UNIVERSITY OF BIRMINGHAM
SCHOOL OF COMPUTER SCIENCE

Creating a 3D Physics Engine for Rigid Body Simulations in Java

Dominic Cogan-Tucker
MSc Computer Science
1636752

Supervisor: Dr Ian Kenny

September 2020

Table of Contents

List of Figures

Abstract

The goal of the project described by this report was to develop a basic 3D physics engine in java. The resulting system should be able to render entities to the screen, and accurately simulate rigid body dynamics of such entities, in three dimensions, detect when two entities collide and then resolve any collisions. The motivation behind this project comes from my passion for physics, with a BSc in the subject, and from my love of video games, with the possibility of using the engine to develop my own game in the future.

The engine was developed using an iterative methodology and an Entity-Component-System architecture. The rendering system was implemented using the Light Weight Java Game Library's OpenGL binding, allowing me to control the process, providing the the functionality required. Multi step collision detection system was used, with the Gilbert–Johnson–Keerthi distance algorithm used at the final step, and then the Expanding Polytope Algorithm was used to generate contact point data.

Testing shows that the collision detection system can accurately determine when any two convex entities are intersecting. In a range of demo scenarios, collision response is also shown to simulate closely to real world interactions. This results in an engine that can be used to create simple rigid body simulations, with a potential to be used as the base for a 3D game.

Keywords: Game Physics, Rigid Body, Simulation, Collision Detection, Impulse, Rendering & OpenGL.

Acknowledgements

I would like to thank my supervisor Dr Ian Kenny for the guidance and advice he provided throughout the project.

I would also like to thank the developers of the Light Weight Java Game Library and the Java OpenGL Math Library, as without their work this project would not have been possible.

1 Introduction

The aim of this project was to develop a 3D physics engine that can render a 3D scene to the screen in real time. The resulting system could then be used in future projects to develop 3D physics reliant games or be extended, to include additional functionality, akin to a more complete game engine.

A physics engine is a piece of software that can provide an approximate simulation of real life physical systems. Such software has seen many uses: from more scientific uses, such as predicting physical interactions; to use in video games, to provide the user with a more immersive experience. A highly sophisticated scientific engine could possibly simulate the behaviour of a large number of complex physical systems, although might require significant pre-processing time to achieve scientific accuracy. On the other hand a physics engine used in games would likely want to be able to produce the best approximation that can quickly be simulated in real time, as to give the user a sense of being in control of the environment.

The intent of the system developed in this project is to perform highly accurate and repeatable visual simulations in real time. A system like this, capable of simulating a range of physical systems, would require a large team of experienced individuals and a long period of time. Therefore, the engine detailed in this report has be designed to be able to accurately simulate just the single system of rigid body dynamics. Where a rigid body is a physical objects that doesn't deform under the presence of any external forces, with dynamics being how a body's motion is affected by such forces.

For the success of this project the system must be able to: detect collisions between bodies, determining the point of contact; resolve these collisions, by applying the appropriate forces to the bodies involved; working out how these force affect the bodies motion, updating their positions; and then rendering the results to the screen, all in real time. These processes must be efficient and take as little time as possible to compute and perform their task, while still providing high accuracy of results, to ensure a smooth simulation.

1.1 Report Outline

The report will continue with the following sections with the aim of providing a great understanding of the project and the processes taken to complete it.

Background Material: A detailed background into theory behind: rendering, the graphics pipeline and some of the available APIs that can be used; collision detection algorithms and common practices; and finally collision response and how we can go about incorporating physics interactions into our system. Additionally, we will take a look at some existing game engines with 3D physics capability.

Project Specification: The initial requirement specifications of the system, with a list the functional and non-functional requirements, a Use Case Diagram and Activity Diagrams.

Solution Design: A look at the design of the solution, and the decision behind the choices made, including: architectural structure, the rendering process, collision detection algorithms, and the approach to simulating dynamics.

Implementation & Testing: An explanation of the libraries used in developing the system and how the chose algorithms for collision detection were implemented and optimised. Finally talks about the testing and validation process of the system to ensure it was fully functioning as expected.

Results & Evaluation: Demonstrations of the final product, showing it's capabilities, and an evaluation of how well the system met the specification requirements.

Discussion: A summary of what has been achieved in the project and any of it's short comings. We will also look at what could have been done to improve the system, as well as possible projects for the future.

Conclusion: To wrap up the report a brief statement on how the end product has addressed the goal in this introduction, and a statement to sum up results and what they mean.

2 Background Material

In this section we will cover the areas vital to developing a physics engine. With a brief introduction to rendering and the graphics pipeline, a detailed look into collision detection and a range of algorithms that can be used to achieve it, and finally rigid body dynamics.

2.1 Graphics Pipeline & APIs

The graphics pipeline refers to the set of interconnected steps a graphics system needs to take to process data that describes a scene to be rendered (Wayback Machine 2017). Where a scene is made up of models, with each model typically being made up of polygons (usually in the form of triangles). In the case of three dimensions, the model data, vertex coordinates in 3D space will be input into the pipeline and transformed into the 2D screen space where an image will be displayed through the lighting of pixels. An outline of the steps taken can be seen in figure 1.

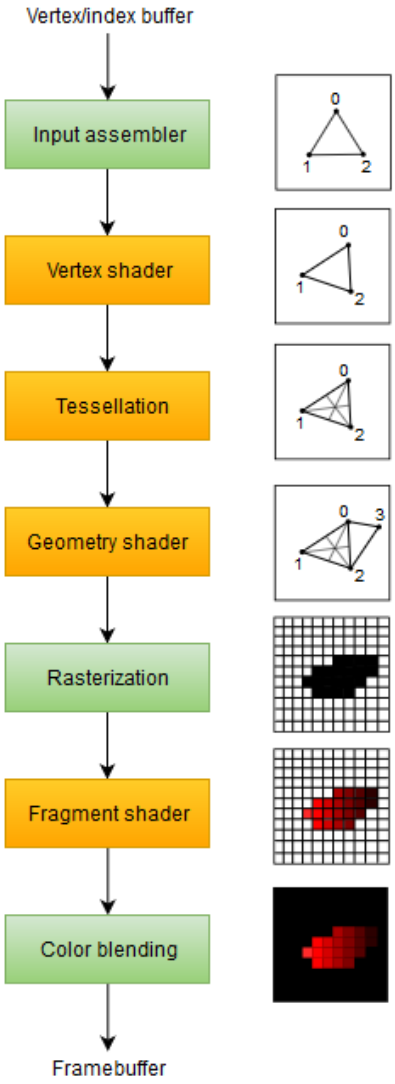

Figure 1: Basic steps in a graphics pipeline (Vulkan Tutorial 2020).

Vertex Shader

A Shader is a user-defined program that is designed to run at a stage in the graphics pipeline. The Vertex Shader is the programmable Shader that handles the processing of individual vertices, they receive a single vertex at a time and generate a single vertex for output. Typically some sort of transformation is applied to the vertices, as the input vertices are relative (OpenGL 2020).

Tessellation

Tessellation is typically an optional vertex processing stage in the graphics pipeline, where sections of vertex data are subdivided into smaller Primitives (OpenGL 2020).Primitive are the simplest geometric shapes that be drawn in n dimensions: in 0-dimensions being a point; 1-dimension being a line; and 2-dimension being a polygon, typically a triangle. This process allows for detail to be added and removed from 3D polygon meshes dynamically.

Geometry Shader

Typically another optional stage in the graphics pipeline, that takes a single Primitive at a time as it's input, and can output any given number of primitives. The limit to the number of output Primitives is defined through the implementation of the Geometry Shader, which also determines the type of Primitives to accept at input, and the type to output (OpenGL 2020).

Rasterization

This is the process of taking each Primitive described in vector format and converting it into a raster fragment (a set of pixels, that when displayed on screen, resemble the input Primitive).

Fragment Shader

The Fragment Shader processes the fragments generated in Rasterization into a set of colours. Each pixel in the fragment is given a colour, typically mapped from an input texture. Along with the set of pixel colours, the Fragment Shader outputs the depth of the fragment, which can be used for process such as shadow mapping.

2.1.1 OpenGL

OpenGL is the most widely used 2D and 3D graphics API. It is window-system and operating-system independent, and network-transparent. Enabling developers to create high-performance, visually compelling graphics software applications (OpenGL 2020). OpenGL is a high level API giving the user abstract control of the macro systems functions.

2.1.2 Vulkan

Vulkan is a new generation graphics API that provides high-efficeiney with cross-platform acces to GPUs used in a wide range of devices (Vulkan 2020). Vulkan is a low level API, unlike OpenGL, meaning instead of giving the user control of macro

systems, they have greater access to the individual components, allowing more control over process such as memory management.

2.2 Collision Detection Algorithms

Collision detection is the process of analysing the geometry of objects to determine if two or more objects are intersecting in a scene. However there isn't just one method to achieve this, there are a large number of methods and algorithms that can be combined to provide different accuracy and efficiency.

2.2.1 Simplified Bounding Volumes

A common step used in the process of collision detection is simplifying the shape of the objects in question, to improve efficiency. By creating a simple geometrical shape whose volume encompasses the entirety of the more complex object, you can perform intersection test using these simple volumes, which will enable faster and simpler methods at determining intersections at the cost of accuracy.

Axis-Aligned Bounding Boxes:

The simplest type of bounding volume, consisting of a non-rotated box aligned to the x, y and z axes. Checking if any two non-rotated boxes are overlapping can be done with simple logical comparisons. Simply we need to determine for a each axis whether the range of each box's min to max values overlap. As can be seen in 2D example in figure 2, both the x and y min to max range on both axes are overlapping. Similarly in 3D we can say that two boxes whose ranges overlap on every axis (x, y & z), must be intersecting.

We can write this as a logical statement, as follows;

$$\begin{aligned} f(A,B) = (A_{minX} \leq B_{maxX} \ \wedge \ A_{maxX} \geq B_{minX}) \\ \wedge \ (A_{minY} \leq B_{maxY} \ \wedge \ A_{maxy} \geq B_{minY}) \\ \wedge \ (A_{minZ} \leq B_{maxZ} \ \wedge \ A_{maxZ} \geq B_{minZ}) \end{aligned} \tag{1}$$

(*Albeza* 2015)

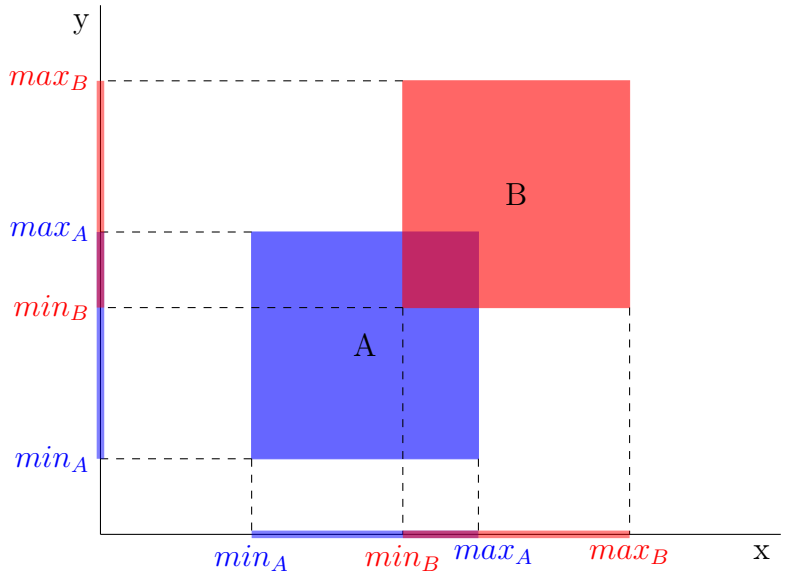

Figure 2: A visualisation of two 2D bounding boxes intersecting and how all axis min-max ranges overlap.

Although determining an intersection is quick, if the objects in question rotated, then the bounding boxes would need to be updated so it still fully encompasses the object.

Bounding Spheres:

Although slightly more complex, a sphere is invariant to rotation so if the object rotates the bounding sphere remains unchanged. To determine if any two spheres are intersecting, we need to compare the distance between the two sphere's centers and the sum of their radii. As we can see in our 2D example in figure 3, when the sum of two circles radii is greater than the distance between their centers, they must be overlapping.

We can write this as a logical statement, as follows;

$$f(A,B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2} \leq A_{radius} + B_{radius} \quad (2)$$

(*Albeza* 2015)

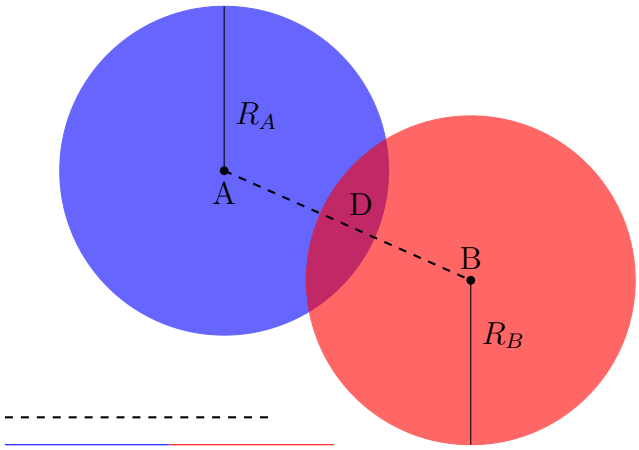

Figure 3: A visualisation of two 2D bounding circles intersecting. The sum of their radii is greater than the distance between their centers.

However, unless the object in question is uniform and spherical in nature, it could result in a significant number of false positive detections, as seen, in figure 4, where a box is a much better fit.

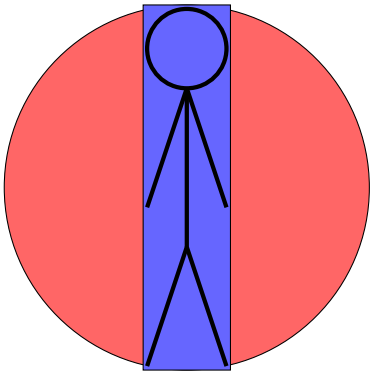

Figure 4: Comparison of a bounding sphere and bounding box encompassing a stick figure resembling a person.

2.2.2 Separating Axis Theorem

In two dimensions the Separating Axis Theorem is a very intuitive way to determine if two polygons are intersecting. Simply put, the theorem states that if you are able to draw an axis (line) that separates two polygons, then they can't be intersecting (Chong 2012). In practice however, this is done by projecting the polygons onto an arbitrary axis and seeing if their projections overlap. If there is an axis where these projections don't overlap then we can say that the polygons are not intersecting.

The main problem is how we determine the axes to check, as we want to be as efficient as possible. In 2D space the axes to check are those parallel to the normals of all the edges of both polygons, if multiple edge normals are parallel to each other then only one of these needs to be checked. This will happen when a polygon has parallel edges with itself or the other polygon in question. Figure 5, shows an overlap between two squares in one axis but none in the other, so as we can see can't be intersecting. In this case the maximum number of axes that need to be checked is 4 (2 normals for each square, due to their parallel edges).

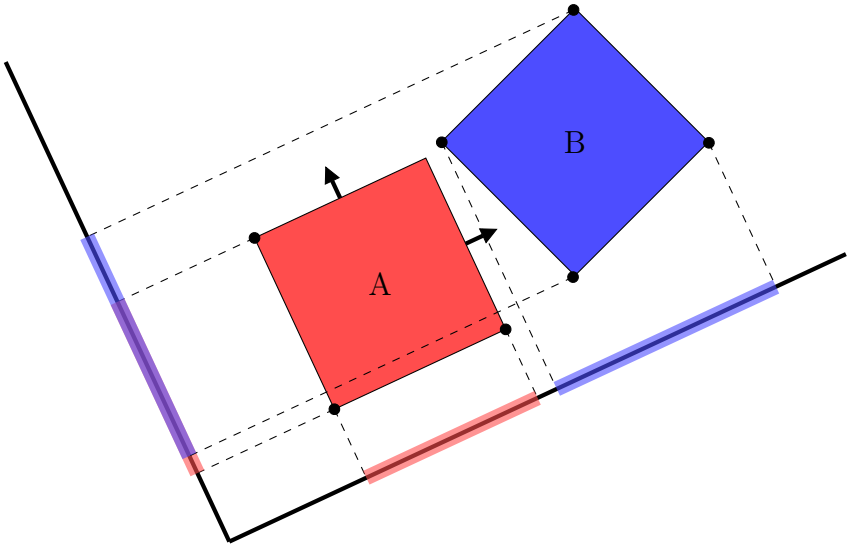

Figure 5: Separating Axis Theorem for two rotated cubes.

In cases with more complex polygons the number of axis to check can increase significantly, with fewer parallel normals and more edges to check. This can be seen in figure 6, where a maximum number of 8 axes may need to be checked.

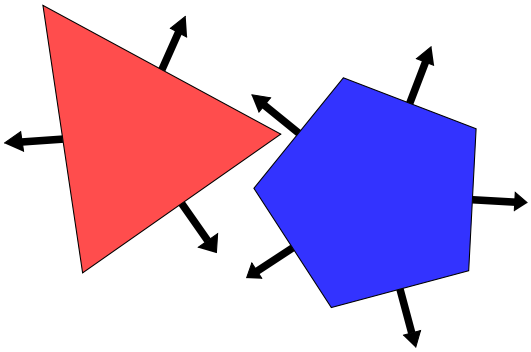

Figure 6: The normals of a triangle and pentagon to be used as axes in separating axis theorem.

The issue with this method to determine if two polygons are intersecting is that it only works accurately with convex polygons (a polygon where all interior angles are no greater than 180° (Math Open Reference 2011)). A false positive result can be seen in figure 7, where due to one of the polygon's concave nature, for all of the four unique axes parallel to the polygons' normals, the projections overlap.

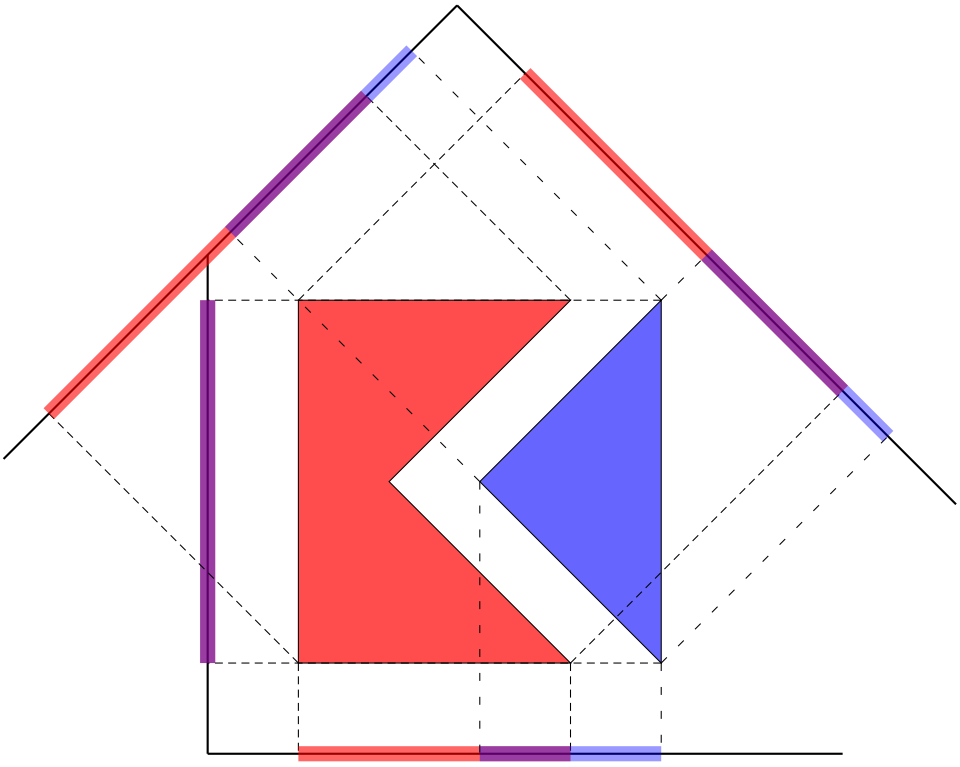

Figure 7: False positive intersection detection of a concave and convex polygon.

For three dimensions the approach is very similar but has a few differences, mainly being the method to determine the axes to project onto. Firstly we check the axes parallel to the normals of the faces, then check the axes parallel to the cross product of every pair of edges (i.e. each edge of polygon A, with each edge of polygon B). As with two dimensions, any duplicate axis wouldn't need to be checked.

2.2.3 Gilbert-Johnson-Keerthi Distance Algorithm

The Gilbert-Johnson-Keerthi distance algorithm is able to determine the minimum distance between two polygons. As with SAT, these polygons must be concave for the algorithm to work. GJK generates a simplex in the Minkowski space of the two polygons using support points, then determines the closest point of the simplex to the origin. If the origin is enclosed in the simplex, then the entities are intersecting.

Simplices

A simplex is the generalisation of a tetrahedral region of space to n dimensions (Wolfram Math World 2020). In the GJK we will generate simplices of $n = [0..3]$, giving us a point, line segment, triangle and tetrahedron in each dimension respectively, see figure 8.

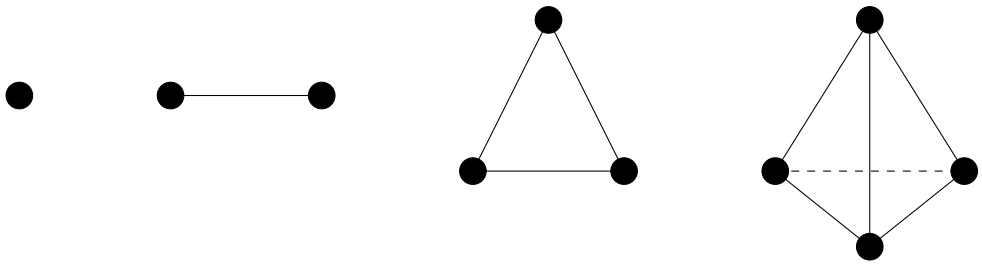

Figure 8: Simplices in 0, 1, 2 & 3 dimensions.

Supporting Points

A support point is the furthest point on a convex shape in a given direction. To find this point you start at an arbitrary vertex, $\vec{p}$, and calculate the dot product with the direction $\vec{d}$. You then cycle through every vertex in the shape, repeating the process and keeping a note of the largest value. Once every vertex has been checked, the largest dot product value is your furthest point.

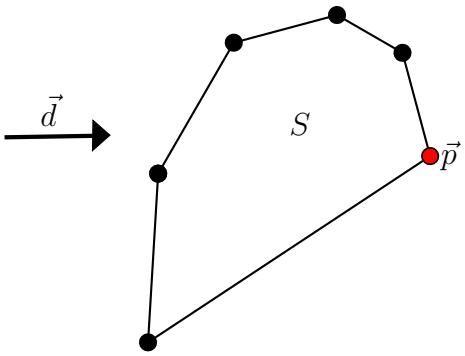

Figure 9: Diagram showing the supporting point, $\vec{p}$, of a convex shape, S, in the direction, $\vec{d}$.

A more efficient method is possible if the adjacency data for the vertices is available. Knowing which vertices are neighbours will allow us to use hill climbing to find the support point, as can be seen in figure 10.

1. Pick an arbitrary vertex, $\vec{p}_{current}$ as the initial solution and calculate it's dot product in the chosen direction, $\vec{p}_{current} \cdot \vec{d}$.
2. Repeat:
    - 2.1 Calculate the dot product of $\vec{d}$ with every neighbour vertex to the current vertex.
    - 2.2 Get the best neighbour of the current vertex, $\vec{p}_{best\_neighbour}$ (neighbour with the largest dot product).
    - 2.3 If $\vec{p}_{best\_neighbour} \cdot \vec{d} \leq \vec{p}_{current} \cdot \vec{d}$ dot product of current
        - 2.3.1 Return $\vec{p}_{current}$
    - 2.4 $\vec{p}_{current} = \vec{p}_{best\_neighbour}$

Figure 10: Hill climbing approach to finding the support point of a convex shape.

The support point function operates as simply in 3-dimensions, requiring no additional steps, just all vectors are now 3D as well.

Minowski Sums & Differences

The Minkowski Sum of two shapes A and B is the set consisting of the sum of each point in A with each point in B, as can be seen in this formula (Harrison 2012);

$$A \oplus B = \{a + b \,|\, a \in A, b \epsilon B\} \tag{3}$$

The Minkowski difference is similar to the sum, however instead of adding every point of B to A, we subract every point of B away from every point in A, as can be seen in this formula (Harrison 2012);

$$A \oplus B = \{a - b \,|\, a \in A, b \epsilon B\} \tag{4}$$

The Minkowski difference has a property which is extremely useful for collision detection, and is used by the GJK distance algorithm. In two dimensions, if two shapes are intersecting then the Minkowski difference will include the origin, $(0, 0)$. This is trivial to understand as we know if two shapes are intersecting then they must share at least one point, p. Therefore, we will have a value in Minkowsi difference equal to $p - p = 0$. An example of the Minkowski difference for intersecting and non-intersecting shapes can be seen in figures 11 & 12 respectively, where the intersecting shapes create a new shape that that engulfs the origin.

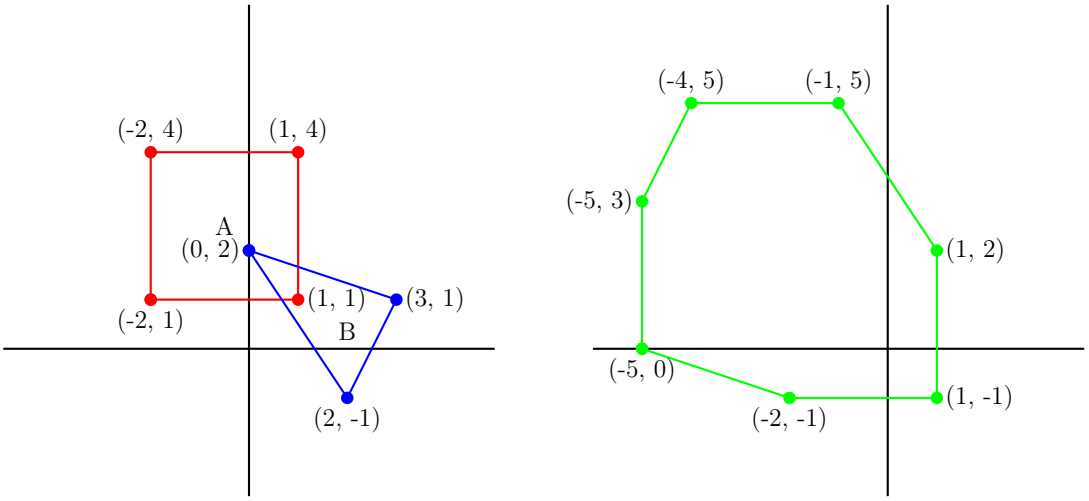

Figure 11: The Minkowski difference of two intersecting shapes.

The issue with calculating the Minkowski difference using the given formula is there's an infinite number of points within both shapes A and B. However, it is not necessary to find every point in Minkowski difference, just the vertices. This can conveniently be done using the support function, where a support point in a given direction on the Minkowski difference is equal to the support point of A in that direction minus the support point of B in the opposite direction. This can be written in equation form as follows:

$$S_{\text{minkowski}}(\vec{d}) = S_A(\vec{d}) - S_B(-\vec{d}) \tag{5}$$

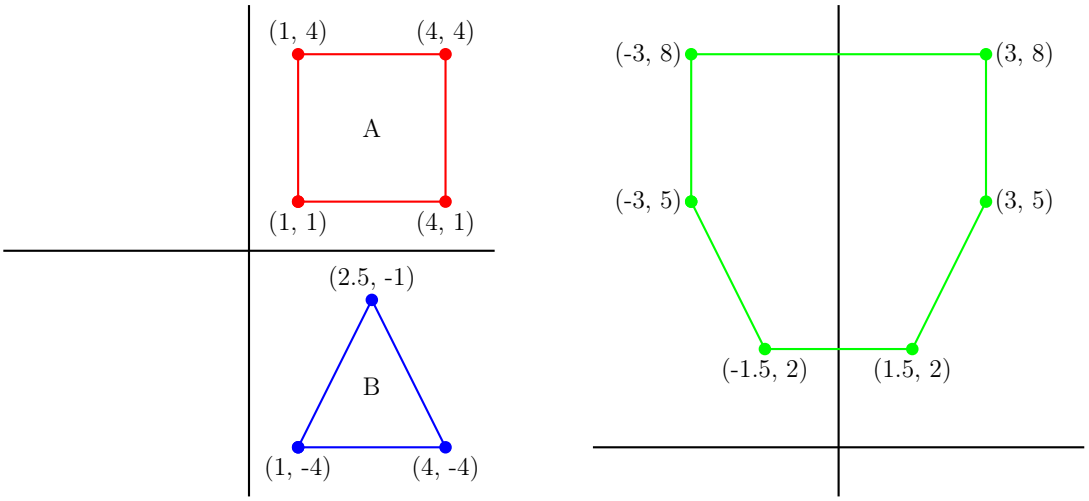

Figure 12: The Minkowski difference of two non-intersecting shapes.

The Minkowski difference, as the support function is, can be simply done in 3-dimensions. The Minkowsi difference of two 3-dimensional convex polygons is just a new convex polygon that will encompass the origin if the original polygons are colliding.

The Algorithm

Now that we understand the ideas of support points, simplices and the Minkowski difference, we can look at how the GJK distance algorithm works (figure 13).

1. Generate a support point, $\vec{s}$, for the Minkowski difference in an arbitrary direction, $\vec{d}$.
2. Push this point onto the simplex.
3. Change the search direction to $-\vec{s}$, i.e. $\vec{d} = -\vec{s}$
4. Repeat:
    - 4.1 Generate a new support point, $\vec{s}$, in the current search direction.
    - 4.2 If $\vec{s} \cdot \vec{d} \leq 0$
        - 4.2.1 Return false, there is no intersection.
    - 4.3 Push this point to the simplex
    - 4.4 If the current simplex is a 4-simplex and the origin is contained within it.
        - 4.4.1 Return true, there is an intersection.

Figure 13: The basic steps of the GJK distance algorithm.

2.3 Contact Point Generation

It is no good knowing that two shapes have collided if we aren't able to determine where the collision took place. We want to be able find out where on the edges (or surfaces) of the two shapes did the collision occur, how deep the penetration between the two shapes is when collision takes place and the direction normal to the collision.

2.3.1 Expanding Polytope Algorithm

EPA requires the support mapping of the polygons involved in the collision, as well as the support mapping of the Minkowski difference of these polygons. It also requires

an initial polytope, which it will use to attractively search for the closest face to the origin. This face can then be used to extrapolate contact data. A simple breakdown of the steps of the algorithm can be seen in figure 14.

1. Generate an initial polytope.
2. Repeat:
    - 2.1 Pick the closest face on the polytope to the origin.
    - 2.2 Find the support point of the Minkowski difference in the direction of the closest face's normal.
    - 2.3 If this point's distance $\leq$ closest face's distance from the origin.
        - 2.3.1 Use this face to extrapolate the contact data.
    - 2.4 Expand polytope to include this new point.

Figure 14: The basic steps of the Expanding Polytope Algorithm in 3-dimensions, modified from the steps given by Tyndall (2014).

The algorithm is much easier to visualise in 2-dimensions, where instead of looking at faces, we look at line segments. We would start with the polytope being a triangle connected to 3 vertices of the Minkowski difference, as seen in figure 15.

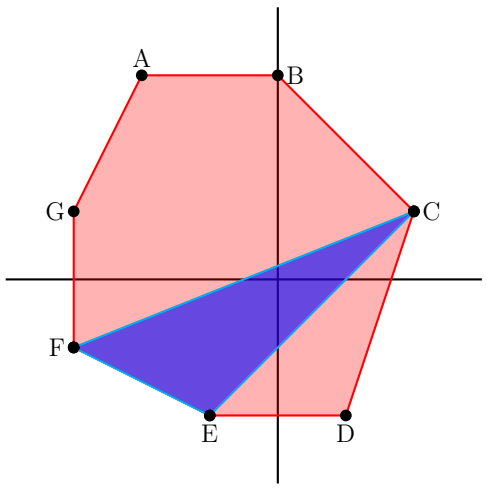

Figure 15: An initial polytope within the Minkowski difference of a pair of colliding polygons.

We would then check every segment to find the one closest to the origin. As we can see in figure 16, the segment closest to the origin is CF, and the support point in the direction of CF's normal is A.

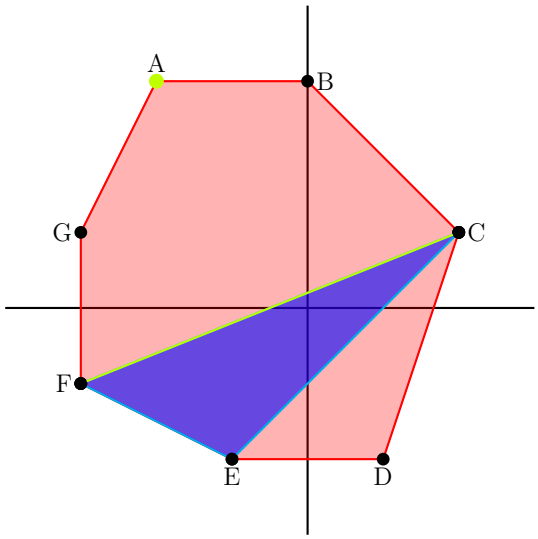

Figure 16: Closest line segment to the origin of the initial polytope, and support point in the direction normal to it.

The polytope will then be expanded to include this new point and the process repeated, until the point is no further from the origin than the segment, as seen in figure 17.

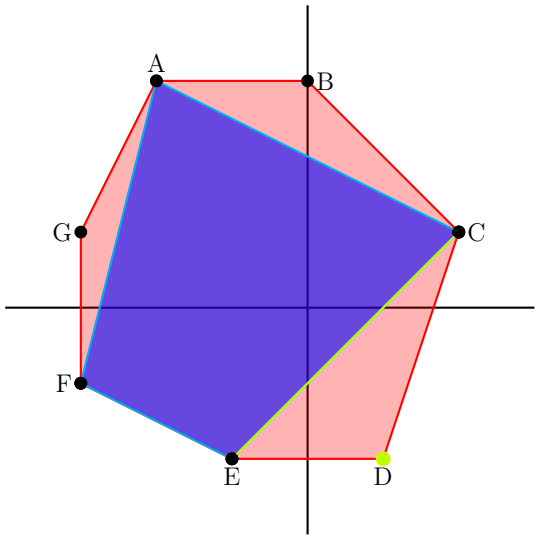

Figure 17: Expanded polytope, with a new closest line segment to the origin, CE, and new support point being D.

After expanding the polytope to include D, we find that the new closest segment is CD, which is on the exterior of the Minkowski difference, so our support point ends up at D, as seen in figure 18. D is one of the points on the line segment so is considered no further from the origin than the segment itself. Therefore we can take the segment CD and use it to extrapolate the contact data.

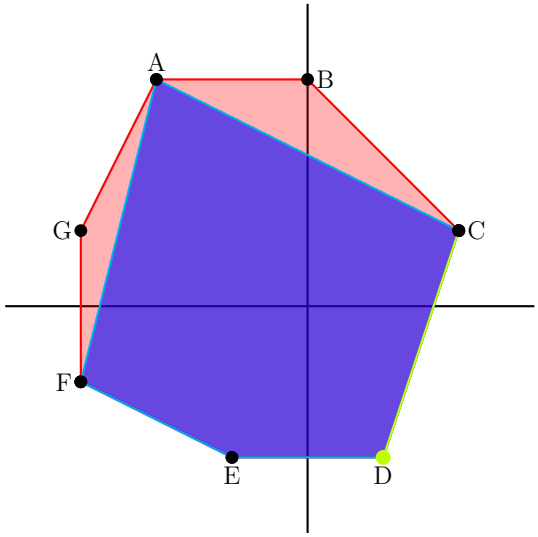

Figure 18: Expanded polytope even further, with a new closest line segment to the origin, CD, and support point being D again.

Extrapolating Contact Data

Now we have the line segment, if we project the origin onto it, along the segment normal, and determine the barycentric coordinates of this point, we can transform from the Minkowski difference space to world space. This can be done by linearly combining the coordinates with the support point of polygon A, that make up the support points of the line segment. This gives us the point of contact of the collision, with the normal to the collision being the negative of the normal, and the penetration depth being the distance from the line segment to the origin.

Summary

The Expanding Polytope Algorithm is a great partner to the GJK distance algorithm, as GJK provides exactly what EPA needs. Using the simplex already generated by GJK as the initial polytope, and its support point mapping, EPA can efficiently generate the contact data in real time (Tyndall 2014).

2.4 Collision Response

Once you are able to determine the points of contact, and the normal to collisions, you need to be able to determine what the colliders should do next to accurately simulate real life interactions. One of the most common ways is to calculate the impulse involved in a collision. Once you have this impulse you can calculate then change in momentum, and angular momentum, using this to update the body's position and rotation in space each frame.

2.4.1 Reaction Force

When a collision between two bodies occurs an impulse, J, is generated in the direction to the normal of the collision. Where impulse is a force applied over a period of time equal to the change in momentum, as given by equation 6.

$$J = \int F\,dt = \Delta p = m\Delta v \tag{6}$$

The coefficient of restitution, e, between the two bodies is also key to determining the impulse applied. Where, e is the ratio of the relative velocity after and before the collision occurred, as equation 7 shows.

$$\text{Coefficient of Restitution (e)} = \frac{v_a - v_b}{u_a - u_b} = \frac{v_r}{u_r} \tag{7}$$

The change in velocity of the system relative to the point of collision is equal to $\Delta v = -(v_f + v_i)$, where v_f and v_i are the relative final and initial velocities of the system. This can be rewritten in terms of, e, and initial relative velocity as:

$$\Delta v = -(v_i e + v_i) = -(e+1)v_i. \tag{8}$$

However we are only actually interested in the velocity that is in the direction of the collision normal $\hat{n}$. To find the magnitude of velocity change along the normal we just dot Δv and $\hat{n}$, giving us (Newcastle University 2020):

$$v_j = \Delta\vec{v} \cdot \hat{n} = (e+1)\vec{v}_i \cdot \hat{n} \tag{9}$$

If we then take this velocity and divide it by the sum of the bodies' inverse mass, we will get the magnitude of the impulse for the collision.

$$J = \frac{v_j}{m_a^{-1} + m_b^{-1}} \tag{10}$$

As force, assuming it is constant, is equal to the impulse divided by the time it was applied for, we get a force on our bodies equal to:

$$R = \frac{J}{dt} = \frac{(e+1)\vec{v}_i \cdot \hat{n}}{m_a^{-1)+m_b^{-1}}} \cdot \frac{1}{dt} \tag{11}$$

2.4.2 Torque

As well as a linear force applied to the bodies during a collision there is also a rotational force. This rotational force is known as torque. Torque is calculated by taking the cross product of the distance vector, $\vec{d}$, and the force being applied $\vec{F}$, (12), where the distance vector is the distance between the bodies center of mass and where the force is being applied.

$$\tau = \vec{r} \times \vec{F} \tag{12}$$

2.4.3 Frictional Force

When the coefficient of friction, μ between a collision is greater than 0, it means there will be a resistive force, $\vec{F}$, tangential to the collision. The size of this force is proportional to the reaction force, $\vec{R}$, by the factor of μ, as seen in equation 13.

$$\vec{F} = \mu\vec{R} \tag{13}$$

2.5 Dynamics

Once you know the forces acting on a body, you are able to calculate its motion. This is done through integrating the forces to get velocity and angular velocity, then integrating that to get our change in position and rotation. Firstly there are two ways we can calculate our change in position, starting from $F = ma$ and $F = \frac{dp}{dt}$ (Fiedler 2004).

Looking at the first equation, to get to position, we rearrange for acceleration, integrate to get velocity then integrate again to get distance moved:

$$a = \frac{F}{m} \longrightarrow v = \int a\,dt \longrightarrow x = \int v\,dt \tag{14}$$

For the second equation, we integrate to get impulse, we add this impulse to the current momentum, divide the new momentum by mass to get velocity, then integrate velocity to get distance moved:

$$J = \int F\,dt \longrightarrow p = p + J \longrightarrow v = \frac{p}{m} \longrightarrow x = \int v, dt \tag{15}$$

2.6 Explicit Euler Integration

Explicit integration is where the next system state is generated solely from the current state. Explicit Euler works by first calculating the new position by adding the change in position over the time step Δt, then calculating the new velocity by adding the change in velocity.

Explicit Euler gives decent accuracy when the time step is small enough and acceleration is constant, however, if acceleration isn't constant issues can arise. This can be seen when trying to simulate a damped harmonic oscillator, where instead of damping and converging to zero, it gains energy (Fiedler 2004).

2.6.1 Semi-Implicit Euler Integration

There is one simple difference between Semi-Implicit and Explicit Euler, being the order in which position and velocity are updated. So instead of position, then velocity, Semi-Implicit Euler updates the body's velocity first, using this updated velocity then calculates the body's change in position.

Even though semi-implicit has the same order of accuracy as explicit Euler, it performs much better, like with damped harmonic oscillators, and in general when integrating equations of motion (Fiedler 2004).

3 Project Specification

3.1 Analysis and Formulation

The requirements specification was formalised by examining the features of the Bullet Physics engine (Erwin Coumans 2020), and evaluating what could be reasonably accomplished in the time frame. Through doing this the conclusion was made to focus on simulating rigid body dynamics of convex objects.

3.2 System Overview

The system will use a Graphics API to allow for control over the implementation of the graphics pipeline, as to provide only the minimal requirements needed to render basic 3-dimensional polygons to the screen. Collision detection will be achieved through 2 steps, a fast acting broad phase, to determine if a pair of entities are potentially colliding. Then the more computational, but accurate, narrow phase will check these potentially colliding pairs for actual collisions. Contact data will then be generated from the collisions, and used along with entity data, to calculate collision impulses. These forces will be applied to entities involved in collisions to determine their movement in the scene, and update their position and rotation each frame.

3.3 Functional Requirements

1. Rendering

    1.1 Window: The system must create a window to display the rendered scene when run.

    1.2 Input: The user must be able to input files to create and texture meshes in a scene.

    1.3 3-dimensional Scene: The system should be able to render textured 3-dimensional meshes onto the screen.

2. Entity Properties

    2.1 Mesh: It must be possible to add mesh data to an entity to use during rendering.

    2.2 Textured: It must be possible to add texture data to an entity to use for texture mapping.

    2.3 State: It must be possible to store an entities position, rotation and scale in the scene.

    2.4 Static Collidable: It must be possible to make an entity that can't move but can still influence the physics of other entities.

    2.5 Dynamic Collidable: It must be possible to allow an entity to be influenced by physics.

    2.6 Controllable: It must be possible to be able to control an entities movement with the mouse and keyboard.

    2.7 View: It must be possible to view the scene from the perspective of an entity (camera).

3. Entity Operations

    3.1 Create: The user must be able to create entities.

    3.2 Add Property: The user must be able to add property values to an entity.

    3.3 Remove Property: The user must be able to remove any property from an entity.

    3.4 Clear Entity: The user must be able to remove all properties from an entity.

    3.5 Get Property: The user must be able to retrieve a chosen property from an entity.

    3.6 Has Property: The user must be able to query if an entity contains a given property.

    3.7 Get ID: The user must be able to retrieve the unique ID of a given entity.

    3.8 Delete: The user must be able to delete an entity removing it from the system.

4. Collisions

    4.1 Detection: The system must be able to detect collisions of entities.

    4.2 Response: The system must be able to produce a realistic collision response.

5. System Controls

5.1 Mouse: The system must be able to recognise mouse inputs, (movement, scrolling, clicking).

5.2 Keyboard: The system must be able to recognise keyboard key presses.

5.3 Keybinds: The user must be able to set a function to any mouse or keyboard input.

5.4 Pause: The user must be able to pause the physics while still rendering the scene. i.e. all entities should stop moving.

3.4 Non-Functional Requirements

1. Performance

    1.1 Real-Time: The system should be able to perform in real-time with 60+ fps.

    1.2 Rendering Load: The system should be able to render 100 of static entities without frame stuttering.

    1.3 Collision Load: The system should be able to process the collision detection and resolution for at least 10 entities without significant performance loss.

    1.4 Consistency: Simulation with given initial parameters should consistently produce the same results.

2. Usability

    2.1 File types: The input file types taken accepted be .obj and .png.

    2.2 Intuitive: Creating an application should be intuitive to the user.

    2.3 Moving the camera: The user should be able to move the camera through the scene by using WASD keys, as standard in video games.

    2.4 Turning the camera: The user should be able to control the yaw and pitch of the camera through moving the mouse left/right and up/down respectively.

3. Extensiblity

    3.1 Integrating a GUI: Should be able to integrate a user interface in the future, that would allow users to place entities into a scene without the need for coding.

    3.2 Combining engines: Should be able to combine with different engines that provide functionality needed for a complete game engine e.g. audio & AI.

    3.3 Physical systems: Should be able to integrate the ability to simulate other types of physical systems i.e. soft body dynamics

3.5 Use Case Diagram

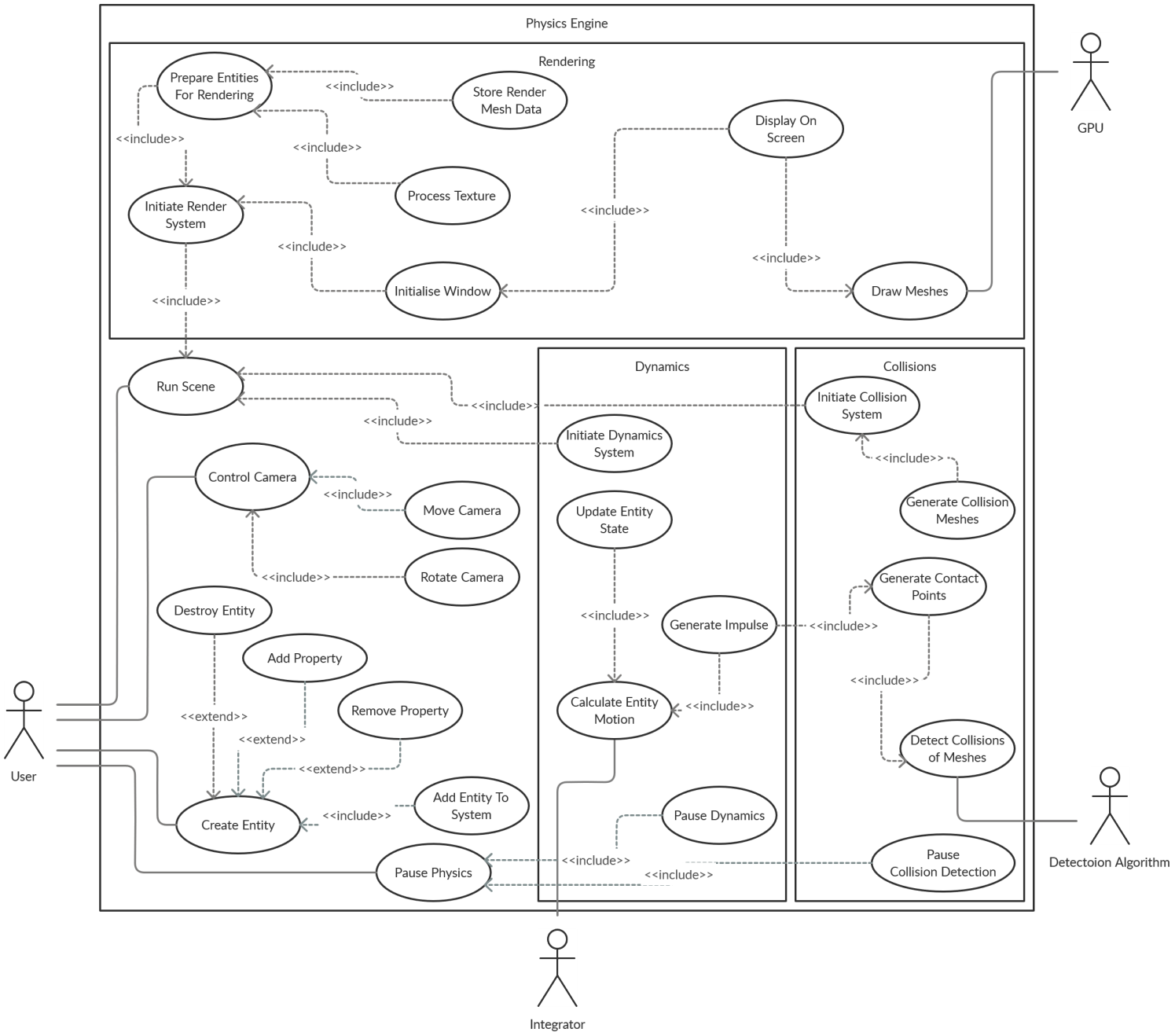

Figure 19: Use Case Diagram showing the core functional requirements.

3.6 Activity Diagrams

3.6.1 Rendering a Cube

In figure 20 we see the process the system should take to render a static cube to the screen. This process is broken down into the 4 swimlanes: User, Display, Renderer and Entity Manager. We see the user create a new application and add a static cube to the scene, then finally running the application. This triggers the display to create a window to be rendered to. With the rendering system retrieving the cubes data from the entity manager, and passing it through the graphics pipeline, it draws the cube displaying it on screen. Then every frame the screen is cleared and the renderer re-draws the cube.

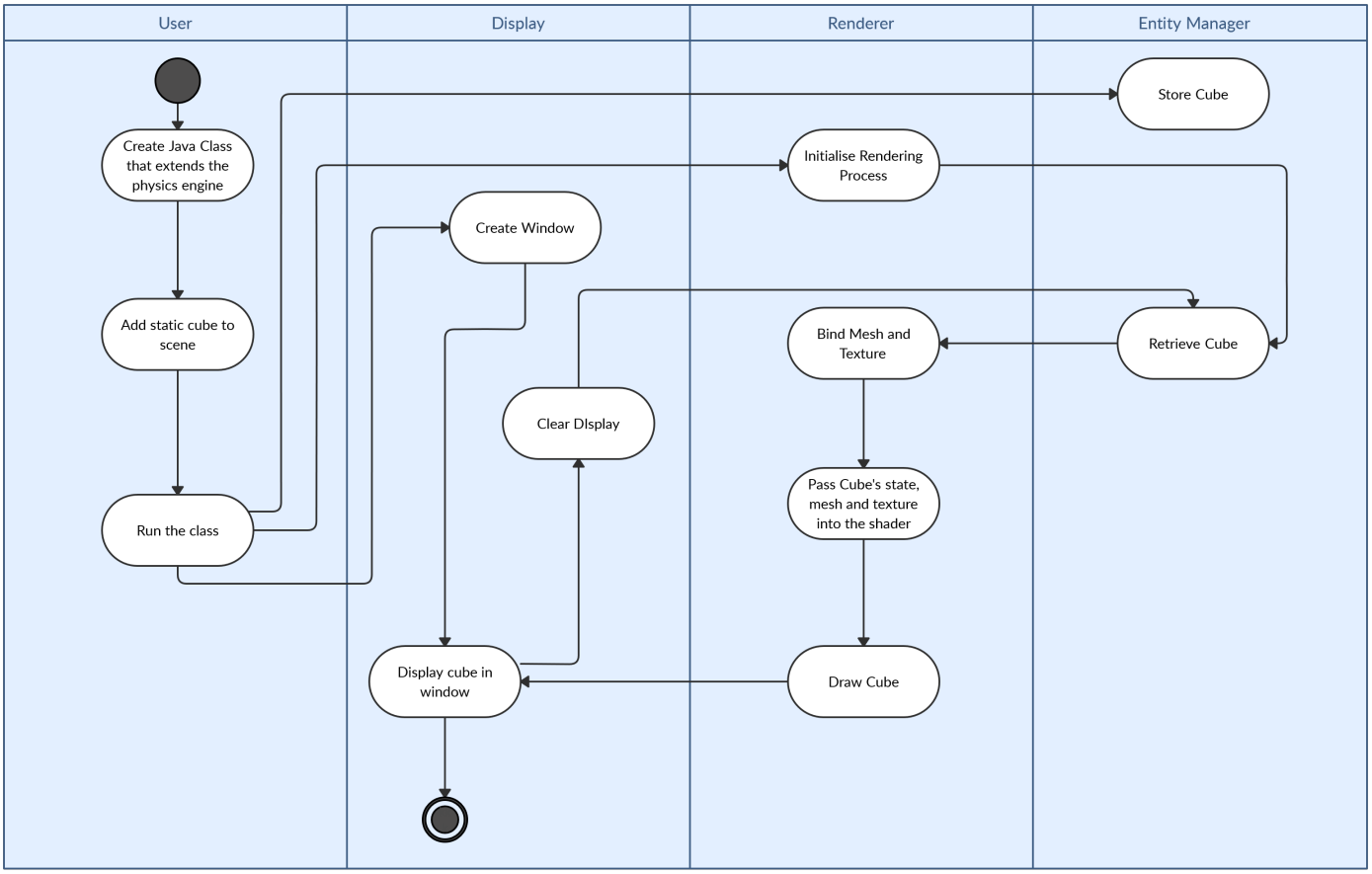

Figure 20: An activity diagram for rendering a single cube to the scene.

3.6.2 Collision Detection Process

In another Activity diagram, figure 21, we see the process of collision detection. The collision detection process will be broken down into two phases. In the broad phase, all

pairs of entities have their axis-aligned bounding boxes checked for intersections, with any intersecting pair getting added to a list of possible collisions. This list is then sent to the narrow phase, where each collision is checked more accurately, with any pair not actually colliding getting removed from the list. This final list is then sent of to the collision response system.

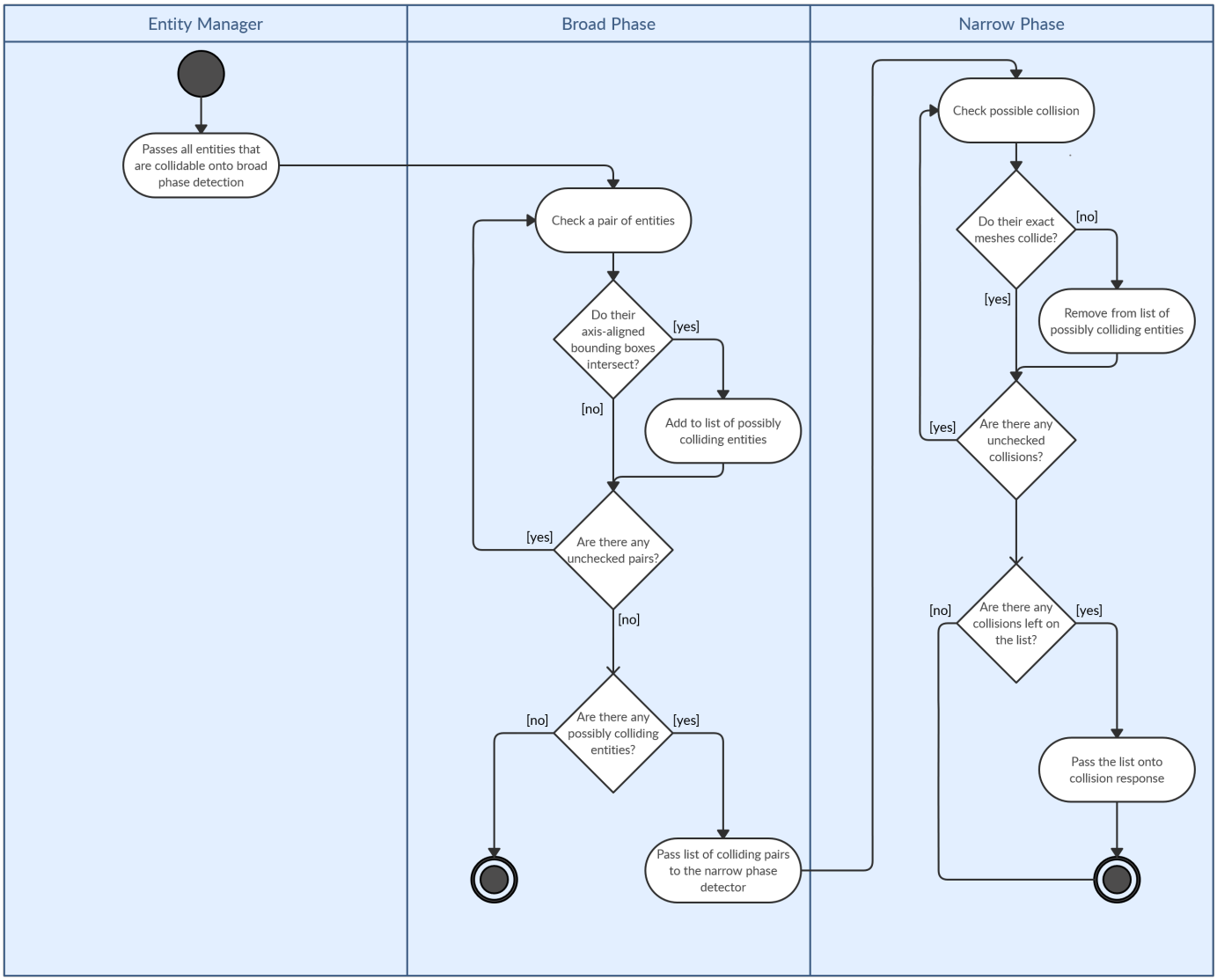

Figure 21: Activity diagram for the process of detecting the collisions of entities.

4 Solution Design

In this section we will look at: the chosen architectural approach taken in developing the software, how the rendering process will be implemented, and the techniques and algorithms that will be used for collision detection.

4.1 Architecture

An Entity-Component-System (ECS) architecture was chosen to be used for the physics engine. An entity is just a container for components, with data contained within these components. This structure follows the principle of composition over inheritance, by having an entity contain instances of different components which then determine the functionality. We then have a range of systems that run independently, performing actions on entities with certain components, i.e. the Render System would iterate through every entity with a mesh and render it to the screen.

The architecture is most commonly used in video game development (Entity Systems Wiki 2014), and has become popular due to its ability to eliminate the problems that can arise in a more inheritance focused approach. With this more traditional inheritance driven OOP approach, all types of entities would lie somewhere on the inheritance chain from the base class. Over time, as new unique entities are implemented, the inheritance tree gets wider and deeper. However, there is a fundamental limitation on inheritance, any given class can only inherit directly from a single parent.

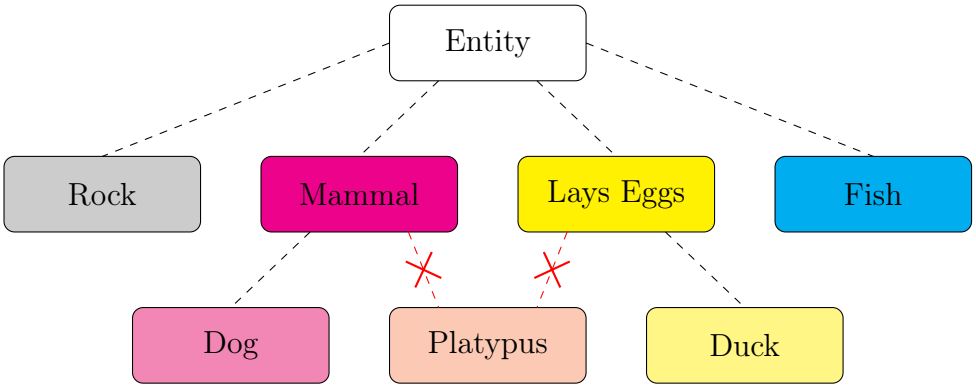

Figure 22: Platypus cannot simply be placed into this inheritance hierarchy, due to the fact it's both a mammal, and lays eggs (Board To Bits Games 2019).

As seen in figure 22, if we were to try and add a Platypus class to the existing inheritance tree we see that due to the fact they are mammals, but also lays eggs, there isn't an intuitive place for it on the tree, due to the single parent limitation. To fit the Platypus into our system would require rearranging of the hierarchy, which is time

consuming. This problem leads to making it very difficult to maintain and extend the structure of the classes in the future as well.

ECS, on the other hand, with the ability for each entity to have a mix of any properties, means we never come across this issue while still practicing code reuses. As seen in figure 23, we would be able to have Dog, Duck, and Platypus entities just by managing the components within each.

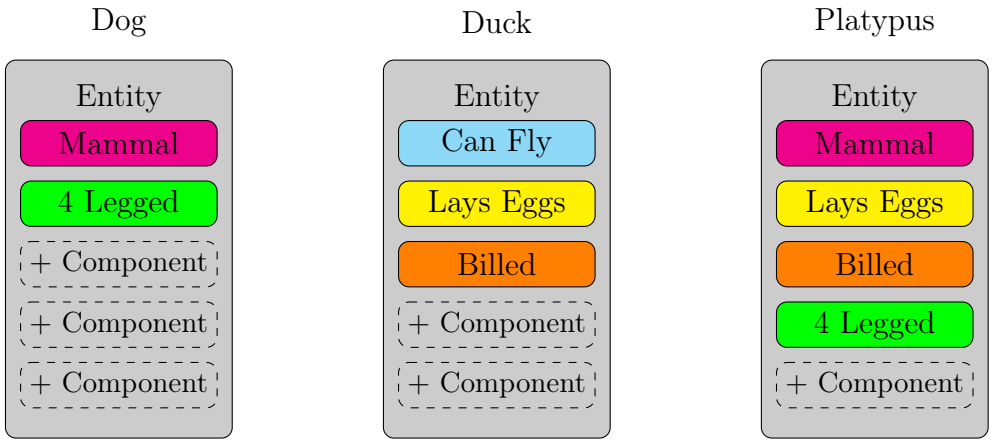

Figure 23: Entities can be built up by adding components, which provide the entity with the data needed to function. We can see that only the relevant components of a Platypus have been added.

This feature would be extremely useful in developing the physics engine as we would be able to have great customisation over the types of entities in the scene, e.g. an entity that renders to the screen but doesn't interact with the physics, or a collidable entity that can't be moved.

What is even more attractive about ECS, is how it uses independent systems to perform the functions on entities based on their component structure. As the systems are independent of one another, it enables the whole program to be developed in stages, helping the flow of the project, with the ability to test each system individually.

4.2 Package Diagram

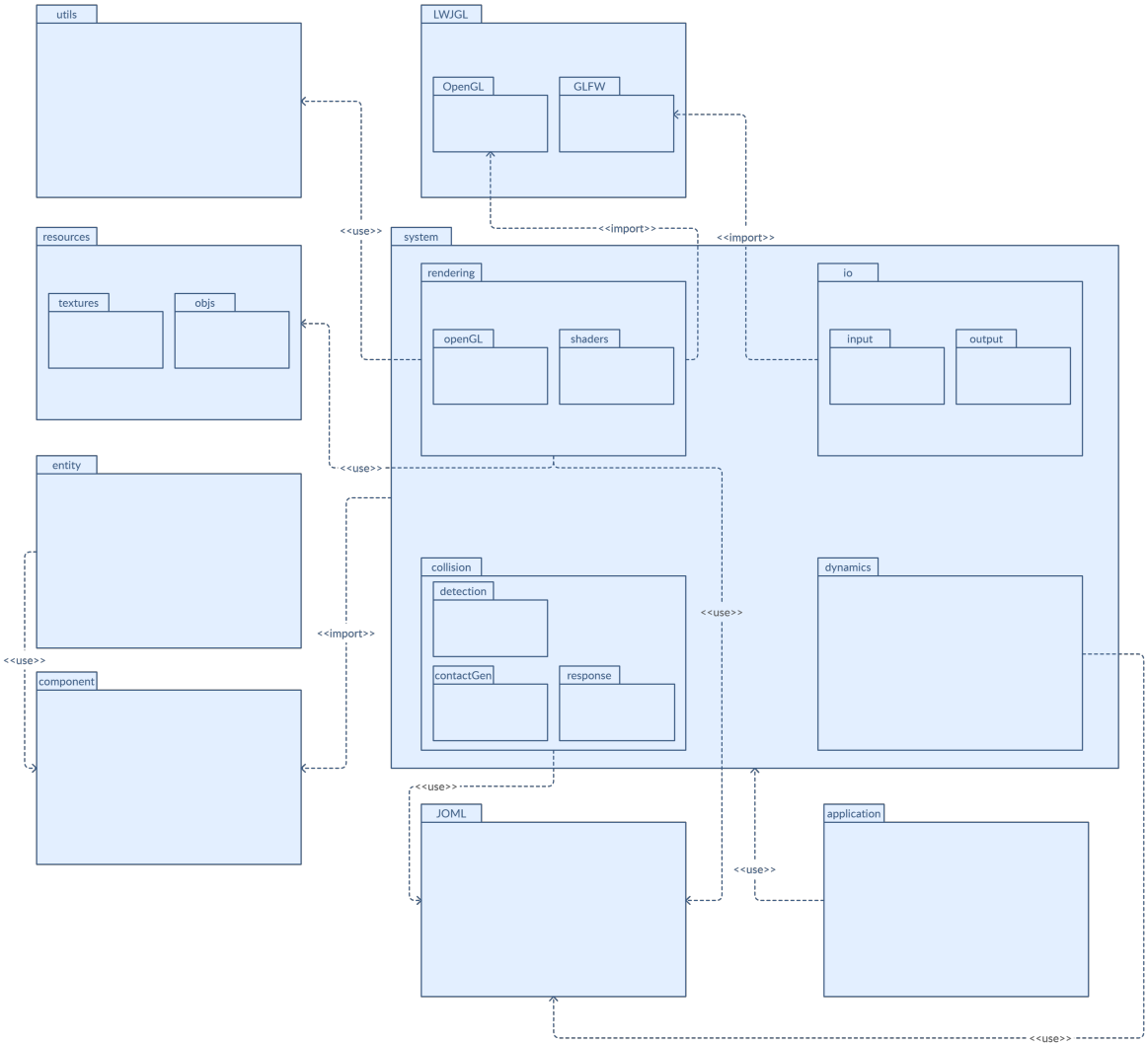

Figure 24: Package Overview of the System.

4.3 Class Diagrams

4.3.1 Simple ECS Class Diagram

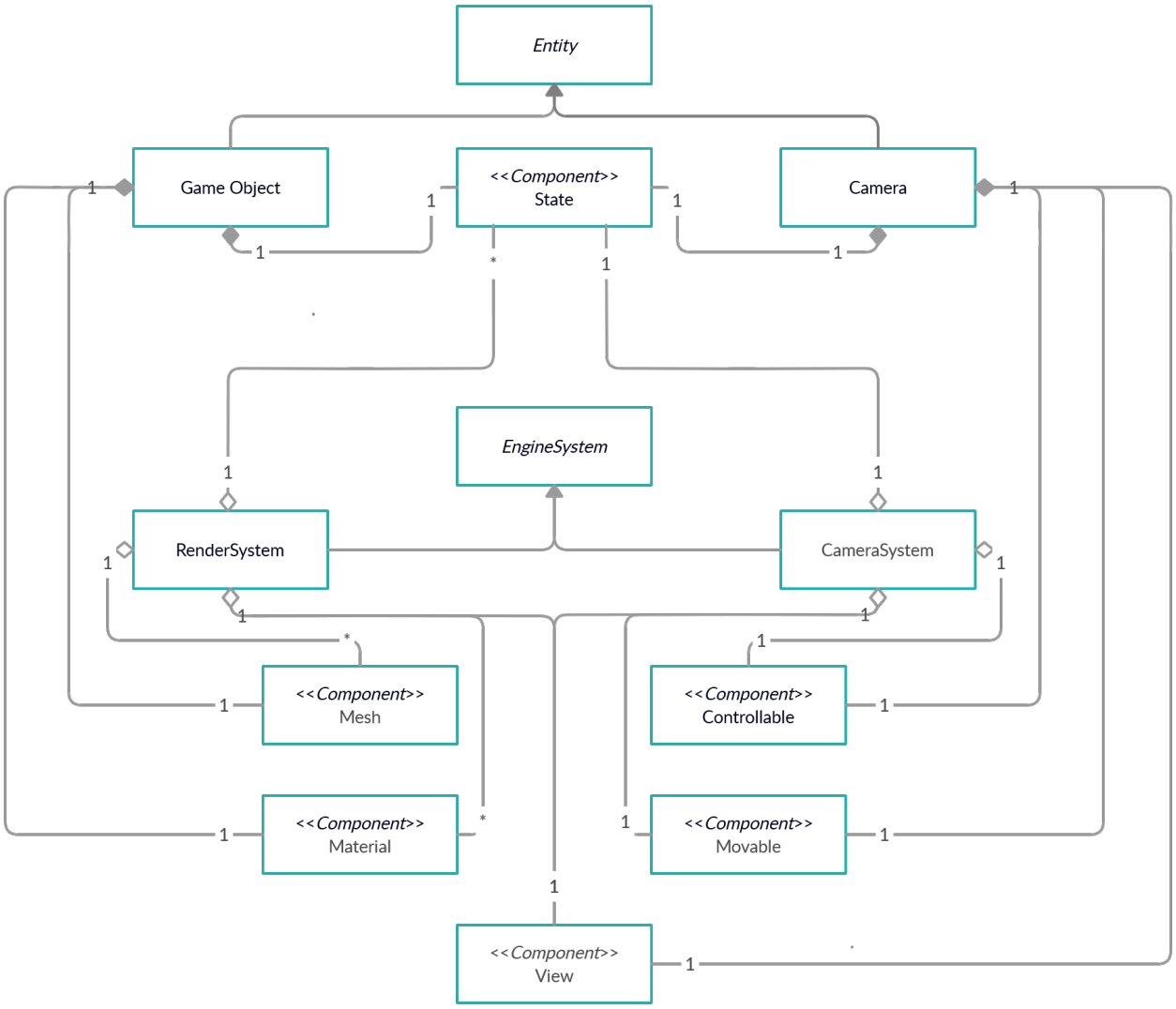

Figure 25: Simple class diagram of a stripped down ECS system.

4.3.2 Rendering System Class Diagram

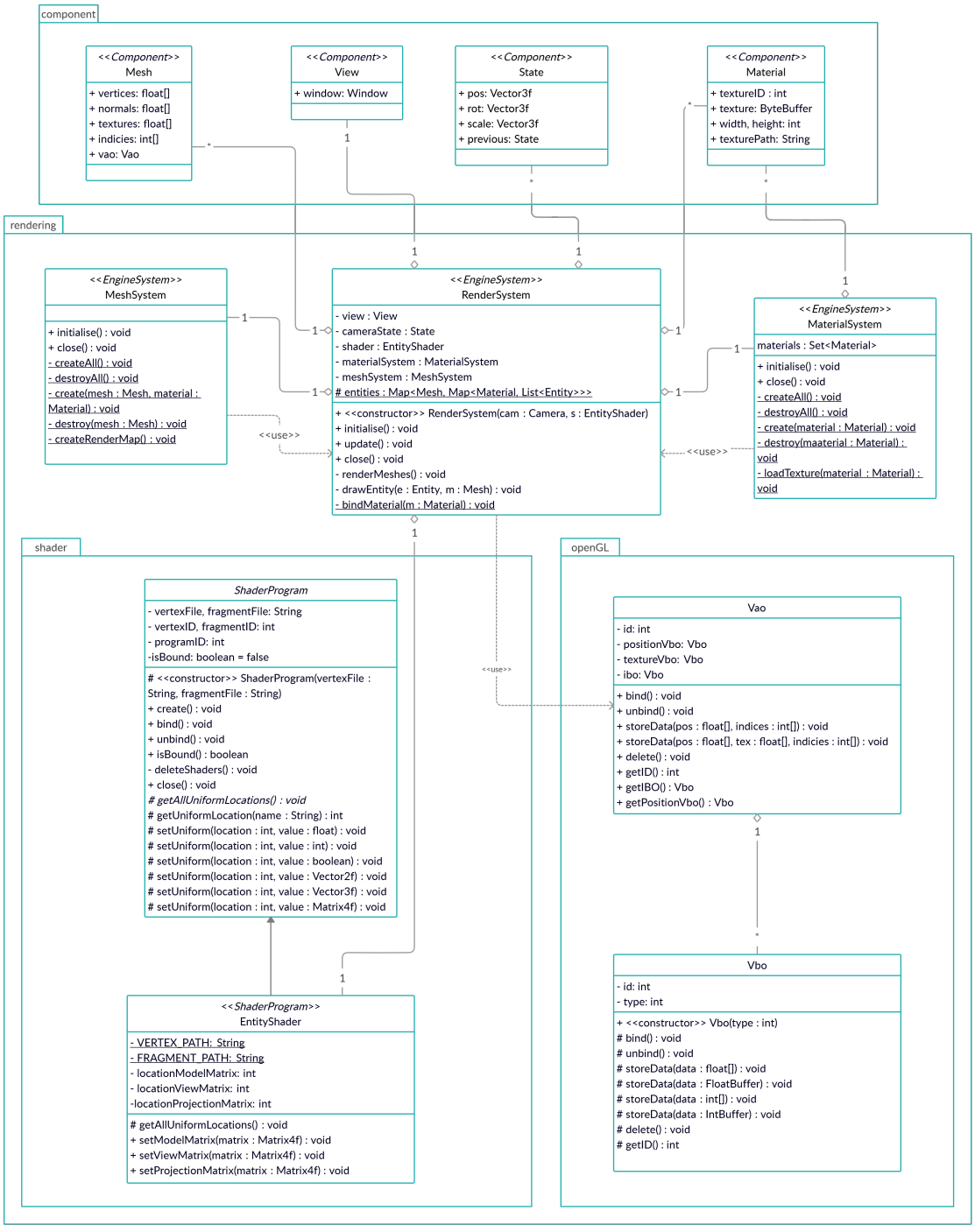

Figure 26: Class Diagram of the Rendering System.

4.4 Dynamics System Class Diagram

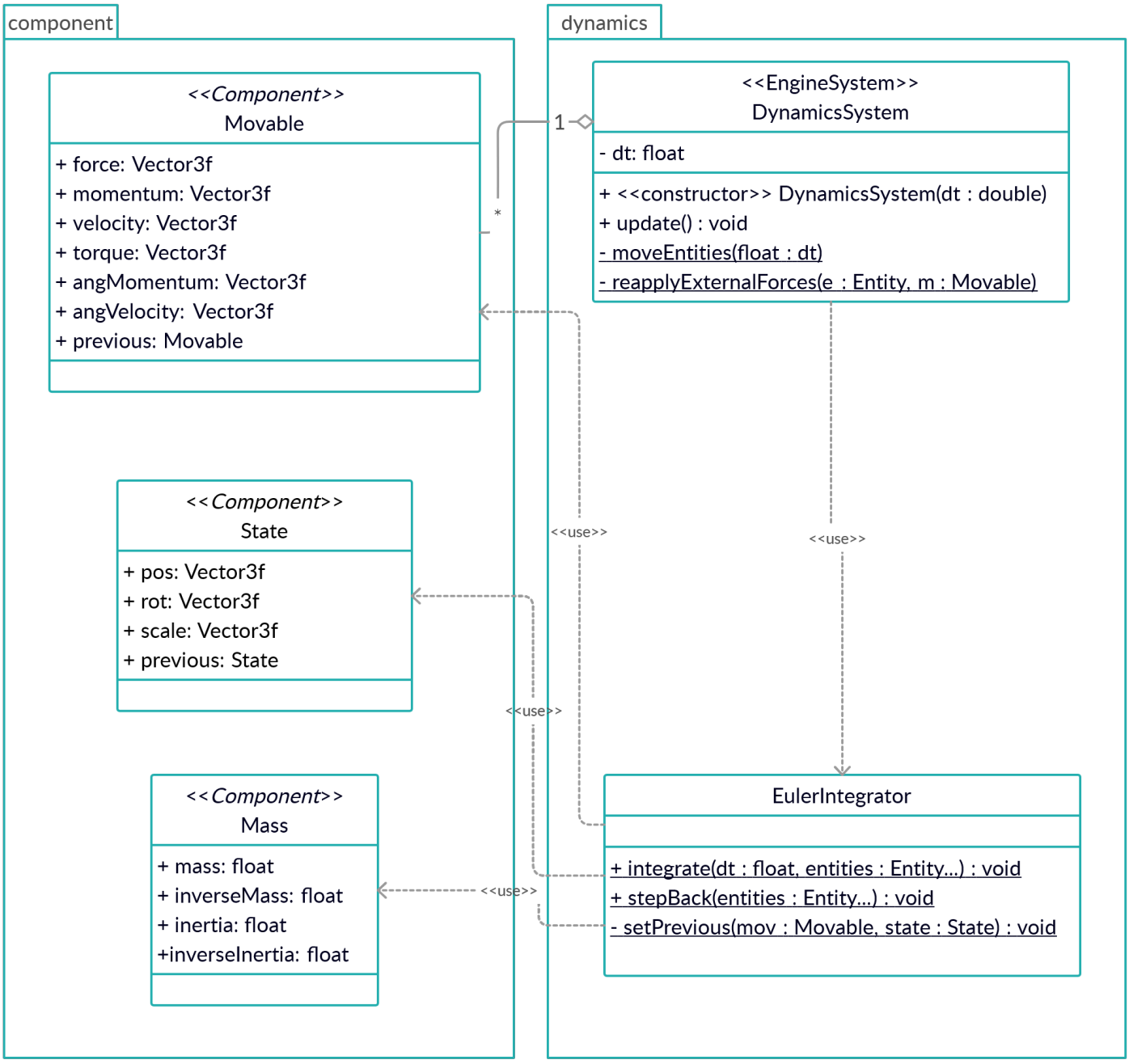

Figure 27: Class Diagram of the Dynamics system.

4.4.1 Collision System Class Diagram

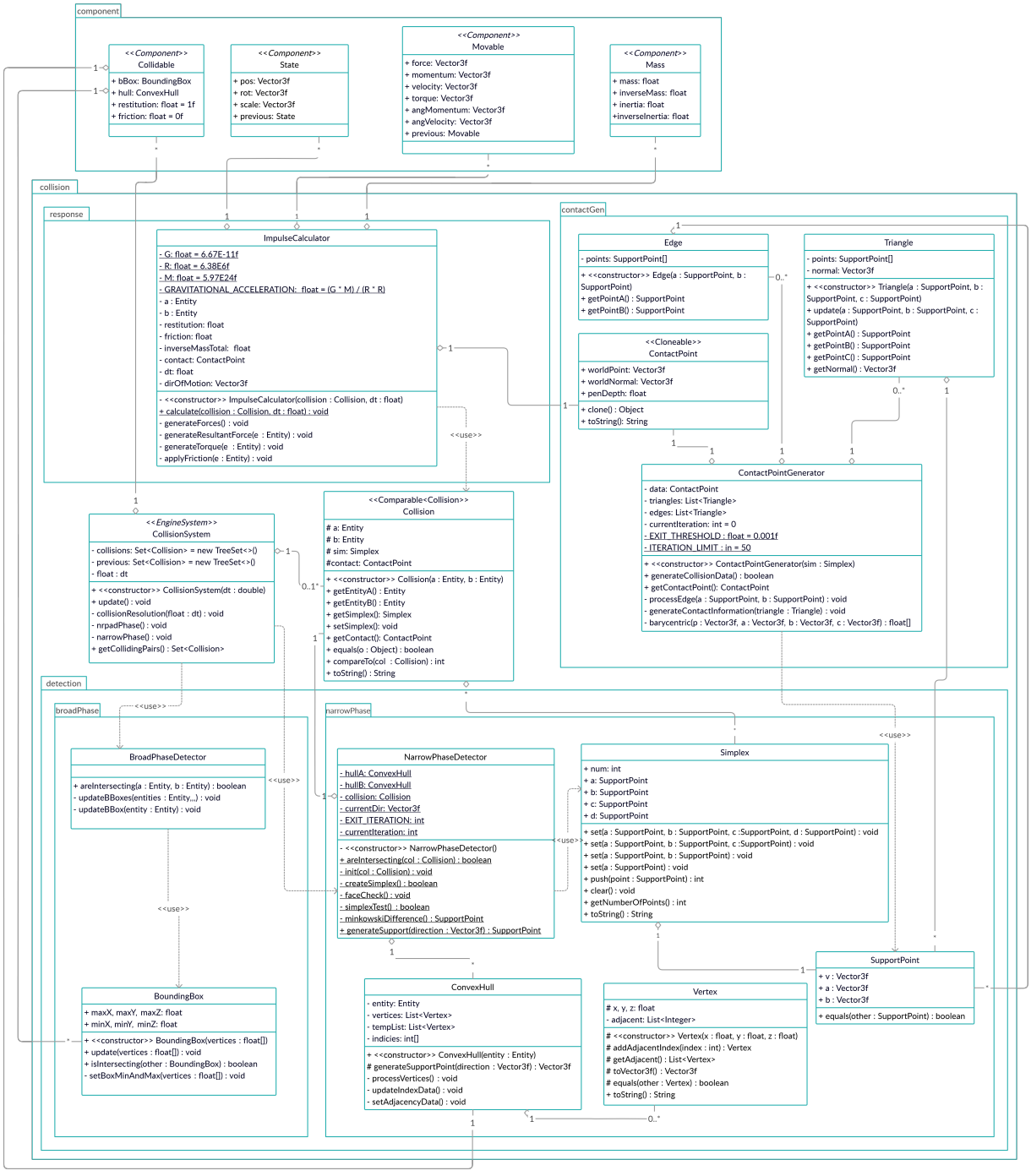

Figure 28: Class Diagram of the Collision System.

4.5 Rendering System

For the rendering system, due to coding experience primarily in Java, it was important that the Graphics API of choice was programmable in Java. The Light Weight Java Game Library's (LWJGL) has an OpenGL binding, which allows the user to control the implementation of the Graphics Pipeline. The OpenGL binding was chose over Vulkan, due to OpenGL being a higher-level API, thus being simpler, giving the user access to only the macro scale system functions.

The file formats to be used for the objects and their textures will be .obj and .png respectively. OBJ being a standard 3D image format that can be exported and opened by various 3D image editing programs (FileInfo 2020).

4.6 Collision System

The collision system will be split up into 3 subsystems, focusing on the detection, contact data generation and the collision response. The process of each of these systems should complement each other to provide the best possible real-time performance.

4.6.1 Detection

For the first broad phase of the detection process the system will use axis-aligned bounding boxes. These boxes are very cheap to store in memory, only needing to store 6 float values (min and max x, y, z coordinates), equating to 24 bytes. It is also really fast to determine if any two boxes are intersecting, due to being aligned to the axes, with simple logic checks, as mentioned back in Section 2.2.1 figure 2. After generating a list of the bounding boxes that are colliding we can go onto the narrow phase.

In the narrow phase we will be using the Gilbert-Johnson-Keerthi distance algorithm. Variations of the algorithm are very commonly used in video games and simulations.

4.6.2 Contact Generation

For contact data generation, the Expanding Polytope Algorithm was chosen due to its complementary nature with the GJK distance algorithm. Being able to use the results from GJK as the start point allows for a fast convergence of the algorithm, and extrapolation of the contact data.

4.6.3 Resolution

The resolution process will take the two entities involved in a collision and their contact data and generate an impulse to apply along the collision normal. Using impulse over energy conservation will allow for a better simulation of multiple bodies colliding

together, due to the ability to apply all the impulses of the collisions together, and then using that work out the motion of the bodies.

4.7 Dynamics System

Semi-Implicit Euler Integration will be used to integrate the forces being applied to the bodies to calculate their change in position and rotation each frame. Using Semi-Implicit Euler is very quick as a first order integration method, only performing simple division and addition to calculate the new state of a body each frame.

5 Implementation & Testing

In this section we will explore how the requirements from section 3, along with the design choices made in section 4 have been implemented to develop the functioning system.

The majority of the project was developed in Java, due to developer experience with the language and its ability to run on range of operating systems. However, the OpenGL Graphics Pipeline Shader Programs had to be written in the OpenGL Shading Language (GLSL), which shares some similarities with C and C++. However, only the simplest functionality was implemented, with plenty of documentation to ensure no significant time was needed to learn the language.

5.1 External Libraries

5.1.1 LWJGL 3

LWJGL 3 is a Java library that provides access to cross-platform APIs for the development of graphics, audio and parallel computing. It enables direct accesses, with high-performance, while also providing a type-safe, user-friendly, wrapping layer (LWJGL 2020). This project uses its low-level binding of OpenGL and GLFW, being responsible for graphics handling and input/output management respectively. The use of the library allowed for the project to be kept solely to Java, ensuring no need to develop a deeper understanding of other languages.

5.1.2 JOML

JOML is the Java OpenGL Math Library, intended for OpenGL rendering calculations in Java. The library provides easy to use, feature-rich and efficient linear algebra operations needed for 3D applications (Burjack 2020). Using this library for this project means that we will be able to skip implementing a Maths Engine from scratch, which in itself would take a reasonable amount of time. The library will also be conveniently used to perform the calculations for entity dynamics.

5.2 Entity-Component Management

With the main architectural structure of the program being ECS, there were decisions that needed to be made about the data structures to use to allow for easy management of entities and their components. The choice of data structures should: limit each entity to only contain a single component of a given type; be able to check if an entity contains a given type of component, and allow for said components to be fetched; and finally allow systems to access groups of entities based on types of components.

With these constraints there is only one real possible contender, and that is maps. With a map we could have the type (class) of component as the key, with the entities instance of that component as the value, written as Map<Class<? extends Component>, Component> in Java. As each key of a map must be unique, this would ensure only one component of each type is stored for each entity, meeting our first constraint. Next, we could use a maps function to check if it contains a given key, and get the value attached to a given key, to determine if our entity contains a given type of component and fetch it, satisfying our second constraint. Finally, for the final constraint we would need to define another map, with component type as the key and a nested map as the value containing all the entities that contain that component and the instance of it.

5.3 GJK-EPA

5.3.1 Implementing

For the implementation of both the GJK and EPA algorithms we adapted an algorithm by Tyndall (2014) written in C++. In Tyndall's approach to GJK, he refrained from generating the complete Minkowski difference of the two entities being checked. Instead he solely used the support function to calcualte the support points on the Minkowski difference to generate the simplex. This allows for minimal upfront computational load, and the benefit of only having to generate the points needed. However the points generated are disposed of if not currently in use (i.e. the points of the simplex), meaning possible need to recalculate points.

A Simplex and SupportPoint class have been implemented to allow for easy management. The Simplex class contains methods for pushing a new point to it, as well as overloaded methods for setting the whole simplex as suggest by Tyndall (2014). The SupportPoint class holding the value on the Minkowski difference along with the values used to calculate it.

For EPA we set up data structures to store the edge and face data of our expanding polytope. With an Edge class storing two support points that form an edge of the polytope, and a Triangle class storing three support points that form a face, along with faces normal. The simplex that GJK terminated with is converted into its faces and stored in a list of triangles. We then expand the polytope until we find the closest face

to the origin on the Minkowski difference. An algorithm, adapted from Tynall's use of an algorithm by Ericson (2004) from his book Real-Time Collision Detection, is used to calculate the barycentric coordinates of this face and then used to extrapolate the contact data.

5.3.2 Optimising

After implementing GJK and EPA, although performance was above expected for just a pair of entities, as more entities were added to the scene there was room for improvement. The biggest and most significant optimisation feature being the addition of enabling the simplex of a collision to be used as the initial simplex in the GJK algorithm in the next frame. As a high frame rate would mean minimal movement between frames, a collision is likely to occur over multiple frames. Enabling the use of the previous simplex, enables us to skip the initial generation to a 4-simplex after the first contact.

The second optimisation feature made was manipulating the vertex data of entities to store the position of their neighbours. By having the adjacency data of the meshes, we would be able to generate support points faster by avoiding randomly checking vertices and instead perform a hill climb algorithm, see figure 10, to find the maximum.

5.4 Validation & Testing

The system was tested through the use of JUnit tests to confirm that: the Entity-Component management was handled correctly, the collision detection system appropriately determined intersections, and the dynamics system accurately integrated entity motion. Additionally the visual examination of simulations were performed to validate results, ensuring that the system was able to run in real-time, and the performance was not effected during simulations involving large numbers of entities.

5.4.1 JUnit Tests

On development on the Entity-Component management system, JUnit tests were made to check whether the functionality for adding removing, and retrieving components was working as intended. Al tests passed and the system was able to manage components stored in entities as well as organise components with component types in common, see figure 29.

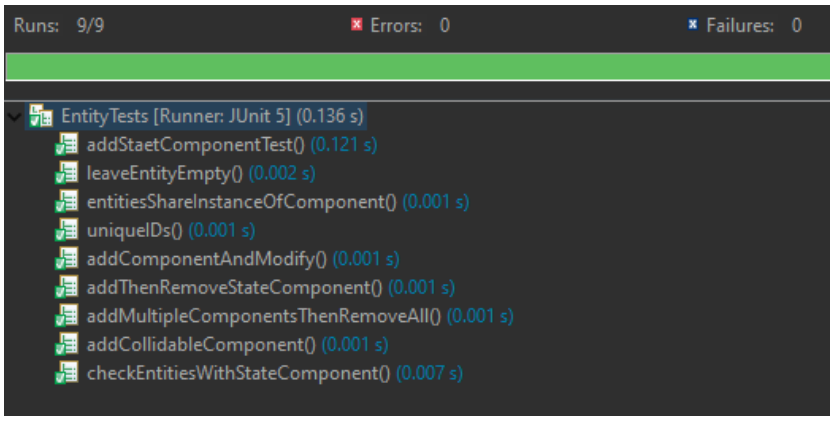

Figure 29: Shows 9 passed tests checking the functionality of the entity-component management.

Then testing was done on the collision detection system, checking the methods for determining intersections between entities in the broad and narrow phase. We see that the system is able to determine intersections of the axis-aligned bounding boxes of entities and their actual mesh, using GJK, in 100% of test cases as seen in figure 30.

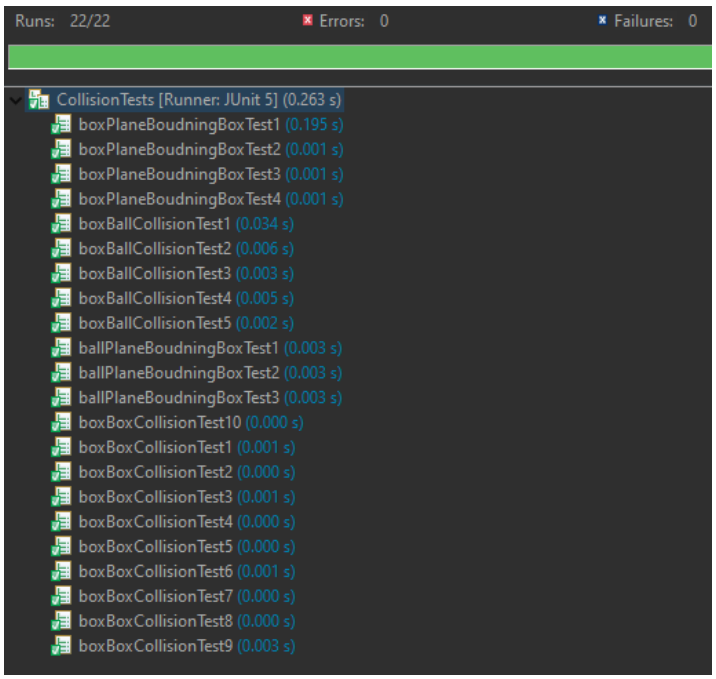

Figure 30: Shows 22 passed tests, where each test involved checking the bounding box collision and GJK collision algorithms.

The narrow phase detector also accurately returned a non-intersection of actual meshes,

where the axis-aligned boxes provided expected false-positives on the collisions, due to the extending hit box region, see figure 31.

```
@Test
public void boxBoxCollisionTest5()
{
    CollidableBox a = CollidableBox.create(new Vector3f(), new Vector3f(), 1, 1);
    CollidableBox b = CollidableBox.create(new Vector3f(1, 0, 1), new Vector3f(0, 45, 0), 1, 1);

    // Rotated box results in a false positive in the initial phase.
    boolean bBAreIntersecting = a.getComponent(Collidable.class).bBox.isIntersecting(b.getComponent(Collidable.class).bBox);
    assertTrue(bBAreIntersecting);

    Collision col = new Collision(a, b);

    // GJK reveals entities aren't actually intersecting.
    boolean gjkIntersection = NarrowPhaseDetector.areIntersecting(col);
    assertFalse(gjkIntersection);
}
```

Figure 31: JUnit test to show initial false-collision detections of close entities are correctly determined to not be intersecting by GJK.

A few tests were also made to check that entities that had a force or torque applied to them over a given time step, had their positions and rotations updated accordingly, with all of the tests passing, see figure 32.

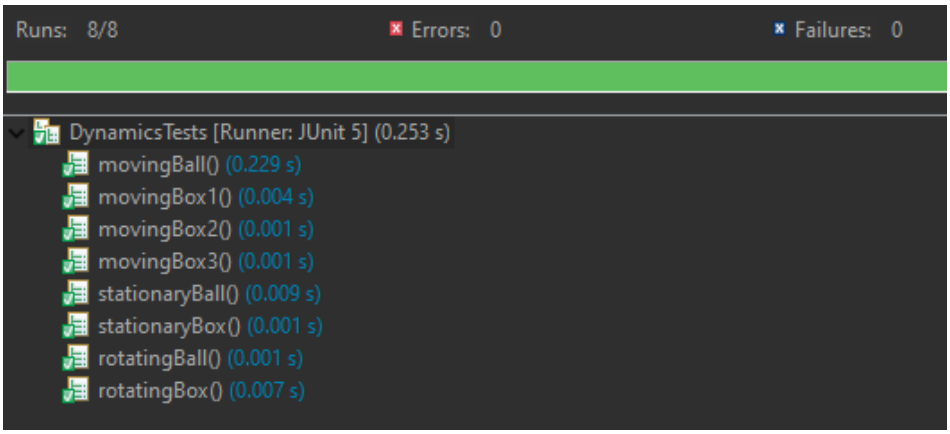

Figure 32: Shows 8 passed tests, where forces and torques were applied to entities over a time step.

5.4.2 Visual Tests

To test performance, the application was set to render 100 static cubes to the scene, see figure 33. The system was able to achieve this with no noticeable effect on the performance, with no stuttering of the frames.

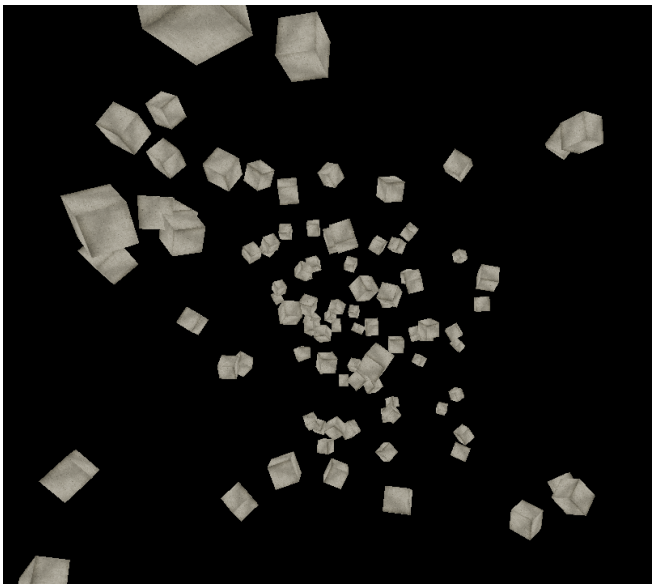

Figure 33: Shows 100 static cubes rendered to the scene.

The performance of the collision system was also tested by dropping 18 cubes onto a plane, see figure 34. The scene ran smoothly, with all 18 cubes ending up resting on the plane, see figure 35.

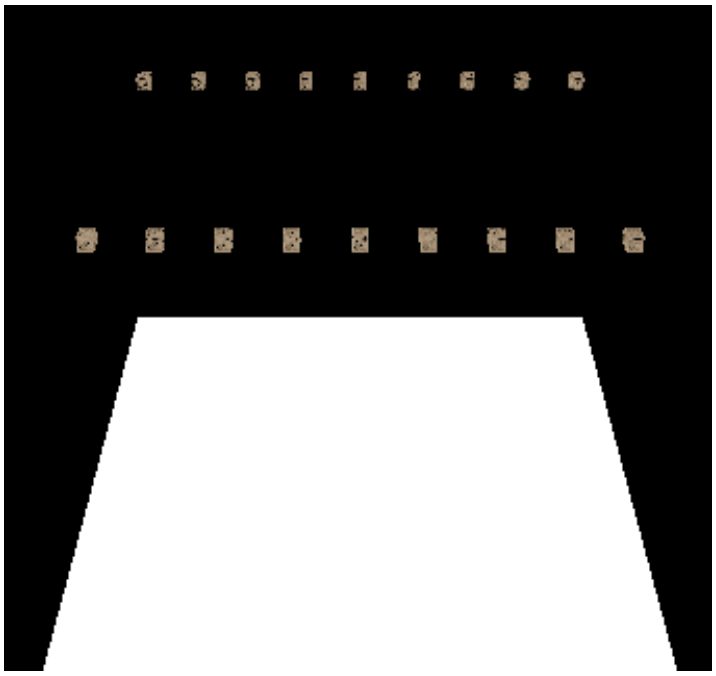

Figure 34: 18 cubes dropped above a plane.

Figure 35: 18 cubes resting on a plane after being dropped onto it.

Lastly, a visual collision check was set up, placing 3 pairs of entities into the scene with two of the pairs colliding and the 3rd not. The current collision can then be printed to the console by enabling collision debugging, which shows that the collisions are correctly detected, see figure 36.

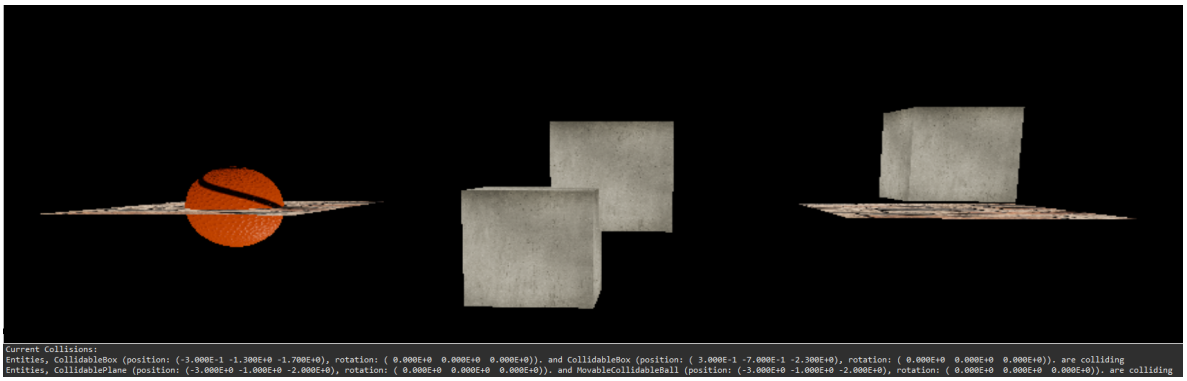

Figure 36: 3 pairs of entities, with the first 2 pairs colliding and the printed results confirming the current collisions occurring.

6 Project Management

The project followed a plan driven approach, where a stage of initial research, requirement defining and software planning took place. The software development process then followed an incremental methodology, where design, development and then testing was done for each new increment, providing increased functionality from the previous. This approach worked very well with the Entity Component System architecture of the project, with its independent systems, it allowed for each increment to consist of a new system, allowing for each system to be functioning before moving onto the next, see figure 37.

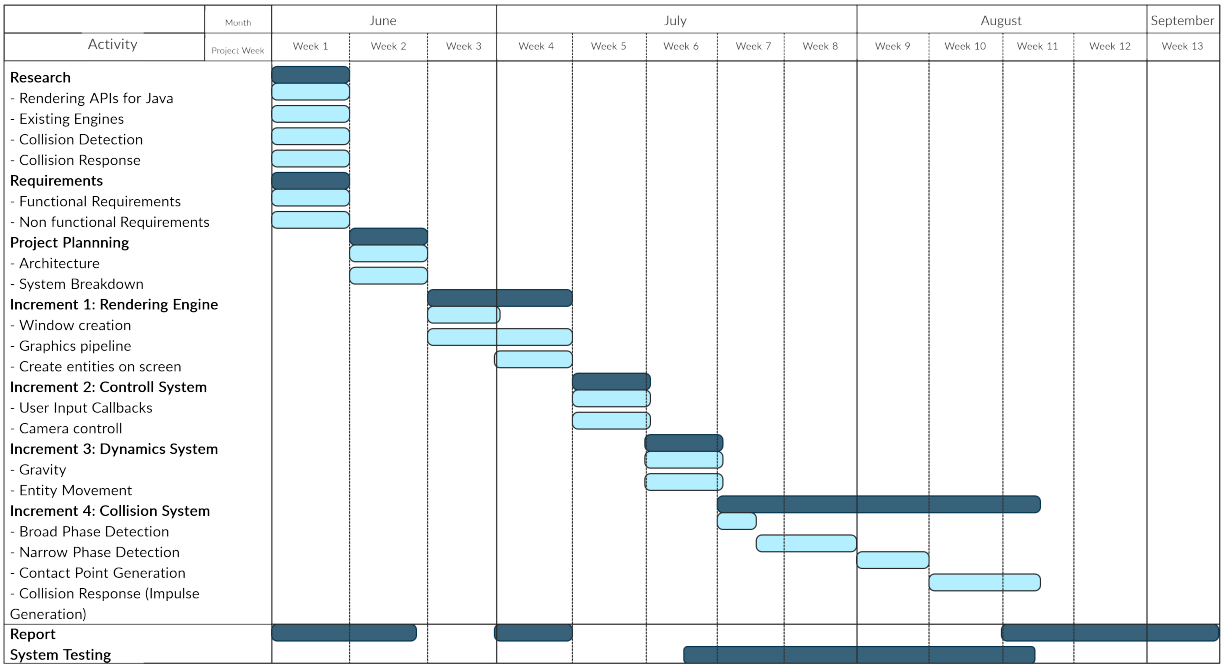

Figure 37: Ghantt chart of project work flow.

The software's initially set requirements, were used to get a breakdown of the systems that had to be developed. This resulted in a decision to run 4 main increments of development: implementing a rendering system that would allow for 3-dimensional static objects to be displayed on screen; an input control system that would allow for a controllable camera to be moved around the scene; the dynamics system, to allow force to be applied to entities, with these forces resulting in the motion of entities on screen; and the collision system, which would consist of sub-systems of broad phase detection, narrow phase detection, contact data generation and collision response (figure 38).

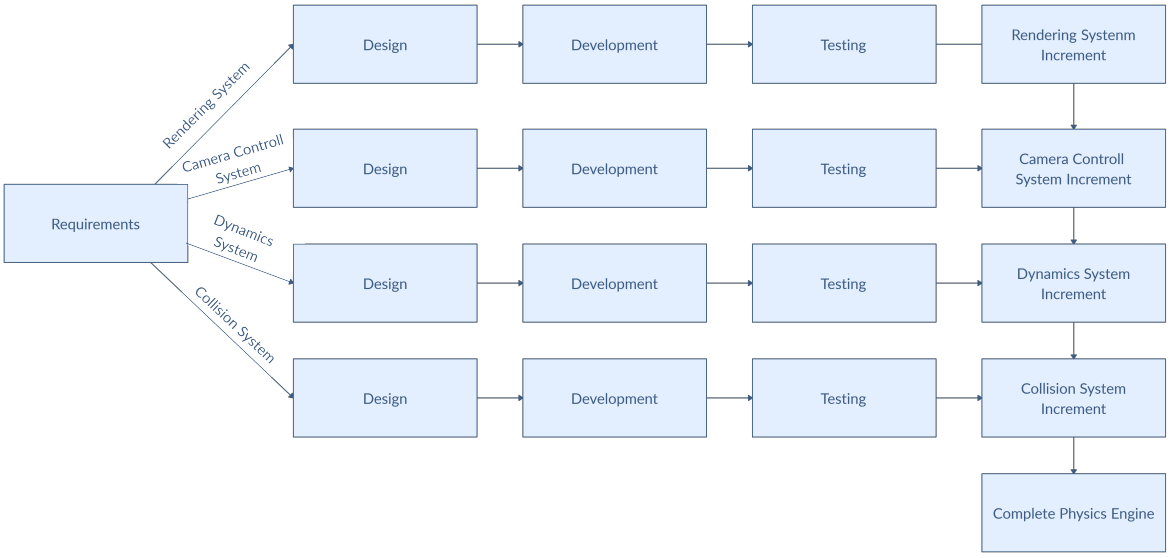

Figure 38: The incremental development steps taken in development of this project.

7 Results & Evaluation

In this section we will examine the final piece of software, looking at a few demonstration scenes, to see what the engine is capable of. We will then look back at the requirements specification, and evaluate how well these requirements have been met, and what this means for the overall success of the project.

7.1 Demo Scenes

A few demonstration scenes were created to demonstrate the capability of the engine, attempting to show a range of physical properties such a friction and elasticity. These scenes were created by extending the Application class, which provides the ability to set entities to be present in a scene, along with their properties.

Box on a Slope:

This first demo consists of a simple slope at 20°, and a box with a of mass 1 kg is dropped from a height onto it. The box hits the slope, resulting in an impulse force at 90°to the slope, however the low coefficient of restitution between the cube and slope results in only a small rebound, as the torque generated from the collision causes the cube to rotate as it slides down the slope, before coming to a stop due to friction. This process can be seen in the screen shots shown in figure 39.

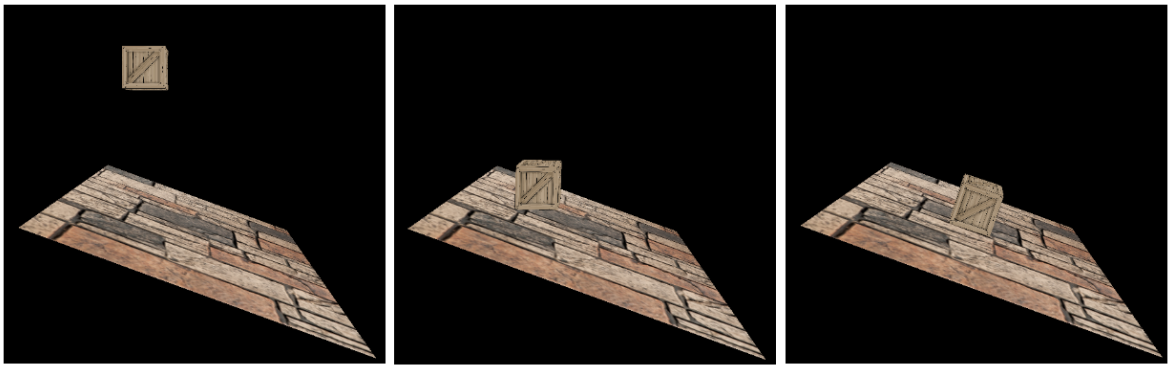

Figure 39: Screen shots from a demo showing a box dropped onto a slope.

Stack of Boxes:

The next demo consists of 4 boxes, which are dropped from varying height directly above the centre point of a plane. The boxes fall to the floor landing one on top of each other, coming to rest almost instantly due to their extremely low coefficient of restitution, see figure 40. By playing around with the coefficient of restitution, you are able to see the boxes rebound more on a collision, with a restitution of 1, the boxes never come to rest and bounce up and down between each other and the floor indefinitely.

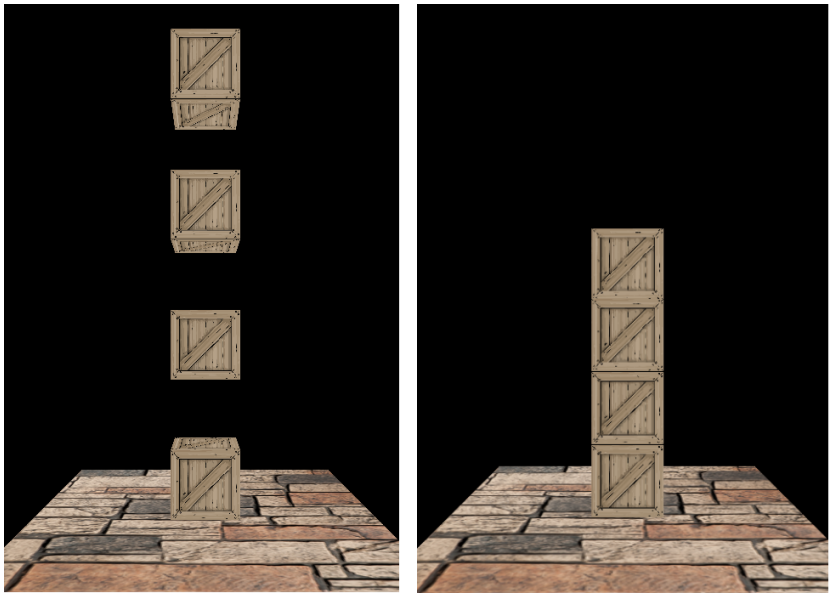

Figure 40: Screen shots from a demo showing a stack of boxes being dropped onto a plane.

Ball on a Slope:

As we have looked at two low elasticity collisions, the third demo shows a basketball being dropped onto a small inclined slope of 15°. The ball falls and hit the slopes, bouncing up along the slopes normal. The ball then performs an arching motion, rotating as it does so, before hitting the slope again at a shallower angle. After this second collision the rebound is at a shallower angle due to the horizontal motion generated in the first impact. The ball then bounces off the slope and falls into the void.

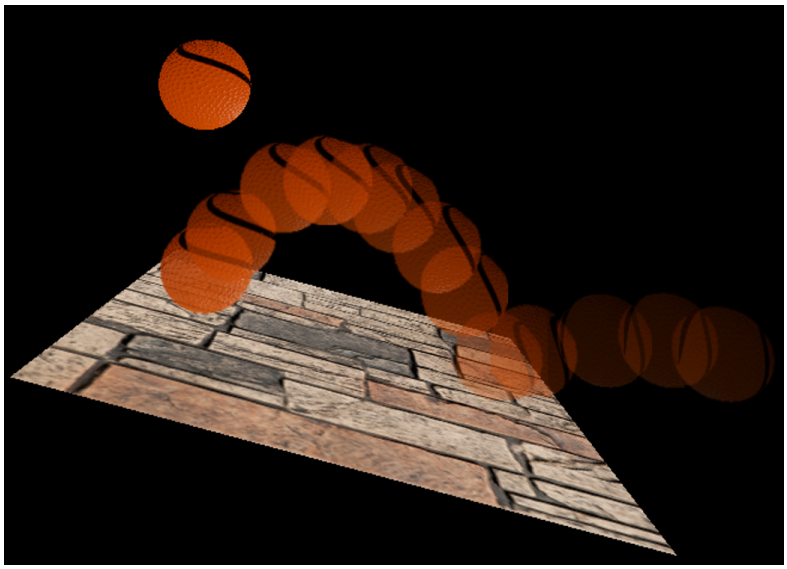

Figure 41: Layered screen shots from a demo showing a balls motion when dropped onto a slope.

Mixed Demo:

The final demo, a mix of 2 basketballs and 2 boxes are dropped, within the confines of a "room", with one of the boxes and balls given an initial momentum towards a wall to replicate the act of throwing. After the box is thrown at the back wall it proceeds to rebound off it slightly, falling down onto the floor and coming to a rest. The ball on the other hand bounces far off the wall, hitting the floor bouncing back, up and flying out the "room". Meanwhile the box and ball at the front are dropped from the same height, with the box coming to rest almost instantly on impact as we have seen before, and the ball bouncing 4 times before coming to rest. This process can be seen in the breakdown of screenshots taken from the demo in figure 42.

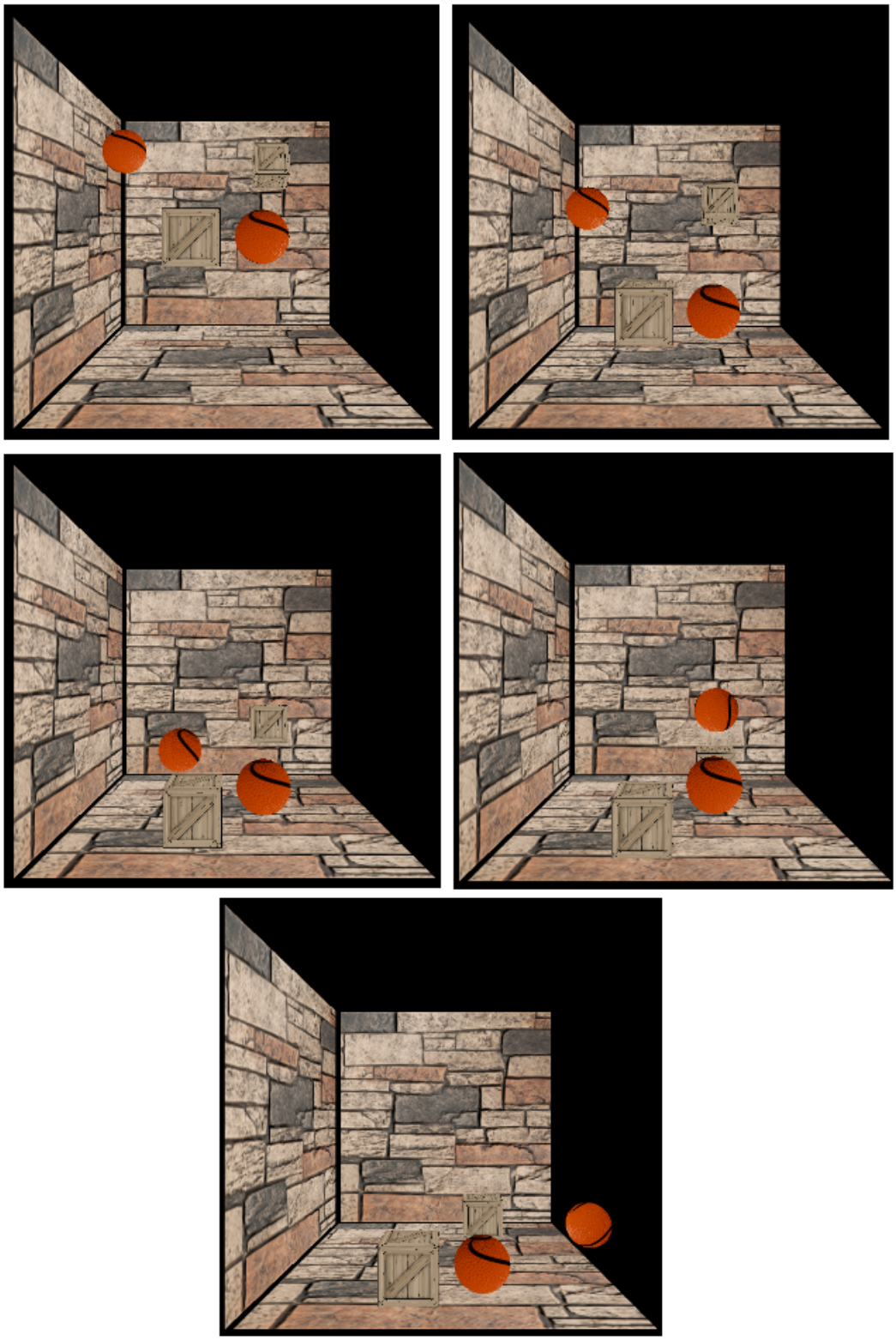

Figure 42: Screen shots from a demo showing some boxes and balls being thrown against some planes.

7.2 Requirements Evaluation

Meeting the specification requirements was key to developing a functioning and usable physics engine. The 24 functional requirements, see section 3.3, along with the performance and usability non-functional requirements, were all met. The clear system structure and development methodology ensured implementing the functionality was achieved.

Rendering

The system as we have seen in our testing and demos is capable of rendering a scene of static and dynamics entities. Our load testing showed that we were capable of rendering 100 static cubes into the scene without any performance loss.

Entity-Component

The use of the ECS architecture allowed for all the entity property and operation functional requirements to be met. With ECS architecture focused on allowing great control over the data an entity contains, and as mentioned previously, the use of maps as a data structure to manage entities and their components has allowed for all must have entity operations to be implemented.

Collisions

As our testing showed, the system is fast and effective at determining if two entities are colliding. Using the two phase implementation allowed for an initial pruning to ensure GJK was only used on entities that were possibly colliding.

The collision response and dynamics, as we saw in the demos, provides accurate interactions between entities in the scene, giving it a realistic feel.

8 Discussion

This section will summarise the achievements and deficiencies of this project, as well as look at what could be done in the future to improve it.

This project has succeeded in meeting the initial project aim of developing a functional physics engine, capable of simulating the rigid body dynamics of 3-dimensional objects, and render the results to the screen in real-time. Demo scenes have visually shown that it can do so reliably and accurately, and through testing have shown that the engines system operate as expected.

Although the system is fully capable of simulating rigid bodies, due to the precision of floats, there are obscure cases where collisions aren't properly detected, such as entities getting stuck in each other when colliding at a specific angle or the generated normal to the collision is not as expected resulting in entities rebounding in unnatural ways. This would unlikely be simply fixed with more time and instead would require extensive research in looking at more advance and complex collision detection methods, along with ways to generate more than a single contact point for stability.

One downfall to the system is it is only able to detect collisions accurately between convex shapes. If there was more time, we could have explored the methods used to enable more accurate collision detection of concave shapes. From simply generating a convex hull around any concave shape and use that to detect collisions, or by splitting a concave shape up into smaller convex components and testing for collisions against these sub meshes.

Also as the system only provides basic rendering capability, the visual aesthetics definitely have room for improvement. Additional work on the graphics pipeline could provide a more immersive visual experience, with possibilities to add lighting, reflection and improved texture mapping to entities on screen. Although the goal was only to provide the simplest of rendering capability, this would be have added an extra touch if there was more time available.

Another feature that would have been beneficial to implement if there was more time would have been a user interface. As of right now, the only way to create a scene, is to create a Java class extending Application and run it. Although this isn't difficult for an experienced Java user, it stops non Java users being able to confidently use the engine. A user interface on the other hand could allow anyone to drag and drop entities into a scene, fiddle with their physical properties then press play and the scene would play out.

The next steps for this project could involve developing a simple 3D physics base game with this engine, as suggested in the introduction. Or taking it even further, expanding the engine for it to include wider functionality, such as audio control, AI systems, to end up with what could be called a complete game engine. This type of project would take a significant amount of time, with there being a lot of optimisation needed to be done, though it would be an exciting prospect, and possibly something you will see in the future.

9 Conclusion

This project succeeded in delivering a 3D physics engine in Java, with capabilities of simulating convex rigid body simulations and rendering the results to a screen in real-time. Users can create a class in Java, which extends an Application class, where they can initialise entities to be added to the scene. When this class is run, the scene will be generated and the simulation will run from the initial state set by the user. The user also has the ability to apply updates to the scene each frame, such as applying forces to an entity or removing components. Based on testing, it is seen the system is able to handle a large number of rendered entities to the screen, as well as the detection of a significant number of simultaneous collisions.

The engine offers all the functionality addressed by the specification requirements, which was achieved through an Entity-Component-System architecture. Following this design structure allows users to have fine control of entity properties, without the need for large and complex inheritance hierarchy trees, and the ability for these properties to be added, removed and manipulated at run time. The engines functionality is split up into independent systems, such that disabling a single system, i.e rendering, would have no effect on the rest of engine.

Collision detection and response is generally reliable, however can suffer from floating point precision errors. These errors aren't frequent, but can cause undesired interactions between entities, though exploring methods to generate an increased number of contact points could ensure an even more reliable system.

The rendering system is simple but effective, allowing users to create entities with custom meshes using the common OBJ file format for 3D geometry, and apply textures to these entities by providing a PNG. However, it would benefit from an overhaul, adding additional features such as lighting would produce a more attractive and immersive viewing experience.

Finally, this engine provides a solid baseline for the development of simple 3D physics reliant games in the future. The engine also has the possibility to be expanded, and resemble a more complete game engine due to the Entity-Component-System architecture. New functionality could be added to the engine through the addition of new systems, such as a sound engine or AI components, which would allow for the development of more complex games.

References

Albeza, B. (2015), '3d collision detection', https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection. [Online; accessed 25-August-2020].

Board To Bits Games (2019), 'Entity component system overview in 7 minutes', https://www.youtube.com/watch?v=2rW7ALyHaas. [Online; accessed 2-September-2020].

Burjack, K. (2020), 'Joml – java opengl math library', https://github.com/JOML-CI/JOML. [Online; accessed 3-September-2020].

Chong, K. S. (2012), 'Collision detection using the separating axis theorem', https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169. [Online; accessed 26-August-2020].

Entity Systems Wiki (2014), 'What's an entity system?', http://entity-systems.wikidot.com/. [Online; accessed 2-September-2020].

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

Erwin Coumans (2020), 'Bullet physics sdk', https://github.com/bulletphysics/bullet3. [Online; accessed 3-September-2020].

Fiedler, G. (2004), 'Physics in 3d', https://gafferongames.com/post/physics_in_3d/. [Online; accessed 1-September-2020].

FileInfo (2020), '.obj file extension', https://fileinfo.com/extension/obj. [Online; accessed 3-September-2020].

Harrison (2012), 'Minkowski sums and difference', http://twistedoakstudios.com/blog/Post554_minkowski-sums-and-differences. [Online; accessed 31-August-2020].

LWJGL (2020), 'What is lwjgl 3?', https://www.lwjgl.org/. [Online; accessed 3-September-2020].

Math Open Reference (2011), 'Convex polygon', https://www.mathopenref.com/polygonconvex.html. [Online; accessed 27-August-2020].

Newcastle University (2020), 'Physics - collision response', https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/5collisionresponse/Physics%20-%20Collision%20Response.pdf. [Online; accessed 1-September-2020].

OpenGL (2020), 'Opengl overview', https://www.khronos.org/opengl/. [Online; accessed 31-August-2020].

Tyndall, J. (2014), '3d convex collision detection, part one: Gjk', http://hacktank.net/blog/?p=93. [Online; accessed 29-August-2020].

Vulkan (2020), 'Vulkan', https://www.khronos.org/vulkan/. [Online; accessed 31-August-2020].

Vulkan Tutorial (2020), 'Introduction', https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction. [Online; accessed 31-August-2020].

Wayback Machine (2017), 'Lecture: Graphics pipeline and animation', https://web.archive.org/web/20171207095603/http://goanna.cs.rmit.edu.au/~gl/teaching/%263dgp/notes/pipeline.html. [Online; accessed 25-August-2020].

Wolfram Math World (2020), 'Simplex', https://mathworld.wolfram.com/Simplex.html. [Online; accessed 30-August-2020].

Appendix A: Git Repository Breakdown & Software Running Instructions

The latest version of the physic engines source code is available at `https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2019/dxc653/`.

In the git repository you will find three folders: src, lib and runnables. The scr folder contains all the code written by me for the project. The code is split up into packages, with the entity, component, systems and application package holding the main structure of the code, with the utils package holding some utility classes. The resources folder contains OBJ, PNG and shader files needed. The demo package holds all the main java files that can be run. Finally the tests package holds all the junit and other testing scenarios used in development.

There are a two ways that the software can be used:

Runnable Jar Files:

1. Download the runnables folder on the git repository.
2. The folder contains 4 runnable jar files.
3. Run one of the jar files (Unfortunatley the jar files are only runnable on windows with Java 11).

Theses runnables jar files contain the demos created to demonstrate the capability of the software.

Run on an IDE:

1. Download the source code and load it up on an IDE.
2. Download the sub folder in lib corresponding to your operating system.
3. Add all the jar files from this folder to a user defined library.
4. For each of the LWJGL jars that don't include the word natives, the native library location must be set (this is just the same folder the jar files are in).
5. Run any one of the demo classes found in the demo package.

You may also create your own scene by creating a class that extends the Application class. You can then can create entities in the initScene method. There should then be a main method to this class, including the line: new ClassName().start().

Appendix B: Libraries Used

- Bindings from the Light Weight Java Game Library (LWJGL3), found at `https://www.lwjgl.org/`.
    - OpenGL - A Graphics API
    - GLFW - Window and Input Handler
    - STB - Image Processing Tools
- Java OpenGL Maths Library (JOML), found at `https://github.com/JOML-CI/JOML`.