithm, an array of vector threes was created, and since the simplex can be a maximum of four points for 3D space, the array was set to contain four vector threes. The final part of the set up for the GJK is the direction. This is a vector three as this project works in 3D space, and the search direction can be along the x, y or z-axis or any variation of all three combined. This search direction is given an arbitrary starting point as the GJK is incredibly fast and does not require a specific starting point; however, optimisations for finding a more appropriate starting point would improve the system. These variables can be seen in Figure 6.2.2. This

figure also shows the class which are used to build up the simplex and will be explored within this chapter.

```
class GJK : public collisionDetector
{
private:
    SupportFunction* support = new SupportFunction();

    int steps;
    int simplexSize = 0;
    glm::vec3 simplex[4];
    glm::vec3 direction = glm::vec3(1, 1, 1);

    bool initialise(std::vector<glm::vec3>&, std::vector<glm::vec3>&);
    bool ContainsOrigin(glm::vec3&);
    bool triangle(glm::vec3&);
    bool tetrahedron(glm::vec3&);
    bool checkTetrahedron(const glm::vec3&, const glm::vec3&,
        const glm::vec3&, const glm::vec3&, glm::vec3&);
```

Figure 6.2.2 - GJK Private Variables

The GJK class is built up of functions which are called one after the other, as the class attempts to build the simplex around the origin. However, only one of these functions are required to be public as all other functions are used by only the GJK class. This function is the *performDetection* function as shown in Figure 6.2.3.

```
public:
    GJK() { };
    bool performDetection(std::vector<glm::vec3>&, std::vector<glm::vec3>&);
};
```

Figure 6.2.3 - GJK Public Functions

6.2.1 Perform Detection

When the *GJK* class is called to perform a collision detection test between two objects, the application which contains the *GJK* class calls the *performDetection* function. This function takes the two convex hulls of the objects which are being tested against each other as input parameters. This function begins by calling the *initialise* function, as shown in Figure 6.2.6, which is described in the following paragraph *Initialise*. After the initialise has completed the loop counter variable, *steps,* is set to zero, this means that when the loop begins it always starts at zero and not at the point where the last call to the function exited. After this, a while loop of fifty maximum loops is started, this is necessary to stop the GJK from infinitely looping. This loop starts by adding the first point in the triangle to the simplex, which now contains three points in total. Now that a triangle has been formed the new point in the triangle is tested to make sure it crosses the origin, as this point must cross the origin for the simplex to contain the origin. If this new point does not cross the origin, then a collision is not possible. Therefore, the system early exists. Otherwise, the GJK can continue. If the GJK can continue and collision is at least possible, then a function is run to check if there is a collision. This function is called *ContainsOrigin* and attempts to enclose the origin within a 4-simplex tetrahedron. If *ContainsOrigin*

returns true, then a collision has been found, and the GJK finishes. This code can be seen in Figure 6.2.4

```
bool GJK::performDetection(vector<glm::vec3>& hull1, vector<glm::vec3>& hull2)
{
    initialise(hull1, hull2);

    steps = 0;
    while (steps < 50) {
        simplex[0] = support->support(direction, hull1, hull2); //Adding third point to the simplex
        if (dot(simplex[0], direction) < 0) //is this new point further than the origin?
            return false;
        else
        {
            if (ContainsOrigin(direction))
                return true;
        }
        steps++;
    }
    return false;
}
```

Figure 6.2.4 - performDetection GJK Function

6.2.2 Initialise

The initialise function, as shown in Figure 6.2.6, is used to create a line which crosses the origin. This function starts by finding the supporting point between the two hulls, before flipping to search in the opposite direction and using this to find the supporting point in this new direction. These two points are stored within the simplex, creating a 1-simplex line. Now that a line has been created it is possible to check if it crosses the origin if it does then it is possible to find a collision. Otherwise, the system exits early. If the line crosses the origin, then performing the double-cross product between the line and the origin changes, the direction to face the origin. Figure 6.2.5 shows the new search direction after the double-cross product has been performed. This sets up the GJK algorithm to enclose the origin within a tetrahedron. Now that a line which crosses the origin has been found, the rest of the *performDetection* function can run and eventually reach the following function, *ContainOrigin*.

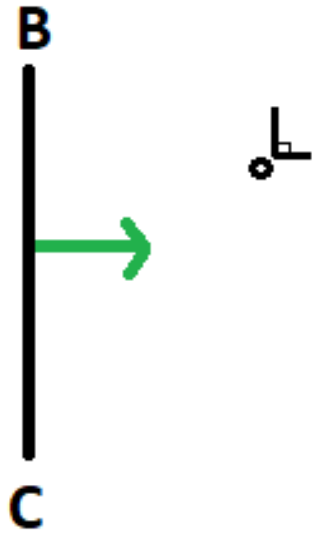

Figure 6.2.5 - Finding Origin Towards Right of Line BC

```
bool GJK::initialise(vector<glm::vec3>& hull1, vector<glm::vec3>& hull2)
{
    simplex[2] = support->support(direction, hull1, hull2); //Adding first point to the simplex
    direction = -simplex[2];                                 //opposite direction
    simplex[1] = support->support(direction, hull1, hull2); //Adding second point to the simplex

    //is this new point further than the origin?
    if (dot(simplex[1], direction) < 0)
        return false;

    //updating search direction
    direction = Maths::doubleCross(simplex[2] - simplex[1], -simplex[1]);

    simplexSize = 2; //begin with a line
}
```

Figure 6.2.6 - Initialise Function of GJK

6.2.3 Contains Origin

When the *performDetection* algorithm adds a new point to the simplex, stored in *simplex[0],* and this point crosses the origin, the *ContainsOrigin* function is called. This function determines which function to call next. If the algorithm has built a triangle and is attempting to find the fourth point to build a tetrahedron, then the *triangle* function is called. If this fourth point has already been found, then the *tetrahedron* function is called instead. If neither is true, then this function returns false. Figure 6.2.7 shows this function.

```
bool GJK::ContainsOrigin(glm::vec3 direction)
{
    if (simplexSize == 2)
        return triangle(direction);

    else if (simplexSize == 3)
        return tetrahedron(direction);

    return false;
}
```

Figure 6.2.7 - GJK Contains Origin Function

6.2.4 Triangle

Now that the algorithm has successfully created a triangle which could contain the origin, the next step is to transform this triangle into a tetrahedron by adding one more point to the simplex. The *triangle* function attempts to do this by finding which area of the current triangle is closest to the origin, or if the origin is currently enclosed by the current triangle. This function begins by determining the lines which require checking as well as the face of the current triangle. For optimisation purposes, the line BC does not require to be checked as it is impossible for the origin to be behind this line since

the algorithm added the point A In the opposite direction when determining which side of BC to place point A. This code can be viewed in Figure 6.2.8

```
//check which edge/face of this triangle is closest to the origin
bool GJK::triangle(glm::vec3 direction)
{
    glm::vec3 ao = -simplex[0];                //Line AO
    glm::vec3 ab = simplex[1] - simplex[0];    //Line AB
    glm::vec3 ac = simplex[2] - simplex[0];    //Line AC
    glm::vec3 abc = Maths::cross(ab, ac);      //Face ABC

    //origin can't be behind points B,C or line BC

    glm::vec3 ab_abc = Maths::cross(ab, abc);  //Line AB on face ABC
```

Figure 6.2.8 - GJK Triangle Function, Setting Up Lines

Now that the lines and face of the triangle have been determined, they can be checked against the origin. First, the line AB is checked to see if it is closest to the origin, if it is greater than the origin then the point C is replaced by B and then B is replaced by A and the new direction is determined by performing the double-cross product (ABxAOxAB). This aims the new search direction towards the origin. Finally, for this section, the function returns false, and the search for the origin begins again within the loop of *performDetection.* Figure 6.2.9 shows the code for this while Figure 6.2.10 shows which line on the triangle is being checked.

```
//is the origin away from AB?
if (dot(ab_abc, ao) > 0) {
    simplex[2] = simplex[1];
    simplex[1] = simplex[0];

    //not facing origin, find new direction
    direction = Maths::doubleCross(ab, ao);

    //can't build tetrahedron
    return false;
}
```

Figure 6.2.9 - GJK Triangle Function, Testing Line AB

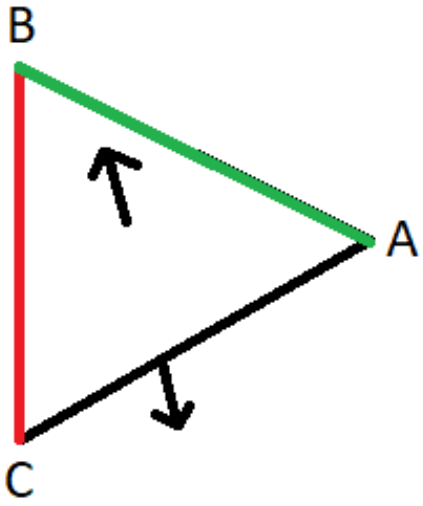

Figure 6.2.10 - Checking the Line AB

This is the repeated for the line AC; however, this time if it is found to be greater than the origin, then point B is replaced with A and C is left unchanged. The same steps are repeated as above with the new search direction aimed towards the origin, and the function returns false. This code is shown in Figure 6.2.11 and the line being checked shown in Figure 6.2.12.

```
glm::vec3 abc_ac = Maths::cross(abc, ac);   //Line AC on face ABC

// is the origin away from AC?
if (dot(abc_ac, ao) > 0) {
    simplex[1] = simplex[0];

    direction = Maths::doubleCross(ac, ao);

    //direction change; can't build tetrahedron
    return false;
}
```

Figure 6.2.11 - GJK Triangle Function, Testing Line AC

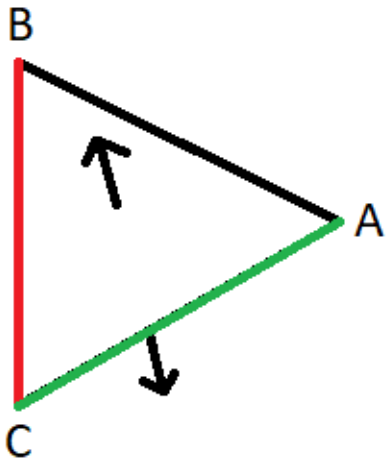

Figure 6.2.12 - Checking the Line AC

If they two previous tests are skipped then the origin must be contained within the current triangle which is being tested as the origin is not outside the lines AB, BC or AC which lie on the triangle ABC. Now the system must check which direction to search in next. The origin can either be above the triangle face ABC or below this face. Figure 6.2.13 shows the code which is performed within the *triangle* function to check which direction to search in next. This test is similar to the line test, however replacing the check between a line and the origin with the face and the origin. If the origin is above the face of the triangle, as shown in Figure 6.2.14, then point D is replaced with point C, C is then replaced with B. Finally, B is replaced by A leaving point A free to become the final point which makes up the tetrahedron. If the origin is below the current face, shown in Figure 6.2.15, then point D is replaced with point B and B is replaced with A, again leaving A to be the final point on the tetrahedron. This section finishes with the *simplexSize* variable being updated to 3, which allows the *ContainsOrigin* function to build the tetrahedron when next called.

```
//above or below face ABC
if (dot(abc, ao) > 0) {
    //base of tetrahedron
    simplex[3] = simplex[2];
    simplex[2] = simplex[1];
    simplex[1] = simplex[0];

    //new direction
    direction = abc;
}
else {
    //upside down tetrahedron
    simplex[3] = simplex[1];
    simplex[1] = simplex[0];
    direction = -abc;
}

simplexSize = 3;
return false;
```

Figure 6.2.13 - GJK Triangle Function, Above or Below ABC Face

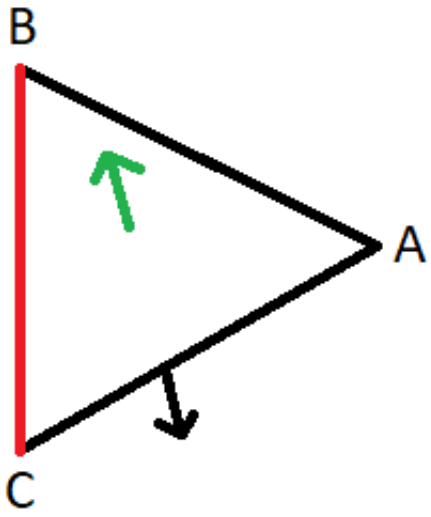

Figure 6.2.14 - Checking Face ABC above

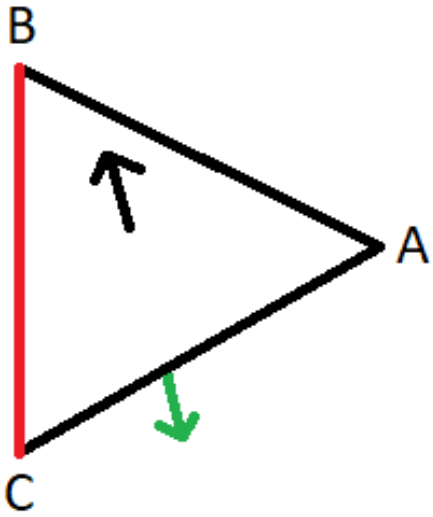

Figure 6.2.15 - Checking Face ABC Below

6.2.5 Tetrahedron

Now that a fourth point has been added to the simplex and a tetrahedron has been built, this tetrahedron must be checked to see if it contains the origin. This check is done using the *tetrahedron* function, which is called from the *ContainsOrigin* function shown previously in Figure 6.2.7. This function begins by determining the lines which make up the face ABC on the tetrahedron and then determining the face ABC itself. This code is shown in Figure 6.2.16

```
//does the current tetrahedron contain the origin
//or do we have to keep checking?
bool GJK::tetrahedron(glm::vec3 direction) {
    glm::vec3 ao = -simplex[0];                 //Line AO
    glm::vec3 ab = simplex[1] - simplex[0];     //Line AB
    glm::vec3 ac = simplex[2] - simplex[0];     //Line AC
    glm::vec3 abc = Maths::cross(ab, ac);       //FACE ABC
```

Figure 6.2.16 - Setting Up the Tetrahedron Function

Now that the face ABC has been found, ABC is tested to determine if the origin is further than the face if so then the origin cannot be within the current tetrahedron and must be closest to this face, therefore the function to find the new point to create the next tetrahedron is run and the face ABC is passed in as the parameter. Figure 6.2.17 shows this test and the call for the *checkTetrahrdon* function.

```
//CASE 1
//in front of face ABC
if (dot(abc, ao) > 0)
    checkTetrahedron(ao, ab, ac, abc, direction);
```

Figure 6.2.17 - Case 1: Testing Face ABC

If the origin is not further than the origin, then it is still possible for the origin to be contained within the current tetrahedron. The second face on the tetrahedron is determined using the cross product between the line AC and the newly created line AD, creating the face ACD. This face is then tested against the origin in the same way the previous face was tested. If the origin is further away than the face ACD, then the origin cannot be contained within the tetrahedron, and the point B is replaced with C while C is replaced with D. The lines which determine the faces are replaced using the new points and then, like in the previous test the *checkTetrahedron* function is run to find the next point to make up the new tetrahedron. This code is shown in Figure 6.2.18.

```
//CASE 2
glm::vec3 ad = simplex[3] - simplex[0];      //Line AD
glm::vec3 acd = Maths::cross(ac, ad);        //Face ACD

//in front of face ACD
if (dot(acd,ao) > 0) {
    simplex[1] = simplex[2];
    simplex[2] = simplex[3];
    ab = ac;
    ac = ad;
    abc = acd;

    checkTetrahedron(ao, ab, ac, abc, direction);
}
```

Figure 6.2.18 - Case 2: Testing Face ACD

If the origin is behind both the faces ABC and ACD, then the next face which is tested is the face ADB. This face is created by performing the cross product of the lines AB and AD. The same check as the previous two is performed to find if the origin is beyond this face. If it is then, as before, the origin is not within the tetrahedron, and therefore the point C is replaced by B and B is replaced with D, and the face is updated using the new points, and the *checkTetrahedron* function is called. Figure 6.2.19 shows this check

A tetrahedron is built up of four faces. Up until now only three of the current faces have been tested for containing the origin. However, this is sufficient in determining a correct result as the final face BCD was already tested back in the *triangle* function when determining which direction to search in next for creating the tetrahedron. This code is shown in Figure 6.2.13 and removes the need to check this face again in the *tetrahedron* function as the origin was already determined to be behind this face.

Finally, if the origin is behind the faces ABC, ACD, ADB and, as previously mentioned, the fourth face BCD then the origin must be contained with the current tetrahedron and collision must be occurring between the two objects and therefore this function can return true which means the *ContainsOrigin* function returns true and therefore the *performDetection* returns true allowing the GJK class to exit with a positive result and allows the application which calls it to continue.

```
    //CASE 3
    glm::vec3 adb = Maths::cross(ad, ab); //ADB triangle

    //in front of face ADB
    if (dot(adb,ao) > 0) {
        simplex[2] = simplex[1];
        simplex[1] = simplex[3];
        ac = ab;
        ab = ad;
        abc = adb;

        checkTetrahedron(ao, ab, ac, abc, direction);
    }
    //origin within tetrahedron already built
    return true;
}
```

Figure 6.2.19 - Case 3: Testing Face ADB

6.2.6 Check Tetrahedron

If the *triangle* function determines the origin to be outside of the tetrahedron built within the simplex, then the *triangle* function calls the *checkTetrahedron* function and passes in the face which is closest to the origin as well as the lines which make up that face. The *checkTetrahedron* begins by performing the cross product on the line AB and face ABC as shown in Figure 6.2.20

```
//tetrahedon doesnt contain origin
//therefor, must check which face on it is closest to the origin
//and set the search direction to this direction
bool GJK::checkTetrahedron(const glm::vec3 ao, const glm::vec3 ab,
    const glm::vec3 ac, const glm::vec3 abc, glm::vec3 direction)
{
    glm::vec3 ab_abc = Maths::cross(ab, abc);       //Line AB on face ABC
```

Figure 6.2.20 - Setting Up checkTetrahedron Function

Next the face is tested against the origin to find out if the origin is beyond the face. If the origin is beyond the face, then a new tetrahedron built using this search direction would not contain the origin and therefore the system replaces points C with B and B with A. Next, the search direction is changed to point towards the origin. Finally, the variable *simplexSize* is updated from three to two which causes the *ContainsOrigins* function to call the *triangle* function the next time it is called. This test ends with the function returning false. Figure 6.2.21 shows this test.

```
if (dot(ab_abc,ao) > 0)
{
    simplex[2] = simplex[1];
    simplex[1] = simplex[0];

    //direction is not ab_abc because it does not point towards the origin
    direction = Maths::doubleCross(ab, ao);

    //start again from triangle checks
    simplexSize = 2;
    return false;
}
```

Figure 6.2.21 - Testing the Face ABC

If the origin is not beyond the line AB on face ABC then the line AC on this face is tested, just as before. This time if the origin is beyond this face then the point B is replaced with A and the rest are left unchanged. The new search direction is found, and the simplex is set back down to two and the test returns false. This code can be viewed in Figure 6.2.22.

```
glm::vec3 acp = Maths::cross(abc, ac);      //Face

if (dot(acp,ao) > 0) {
    simplex[1] = simplex[0];

    //direction is not abc_ac because it's not point towards the origin
    direction = Maths::doubleCross(ac, ao);

    //start again from triangle checks
    simplexSize = 2;
    return false;
}
```

Figure 6.2.22 - Testing the Face ACP

Finally if neither of the previous two tests are found to be true then the origin must be in the direction of the face passed into the function, therefore all of the points are moved up by one, for example; D is replaced by C, C replaced with B. The search direction is updated to point in the direction of this face and the size of the simplex is set to 3. This simplex size makes the *ContainsOrigins* function run the *tetrahedron* function the next time it is called. Finally, this function returns false and allows the loop from *perfromDetection* to continue. This can be viewed in Figure 6.2.23

```
        //build new tetrahedron with new base
        simplex[3] = simplex[2];
        simplex[2] = simplex[1];
        simplex[1] = simplex[0];

        direction = abc;
        simplexSize = 3;
        return false;
    }
```

Figure 6.2.23 - New Search Direction Found

6.3 Support Function

As previously stated, the GJK algorithm implements the support function which returns the supporting point, in a given direction, of a set of points. This support function builds the simplex with the Minkowski difference of two shapes. For this application the support function is contained within its own class called “supportFunction”. This class is comprised of two functions, a private function which returns the index of the furthest point for a given direction within one set of points and a public function which returns the supporting point between two sets of points in a given direction. These functions can be seen in Figure 6.3.1.

```
#pragma once
#include <vector>
#include <glm/glm.hpp>

class SupportFunction
{
private:
    unsigned int furthestPoint(const glm::vec3& direction, const std::vector<glm::vec3>& points);
public:
    glm::vec3 support(const glm::vec3& direction,
        const std::vector<glm::vec3>& points, const std::vector<glm::vec3>& points2);
};
```

Figure 6.3.1 - Supporting Point Class Header File

Since the only public function for the *supportFunction* class is the *support*, this function is ran first. When the GJK calls the *support* function, the current direction and the convex hulls of the two shapes being tested for collision are passed into the *support* function. The *support* function begins by finding the furthest point in the search direction of the first convex hull and stores this variable then repeats this with the second objects convex hull, but in the opposite direction, this is done to give the GJK the furthest possible reach to attempt to enclose the origin. Since the convex hulls are stored as vector containers of 3D points, returning the index of the element within the container is more memory efficient than passing back the entire point within the container (One int value is faster to pass than three floats). Next a vector three is created and used to store the supporting point on the Minkowski difference of the two hulls by taking the furthest point on the second hull and taking it away from the

furthest point on the first hull. This point is then returned to the GJK class and is used as the next point within the simplex being built. The *support* function can be seen in Figure 6.3.2.

```
glm::vec3 SupportFunction::support(const glm::vec3& direction,
    const std::vector<glm::vec3>& points, const std::vector<glm::vec3>& points2) {
    //furthest point in one direction and furthest in the opposite
    int a = furthestPoint(direction, points);
    int b = furthestPoint(-direction, points2);

    //returns a point on the edge of the minkowski difference
    glm::vec3 supportingPoint;
    supportingPoint = points[a] - points2[b];

    return supportingPoint;
}
```

Figure 6.3.2 - Support Function Class Support Function

As shown in Figure 6.3.2 the *support* function begins by finding the furthest point within a container of points, in a direction. This is done by calling the private *furthestPoint* function which returns the index of the element of the furthest point in a container. This function takes the search direction which is passed into *support* function and a container of points. The first step is to create a maximum value by performing the dot product between the search direction and the first element in the container. This point is stored as the current furthest point. An index value is also created and stored as the first element index zero. Next the container of points is looped through until the end of the container and for each loop the current element in the container, which is the element at the index equal to the loop counter number, is checked against the current maximum value. If the current value is further away in the searching direction, then the current point is set to the maximum and the index of that element is stored. The loop continues until the end of the container and the exits and returns the index of the furthest point.

```
unsigned int SupportFunction::furthestPoint(const glm::vec3& direction,
    const std::vector<glm::vec3>& points) {

    float maximum = dot(direction, points[0]);
    unsigned int index = 0;

    for (int i = 1; i < points.size(); i++) {
        float current = dot(direction, points[i]);
        if (current > maximum) {
            maximum = current;
            index = i;
        }
    }
    return index;
}
```

Figure 6.3.3 - Support Function Class Furthest Point

6.4 Shapes

Each shape contains its own implementation of its parent class *Shape*, for this the *Cube* class will be used as the example of this implementation.

When a cube is created three parameters are passed in; The scale, position and texture. The scale is passed into the cubes *transform* class and used to set the scale for that specific object. The cube's "privMove" function is then called and the position is passed in, giving the cube a starting position. Finally, the texture is stored for use later when drawing the cube. This can be seen in Figure 6.4.1.

```
Cube(glm::vec3 cubeScale, glm::vec3 pos, GLuint texture) {
    transform->Scale(cubeScale);
    privMove(pos);
    this->texture = texture;
}
```

Figure 6.4.1 - Cube Constructor

After this the cube's initialise function is run. The initialise function calls the *mesh* class and passes in the file path to the obj file for the *mesh* class to load in this object into the project. After the object has been loaded in the bounding volume is created, this is done by calling the "makeHullContainer" function and passing in the vertices of the object which were stored within the *mesh* class when loading in the object. Figure 6.4.2 shows this initialise code.

```
void Cube::init()
{
    mesh->loadMesh("../Resources/Models/cube.obj");
    makeHullContainer(*mesh->getVerts());
}
```

Figure 6.4.2 - Cube Initialise

The "makeHullContainer" function takes a container of floating points and converts this into a container of vector threes. This is due to the model loading library storing vertices this way. First, the container which will house the bounding volume is resized to fit the vertices required. Next, a counter is created and a for loop is started which loops through each float within the points container and adds the points to the hull container three at a time in x-y-z order until the vector container of hull points contains all the floating-point values stores as vector threes. This function ends with a call to set hull which takes the newly filled hull container as its parameter. The "makeHullContainer" is shown in Figure 6.4.3

```
void Cube::makeHullContainer(vector<float> points)
{
    hull.resize(8);
    int counter = 0;

    for (int i = 0; i < hull.size(); ++i)
    {
        hull[i].x = points[counter];
        hull[i].y = points[counter + 1];
        hull[i].z = points[counter + 2];
        counter += 3;
    }
    setHull(hull);
}
```

Figure 6.4.3 - Cube's makeHullContainer Function

The "*setHull*" function takes the objects base convex hull and transforms it to be in line with the mesh data, giving the object separate collision and mesh objects. The function loops over all the points within the container and manipulates each point one at a time. Firstly, each object contains the "transform" class which handles movement, rotation and scaling. This class uses a single four by four matrix to hold this data. Since the position of the collision object is in 3D the vectors are of size three, however, a matrix holds its transform data on the bottom row, row four. Multiplying a vector of three by a matrix of four by four would not allow the object to be transformed. Because of this the vector three is transformed into a vector four and the last point, Z, is set to one. A copy of the model matrix which stores the transform data is created, the matrix is multiplied by the position vector and the vector four is then transformed back into a vector three. This is repeated for all the points on the hull before the loop ends. Finally, this function ends by passing this updated container of points to the *collidable* class. This class handles the convex hull for each object. The code for this function can be seen in Figure 6.4.4.

```
void Cube::setHull(vector<glm::vec3> points)
{
    for (int i = 0; i < points.size(); ++i)
    {
        glm::vec4 v = glm::vec4(points[i].x, points[i].y, points[i].z, 1.0);
        glm::mat4 m = *(transform->getModelMatrix());
        v = m * v ;
        points[i] = glm::vec3(v.x, v.y, v.z);
    }
    collidable->setConvexHull(points);
}
```

Figure 6.4.4 - setHull Function

6.5 Reacting to Collisions

After a collision between two objects has been detected a response must be invoked. This response is the reason collision is being tested for in the first place. For this project, shapes are given a texture to represent that collision has not occurred, every shape is given a “no collision” texture when they are created. Currently for cube objects the texture shown in Figure 6.5.1 represents a cube which has not collided with another object, while all other shapes have a red texture to show no collision has occurred, shown in Figure 6.5.2. When a collision has been detected the texture of the two objects which are colliding is updated to a texture which represents collision detection. This is the collision response. For cubes Figure 6.5.3 represents a collision has occurred, while Figure 6.5.4 shows a blue texture which represents collision has been detected for an object which is not a cube.

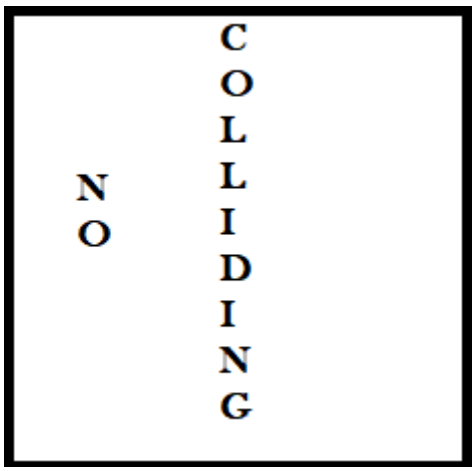

Figure 6.5.1 - No Collision Texture for Cube Objects

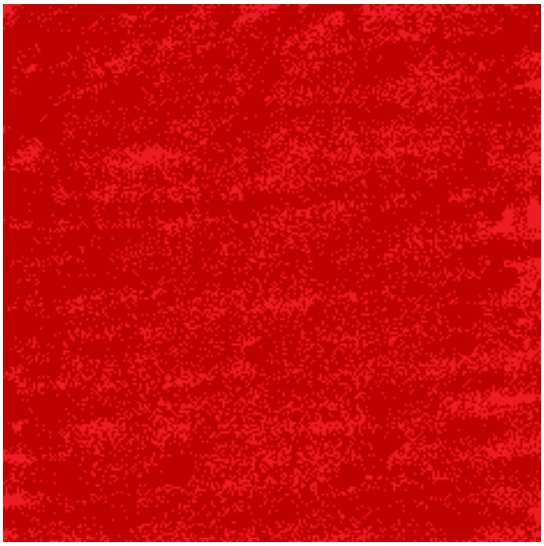

Figure 6.5.2 - No Collision Texture for Non-Cube Objects

Figure 6.5.3 - Collision Detected Texture for Cube Objects

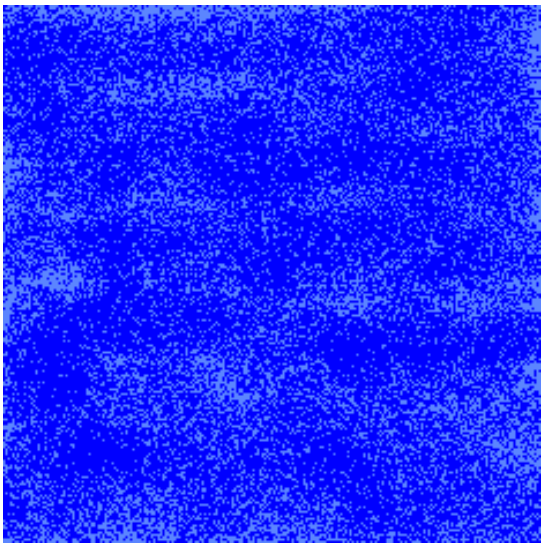

Figure 6.5.4 - Collision Texture for Non-Cube Objects

As mentioned above, when two objects collide their textures are updated accordingly. Textures are loaded in within the Game classes initialise function. The file path for each texture is passed into the *loadTextures* class, this is shown in Figure 6.5.5. This class loads each texture into the project for use later. Each texture is stored as an array of unsigned int values. When each object is created, one of the four textures is passed in and used to draw the box, as shown in Figure 6.5.6 when creating a new *cube* object.

```
//textures
textures[0] = loadTextures::loadTexture("../Resources/Textures/nocollision.bmp");
textures[1] = loadTextures::loadTexture("../Resources/Textures/colliding.bmp");
textures[2] = loadTextures::loadTexture("../Resources/Textures/red.bmp");
textures[3] = loadTextures::loadTexture("../Resources/Textures/blue.bmp");
```

Figure 6.5.5 - Loading in Textures within Game Class

```
boxes[i] = new Cube(glm::vec3(1.0f, 1.0f, 1.0f),
    position, textures[0]);
```

Figure 6.5.6 - Passing in the Texture When Creating New Object

When a collision has been detected, such as in Figure 6.5.7 where two objects are being tested, then a response occurs. For this test, since the first cube is the player-controlled cube only the second cube objects texture is updated. This texture update is performed with a call to the cubes *changeTexture* function which takes a new texture as a parameter, this being an unsigned integer value. Figure 6.5.8 shows a second example of the same collision response, but between a cylinder and a cube object where both textures are updated, using the appropriate textures for each.

```
if (gjk->performDetection(*&cube1->getHull(), *&cube2->getHull()))
{
    //Collision response
    cube2->changeTexture(textures[1]);
}
```

Figure 6.5.7 - Collision Detection/ Response

```
if (gjk->performDetection(*&cylinder1->getHull(), *&cube1->getHull()))
{
    //Collision response
    //Sound::playSample(samples[0]);
    cylinder1->changeTexture(textures[3]);
    cube1->changeTexture(textures[1]);
}
```

Figure 6.5.8 - Collision Detection/ Response example 2

Figure 6.5.9 shows the cube classes implementation of the *changeTexture* function. This function takes the ID of the new texture being passed in and uses this to update the current texture being used to

draw the cube. Figure 6.5.10 then shows how the cube object calls the mesh objects *draw function* and passes in the texture which has just been overridden by the *changeTetxure* functions update. This *draw* function then takes the texture along with all the other information to draw the cube to the screen.

```
void Cube::changeTexture(GLuint newTexture)
{
    texture = newTexture;
}
```

Figure 6.5.9 - Update Texture Function

```
void Cube::draw(GLuint shader,
    std::stack<glm::mat4>* _mvStack, glm::mat4 projection)
{
    mesh->drawMesh(shader, _mvStack, projection, texture,
        transform);
}
```

Figure 6.5.10 - Drawing the Cube

7 Results and Evaluation

This chapter explores the results obtained when the GJK is performed on all colliding objects.

7.1 Collision Detection Results

Figure 7.1.1 shows the users view of the game window when the project is first loaded, and the user has moved the camera to see the scene. Figure 7.1.2 shows six of these objects, which are all close together, but not close enough to collide. Figure 7.1.3 shows two of these objects, the sphere and the cylinder, colliding which has caused the texture for each object to update from the red (no collision) texture to the blue (collision) texture. Figure 7.1.4 shows two cubes which are touching on their faces. These objects are as far apart as they can be to still cause a collision by contacting against the faces only. This shows the accuracy of the system. Figure 7.1.5 is also an excellent example of the accuracy of this collision detection system as the corner of the player's cube object is touching the hull of the sphere, causing a collision reaction.

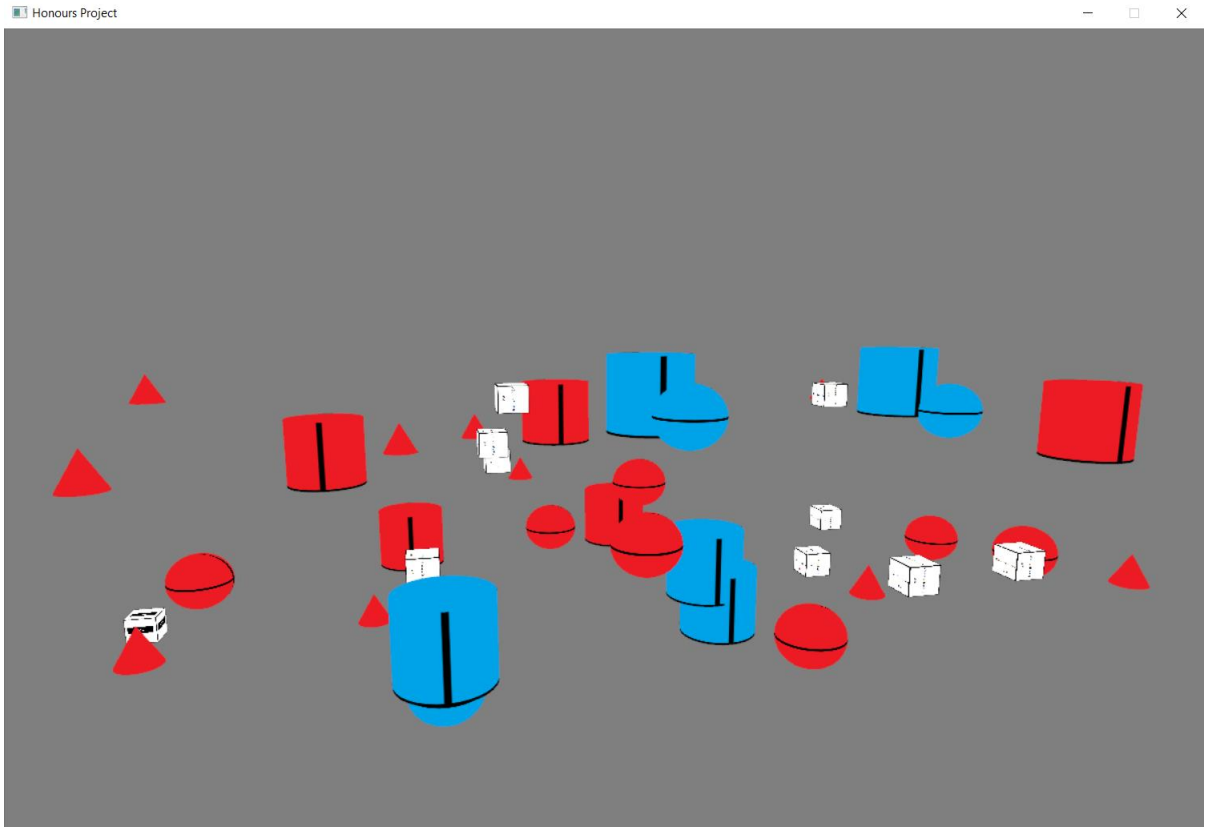

Figure 7.1.1 - Game Window

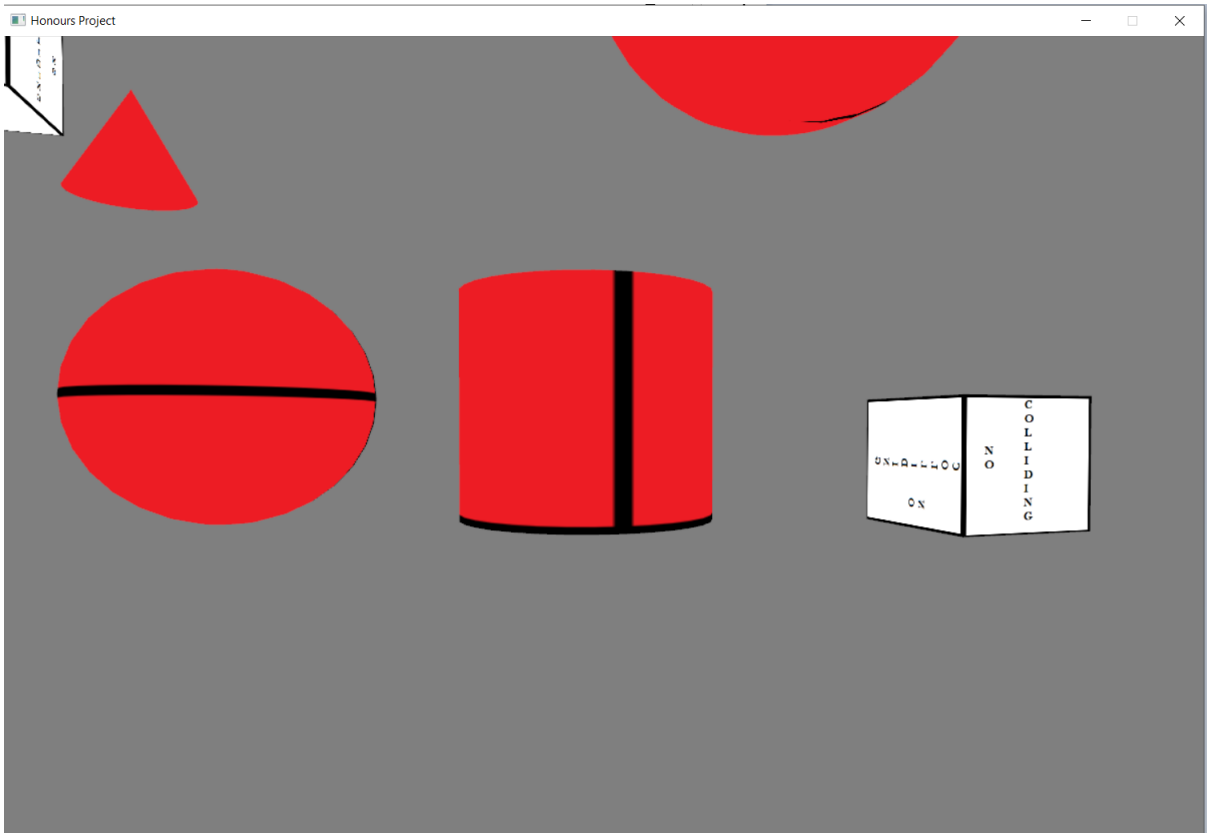

Figure 7.1.2 - No Collision Between Objects

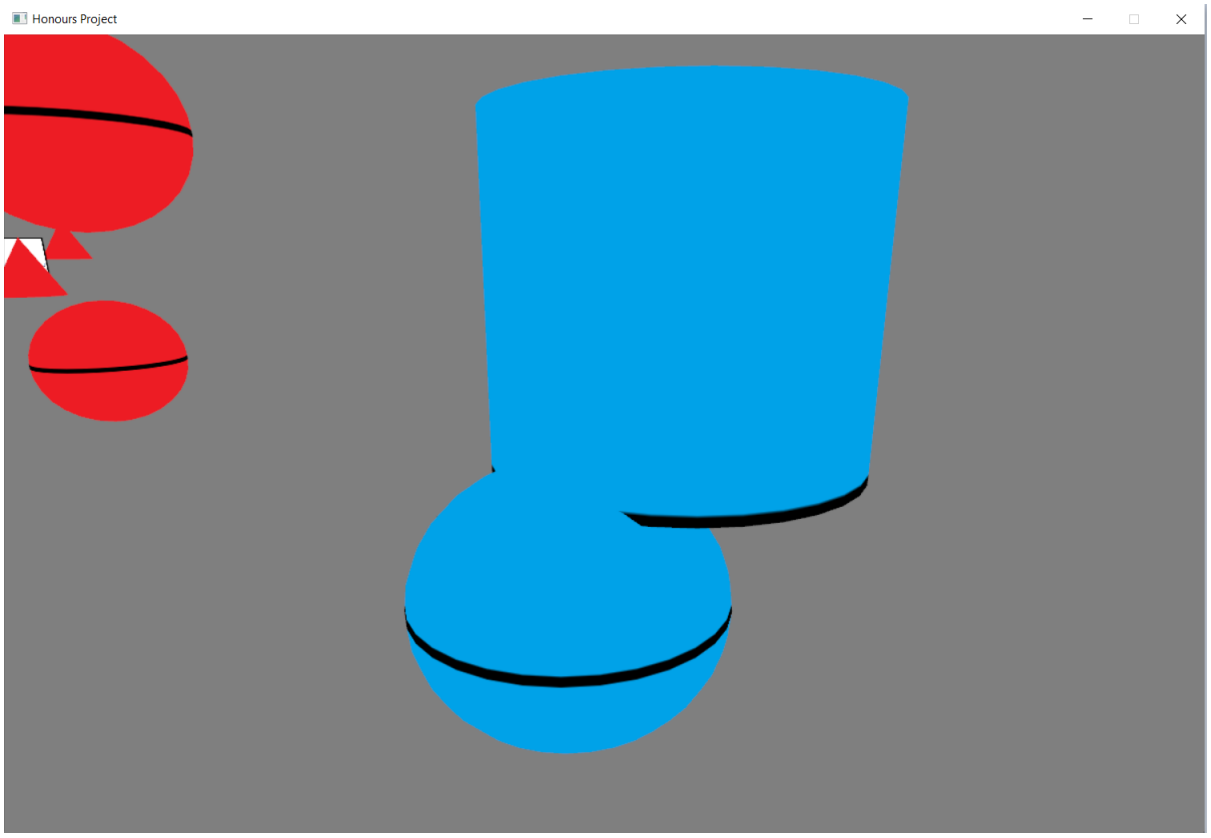

Figure 7.1.3 - Cylinder and Sphere colliding, Updated Textures

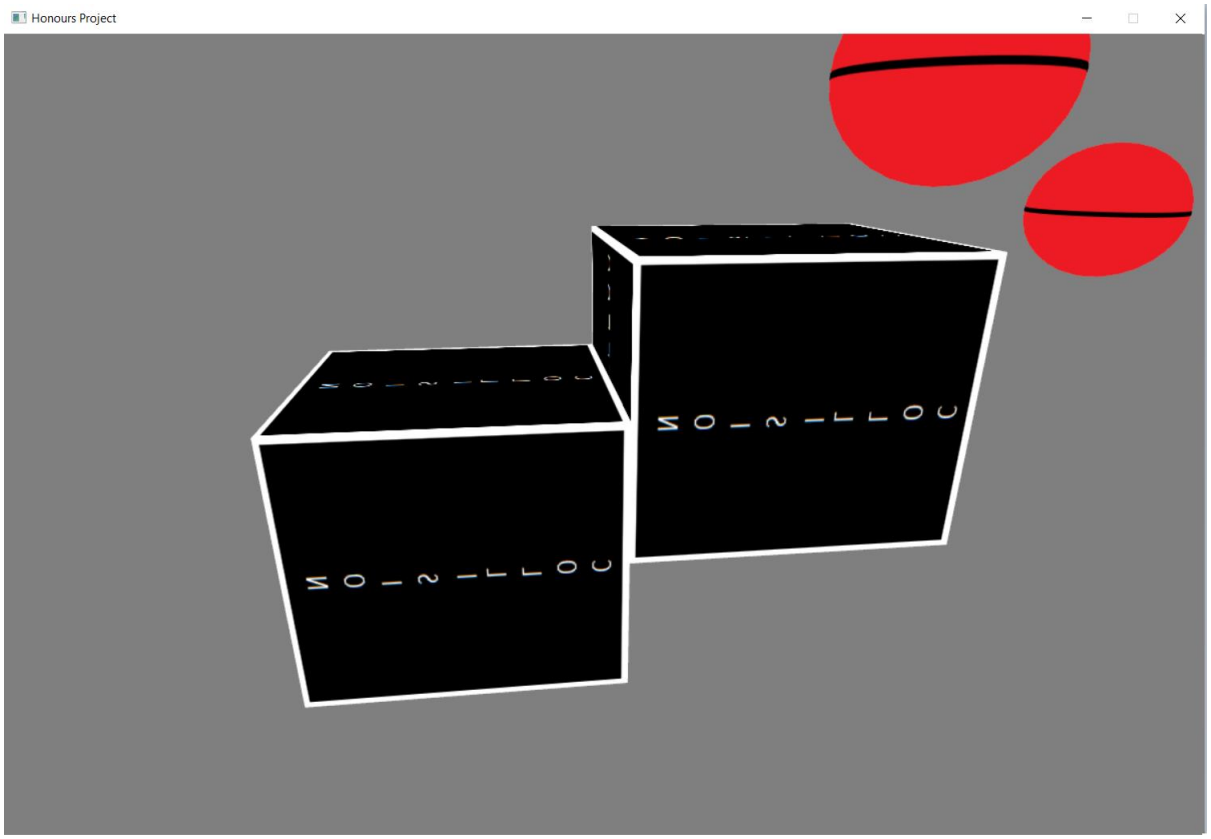

Figure 7.1.4 - Two Cubes Colliding (Touching Faces)

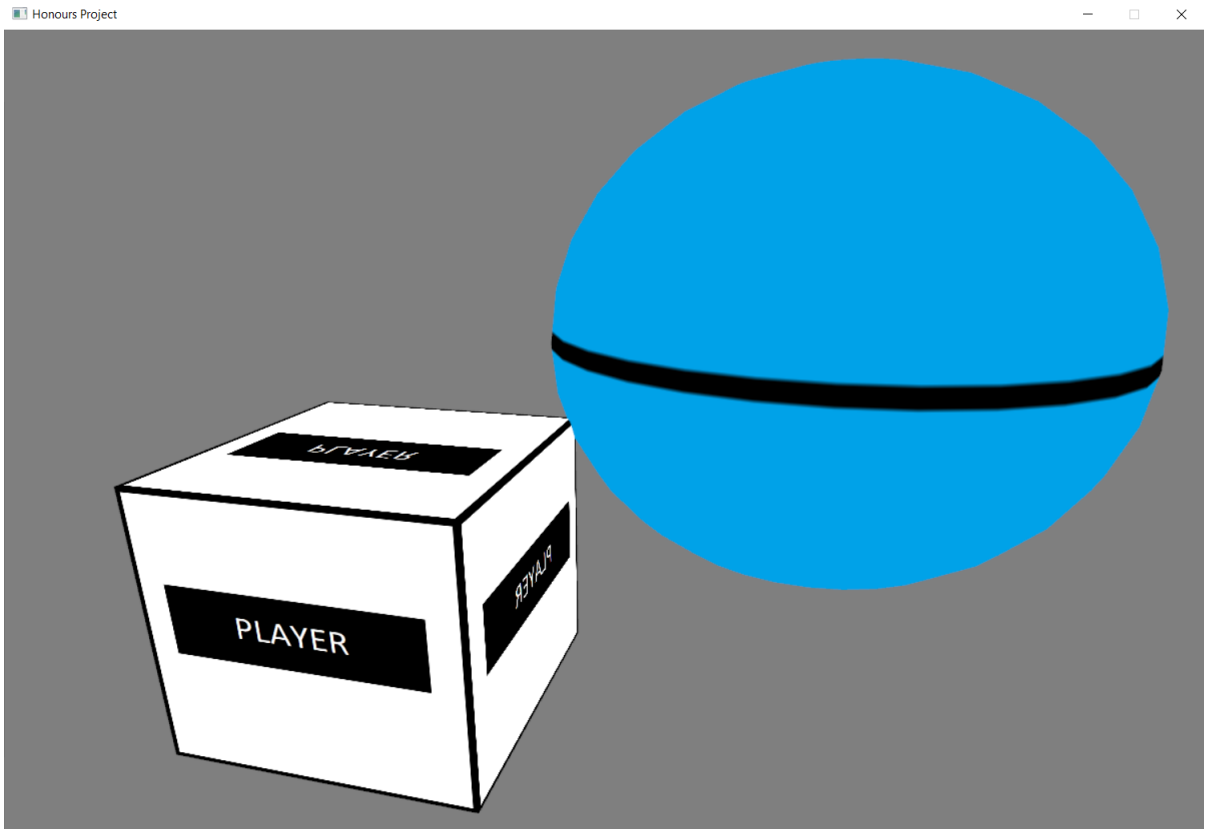

Figure 7.1.5 - Player Object Colliding with Sphere - Corner touching Sphere

7.2 White Box Testing

For the first testing phase, all objects were temporarily removed from the project, and only the objects required for the collision testing were added back. For example, during cube-sphere collision testing, only one cube and one sphere were created and forced to collide. This removes any possibility of outside interference from other objects.

What is immediately apparent from the results shown in

	Shape Tested Against Cube:			
	CUBE	**CONE**	**CYLINDER**	**SPHERE**
	(8 Vertices)	(33 Vertices)	(64 Vertices)	(482 Vertices)
Average Time Taken (Microseconds, rounded to 1DP):	17.3	58.2	154.1	483

and Table 7.2.2 is that the number of vertices which an object contains is directly involved with the calculation speeds. The higher the vertex count, the longer the GJK algorithm takes to determine collision. For example, the average time taken for the GJK to determine collision between two cubes is 17.3 seconds, as shown in Table 7.2.2, where the cubes are made of eight vertices each. The same table shows a shape with a much higher vertex count, such as the sphere which is made of 482 vertices colliding with an 8-vertex cube takes the GJK code an average of 483 microseconds. This trend continues, where the vertex count effects the time taken to perform a detection test and can be viewed in Table 7.2.5, Table 7.2.8 and Table 7.2.11. This is due to the GJK algorithm building the simplex by moving between vertices until the origin is contained, therefore the larger the vertex count, the longer the GJK could take to find the origin as there are more possible vertices to check, the lower the count the less the GJK has to check before determining the same result. The average time taken for the GJK to perform a collision test 328.9 microseconds. The standard deviation was also noted to show the consistency of the software and is presented in

Table 7.2.3, Table 7.2.6, Table 7.2.9 and Table 7.2.12.

Table 7.2.1 - Cube Collision Timing

			Shape Tested Against Cube:			
			CUBE	**CONE**	**CYLINDER**	**SPHERE**
		Test No.	(8 Vertices)	(33 Vertices)	(64 Vertices)	(482 Vertices)
Time Taken	(Microseconds)	1	21	57	151	422
		2	20	57	151	755
		3	17	57	150	407
		4	16	57	171	510
		5	16	57	150	407
		6	16	57	150	508
		7	17	58	150	444
		8	17	57	151	509
		9	16	58	150	407
		10	17	57	170	518
		11	17	68	151	406

Table 7.2.2 - Average Cube Collision Timing

	Shape Tested Against Cube:			
	CUBE	**CONE**	**CYLINDER**	**SPHERE**
	(8 Vertices)	(33 Vertices)	(64 Vertices)	(482 Vertices)
Average Time Taken (Microseconds, rounded to 1DP):	17.3	58.2	154.1	483

Table 7.2.3 - Standard Deviation for Cube Collision Timing

	Shape Tested Against Cube:			
	CUBE	**CONE**	**CYLINDER**	**SPHERE**
Standard Deviation (Microseconds, rounded to 1DP):	1.7	3.3	108.2	8.1

Table 7.2.4 - Cone Collision Timing

		Shape Tested Against Cone:		
		CONE	**CYLINDER**	**SPHERE**
	Test No.	(33 Vertices)	(64 Vertices)	(482 Vertices)
Time Taken (Microseconds)	1	121	193	449
	2	98	193	449
	3	121	193	524
	4	98	336	449
	5	134	243	450
	6	97	193	491
	7	97	235	448
	8	97	201	449
	9	122	285	461
	10	97	191	450
	11	132	262	454

Table 7.2.5 - Average Cone Collision Timing

	Shape Tested Against Cone:		
	CONE	**CYLINDER**	**SPHERE**
	(33 Vertices)	(64 Vertices)	(482 Vertices)
Average Time Taken (Microseconds, rounded to 1DP):	110.4	229.5	461.3

Table 7.2.6 - Standard Deviation Cone Collision Timing

	Shape Tested Against Cone:		
	CONE	**CYLINDER**	**SPHERE**
Standard Deviation (Microseconds, rounded to 1DP):	15.5	48.3	24.3

Table 7.2.7 - Sphere Collision Timing

			Shape Tested Against Sphere:	
			CYLINDER	**SPHERE**
		Test No.	(64 Vertices)	(482 Vertices)
Time Taken	(Microseconds)	1	544	869
		2	573	885
		3	741	808
		4	601	814
		5	722	912
		6	555	880
		7	545	1036
		8	708	804
		9	676	810
		10	546	841
		11	571	863

Table 7.2.8 - Average Sphere Collision Timing

	Shape Tested Against Sphere:	
	CYLINDER	**SPHERE**
	(64 Vertices)	(482 Vertices)
Average Time Taken (Microseconds, rounded to 1DP):	616.5	865.6

Table 7.2.9 - Standard Deviation Sphere Collision Timing

	Shape Tested Against Sphere:	
	CYLINDER	**SPHERE**
Standard Deviation (Microseconds, rounded to 1DP):	78.7	67.3

Table 7.2.10 - Cylinder Collision Timing

			Shape Tested Against Cylinder:
			CYLINDER
		Test No.	(64 Vertices)
Time Taken	(Microseconds)	1	282
		2	301
		3	282
		4	282
		5	290
		6	284
		7	283
		8	319
		9	283
		10	305
		11	311

Table 7.2.11 - Average Cylinder Collision Timing

	Shape Tested Against Cylinder:
	CYLINDER
	(64 Vertices)
Average Time Taken (Microseconds, rounded to 1DP):	292.9

Table 7.2.12 - Standard Deviation Cylinder Collision Timing

	Shape Tested Against Cylinder:
	CYLINDER
Standard Deviation (Microseconds, rounded to 1DP):	13.6

Another aspect which becomes apparent is the consistency of the code. This is visible in the standard deviation shown in

Table 7.2.3, Table 7.2.6, Table 7.2.9 and Table 7.2.12. The standard deviation of all these tests is 36.88 microseconds, which is an average of 10.86% deviation for each shape. This is worth noting as it proves the consistency of the GJK code. The vertex count can increase while the GJK performs consistently at roughly 11% deviation across all shapes.

Due to the likelihood of the GJK being performed while many possibly collision objects exist within an environment, it was worth testing more than just individual shapes colliding. For this testing phase, all shapes were drawn in the same position, forcing collisions to occur at the same time. This test contains results from the start to the end of the *checkCollisions* function. This includes timing not only the time taken for the GJK to perform but also the timings for retrieving data from the grid, casting data, testing data and the collision responses. This test, therefore, shows how the project performs under strain. The timer began when the *checkCollisions* function started and ended when this function exited. There are four shape types in the game, each array of shapes was increased by the same amount each time, for example, if the array size was two then there were eight shapes loaded in and if the array size was ten, then forty objects were loaded into the project. Like previous tests, these tests were performed multiple times, and an average was taken. However, these tests were recorded in milliseconds rather than microseconds, which was used for the previous tests, as *checkCollisions* takes more longer to complete than previous tests.

Table 7.2.13 shows the results for the forced positions while the average is presented in Table 7.2.14.

Table 7.2.13 - Time Taken Per Array Size Forced

	Size of Array								
	2	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Time Taken (Milliseconds)	11	51	113	211	315	526	607	801	1047
	11	49	115	200	318	539	606	814	1048
	11	53	114	204	320	477	619	807	1047
	12	61	114	200	317	499	619	799	1047
	11	50	112	202	316	502	607	802	1037
	11	50	140	201	316	465	615	812	1036
	11	54	127	202	313	501	612	809	1032
	11	52	118	203	319	479	607	803	1024
	12	50	114	201	313	489	612	813	1034
	12	55	123	201	317	544	608	803	1037

Table 7.2.14 - Average Time Taken Per Array Size Forced

	Size of Array								
	2	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Average Time Taken (Milliseconds, rounded to 1DP):	11.3	52.5	119.0	202.5	316.4	502.1	611.2	806.3	1038.9

The system drastically slows down when the array size is eight, loading thirty-two shapes into the project, and as shown in Table 7.2.14, the trend continues as the array size increases. The project very quickly becomes unplayable as the system cannot handle the number of collision tests which are required. This is where a robust 3D spatial data structure would help reduce these numbers and stop the game from crashing under its own weight. Other optimisations would also be required to allow the system to handle more objects at once.

7.3 Conclusion

This chapter discussed the results which were obtained after testing the finished version of this project. Visual results were presented, showing before and after a collision had occurred between objects, as well as a collision between objects touching faces and objects with further penetration. The results from white box testing on the GJK code were then discussed, which shows the speed at which this code can calculate collision when one has occurred. Minor white box testing was also completed on the *checkCollisions* function to view how the project handles the collision detection phase.

8 Discussions and Conclusion

In conclusion, the Gilbert-Johnson-Keerthi distance algorithm appears to be one of the best algorithms to use for collision detection between convex hulls. Its use of the support function to build a simplex which attempts to enclose the origin allows for an early exit and stops the algorithm from performing unnecessary tests. The use of the theory of the Minkowski sum as a basis for the algorithm stands as intelligent use of geometry as a basis for collision detection.

Other algorithms such as Sweep and Prune and the SAT and many more exist which are widely used throughout the gaming industry. However, the GJK was chosen for further study as its intelligent design allows delving into many different areas of mathematics and learn more about different fields than if another detection algorithm was chosen.

Throughout this report, a lot has been learned. Much information has been gathered surrounding the field of collision detection alongside the aspects used for improving these algorithms. Different types of collision detection algorithms have been studied, and the GJK has stood out as the most appropriate algorithm for developing when studying collision detection due to its. The GJK uses different aspects of mathematics which lay as the foundations of the algorithm. These topics were studied to understand the algorithm and how it works. Studying these topics has made understanding how to develop this algorithm as code possible.

The knowledge gathered throughout the report was used to create a project which contains an implementation of the Gilbert-Johnson-Keerthi algorithm, which is used to test for collisions. This project meets the requirements laid out within the project specification. The GJK algorithm was thoroughly researched over the course of seven months. This research builds up the Background Information, Literature Review and Gilbert-Johnson-Keerthi Algorithm chapters of this report. A project was designed and developed which uses the GJK algorithm to test for collision between multiple different 3D shapes; cubes, spheres, cones and cylinders are all present within the software created and can all collide with one another. One major factor which was missed from this project was the use of physics for collision responses. Unfortunately, due to time restraints responses for positive collision tests never got further than basic texture changing.

Taking all of this into consideration, this project should be viewed as a success. The main requirements were met while many extras were implemented to improve the overall system. Further improvements would be required before considering using this software commercially. White and black box testing would be required to find issues, bugs and possible improvements.

8.1 Key Findings

The literature review explored the different aspects of collision detection. Different algorithms were explored, such as the Separating Axis Theorem and Sweep and Prune algorithms which shows how different approaches exist to perform the task of determining collision, each with their benefits and payoffs. Methods for optimisation were also discussed in this chapter. This focuses on reductions for collision detections to improve performance for use in complex environments. Some aspects worth discussing are the use of bounding volumes to reduce the number of vertices tested and spatial data structures which reduce the number of objects being tested for a collision. Without these optimisation techniques complex systems which implement many possible collisions would grind to a halt as these systems without optimisations could not handle the number of calculations taking place at once.

It is apparent from the research conducted that collision detection is a complex field of study with many differing aspects. A good understanding of these aspects gives a developer the knowledge to create a robust system which uses collision detection effectively.

8.2 Future Work

The software created implements the critical features of collision detection, which were researched within the Literature Review and Gilbert-Johnson-Keerthi Algorithm chapters. However, no system is perfect, and many improvements could be implemented to both the collision detection side of the project as well as the design of the software itself. For example; improvements to the GJK code which allow it to detect the range of intersection of the objects. Implementing a second collision detection algorithm for further comparisons would also be worthwhile. The overall system could also be improved by implementing more robust code for creating the convex hulls, which cuts down on excessive vertices. A major improvement would also be to implement a 3D SDS to cut down on excess collision tests which slow the system down, the system currently contains a 2D version for testing purposes.

8.2.1 Collision Detection

The GJK class could be improved by implementing an intersection distance as a possible return function. Currently, the GJK class only returns a Boolean value when testing is complete. However, it is possible to obtain the intersection distance by making changes to the current GJK class. Making these improvements would increase the usability of this code, making it available for use within a broader range of projects. This would be made possible by implementing the algorithm presented with the original description of the GJK, using the Johnson Algorithm, also known as the distance sub-algorithm.

This project only contains one type of collision detection algorithm; however, it is set up to allow more algorithms to be implemented. A possible improvement to this system would be to increase the number of algorithms which are available to the user for testing collisions. This again would allow the system to be used by a more extensive range of projects. One of the possible additions would be an implementation of the SAT algorithm. This implementation would allow for data comparison between the two algorithms. Running both algorithms several times and taking an average of the results would give objective, comparable data between them. A table could then be used to store this data and display these results. One such type of data which could be comparable would be the computational complexity, i.e. is there a $O(n^2)$ or $O(n\ log\ n)$ time for computations and what effect these results have on the overall project. For example, the SAT could result in linear $O(n)$ time, which might be fast enough for some projects, but if the GJK resulted in an $O(n\ log\ n)$ which runs much faster, then this might help a project over choosing the SAT. See Table 8.2.1 for a table of comparisons.

Table 8.2.1 - Comparison Table

Test No.	Test Name	Average GJK Result	Average SAT Result
1.	Compile Time		
2.	Calculation Time		
3.	Computational Complexity		

8.2.2 Software

The software which implements collision detection could be improved as well. For example, the reactions which take place after a collision has been detected could be improved. Currently, the system updates the textures of the colliding objects to show collision, however, stopping the objects from moving when they collide would help visualise the convex hulls more accurately as the objects would be touching along the hulls during a collision.

For optimisations, a spatial data structure was implemented. This SDS was based on one created for the collision detection module by Dr. Gilardi. However, this code was developed for a 2D system and therefore did not separate objects along the Z-axis. Changes to the implementation of this SDS would theoretically improvement calculation speeds dramatically and allow the system to increase the number of objects being tested for collisions.

Currently, the vertices which make up the mesh data are stored when the object is created and used as the objects convex hull. This works within this project as only convex objects are implemented. However, if a non-convex object were to be loaded into the project collision would be possible within the non-convex area of this shape as the GJK only works with convex shapes. Therefore, false positives would occur during collision testing. Implementing a separate library which creates convex hulls from mesh data and separates non-convex shapes down into smaller subsections made of convex shapes would be the next step for this system.

References

Berg. M. D, Cheong. O, Kreveld. M. V, Overmars. M. (1997). *Computational Geometry. Algorithms and Applications.* 3rd Edition. Springer.

Choi A. R, Sung M. Y (2017). *Performance improvement of haptic collision detection using subdivision surface and sphere clustering.* PLoS ONE. 12(9)

Cohen J. D, Ming L. C, Manocha D, Ponamgi M. K. (1995). *I-COLLIDE: An Interactive and Exact Collision Detection System for Largescale Environments*. Proceedings of the 1995 Symposium on Interactive 3D Graphics, pp. 189–196.

Craitoiu, Sergui. (2014). *GJK ALGORITHM COLLISION DETECTION 2D IN C#* [online]. Available from: http://in2gpu.com/2014/05/12/gjk-algorithm-collision-detection-2d-in-c/ In2gpu [Accessed 9th October 2019].

Eberly D. H. (2010) *Game Physics.* 2nd Edition. Morgan Kaufmann, USA.

Ericson, C. (2004). *Real-Time Collision Detection.* Boca Raton. CRC Press.

Finkel, R.A. Bentley, J.L. (1974). 'Quad Trees A Data Structure for Retrieval on Composite Keys.' Acta Informatica. 4th Edition. Springer-Verlag.

Gilbert E.G., Johnson, D.W., & Keerthi, S.S. (1988). *A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE J. Robotics and Automation, 4*, 193-203.

GJK (Gilbert-Johnson-Keerthi) (2010). Dyn4j [online]. Available from: http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/ [Accessed 9th October 2019].

Gottschalk, S. Lin, M.C. Manocha, D. (1997). 'OBBTree: A Hierarchical Structure for Rapid Interference Detection.' *ACM Siggraph Computer Graphics*.

Guru99 (n.d.) *What is WHITE Box Testing? Techniques, Example, Types & Tools* [online]. Guru99 Available from: https://www.guru99.com/white-box-testing.html [Accessed 30th March 2020].

Hubbard, P. M (1995). *Collision Detection for Interactive Graphics Applications.* IEEE Transactions on visualization and computer graphics. P218.

Huynh, J. (2008). *Separating Axis Theorem for Oriented Bounding Boxes*.

Lin, Ming & Gottschalk, Stefan. (1998). 'Collision Detection Between Geometric Models: A Survey.' *IMA Conference on Mathematics of Surfaces*. 8.

Muratori, C. (2016). *Implementing GJK (2006)* [online]. Caseymuratori.com Available from: https://caseymuratori.com/blog_0003 [Accessed 16th October 2019].

Muratori, C. (2018). *Papers I Have Loved (2016)* [online]. Caseymuratori.com Available from: https://caseymuratori.com/blog_0030 [Accessed 16th October 2019].

Nievergelt, J. Widmayer P. (2000). *Spatial Data Structures: Concepts and Design Choices.* Handbook of Computational Geometry. Springer. Berlin.

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. College Park. Morgan Kaufmann Publishers.

SAT (Separating Axis Theorem) (2010). Dyn4j [online]. Available from: http://www.dyn4j.org/2010/01/sat/ [Accessed 22nd October 2019].

Stover, C. & Weisstein, Eric W. (n.d.). *Euclidean Space* [online]. MathWorld--A Wolfram Web Available from: http://mathworld.wolfram.com/EuclideanSpace.html [Accessed 14th September 2019].

Team Linchpin. (2019). *A Beginners Guide to the Agile Method & Scrum's* [online]. Linchpinseo Available from: https://linchpinseo.com/the-agile-method/ [Accessed 16th October 2019].

van den Bergen G. (2003). *Collision Detection in Interactive 3D Environments.* Morgan Kaufmann Publishers. The USA.

Vassilakopoulos. M, Tzouramanis. T (2016). *Quadtrees (and family).* [online]. Encyclopedia of Database Systems. Available from: https://link.springer.com/referenceworkentry/10.1007%2F978-1-4899-7993-3_286-3 [Accessed 28th October 2019].

Vries, J. D. (n.d.). *Collision Detection* [online]. LearnOpenGL. Available from: https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection [Accessed 8th October 2019].

Wang C. (n.d.). *What is a Polytope?* [online]. Berkeley. Available from: https://math.berkeley.edu/~charles/whatis/polytopes.pdf [Accessed 8th November 2019].

Weisstein, Eric W. (n.d.). *Polyhedron* [online]. MathWorld--A Wolfram Web. Available from: http://mathworld.wolfram.com/Polyhedron.html [Accessed 9th October 2019].

Weisstein, Eric W. (n.d.). *Simplex* [online]. MathWorld--A Wolfram Web Available from: http://mathworld.wolfram.com/Simplex.html [Accessed 9th October 2019].

Appendix A: UML Diagrams

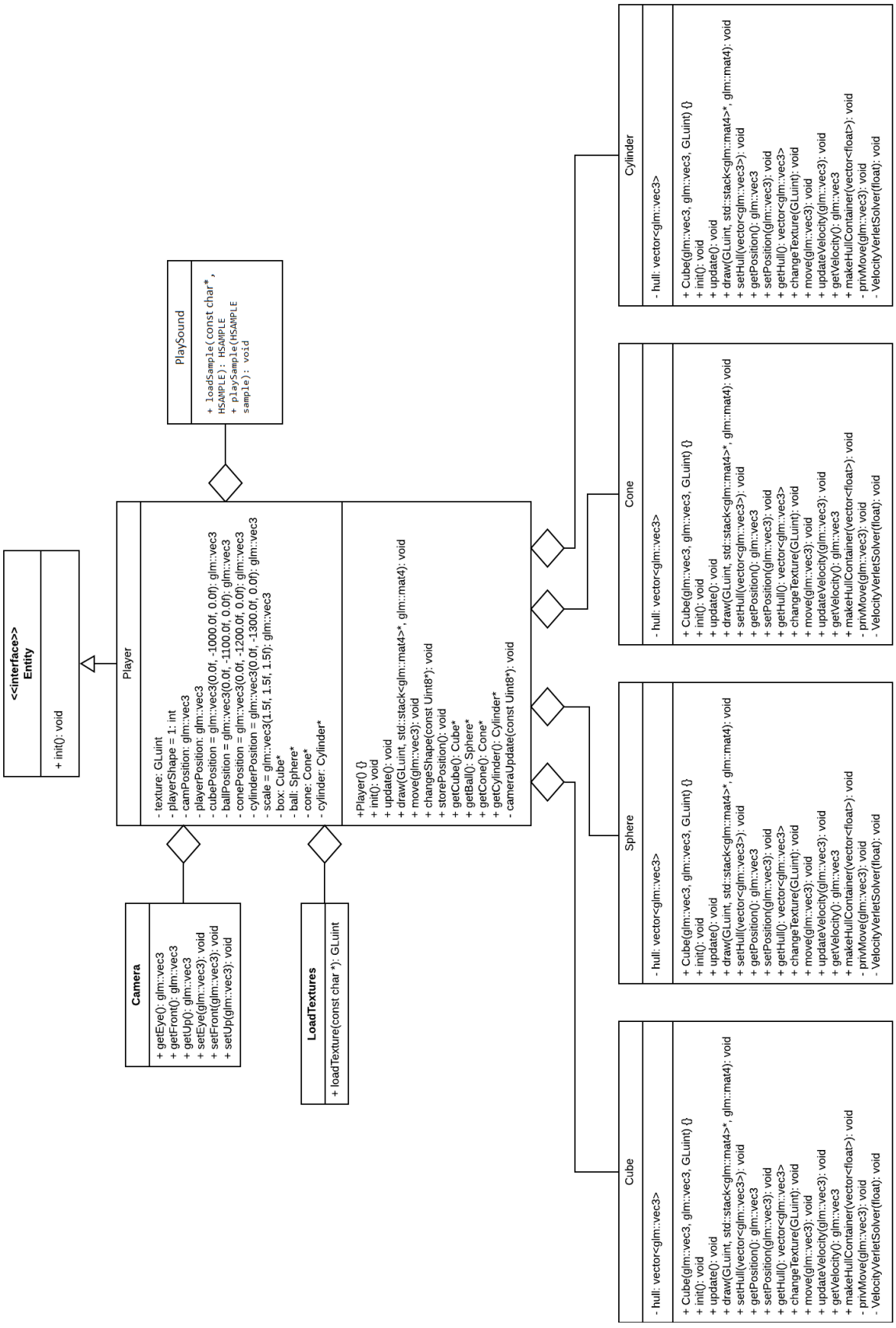

Figure 8.2.1 - Player

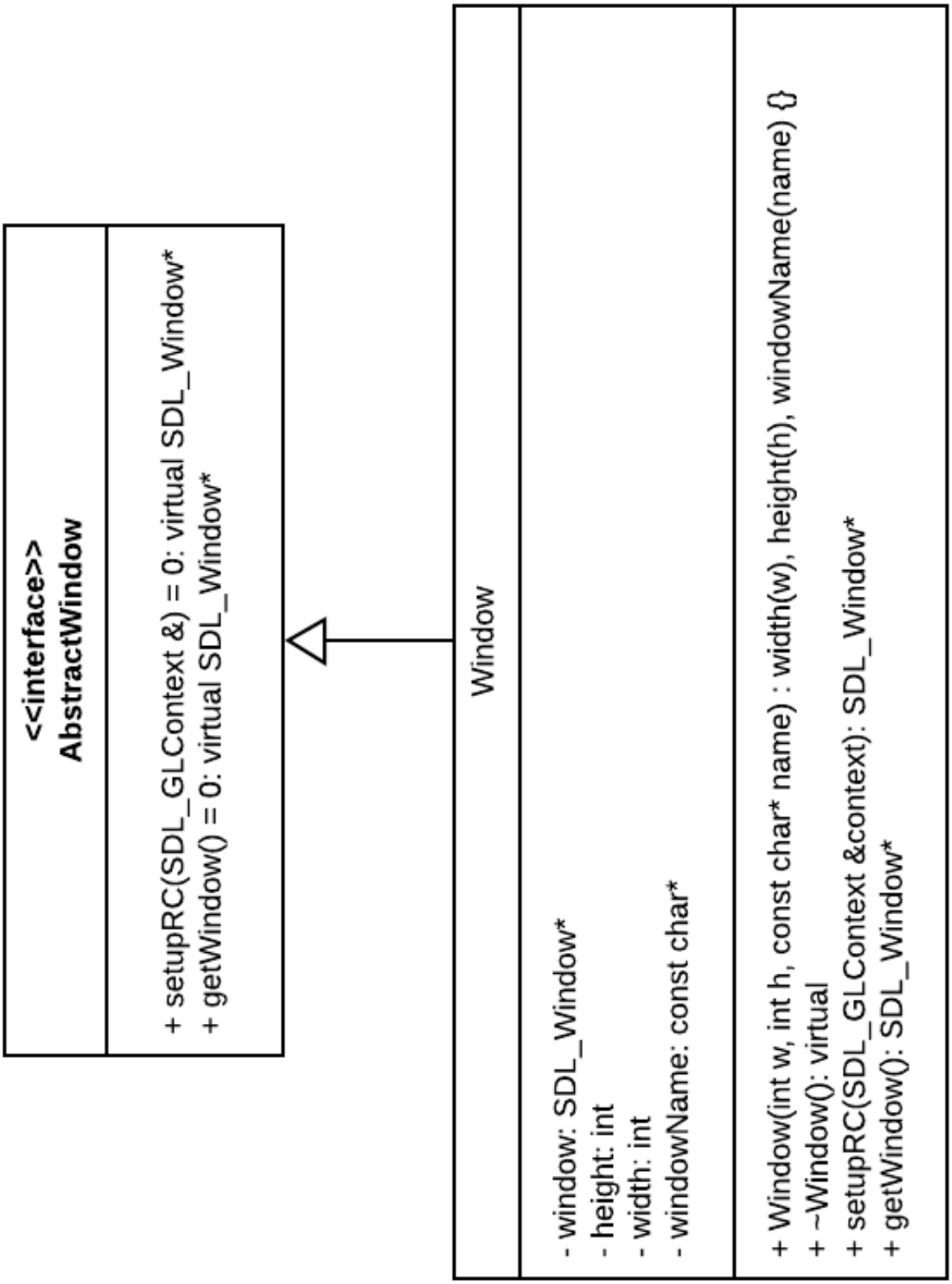

Figure 8.2.2 - Window

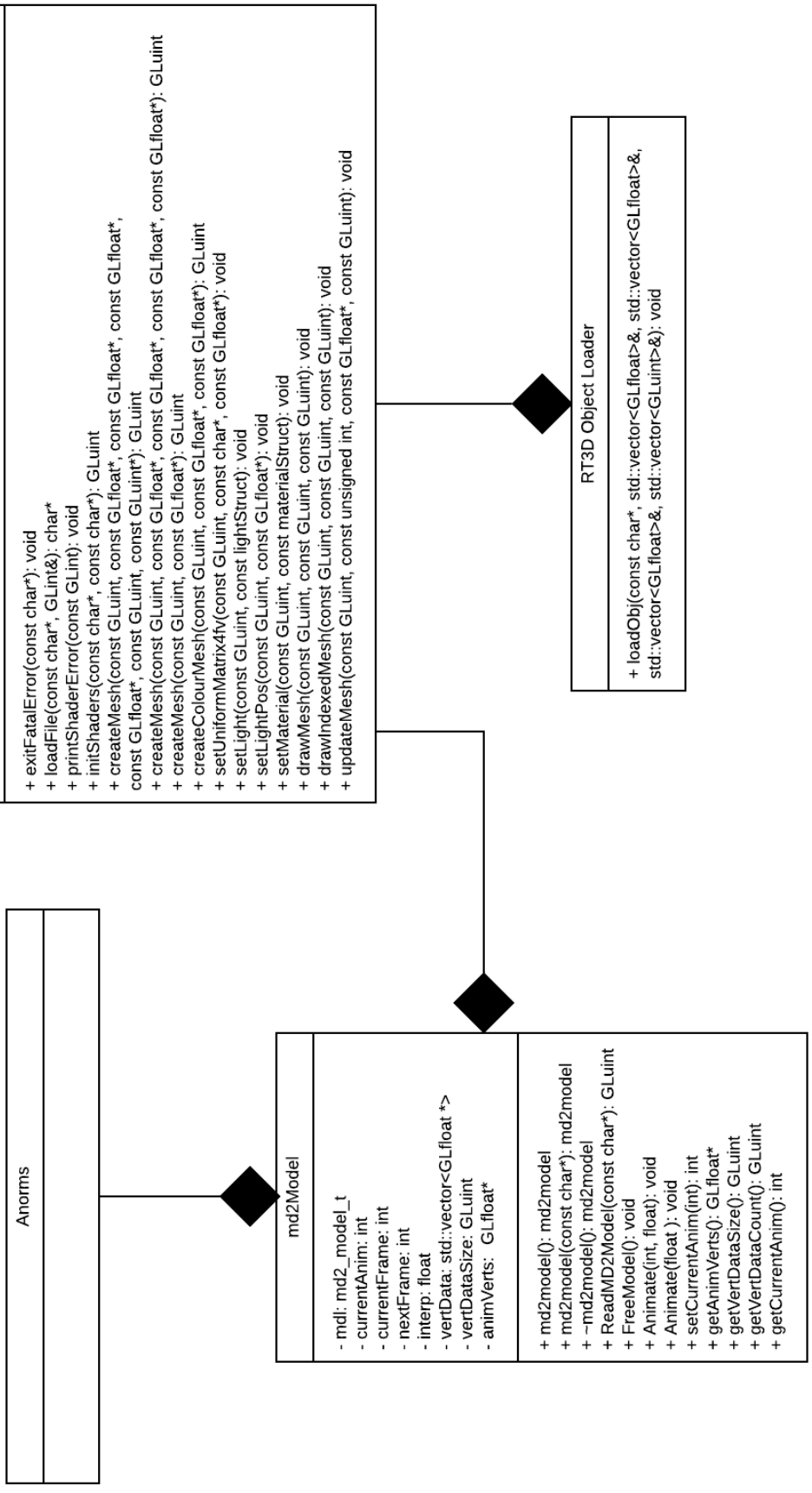

Figure 8.2.3 - Model Loading

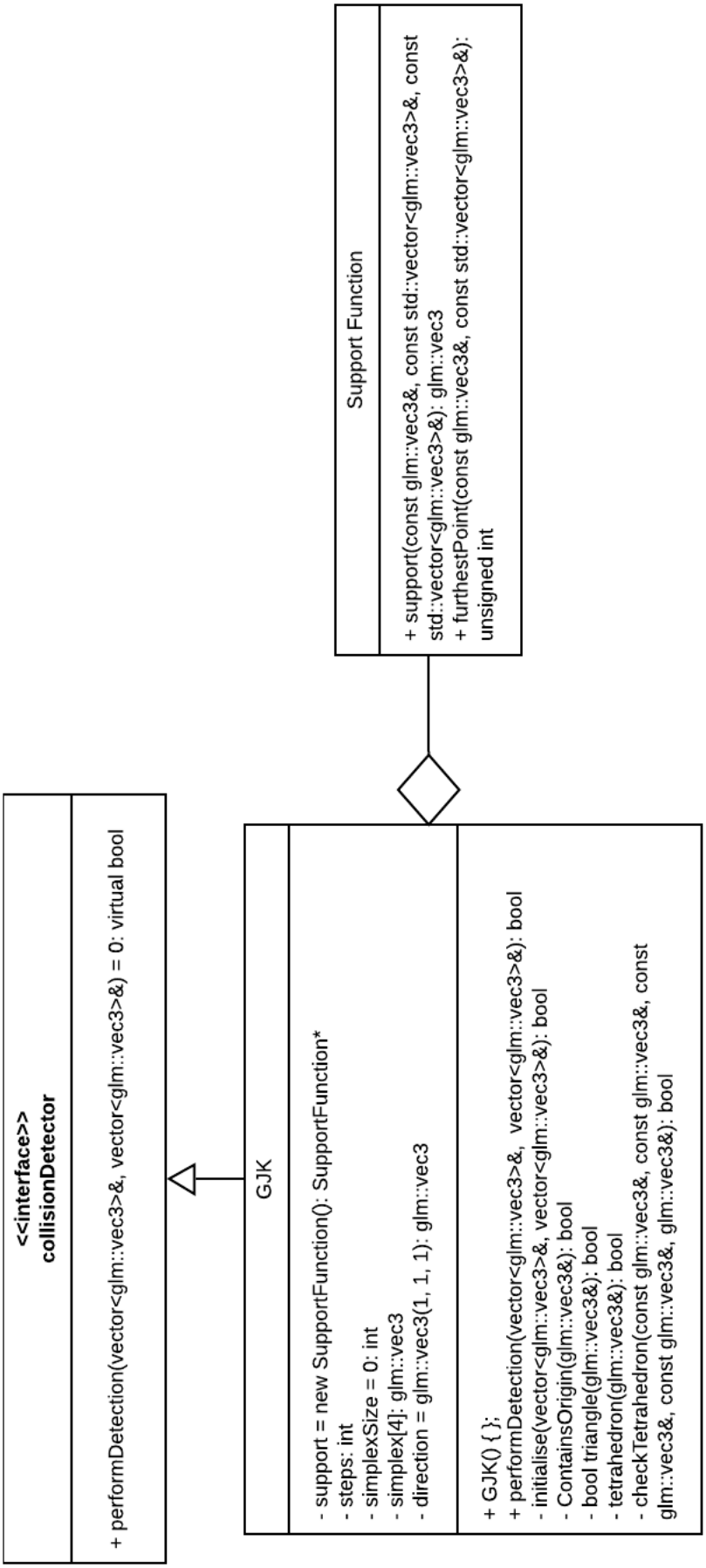

Figure 8.2.4 - Collision Detector

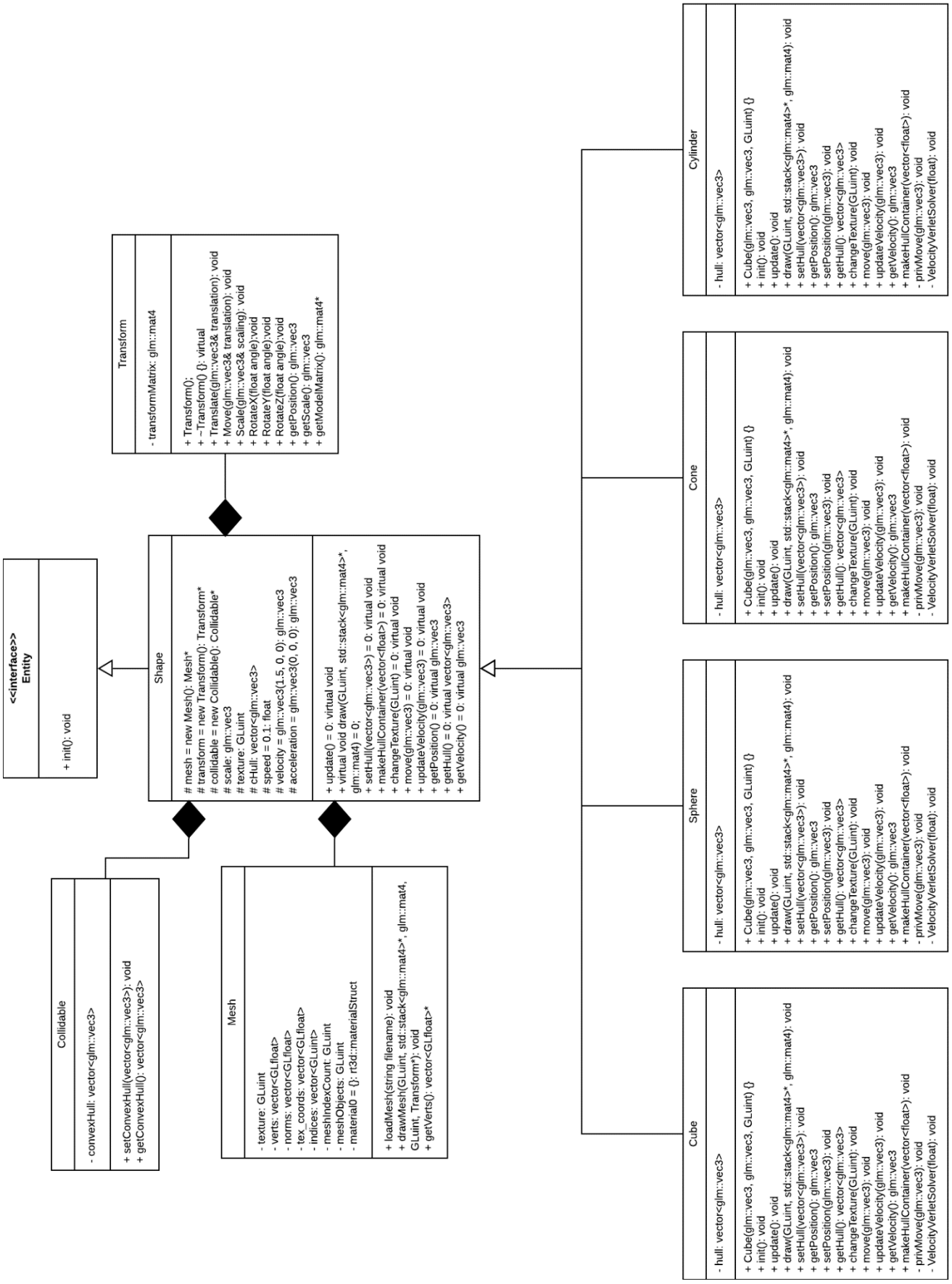

Figure 8.2.5 - Shape

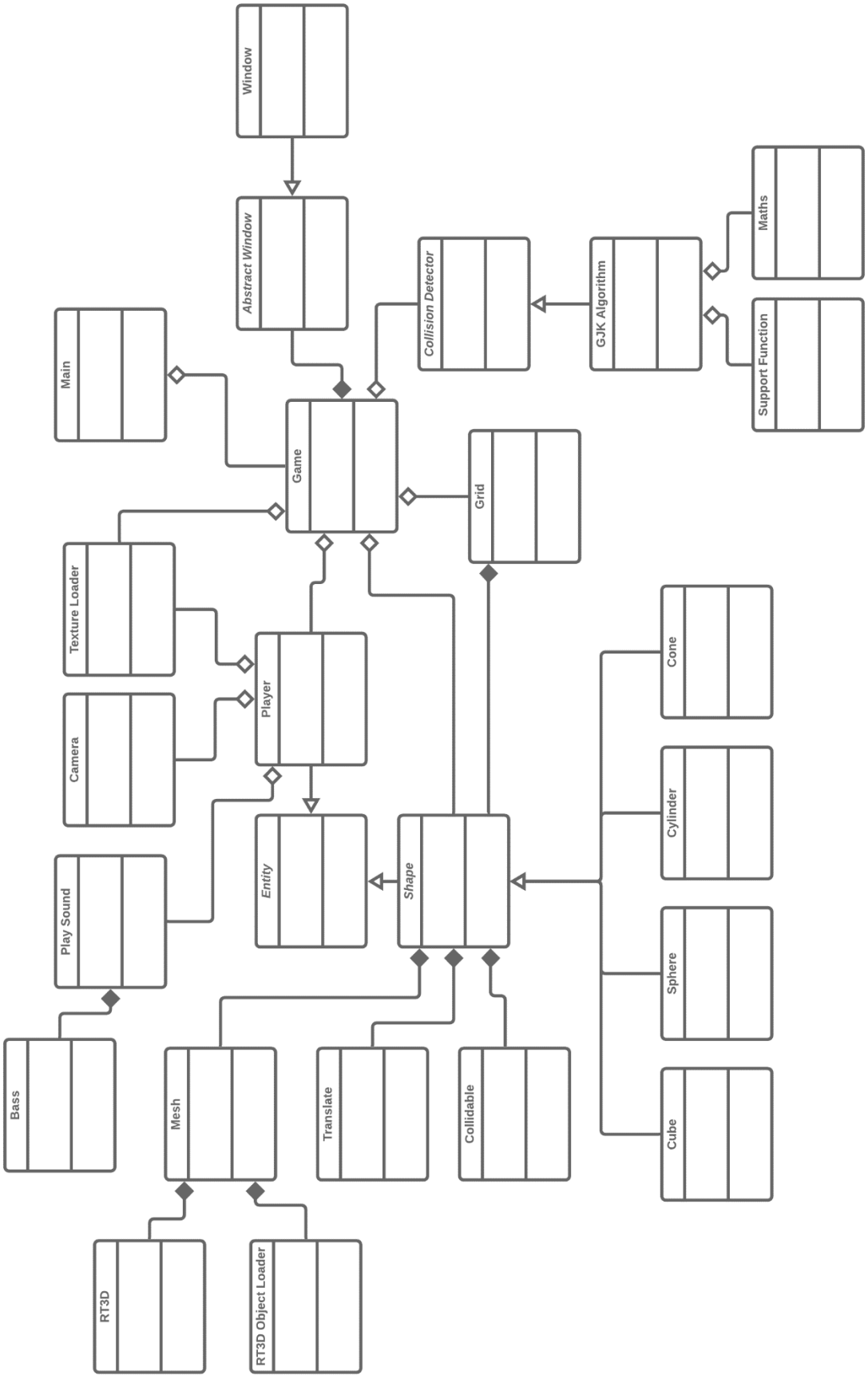

Figure 8.2.6 - Architecture

Appendix B: Critical Appraisal

Collision detection is a broad subject which I had little to no real understanding of before starting this project; however, throughout development, I gained a lot of knowledge and awareness of the subject matter. This project was a considerable challenge from the very beginning as mathematics is one of my weaker skills which had to be improved upon for this project to succeed. I began this journey, unsure if I could develop such a complex project; however, after three months of development, I believe this to be my best work to date.

My focus throughout this past year has always been on getting this project completed to the best of my abilities. My persistence and focus helped me keep the project on track as much as I could. The project did, unfortunately, fall behind and miss some minor deadlines which were set out on the Gantt chart, however, on reflection, the Gantt chart was overly optimistic in some places and unrealistic in others.

One aspect of development which has always hindered my progress is my time management skills, I allowed other projects and work commitments to take too much time away from this project which slowed down progress. I believe this needs to be worked on if future projects are to succeed within strict deadlines. This project was a success, however, a more professional finish, as well as a comparison with another collision detection system, could have been achieved if my time had been managed better.

The project was a success; the GJK code runs fast and works effectively as a collision detection system and has much potential as a viable option for use in future projects. However, much work is required if it was to be used within a commercial project, such as; addition error checking with appropriate feedback and improved optimisations for calculation speeds.

Overall, I pushed myself to complete this project and make progress every day, no matter how small. I worked harder than I have worked on any previous project and throughout this process I have gained more knowledge than I ever imagined that I would. I have vastly improved in my abilities as a programmer, as this project forced me to learn more about C++ and develop my problem-solving skills, to solve issues without relying entirely on my supervisor to talk me through each problem. While I still have a lot to learn about programming and my skills could be greatly refined, I have came a long way since the planning stages of this project. I believe I have improved as a developer more in the last few months than I have over previous years as I learned to focus and take the time to plan and think before jumping into the code.