Contents

[Analysis: 3](#_Toc122268673)

[Problem Definition: 3](#_Toc122268674)

[Why a Computational Solution is Suitable: 3](#_Toc122268675)

[Stakeholders: 3](#_Toc122268676)

[Interview: 4](#_Toc122268677)

[Jack: 4](#_Toc122268678)

[[INSERT TEACHER HERE]: 4](#_Toc122268679)

[Currently Existing Similar solutions: 5](#_Toc122268680)

[PhET: 5](#_Toc122268681)

[The Physics Classroom: 7](#_Toc122268682)

[Features of the proposed solution: 9](#_Toc122268683)

[The limitations of the proposed solution: 9](#_Toc122268684)

[Stakeholder Requirements: 9](#_Toc122268685)

[Hardware/Software requirements: 10](#_Toc122268686)

[Design: 12](#_Toc122268687)

[Decomposing the Solution: 12](#_Toc122268688)

[GUI Design: 12](#_Toc122268689)

[Object Information Panel: 13](#_Toc122268690)

[General Information Panel: 13](#_Toc122268691)

[File Dropdown: 13](#_Toc122268692)

[Options Dropdown: 13](#_Toc122268693)

[Add Object Panel: 13](#_Toc122268694)

[What the Stakeholders Think: 14](#_Toc122268695)

[Graphics 15](#_Toc122268696)

[ImGUI: 15](#_Toc122268697)

[OpenGL: 15](#_Toc122268698)

[Key Classes and Functions: 16](#_Toc122268699)

[Body: 16](#_Toc122268700)

[Rigidbodies: 17](#_Toc122268701)

[Softbodies: 20](#_Toc122268702)

[Simulation: 23](#_Toc122268703)

[Renderer: 24](#_Toc122268704)

[Collisions: 26](#_Toc122268705)

[Collision Detection: 26](#_Toc122268706)

[Collision Resolution: 27](#_Toc122268707)

[Mainloop: 27](#_Toc122268708)

[Diagram: 27](#_Toc122268709)

[Features of the mainloop: 28](#_Toc122268710)

[Why is the mainloop designed this way: 29](#_Toc122268711)

[Utilities: 29](#_Toc122268712)

[Saving and Loading the Simulation: 29](#_Toc122268713)

[Pause, Play and Game-states: 29](#_Toc122268714)

[Testing: 30](#_Toc122268715)

[How I am testing Softbodies: 30](#_Toc122268716)

[Table: 30](#_Toc122268717)

[Glossary 31](#_Toc122268718)

The Project:

# Analysis:

## Problem Definition:

It is currently difficult for teachers to teach about rigid and soft bodies. A lot of students struggle without a visual representation to understand what is happening, however, these visual representations are often limited to simple animations and aren’t customisable. This leads to situations where the student can see the visualisation but can’t understand what is happening it, the students are often even more confused as to what the teacher is trying to teach. When I spoke to a few of my piers about this problem, they agreed that visualising a problem helps them understand it. One of them commented that once he understood the types of problems that questions were asking he was fine, but had issues initially learning about them as he didn’t understand what the diagrams were representing.

The idea for my project is a 2D physics simulation that works together both rigidbody and softbody physics. This simulation will be completely customisable, allowing whoever is using it to set up a scene quickly, run it in real time, and customise a scene however they want. This will make it easier for teachers to teach their students with multiple examples, as well as allow the students to play around with the simulation themselves. A lot of students learn better with an example that they can tinker with to help them to understand what is happening in a specific example.

## Why a Computational Solution is Suitable:

It is difficult to recreate a collision experiment at home, or on the fly. On the other hand, it is much easier to run an application on a computer. A simulation would also be able to make any calculations the student also has to do, this would allow them to check their results. All of the computations that a computer makes are also repeatable, meaning that the user could replay the simulation multiple times and get the same result, unlike real life where there are many unpredictable factors.

## Stakeholders:

My program will be used to help both teachers and students teach and learn about the concepts of collisions3, rigidbodies and softbodies, therefore it makes sense that my shareholders are both. In order for the stakeholders to use the program effectively, it needs to be easy to pick up but also complex enough to be used in every collision system available. To make this possible, I will have multiple menus with different options available for different purposes, as well as a tutorial section so that new users can learn to use the program easily.

My stakeholders will be my friend Jack who both take physics and further maths, and my physics teacher [INSERT TEACHER HERE]. Both physics and further maths include sections in mechanics on collisions, and students from both subjects should be able to use the software without having to take the other. This means that some settings should be optional, as it might be covered it one subject but not another, or some should have multiple options for how to calculate, (eg energy methods vs suvat methods of motion).

As a student, Jack will be more focused on using the software for revision purposes as well as using it to understand the topics. [TEACHER] already understands all of the topics, instead, they are going to be using the software as a demonstration tool. Having both of them as my stakeholders will allow me to take feedback, and tailor the experience to both groups.

## Interview:

### Jack:

Q: How do you currently learn about collisions.

A: Depends on the subject, in further maths - I don’t think collisions are in normal maths – we mainly learn about the theory, we do a lot of calculations as well as draw diagrams to represent what is happening. In physics, we still do calculations, and diagrams, but physical examples are also used, we do some practicals on conservation of momentum and other things.

Q: Do you think that a computer program where you can simulate the system would be of any help to you while learning mechanics?

A: Yes, in maths we dive deeper into the theory, but I never get the chance to actually see the system in action. Even when doing physics, I can’t just set up a practical at home, so having a piece of software I can download and use anywhere would be very useful.

Q: What sort of features would you want to have available to you inside the simulation?

A: Being able to use different equations to model the same situation. Having a menu with all of the options in one place would mean that I could easily set up the scene. Also, being able to click on an object and change its properties.

Q: What kind of objects do you think should be in the simulation?

A: Particles, walls, some polygons, as well as hollow variations.

Q: What have you used in the past to help you understand the concepts?

A: Online videos and physical examples in physics. It is difficult in maths as we can’t set it up as a practical.

Q: What operating system do you use?

A: I use Linux when programming but I use Windows for school work, so I would want the application on Windows.

[INSERT TEACHER HERE]:  
Q: How do go about teaching collisions and mechanics?

A: I try to teach with a lot of practicals so that the students can actually see what is happening. After the students have finished the practicals, I generally link it to the theory and equation behind it. Depending on the topic, I try to get students to derive the equations themselves at A-Level. Showing them the equation and making them link it to the experiment they have just done works well.

Q: Do you think a simulation that you and your students could use would be useful?

A: At A-level, I try to get the students to do the experiments themselves, this is great inside the classroom but very difficult when they go off and do their own study on the subject, I am sure that a lot of my A-Level students would find this very useful for their own study. At GCSE, some of the topics are hard to do with the large class size and limited equipment, so I have to show them all the experiment rather than them do it themselves, repeating the experiment [In multiple lessons] takes a lot of time to set up, being able to just pull up a program with a pre-loaded scene would save a lot of time.

Q: What sort of features would you want to have available to you inside the simulation?

A: Being able to save and load scenes would be a massive time-saver, similarly, being able to send saves to others would let me set prep to students easier as well as send it round the science department. Also, being able to pause the simulation at any point and change what is happening.

Q: As a follow on would you want to be able to drag the objects in the simulation with a mouse during that pause time and when setting up the scene?

A: Yes, that would be good, but also having a window where you can set the details about the object like its velocity, it would be nice if it showed up when you clicked on the object.

Q: What kind of objects do you think should be in the simulation?

A: I think the most needed one is particles – just circles, these are used all the time to model situations. But also shapes such as rectangles, squares and lines. And you should be able to set whether an object is stationary or can move.

Q: What have you used in the past to help students understand the concepts?

A: A real life example works the best, but when that isn’t available, I often draw diagrams. PhET has a lot of really good simulations that simulate things like projectiles, however, some of their simulations haven’t got great interfaces, and you need to restart their simulation every time you want to change a variable.

Q: What operating system do you use?

At school, I use windows, I have an old Macbook at home, but I don’t use it for any work. I would want the program to run on Windows primarily.

## Currently Existing Similar solutions:

### PhET:

Overview:  
PhET is an organisation that provides “interactive simulations for science and math”, it has a website full of simulations, some that run with HTML5 (using JavaScript) and some with Java, and all of their source code is available on GitHub. They have a collision simulation called Collision Lab. This simulator has 4 modes, “Intro”, “Explore 1D”, “Explore 2D” and “Inelastic”, Intro is a 1D simulation (one that only has one dimension of movement, in this case left and right) that has no borders, this means that the balls just fly off the screen after they have collided. Explore 1D is very similar to intro, however, there are borders so that the particles rebound off the edge of the viewport to collide with each other again. Explore 2D is the same as Explore 1D, but as the name suggests, in 2 dimensions, the particles can now move up and down as well as left and right.

Positives:

* UI is very clean and self-explanatory.
* Ability to drag the balls and their velocities around is nice and makes it easy to quickly change the attributes.
* Different coloured and numbered balls makes it easy to distinguish between different balls.
* 2 different speeds available.
* Arrows to show the current velocity.
* Can change the elasticity of the collision.

Negatives:

* Limited to 4 balls at once, and no other shapes exist, however, this is enough to model the majority of simulations that occur at GCSE and A-Level.
* Difficult to input values. Eg, when setting the velocity of the object manually, a pop-up window appears to input the values, and the keyboard doesn’t input anything.
* Not very customisable.

What my stakeholders think:

Jack– I think PhET is a really good website with lots of good simulations, the collisions one is a little clunky, but pretty easy to use. I think that the arrows representing velocity is a nice touch.

[TEACHER 1] – I recommend PhET to my students, the collision simulation almost all that is needed to teach the simulations, I also like the ability to change the elasticity of the collisions for when we are teaching about elasticity.

What I can use:

I will use features such as the dragging of the balls and velocity vectors, these were easy to use and very useful, a definite include. Being able to customise what individual balls look like is a niche but nice feature which extends upon what they have done with the multicoloured balls here. I will also include the overlays such as the velocity of each ball and the momentum. I don’t use Javascript that much, so it will be difficult for me to adapt their collision mechanics into my own simulation, however, all of the source code is available on GitHub (<https://github.com/phetsims/collision-lab>).

Bubble chart

Description automatically generated

### The Physics Classroom:

Overview:

The Physics Classroom is a website with lots of helpful explanations and simulations covering the subject of physics. This website is very similar to PhET, but also has some more explanations and videos describing the topics. Like PhET, all the simulations are HTML5 based, so they all run in the web-browser. They also share some of the same ups and downs as PhET.

The collision simulator is very simple, a two cart system meant to mimic how an experiment would be set up in real life. You can control the mass and initial velocities of the two carts as well as if the collision is elastic or not.

Unfortunately, all of the options were too big to fit on any of the computer screens that I tried. However when I read their homepage, it said that the simulations were “Designed with tablets such as the iPad and with Chromebooks in mind”. Even though this allows the user to use these simulations on their tablets, it does detriment the use on a computer.

Positives:

* Simple to use interface
* Can change the collision type
* Easy reset button
* Shows velocity above cart

Negatives:

* Doesn’t fit on the screen (PC only)
* Not very customisable
* Can only use two carts

What my stakeholders think:

Jack – This simulation is OK, I don’t like how it doesn’t all fit on my laptop screen, but it does have a very easy to use interface. Also, some of the situations that I have to calculate involve more objects as well as in two dimensions, so it wouldn’t be suitable for those.

[TEACHER] – I wouldn’t use this simulation in the classroom. It doesn’t all fit on my screen, and doesn’t have any advantage over just setting up the simulation. I do like the elastic and inelastic collisions, but it would be nice to be able to have a scale for elasticity.

What I can use:

I can take the elasticity, but I would change it to be a value between 0 or 1 that the user can specify, maybe a slider that they can slide. I can also take the reset button to reset it to the last paused location.

Timeline

Description automatically generated

Graphical user interface, application, chat or text message

Description automatically generated

## Features of the proposed solution:

Initial concept:

I will make a program that contains an easy-to-use GUI. This GUI will include a view window that shows the collisions, a drop-down menu with a list of all of the objects, when an object is clicked in the view window, it will bring up a menu that shows all of the object’s data, the user will be able to edit the values here, such as the mass of the object, to create custom scenarios. They will also be able to drag the objects around the view window, as well as pause the simulation and change the speed of the simulation.

The limitations of the proposed solution:  
The main limitation will be how quickly a user can set up a simulation, if it becomes too slow or difficult for them to simulate a situation, they won’t bother and resort to another method. One of the main things I should focus on is the speed of the application as well as creating a simple to use interface so that they can create setup their simulation faster.

Another limitation would be how I will only be simulating in 2D. Even though 3D simulations are a part of the specification for some subjects, I will not be simulating them here. This is because they will add another layer of complete that is not needed for the majority of subjects and exam boards. If you are doing calculations with 3D collisions, you should already have a strong foundation with 2D collisions so this shouldn’t be much of a problem.

## Stakeholder Requirements:

|  |  |
| --- | --- |
| Stakeholder Requirement | Reason |
| Different types of objects | To simulate different unique situations, and for students to play around with |
| Simple interface | So that people only have to learn about how collisions work, not how to use the software as well |
| Able to pause the simulation at any time | So that teachers can explain things. |
| Doesn’t take lots of processing power | School computers are very slow, if they can’t run the program well, it can’t be used for that much teaching |
| Arrows representing velocity | So that users can easily see where the object is going, even when paused. |

General Requirements:

|  |  |  |
| --- | --- | --- |
| General Requirement | Reason | Difficulty |
| A particle collision system | This is the main function of the program, simulating particle collision. | Medium – Detecting the collisions of different types of objects will be the hard part. |
| Able to move objects by dragging them with the mouse. | So that the user can set up custom simulations that will demonstrate what they want to | Medium –translating mouse position to the simulation will be a challenge |
| Pause the simulation with a button | This can be used for multiple uses, such as explaining what just happened or making predictions while teaching | Easy - a simple if statement to check if the simulation is paused or not will do |
| Easy to use, clean interface. | Otherwise the program will be difficult to navigate and won’t be used to its full potential | Medium – Designing user interfaces can sometimes be a challenge |
| Multiple objects can be simulated at once | For simulating larger scenarios | Easy – having a container of objects should solve the problem |
| Multiple shapes and sizes can be used at the same time | To simulate multiple different scenarios | Medium/hard – its not the shapes that’s hard, its calculating the collisions for the shapes |
| Be save and load scenarios. | For fast setup for common scenarios | Easy - shouldn’t be difficult to store object info into files |
| Multiple simulation speeds | To slow down or speed up the collision for clarity or example | Easy - a variable that scales the dt in the simulation should solve the problem |
| Not computationally heavy | So that the slower computers can still run the program without any problems | Medium – efficiency is a challenge every computer program faces. |

## Hardware/Software requirements:

These are general requirements that the user will have to have in order to run the final program.

|  |  |
| --- | --- |
| Requirement | Reason |
| A computer that runs Windows | The majority of users will run Windows, compiling the program for other OS wouldn’t be worth it. I don’t know of any schools that don’t run a version of Windows |
| Mouse and keyboard | To be able to navigate the interface |
| OpenGL compatible graphics card/processor | All of the graphics will be rendered in OpenGL. They will not show anything if it is not compatible. It should be noted, most graphics cards are compatible. |

Measurable Success Criteria:

|  |  |  |
| --- | --- | --- |
| Number | Criteria | How to evidence |
| 1 | Main window that contains the simulation | Screenshot of the window |
| 2 | Simulation detects collisions | Show the code that detects collisions as well as the tests |
| 3 | Simulation accurately simulates the collision, with real time collision detection and accurate collision maths that calculates post-collision velocity | Show the code, tests, and on screen data that shows before and after the collision, as well as hand done calculations on what should have occurred |
| 4 | A pause button that pauses the simulation when pressed. | A screenshot of the main window, as well as the code that shows how it pauses the simulation |
| 5 | Different object shapes and sizes, including: a circle, square, n-gon (a polygon with n sides) softbodies. | Screenshots of the different objects and sizes |
| 6 | Option that changes simulation speed (at least 3 speeds: slow, normal, fast) | Screenshot of the speed input as well as the code of how it works |
| 7 | Help option that describes how to use the program | Screenshot of the help screen |
| 8 | Able to drag objects with the mouse. | Series of screenshots that show how dragging works |
| 9 | A window that shows the object data when clicked on (such as mass, position, velocity etc.) | A screenshot of the window, showing the object data |
| 10 | Objects render to the view window correctly (in the correct position and in the correct shape) | A screenshot that shows objects |
| 11 | Arrows representing velocity render when option is clicked | A screenshot of the velocity arrows on and a screenshot of them off |
| 12 | Option to change the objects’ colours | A screenshot of the different colours |
| 13 | Simulation is fast | A screenshot of the framerate |

The game will be written in C++ and all of the GUI will be done using ImGui. Writing it in C++ will mean that the simulation will be extremely fast, however, it does mean that I will need to compile the code for any potential platforms. However, most schools use windows, this means that compiling for x86 windows should cover most use cases. If not, the code will be open source on GitHub, so anyone can recompile for their machine. ImGui is an open source GUI interface for C++, it is fast, and can be used with most graphics engines, including OpenGL, which is what I will be using.

# Design:

## Decomposing the Solution:

This involves splitting up the solution into more manageable modules. There will be three main modules: the simulation, the GUI, and utilities. The simulation will include anything to do with the maths, collisions, bodies, etc. The GUI will encompass all the interaction with the user, displaying the simulation and all of the menus. The utilities will involve all of the other small parts that don’t fit into any of the previous section, this includes saving and loading simulations, the options and others.

Chart, box and whisker chart

Description automatically generated

// TODO: Diagram can be resized later

## GUI Design:

The UI will use movable panels that hold different information, for instance, one panel will hold the data of the selected object, another will be the viewing window that shows the simulation. These can be dragged anywhere and ‘docked’ to the main window. Below is an example of what the app could look like (it will also be the default layout when the app first loads).

Table

Description automatically generated

### Object Information Panel:

This panel will contain information of the focussed object. You can focus on an object by clicking on it, the user can defocus by clicking on empty space. The user will be able to change all of the information in the panel. It will contain things like the mass, the velocity, the position, the force exerted on the object etc. If nothing is focused, the panel will contain options to add a new object

### General Information Panel:

This panel will contain information about the overall simulation such as the framerate, the number of objects currently being simulated etc. This panel will be useful while debugging the code as well as for any users. It will also contain options for pausing the simulation and simulation speed.

### File Dropdown:

This dropdown menu will contain options for loading and saving simulations. The save option will open a file explorer where the user can choose the folder the save-file is saved in. The load function will open a file explorer which allows the user to select the save-file the user wants to open.

### Options Dropdown:

This will contain general options such as the aesthetics, or which panels are displayed. For instance, the user could decide that they do not want the General Information Panel open, so they could go into options, and tick the togglable General Information Panel option to toggle it on or off.

### Add Object Panel:

This panel will contain options for the added object, such as its position, velocity etc. There will be a dropdown menu where the user can switch between different object types (such as a circle or a rectangle). Below this will be the options for the selected type. For a circle, these would be the position, velocity, initial force, mass, radius. At the bottom of the panel, there will be a button which will add a new object with the selected options to the simulation.

Text

Description automatically generated

### What the Stakeholders Think:

#### Jack:

I think that the design layout looks good so far, I like the fact that I can move about the windows and customise where they go, it is a nice touch. I also like splitting the options into different panels. It will make it much easier to find the option that I am looking for. I like the idea of clicking on an object to bring up more information on it, but I think that there should also be a dropdown menu that allows you to pick which object.

#### [Teacher]

I am pleased with the aesthetics layout of the panels, and all of the options, but I am worried about there being a learning curve to navigate the panels. Would there be a way to have a guide that can show new users what to do?

#### Feedback Conclusion:

The stakeholders like how the simulation could look and provided some feedback for extra features. The dropdown menu for selecting objects is a good idea and could be very useful for a fast moving simulation, I will implement this in the Object Information Panel. [TEACHER] has a good point about new users, while I will try to keep the menu as simplistic as possible, most people will need some extra guidance when first using the simulation, I will create a help page in the dropdown menu. I will also implement some tooltips around the options so that users can tell what each option does.

## Graphics

### ImGUI:

ImGUI is a GUI library for C++ that focuses on being simple and easy to set up, it is also extremely customisable which is what lead me to choose this as my GUI library. It is all open source and can be found on GitHub, they also contain some useful examples so that beginners can see how the code works and is put together. Below is one of the windows in the examples.

Graphical user interface, application

Description automatically generated

And here is the code that creates this window.

Text

Description automatically generated

As you can see, it is very simple to add new GUI elements. It also uses pointers to edit the variables. This code can be contained in a function, so it very easy to group the different windows.

It also contains functionality to draw directly onto the windows by using textures.

### OpenGL:

OpenGL is a graphics engine supported by most graphics cards. It utilizes buffers stored in memory that are parsed through shaders run on the GPU to produce an output to the screen. ImGUI uses an API to draw to an OpenGL screen. As the simulation does not require complex graphics, I will be using the legacy OpenGL. While this gives me less control over the graphics, it is plenty for drawing shapes to a screen and will be much easier and faster to write.

However, by default, OpenGL draws directly onto a window, however, I want it to draw onto an ImGUI window so that it can be moved about and customised. To do this, I need to utilize framebuffers and textures. ImGUI cannot render pure buffers, however, it can render textures. In order to render the simulation to a texture, we bind a special buffer called a framebuffer to a texture and then render the simulation to the frame buffer, then render the texture with ImGUI.

## Key Classes and Functions:

This is an overview of the different classes that the simulation will use, all of these classes will also have getter and setter methods as well as the ones outlined here. Most classes will also contain a getter method for pointers to attributes, while this is not considered generally safe, ImGui interacts with values using pointers, so I will have to use pointer getters so that they can interact with objects.

This is a class diagram that relates all of the body subclasses, it shows how each of the classes relate.

Chart, box and whisker chart

Description automatically generated

### Body:

This is a base class for all bodies, is then split into the rigidbody and softbody classes. Contains base functions that will be called by any accessor, these functions are overridden in the subclasses. It contains no functionality as a body will always be a rigidbody of softbody.

The render function is called by the renderer and draws the object onto the currently bound buffer

class Body{

public:

render(){}

project(){}

update(){}

getCollisionObjects(){}

private:

Vector position

Vector velocity

Vector force

Vector impulse

double mass

double rotation

Boolean static

}

### Rigidbodies:

#### Rigidbody:

The base Rigidbody class contains all of the base features needed for a rigidbody, including the relative vertex positions, position of the object, its velocity, mass and rotation. It’s update function calculates the force on the object, updates the velocity using and . It then updates the position using , however, it only updates if the body is not static. When an object is static, it does not move, however objects can still collide with it.

ImGUI uses pointers to interact with variables, so this class will also contain getters for pointers to some of the attributes so that the user can edit them using the Object Information Panel.

class Rigidbody{

private:

List<Vector> vertices

public:

constructor(vertices, position, velocity, force, mass,

rotation){

this.position = position

this.velocity = velocity

this.force = force

this.mass = mass

this.rotation = rotation

}

method update(dt){

if (!static){

calculateForce()

velocity += (this.force/this.mass)\*dt

this.position += this.velocity\*dt

}

}

method draw(renderer\*){

renderer->drawPolygon(this.position, this.vertices, this.colour)

}

method project(axis){

double min = dot(axis,vertices[0]+position)

double max = min

for (int i=1; i<shape.vertices.length; i++){

double p = dot(vertices[i]+position)

if (p < min) min = p

else if (p>max) max = p

}

return {min, max}

}

}

#### Circle:

The projection for a circle is slightly different than one for a normal polygon as the circle has no vertices stored, the traditional project will not work. The alternative method is to project the centre position and then the minimum point will be that minus the radius, and the maximum will be that plus the radius. The circle is initialised as having 0 rotation, this is because the circle will be rendered the same way no matter the location.

class Circle inherits Rigidbody{

private:

double radius

public:

constructor(position, velocity, force, mass, radius){

super([], position, velocity, force, mass, 0)

this.radius = radius

}

method draw(renderer){

renderer.drawCircle(this.position, this.radius)

}

method project(axis){

point = dot(axis, position)

return [point-radius, point+radius]

}

}

#### Rectangle:

The rectangle is a subclass of rigidbody, the only change it makes is it takes in a width and a height and works out the position of the vertices and passes it to the super constructor. The rendering and the projection is the same as the rigidbody class, so we don’t override these functions.

class Rectangle inherits Rigidbody{

private:

double width

double height

public:

constructor(position, velocity, force, mass, width, height,

rotation){

this.width = width

this.height = height

vert1 = Vector(-width/2, -height/2)

vert2 = Vector(width/2, -height/2)

vert3 = Vector(-width/2, height/2)

vert4 = Vector(width/2, height/2)

vertices = [vert1, vert2, vert3, vert4]

super(vertices, position, velocity, force, mass,

rotation)

}

}

#### Square:

A square is a special rectangle where the width and height are the same, there are no other differences.

class Square inherits Rectangle{

private:

double length

public:

constructor(position, velocity, force, mass, length,

rotation){

this.length = length

super(position, velocity, force, mass, length, length,

rotation)

}

}

### Softbodies:

Softbodies are notoriously difficult to simulate. They are incredibly computationally intensive and are known to break apart with a small force. There are multiple ways of simulating softbodies, but I will go with the Spring-Mass model.

This model uses springs connected to mass points to simulate the forces of the softbody. The force provided by the spring is given by where k is the spring constant and x is the extension. This provides the force both outwards when the spring is compressed and inwards when it is stretched. The extension can be calculated by subtracting the natural length of the spring from the distance between the two ends.

The points on the model can modelled as circles, so we will just reuse our Circle class from the Rigidbody family, this contains all of the features needed to simulate the points including experiencing forces. During collisions, each point will be separately simulated, this will allow each point to interact with other bodies as well as each other. We simulate them this way rather all together because if we simulated it as a single object, we would have to know where the collision occurred on the object, this adds unnecessary complexity.

The Softbody class will be a superclass to different variants of the softbody class, such as a sphere softbody, a rectangular softbody etc. There will also be an option to create a custom softbody.

#### Spring Class:

We will need a spring class. This class will connect 2 points, so we need to store those 2 points as pointers, it will also need to store its natural length and the spring constant. The update function will add a force to each of its stored points.

class Spring{

private:

Circle\* points[2]

double naturalLength

double springConstant

public:

constructor(point1, point2, naturalLength, springConstant){

points[0] = point1

points[1] = point2

this.naturalLength = naturalLength

this.springConstant = springConstant

}

method update(){

length = magnitude(point1.getPos()-point2.getPos())

extension = length-natural

forceMag = springConstant\*extension

forceDir = (point1.getPos()-point2.getPos()).normalise()

force = forceDir\*forceMag

points[0].addForce(force)

points[1].addForce(force)

}

}

#### Softbody Class:

class Softbody{

private:

list<Circle> points

list<Spring> springs

double totalMass

Double springConstant

public:

constructor(vecPoints, pointRadius, position, velocity,

totalMass, springConstant){

this.totalMass = totalMass

mass = totalMass/vecPoints.size()

for vecPoint in vecPoints{

circle = Circle(position+vecPoint, velocity,

Vector(0, 0), mass, 10)

points.add(circle)

}

for (i=0, i<points.length, i++){

for (j=0; j<vecPoints.length, i++){

if (i == j) continue

length = points[i]

springs.add(Spring(&points[i], &points[j], ))

}

}

}

method update(double dt){

for spring in springs{{

spring.update()

}

for point in points{

points.update(dt)

}

}

}

### Simulation:

The main variables are the simulation size and all of the bodies. Due to the way vectors are handled in C++, the bodies will actually be a vector of shared pointers, this will allow us to use polymorphism, so that all subclasses of the Body class can be contained in one list. It also contains an update function that progresses the simulation by some change in time (dt). The class will also contain a setter for simulationSize and a getter for the bodies list.

class Simulation{

private:

vector2 simulationSize

list<Body> bodies

public:

constructor(simulationSize){

this.simulationSize = simulationSize

}

update(dt){

for body in bodies{

body.update(dt)

}

checkCollisions()

}

checkCollisions(){

// This function is covered here

// TODO: insert link to checkCollision funtion

}

}

### Renderer:

The renderer renders everything to the screen. It contains a pointer to the simulation to access the state of the simulation.

class Renderer{

private:

// Window

GLFWwindow\* window

// Pointer to the simulation

Simulation\* simulation

// Window properties

Vector2 wsize

Vector2 pos

// Framebuffer

Framebuffer framebuffer

Mat4 projectionMatrix

// Graphics options

...

public:

constructor(GLFWwindow\* window, Simulation\* simulation){

this.window = window

this.simulation = simulation

}

newFrame(){

imgui::newFrame()

openglUpdateWindowSize()

openglClear()

}

renderSimulation(){

// Renders the simulation to a framebuffer

// First sets the framebuffer as the active buffer

// Then calls of the body render functions

// Each of these call the correct render shape in

// this class, which render to the buffer

// Renders simulation border

// Sets active buffer to the default buffer

// Then copies the framebuffer to ImGui “viewport” window

}

renderImGui(){

// Renders the ImGui interface

}

// Render shapes functions

renderCircle(radius, position, colour){

// Renders a circle to current opengl buffer

}

renderPolygon(vertices, position, colour){

// Renders a polygon to current opengl buffer

}

RenderLine(start, end, thickness, colour){

// Renders a line to current opengl buffer

}

## Collisions:

### Collision Detection:

#### Axis-Aligned Bounding Box (AABB):

AABB is a basic detection method that only works with rectangles whose edges are parallel to the axis. It works by comparing the positions and sizes of the 2 rectangles. A pseudocode example would look like:

function AABB(position1, position1, size1, size2){

return position1.x < position2.x + size2.width &&

position1.x + size2.width > position2.x &&

position1.y < position2.y + size2.height &&

position1.y + size1.height > position2.y);

}

#### Separating Axis Theorem (SAT):

The SAT is a collision detection method that uses vectors maths to project the objects onto a “separating axis” which then compares to find collisions. One major advantage of the SAT is that it works on any convex shape, this will be very helpful as I will be simulating more than just rectangles.

The projection is done by finding a perpendicular line which passes through a vertex. The point where this line intersects the axis is the projected vertex, we call the projected vertices its shadow.

We compare both shapes’ shadow on the axis to check if they overlap. If they overlap on every axis, then the shapes have collided. The axis are the lines perpendicular to the faces of both shapes.

The pseudocode for SAT is below

function separatingAxisTheorem(object1, object2){

edges = object1.getEdges() + object2.getEdges()

axes = []

for edge in edges:

axes.append(perpendicular(edge))

for axis in axes:

projection1 = object1.project(axis)

projection2 = object2.project(axis)

if not overlapping(projection1, projection2):

return false

return true

}

I will be using the Separating Axis Theorem to detect collisions, as it allows for more complex shapes than just rectangles. However, when using SAT, there are 2 major of problems, it is computationally intensive and objects moving fast enough can just pass straight through each other (tunnelling).

The first problem can be minimised by using optimisation techniques, however, the second is harder to solve. One solution would be to check the collision at multiple times per frame, the problem with this solution it multiplies the number of computations by the number of checks per frame, doing this a lot would slow down the simulation, this would fail success criteria number 13. Another solution would be to have a speed limit for the simulation, while limiting the simulation, this would prevent all tunnelling. High speed collisions aren’t a part of any physics or maths curriculum, so this wouldn’t be a problem from a teaching/learning standpoint.

### Collision Resolution:

#### The Linear Equation Route:

This route calculates the final velocities using equations that the students using the program will learn. The first will be the conservation of momentum, this says that the total momentum of a system will remain the same without an external force. The second is the coefficient of restitution (e), this tells determines how much of the energy is conserved in a collision, in a perfectly elastic collision, e is 1, in a inelastic collision e=0. The coefficient of restitution is defined as: . By rearranging these two equations, we can set up a set of linear equations that can be solved to find the final velocities of the two objects.

#### The Impulse Route:

An impulse is the change of momentum of an object, it can also be defined as a force multiplied by the time the force is acting upon the object. The impulse route is very similar to the linear equation route. It still uses the coefficient of restitution, and conservation of momentum. However, the difference is in how the velocity is changed, where the linear equation route simply changes the velocity, the impulse route applies an impulse to the object.

This solution is better suited for my needs because softbodies deal with collisions differently to rigidbodies. Using an impulse will make it easier to simulate the softbody.

## Mainloop:

### Diagram:

This is a flowchart of the mainloop of the program, the loop executes once per frame.

Chart, box and whisker chart

Description automatically generated

### Features of the mainloop:

#### Handle Input:

This function handles all of the inputs that ImGUI detects, this includes clicking on the screen, dragging, keyboard inputs etc. This function is further described in the IO section.

#### Update Simulation:

This calls the update function in the simulation which updates all of the bodies as well as detecting and resolving any collisions

#### New Frame:

This is called in the renderer to setup a new frame, it clears the previous screen as well as sets up the ImGUI frame.

#### Render Simulation:

This function renders the simulation to an ImGUI window. It binds the framebuffer, then renders all of the objects to the framebuffer before unbinding the framebuffer. It then renders the connected texture the ImGUI window.

#### Render ImGUI:

This function renders the rest of the interface, including the object information panel, as well as the panel to add a new body.

#### End Frame:

This function renders all of the OpenGL and ImGUI to the screen by swapping the buffers to the output buffers, it is necessary to actually see any output onto the screen.

### Why is the mainloop designed this way:

The mainloop is structured so that it is a linear set of events every frame. The mainloop is abstracted to make it easier to understand, and debug. The only check within the mainloop is the check to exit the mainloop. This isolates problems, so that problems affecting one part of the program will not affect other parts, ie. a rendering problem will not affect a problem with the simulation.

## Utilities:

### Saving and Loading the Simulation:

One big part of the program is saving and loading simulations. These features will allow users to quickly load frequently used simulations without having to set them up each time. The user should only have to pick a location to save or select the file to load, they shouldn’t have to do any of the background work.

#### Save file format:

I will use a custom save format to save the game. The only parts of the game that need saving are the bodies, so the JSON file will just contain a list of all the bodies. As each type of body will save different data, each class will have unique save and load functions. The load function will be a static function and will return a new object with the save data.

Each object will have an identifier and a body. The identifier will be unique to each Body subclass and will determine which subclass it is. The body contains the data of the object. Each piece of data will be separated by a comma. Only the data used by the constructor needs to be stored.

#### Validation:

The save file will be validated to make sure that it is a valid save file. One check will be the file extension, to check if it is a save file. Another will be to check that the syntax of the save file is correct, this makes sure that the save file is not corrupted, or another random file. Both validations should be made before we use the data of the save file to load any objects.

### Pause, Play and Game-states:

The pause and play function will allow the user to stop the simulation at that specific time and then resume the simulation. Pausing the simulation should not stop the render loop, only the simulation itself, and as it has only 2 states (paused or play) it can be stored as a boolean. We can store this boolean variable within a game-state class. This class will also store other states such as the current speed of the simulation.

## Testing:

Testing is an important part of development as it makes sure that every function does what it should. The testable parts of my solution are:

* Collision detection – projection, overlap
* Collision resolution – momentum calculations, final velocity
* Spring calculations
* Save function – individual objects, save file
* Load function – individual objects, simulation
* Individual update functions – acceleration, velocity and displacement calculations
* UI elements that change simulation values – change game-state, change simulation objects

### How I am testing Softbodies:

Softbodies are difficult to test. Certain parts of them can be tested, such as individual springs, or general velocity calculations, however, to test a full collision with a softbody, would require to much maths to do manually. Therefore, I will only test individual spring calculations, which can be scaled up to represent the whole softbody.

### Table:

|  |  |  |
| --- | --- | --- |
| Test No. | What I will test. | How to test |
| 1 | Shape projection onto axis | Manually do the maths, then check using a unit test. |
| 2 | Checking overlap of projected items | Write a test for a case with and without an overlap. |
| 3 | Collision detection | Check a case with and without a collision for all combinations of special bodies (such as a circle or softbody). |
| 4 | Momentum Calculation | Manually do the maths, check with unit test. |
| 5 | Final velocity calculation | Manually do the maths, check with unit test. |
| 6 | Spring calculations | Manually do the maths, check with unit test. (F=kx, force on points, etc). |
| 7 | Save individual object |  |
| 8 | Save file in correct location | Check if file is saved in correct location – test multiple locations. |
| 9 | Open file correctly | Write unit test to test if file opens. |
| 10 | Loads objects from save correctly | Write unit test with tested save file. |
| 11 | Object update functions | Write unit test with manually calculated values. |
| 12 | Test if UI elements change objects | Manually test using debug mode. |
| 13 | Test if UI elements change game-states | Manually test using debug mode. |
| 14 | Game opens onto main screen | Check screen when game is first opened. |
| 15 | Game windows are moveable | Check if you can move the windows around (including outside the main window). |
| 16 | Bodies are shown in viewport | Check if bodies are displayed in the viewport. |
| 17 | Bodies display correct colour | Check that bodies colour are the same as the stored RGB values . |
| 18 | Object Information Panel works | Brings up Object Information panel when an object is clicked on. |
| 19 | Pause function | Simulation is paused when paused is toggled. |
| 20 | Play function | Simulation is resumes after pause |
| 21 | Simulation Speed | The simulation moves faster/slower depending on the simulation speed. |
| 22 | General Information Panel is displayed | Check that the General Information panel is displayed when toggled. |
| 23 | Framerate is updated | General Information Panel displays correct framerate. |
| 24 | Add Object Panel | Add object panel is displayed. |
| 25 | Add Object button adds object | Check that a new object is added to the screen when the add object button is pressed. |
| 26 | Add object panel displays correct object options | Verify that the panel displays the correct options for the selected object, (radius for circle etc) |
| 27 | Adds object with correct attributes | Verify that the object added has the correct attributes |
| 28 | Help menu is displayed | When clicked on, the help menu is displayed |

// TODO: add more tests while in development stage

# Development:

## Development Plan:

The development will be split into multiple stages, each stage will be developed separately, and separately tested.

The phases are as follows:

1. Project Setup
2. Application class and main function
3. Simulation class + Body class
4. Renderer class
5. Rigidbodies
6. Collisions
7. Softbodies
8. Utilities
9. UI
10. Testing

## Project Setup:

### Visual studio:

I will be using Visual Studio as my IDE. I have chosen it because of its powerful debugger and UI. I am also very familiar with it so wont have to learn a new editor.

Setting up a new project in Visual Studio is very easy. On the start-up window, press “Create a new project”, I then selected “C++ Empty project”, selected a location and a named the project. After I pressed create, a new project was created.

### Linking external libraries:

In order to use GLFW and GLEW, I first need to link the libraries. To do this, I first create a “libs” folder in my project. I then downloaded the library files from the GLFW and GLEW websites and placed them in the libs folder.

Text

Description automatically generated

To link them to the C++ project, I first had to add the location of the libraries to the project settings. So I went into Properties->Linker->General, and pasted the relative file path into Additional Library Directories

Graphical user interface, text, application

Description automatically generated

Then I linked the files by going into Properties->Linker->Input->Additional Dependencies. I then inputted, opengl.lib, glfw3.lib, and glew32s.lib. Now our external libraries are linked.

Graphical user interface, text, application

Description automatically generated

### Additional Include Directories:

I also had to add any extra directories I wanted to include header files from, to do this, I went to Properties->C/C++->General->Additional Include Directories, there I added all of the extra include directories that I needed.

A picture containing text, screenshot, indoor

Description automatically generated

## Application Class and Main Function:

The main function in C++ is the entry point to the program, our main function will only create an instance of the application class, check if the window has setup properly, and then run the application.

### Application:

#### Application.h

Text

Description automatically generated

The “imguiInclude.h” file is a header file which includes all of the ImGUI dependencies.

#### Application.cpp

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

The constructor calls the setupWindow method which sets up the GLFW and ImGUI instance, it also updates the error enumerator with any errors it finds and returns out of the function.

The destructor shuts down the ImGUI instance and terminates the GLFW instance.

The run function currently contains an empty mainloop, more will be added to this when the Renderer class and Simulation class are made.

The isSetup method checks if there was any error, reports it and returns a Boolean value if the program can run.

### Main Function:

#### main.cpp

Text

Description automatically generated

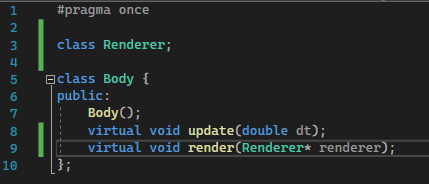
It creates an instance of Application, checks if the program can run, if it can, it runs the application, if not it returns an error code.

## Simulation Class + Body Class:

### Body:

I started to write the Simulation class, when I realised that for most of the methods, I would need to also implement the body class, so I started to write a bare bones Body class. I only implemented the update and render functions, but this is all I need to make the Simulation and Renderer classes.

#### Body.h



The virtual keyword specifies that this function can be overwritten by child classes.

#### Body.cpp

Text

Description automatically generated

The update method doesn’t do anything in Body, this is because it is overwritten in Rigidbody and Softbody.

### Simulation

The simulation contains a list of bodies, the simulation size, an update function as well as a few getters and setters.

#### Simulation.h

Text

Description automatically generated

An “std::vector” is part of the C++ standard library, it is a dynamic array, with lots of inbuilt functions, however, it shares a name with the mathematical vector, to avoid confusion, I will refer to mathematical vectors as “vec2” (a 2D vector), this is included in glm, a maths library that is useful for OpenGL.

#### Simulation.cpp

##### Constructor:

Text

Description automatically generated

The constructor initialises the bodies

##### Update function:

Text

Description automatically generated

The update function iterates through each of the rigidbodies and calls their update function

##### Getters/Setters:

Text

Description automatically generated

### Stage 3 Conclusion – Adding to mainloop:

That is all I need for the simulation class for now, later we will add some more things for collisions, we can now implement it into the mainloop. First we initialise a simulation in the run function, then we get the time from the last frame and store it in “dt” then we update the simulation. The run function in Application now looks like this:

#### Application.cpp

Text

Description automatically generated

## Renderer Class

The Renderer class is where all of the graphics will be implemented, including the GUI and rendering the simulation.

### Setup:

Here I set up the Renderer header file by declaring most of the main functions. It also includes a pointer to the GLFW window, a pointer to the simulation, as well as some options. The window pointer is used to render to the screen, and the simulation pointer makes it easier, so that the program doesn’t have to copy the whole class every time it renders a frame.

#### Renderer.h:

Text

Description automatically generated

#### Renderer.cpp:

Here I left most of the functions blank, only initialising values in the constructor and filling out a getter method.

A screenshot of a computer

Description automatically generated

### New Frame:

The new frame function clears the current window and sets it up to be rendered on. It has three main sections: Setting up a new ImGUI frame, updating the window size, and clearing the OpenGL buffers.

#### Renderer.cpp

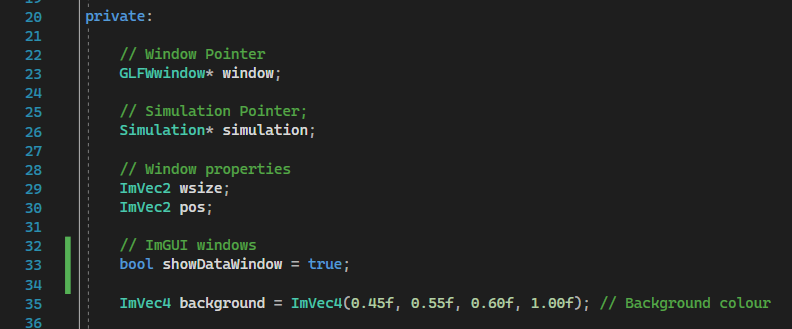
Text

Description automatically generated

### Render ImGUI:

In this frame, I render all of the GUI. For now I will just add a fps (frames per second) counter and all of the setup to allow the ImGUI frames to render. First, I will add a private boolean to the Renderer class called “showDataWindow”, this variable will determine whether the window is shown or not, it will be on be default.

#### Renderer.h



#### Renderer.cpp

Text

Description automatically generated

### Render Simulation:

Before I render the simulation, I will first abstract OpenGL to make it easier to work with. The main things to abstract are the framebuffers. Framebuffers will allow me to render a frame to a texture which can then be passed to an ImGUI window.

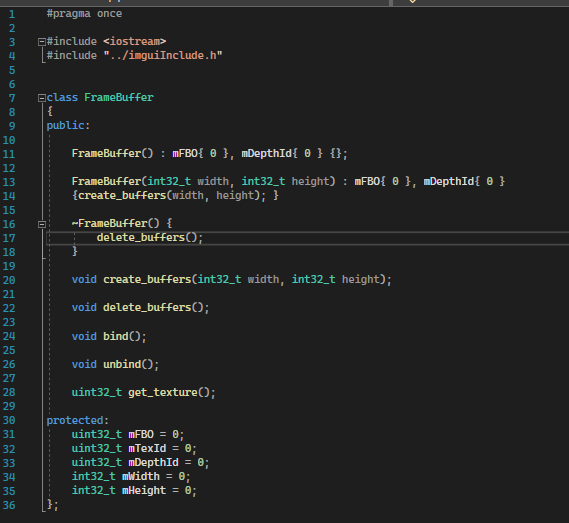
#### OpenGL Abstraction

First, I created a class to contain the framebuffer.

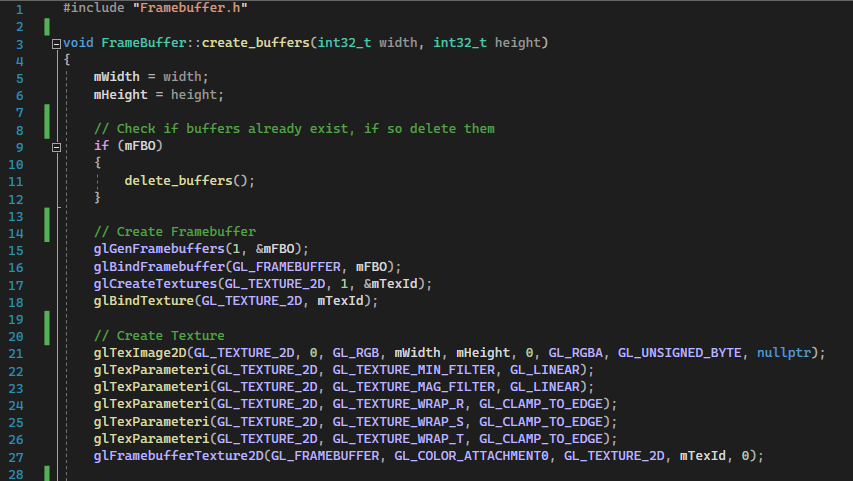
##### Framebuffer.h

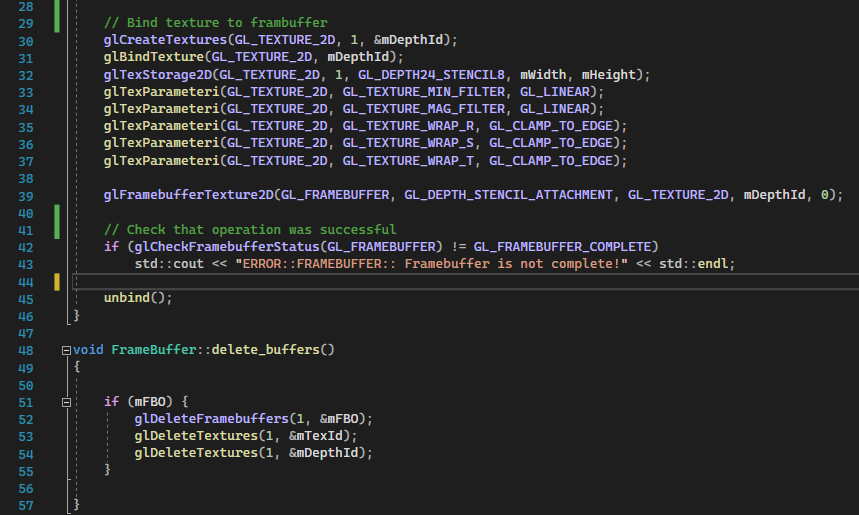
(new file)

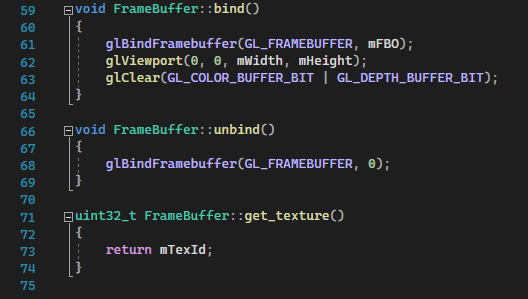
This class contains a few members, these contain information such as the IDs that relate to the individual buffers and textures, as well as some information such as the width and the height of the buffer. The create buffers method creates a framebuffer and a texture and binds them. The delete buffers method destroys the framebuffer and frees the memory allocated to it. The bind methods binds the framebuffer to the current buffer, the unbind method binds the default buffer, and the get texture method returns the texture ID which can be used to draw the texture.



##### Framebuffer.cpp







#### Renderer.h

Here I include the Framebuffer.h file

Text

Description automatically generated

#### Renderer.cpp

In the renderSimulation function, I created a new ImGUI window called “Viewport”, then I create a new framebuffer, bind it, render the simulation. After that, I unbind the framebuffer, and draw the texture onto the ImGUI window.

Text

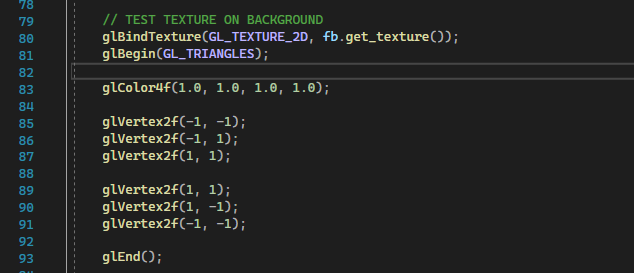
Description automatically generated

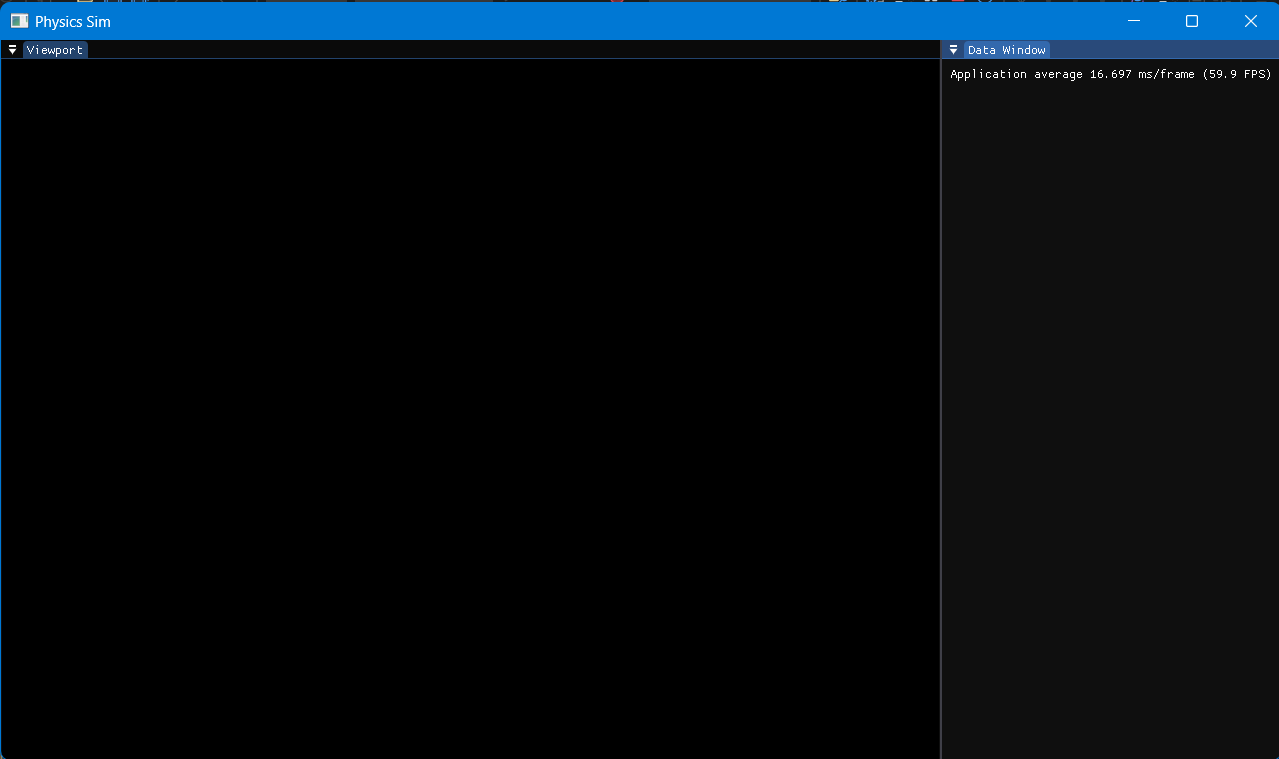
Text

Description automatically generated

#### Debugging:

While testing this section after adding it to the mainloop (next section), I found that the test rectangle (code below) that I hard coded in wasn’t showing up, the screen was just black.





After spending some time trying to figure out what was causing it, I came up with a hypothesis. The framebuffer is a local variable, meaning that the destructor is called when the scope ends. My theory is that ImGUI doesn’t actually copy the texture when the addImage function is called, but rather when the final image is rendered. As the destructor of the FrameBuffer class destroys the texture, the final render function is trying to access some memory that no longer contains a texture, resulting in a black screen.

To test if this was the case, I made the fb object into a member field in the Renderer class, and removed the variable from the renderSimulation function.





After finishing that it now works properly, the background colour comes up, and the test rectangle shows up.

### Adding to Mainloop

Like the Simulation class, adding it to the mainloop is not that hard.

#### Application.h

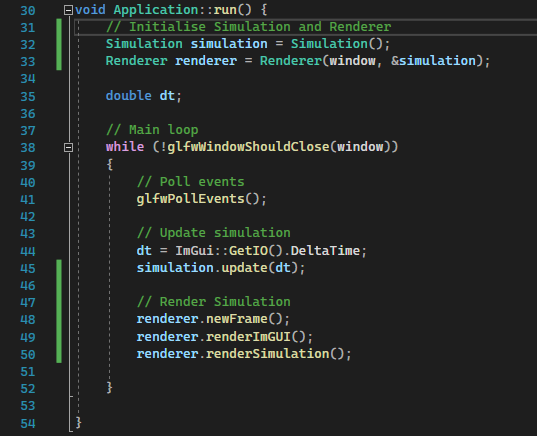
First I included the file in the header file.

Text

Description automatically generated

#### Application.cpp

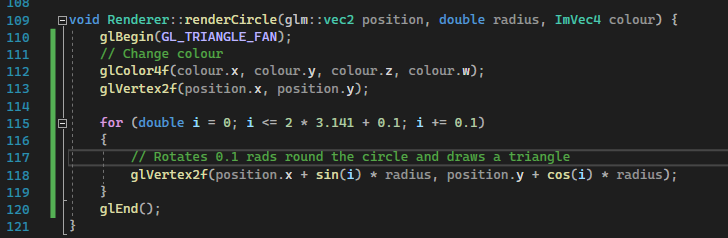
Then I initialised the Renderer class and added the rendering functions to the main loop.



### Rendering Shapes:

#### Render Circle:

The render circle function essentially draws lots of triangles from the centre of the These triangles form a shape that looks like a circle. This works as the individual triangles are very thin, the angle at the centre of the circle is only 0.1 radians.



#### Render Polygon:

The render polygon function iterates through all of the relative position of the vertices and draws a triangle from the shape position to the actual position of the vertex.

Text

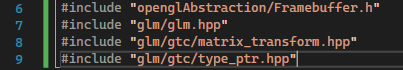
Description automatically generated

### Projection Matrices

When rendering to an OpenGL window, the centre of the screen is position (0, 0), the top left is position (-1, -1), the top right is (1, -1), etc. This coordinate system can be difficult to work with, especially as I want the window to be resizable. Rather than calculating the position every time I want to render to a screen, I can use a projection matrix, this means that the calculation will be done on the GPU and will be much more efficient and easier to work with.

#### Renderer.h

To implement the projection matrix, I first included the matrix and projection matrices hpp files from the glm library. Then I added a projection matric member to the Renderer class.





#### Renderer.cpp

Then I setup the projection matrix at the beginning of the renderSimulation function.

Text

Description automatically generated

Now, the projection matrix is setup, this also allows the window to be resizable.

## Rigidbodies:

This section is focused on creating all of the rigidbody classes.

### Body

First we have to update the Body class, we add a position vector, a velocity vector, a force and impulse vector, mass, colour and an isStatic variable.

#### Body.h

In Body.h, we predeclare the Renderer class, we do this to prevent circular imports as the Renderer class will include the Body class.

Graphical user interface, text

Description automatically generated

#### Body.cpp

Then we add the getters to Body.cpp

Text

Description automatically generated

### Rigidbody:

The Rigidbody class is the base class of all of the rigidbodies. It is a subclass of Body.

#### Rigidbody.h

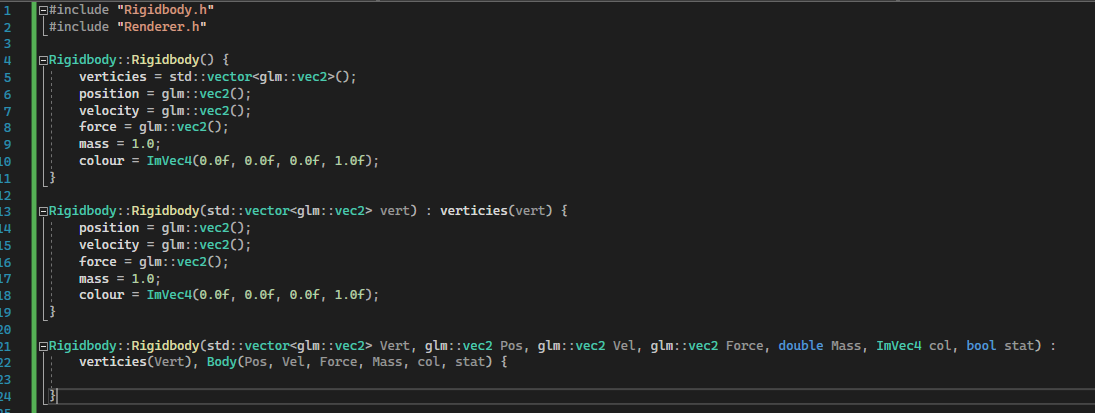
The Rigidbody class has 3 constructors, each containing a different number of inputs, most of the time, I will be using the third one as it specifies all of the member variables. It also overrides the render and update function of Body. The main difference between this and the Body class is that this class contains a list of vertices.

Text

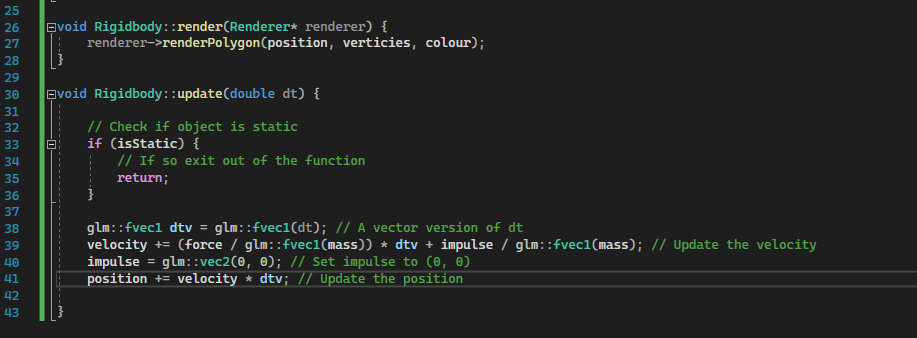
Description automatically generated

#### Rigidbody.cpp

These are the constructors, they simply initialise all of the member variables.



The render function takes in a pointer to a renderer, and then calls the renderPolygon procedure from said renderer. The update function takes in a double dt, this is the change in time, it checks if the object is static, if it is, then it returns out of the function. If not, then it calculates the velocity of the object, and then updates the position.



### Circle:

The circle is a Rigidbody that has no vertices and instead has a radius. This means that it will have a few different parts than Rigidbody.

#### Circle.h

The circle contains a member variable radius, the constructor takes in the

Text

Description automatically generated

#### Circle.cpp

The constructors initialise the radius and then calls the Rigidbody constructor with no vertices. The render function calls renderCircle instead of renderPolygon.

Text

Description automatically generated

### Rectangle:

The Rectangle class is a sub class of Rigidbody with a set 4 vertices, it takes in width and height as parameters. All of the other parts of the Rectangle are the same as the rigidbody, including rendering and updating.

#### Rectangle.h

It contains a method called sizeToVertices, this converts width and height to a list of verticies.

Text

Description automatically generated

#### Rectangle.cpp

The sizeToVertices halves the width and height and then uses those to find the 4 corners.

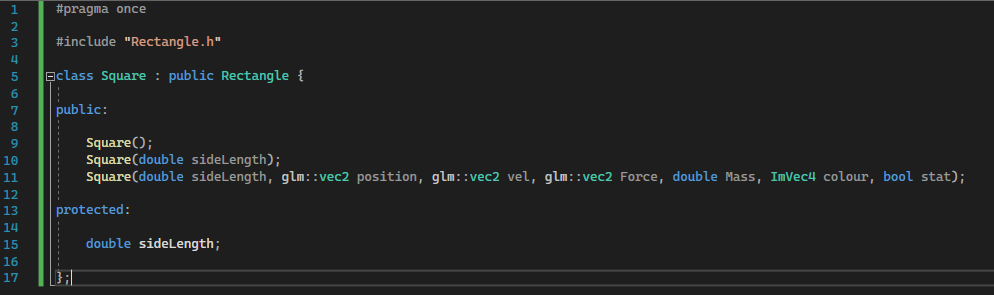
Text

Description automatically generated

### Square:

The square is a special case of the Rectangle where the width and the height are equal, because of this, we can just implement the square as a subclass of rectangle where the width and the hight variables are one singular variable.

#### Square.h



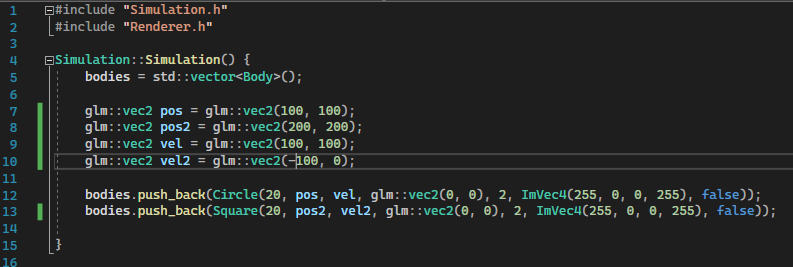
#### Square.cpp

A screenshot of a computer

Description automatically generated with medium confidence

### Bugs:

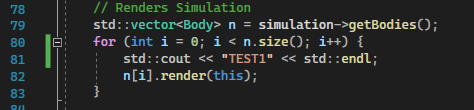
While checking if the rigidbodies rendered. I discovered an issue, none of the rigidbodies are rendering. I added this code to the Simulation construction:



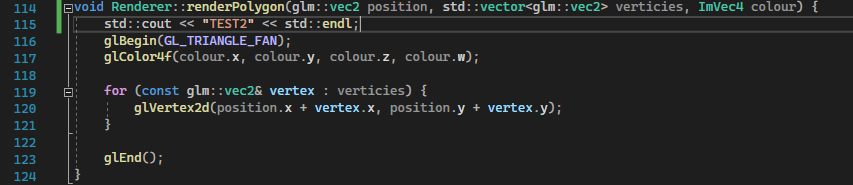
This adds a circle and a square to the bodies list. But when I compiled the program, neither of them showed up.



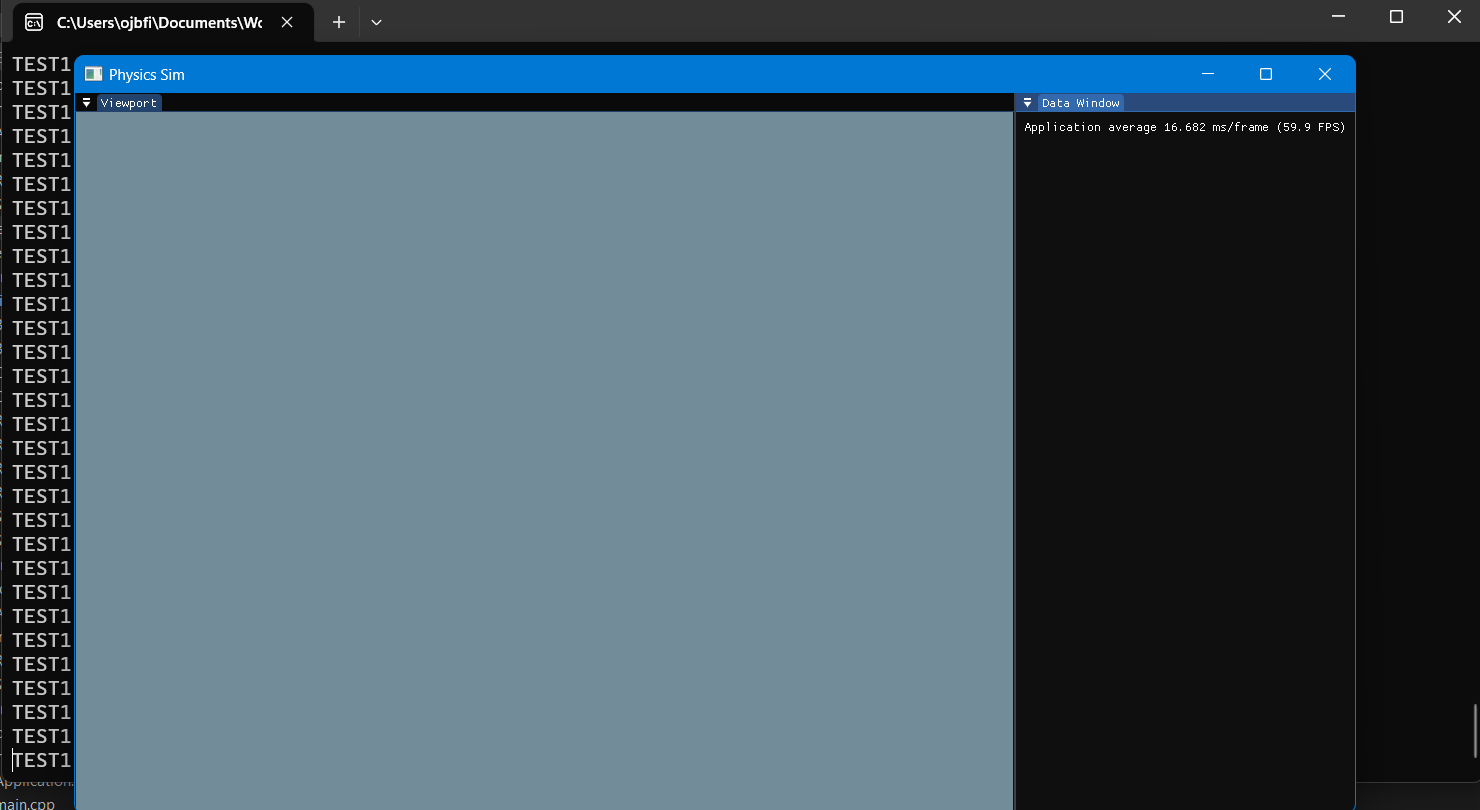
To test if the render loop was working, I added this line to the renderSimulation function in Renderer.cpp to check if it was being called (std::cout outputs to the console, a bit like a print function).



And this line to renderPolygon.



When I compiled it this time, nothing showed on the screen, but “TEST1” showed on the terminal window, but not “TEST2”.



This told me that the render loop function was being called, but not the renderPolygon function. I added another std::cout to Rigidbody in the render function.

Text

Description automatically generated

This didn’t show up in the console either.

Graphical user interface

Description automatically generated with medium confidence

This meant that the Rigidbody render function wasn’t being called. I then added another std::cout to the Body render function. I did this to test if my polymorphism was working correctly.

Text

Description automatically generated

Shape

Description automatically generated

This test was showing in the console. This told me that polymorphism wasn’t working properly.

When doing some research online, I discovered that std::vector (the dynamic array that I was using to store the objects) didn’t work properly with polymorphism. To get over this, one page suggested using an std::vector of pointers to objects.

First, I defined PBody as a shared pointer, I used a shared pointer as it does not free when the original goes out of scope, and can be used by multiple different functions. I defined it as PBody so that I didn’t have to write out the whole thing every time.

I defined it in Simulation.h

Graphical user interface, text

Description automatically generated

Then I changed all of the std::vector<Body> to std::vector<PBody> in Simulation.h.Text

Description automatically generated

Then I changed it over in Simulation.cpp, I also had to make a few changes to the syntax used, when adding new objects, I had to use std::make\_shared to create a shared pointer, and I had to change any bodies[i].function() to bodies[i]->function() as they are pointers now.

Text

Description automatically generated

And another change in Renderer.cpp

Text

Description automatically generated

Now when I compile, I get a square and a circle appear on the screen, and my test functions are also being called.Graphical user interface

Description automatically generated with low confidence

## Collisions:

The collisions have two main parts, detecting collisions and resolving collisions. As you need to detect a collision before you can resolve it, I will focus on detecting collisions first.

### Detecting Collisions:

This is where the SAT algorithm comes into play. But before that, I need to add a method to get the bodies that will collide. For most rigidbodies, this will just return themselves, but for softbodies (which will be implemented later), it will return all of the points that make up the softbodies.

I add a method to the Body class.

Text

Description automatically generated

Text

Description automatically generated

Next, in the Rigidbody class I override the function to return itself in the vector.

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

#### Projection:

Projecting a shape onto a line involves projecting each individual vertex onto the line, and then finding the two most extreme values that represent the max and min of the “shadow”. To project a vertex onto a line, we find the perpendicular to the line that passes through the point and find the point of intersection between the line and its perpendicular.

Chart

Description automatically generated

To store the min and max, I will use a glm::vec2, with the x value representing the minimum and the y value representing the maximum. I will also a glm::vec2 to represent the projection line, which I will call the axis.

##### Body:

To implement it, I first added a project method to Body, the method only returns an empty vec2. This is because the default Body does not have any vertices to project

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

##### Rigidbody:

We override the method in Rigidbody. First, we declare it in Rigibody.h

A screenshot of a computer

Description automatically generated with medium confidence

Then we add the implementation in Rigidbody.cpp

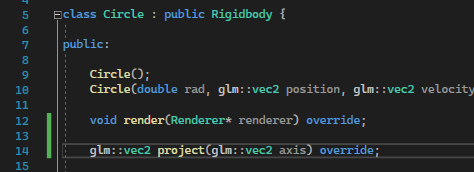


To find the distance along the axis, we use the dot product between the axis and the position of the vertex.

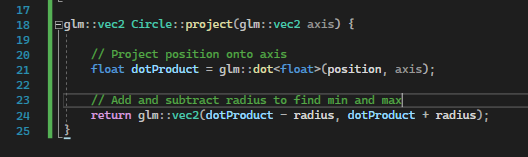
##### Circle:

Circles are a special case because they have no vertices to project. Instead, we can project the position of the circle onto the axis, then add the radius to find the maximum, and subtract it to find the minimum.

First, I add the project to Circle.h



Then I add the definition to Circle.cpp



#### Finding Axes:

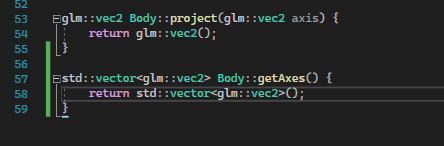
The next step is to find the axes to compare the projections to, as mentioned in the design, the axes are the perpendicular to the edges of the rigidbody. An edge can be found by finding the difference in position between two vertices (ie vertex1 – vertex2), an edge is also stored as a vec2.

##### Body:

First we add a method to the Body class, declaring it in Body.h

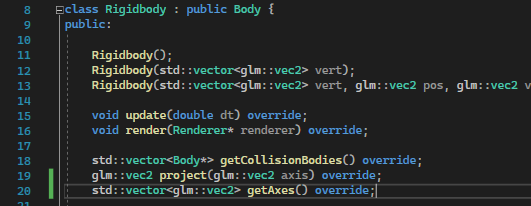


And then defining it as return an empty vector in Body.cpp

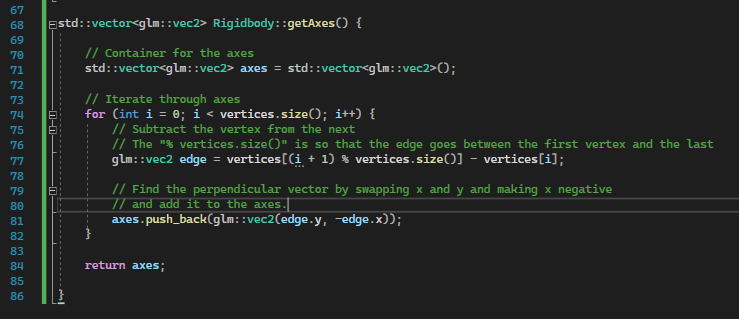


##### Rigidbody:

Then in Rigidbody.h we override it.



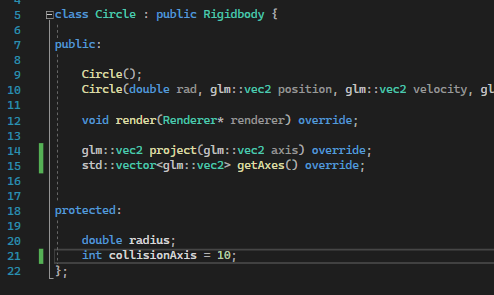
In Rigidbody.cpp, we define the function. It iterates through the vertices, and then finds the edge between that vertex and the next one in the list. If the vertex is the last one in the list, it finds the axes between the last vertex and the first.



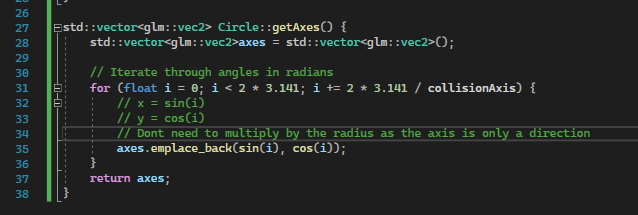
##### Circle:

Circle is a special case again, because it only has one edge (or infinite depending on how you look at it), to save computations, I will model the circle as though it has ten edges, this makes finding the axes very easy.

First, we override the function in Circle.h, I also added a variable to store how many edges I was modelling, called collisionAxis.



Then I implement it in Circle.cpp.



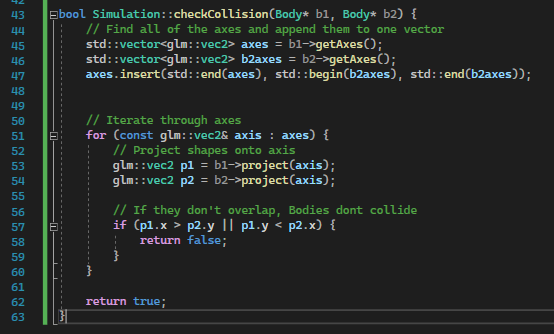
#### Check Collision function:

The checkCollision function is the function that checks if two Bodies have collided, this is where the SAT algorithm is executed. It returns a boolean value that tells if the objects have collided or not. It finds all of the axes to project onto, then compares if the objects overlap on those axes, if they are separate on any axis, it returns false, otherwise, it returns true.

First, I declare the function in Simulation.h



Then, I add the functionality in Simulation.cpp.



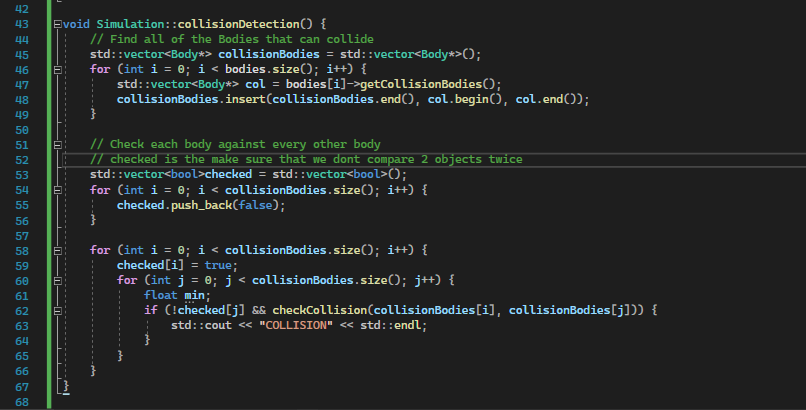
### Collision Detection function:

This is the function that calls the collision detection and collision resolution functions. I am choosing to implement it before I implement collision resolution to test if the collision detection algorithm works. That way I know that if it doesn’t detect the collision, it isn’t a problem with the collision resolution (and visa versa). For now, it iterates through all of the bodies and checks for a collision between them. If there is, it will output “COLLISION” to the console.

First, I declare it in Simulation.h.



Then I add functionality in Simulation.cpp.

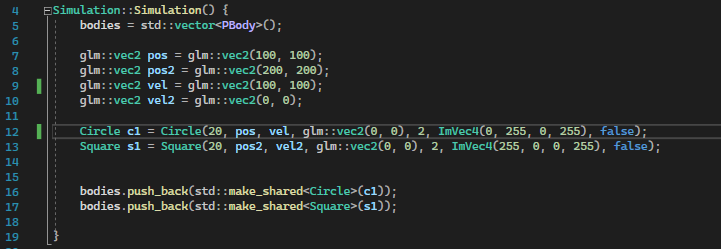


Next, I needed to add the collisionDetection function to the update function.

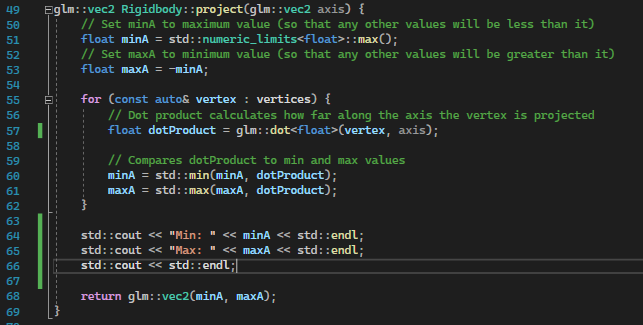


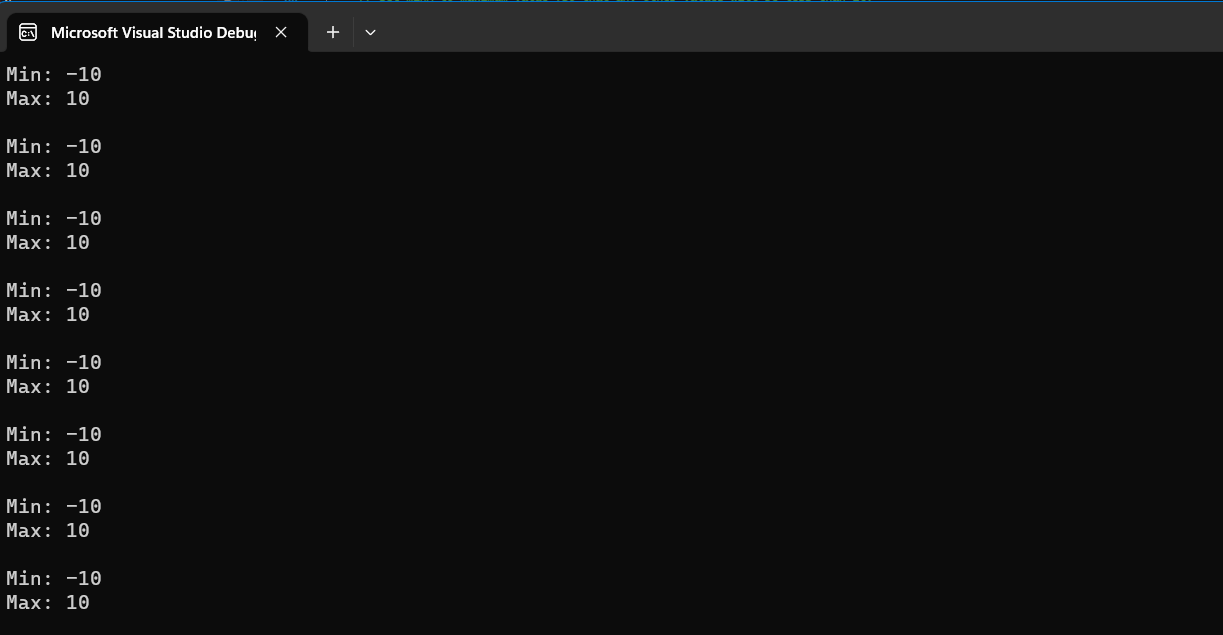
### Testing Collision Detection:

To test if my collision detection worked, I changed the two starting objects in Simulation.cpp to have one stationary, and one moving towards the other, and the circle to be green, this makes it much easier to see if a collision occurred with my eye.

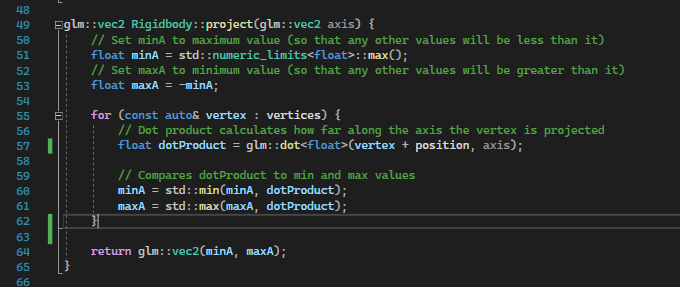


When I ran the program, I saw that no collisions were being detected, (see video named CollisionDetectionTest1.mp4). I checked, all of the functions were being called, but no collisions were being detected. When I checked the values of the dot product being produced with the code below, I found that the rigidbody projections were off, and weren’t changing based on the position.





When looking over the code, I realised that I was only projection the relative positions of the vertices onto the axis, not the actual position, so I changed the dot product to be between the axis and the position plus the relative vertex position.



When I ran it, it worked, collisions were being detected, (see video named CollisionTest2.mp4).

### Resolving Collisions:

Resolving collisions is made up of two parts, position correction and impulse calculations. Position correction makes sure that the two objects never actually intersect, the impulse calculations applies an impulse to each Body, making it so that the objects velocities are accurate post collision. Impulse also allows multiple collisions to happen at once.

Before I implement those, I will first add a resolve collision function to the Simulation class.

Declaring it in Simulation.h

Text

Description automatically generated

Then I define it in Simulation.cpp.

Graphical user interface, text, website

Description automatically generated

Then I replace the ‘std::cout << “COLLISION” << std::endl;’ with collisionResolution in Simulation.collisionDetection().

Text

Description automatically generated

#### Position Correction:

Position correction makes sure that neither of the objects are clipping. To do this, it calculates the “most penetrating vertex”, this is the vertex that is closest to the other objects centre, then it finds the closest edge to that object, then moves the objects so that they no longer intersect.

First, I declare the positionCorrection function in Simulation.h



Then I add functionality in Simulation.cpp:

# Glossary

Rigidbody – A body which cannot be deformed (think a plank of wood).

Softbody – A body that can be deformed (think a bouncy ball or jelly).

C++ vector – Part of the C++ standard library, acts as a dynamic array.

Polygon – A closed 2D shape made of 3 or more vertices.

Linux – A group of free open source operating systems, often used by developers due to its customisability

Buffer – A set of data that is passed to the GPU, or used to store data. Often stored as a large array.