

## Contents

<b>Analysis.....</b>	3
<b>Problem Identification.....</b>	3
<b>Stakeholders.....</b>	4
<b>Research The Problem.....</b>	5
<b>Existing Similar Solutions .....</b>	5
Parts which can be applied to my solution (1).....	5
Parts which can be applied to my solution (2).....	6
Parts which can be applied to my solution (3).....	8
<b>Features of the Proposed Solution .....</b>	9
Initial Concept .....	9
Limitations of my solution .....	9
<b>Specify the Proposed Solution.....</b>	10
<b>Requirements.....</b>	10
Hardware .....	10
Software.....	10
Stakeholder Requirements .....	10
<b>Success Criteria .....</b>	13
<b>Design &amp; Development .....</b>	15
<b>Decompose the Problem .....</b>	15
<b>Top-Down Design:.....</b>	18
<b>Prototype 1 Design .....</b>	19
<b>Test Plan.....</b>	51
<b>Key Classes, Subroutines and Variables .....</b>	52
<b>Prototype 1 Development .....</b>	57
<b>Testing Prototype 1 .....</b>	93
<b>Success Criteria Review .....</b>	99
<b>Prototype 2 Design .....</b>	100
<b>Dynamics (objects moving) .....</b>	100
<b>Object rotation.....</b>	101
<b>Finding angular velocity from applied forces.....</b>	104
<b>Collision Detection.....</b>	106
<b>Collision Response .....</b>	109
<b>Note on Post-Development Testing .....</b>	124
<b>Test Plan.....</b>	125

<i>Key Classes, Subroutines and Variables</i> .....	127
<b>Prototype 2 Development</b> .....	130
<i>Testing Prototype 2</i> .....	151
<i>Success Criteria Review</i> .....	167
<b>Prototype 3 Design</b> .....	168
<i>Menus</i> .....	168
<i>Accessibility</i> .....	170
<i>Differences between each simulation</i> .....	173
<i>Note on Post-Development Testing</i> .....	173
<i>Test Plan</i> .....	174
<i>Key Classes, Subroutines and Variables</i> .....	176
<b>Prototype 3 Development</b> .....	178
<i>Main Menu &amp; Settings</i> .....	178
<i>Input Validation</i> .....	183
<i>Implementing The Ground Plane</i> .....	187
<i>Simulation Menus</i> .....	201
<i>More Input Validation</i> .....	211
<i>User Created Objects</i> .....	216
<i>Refining/Differentiating Each Simulation</i> .....	246
<i>Testing Prototype 3</i> .....	264
<i>Success Criteria Review</i> .....	283
<b>Evaluation</b> .....	284
<i>Testing to inform evaluation</i> .....	284
<i>Testing for Function</i> .....	284
<i>Testing for Usability</i> .....	306
<i>Evaluation of Solution</i> .....	309
<i>Cross referencing evidence with success criteria:</i> .....	309
<i>Addressing incomplete criteria in further development</i> .....	310
<i>Explain how tests from before demonstrate usability</i> .....	311
<i>Maintenance issues and limitations</i> .....	311
<b>All project code:</b> .....	312

# Analysis

## Problem Identification

Physics can be a very difficult subject to learn. One very important topic in physics is forces and how they cause objects to move. Some students learn best from diagrams, though others may find this style of learning challenging and are visual learners. It can be difficult for a teacher to set up a physical experiment to demonstrate forces and how objects interact when they collide. It may therefore be useful to have a piece of software capable of simulating these interactions, as the teacher would have full control over the conditions, e.g., forces used, the strength of gravity etc., which can make it easier for the teacher to teach and for the students to learn.

This problem will require problem recognition because it is important to clearly identify what the goal of a given algorithm is and understanding clearly what the problem is will be necessary to solve it. For example, an algorithm to project three-dimensional objects to a two-dimensional screen would need to be clearly defined before it can be solved by an algorithm. The problem will also require decomposition as this problem consists of many parts such as creating a function to render a three-dimensional object, calculating how object should move and interact, and simply creating a window and menu. The problem can be decomposed into these parts, each of which can be decomposed into smaller and more specific parts until each individual problem is simple and easy enough to solve. I will be using this computational method because it will make it much easier to design and implement more complicated features. Abstraction will also be useful because there will be many situations where certain pieces of information will not be necessary for a given algorithm. For example, the force acting on an object will not be necessary in a rasterisation algorithm to display the object on the screen.

This project is amenable to a computational approach because other methods may not be as effective. As stated earlier, different students learn differently. For some, a program like this could be far more beneficial than a diagram or paragraph explaining the concepts of forces and collisions. The software could also be used to predict real-life events, such as where a ball will end up if thrown with a certain force in a certain direction. A piece of software would be better than doing these calculations by hand as that would be very difficult to do if there were many objects interacting with each other and many different forces involved. Overall, this project is suited to a computational approach as it involves many complex algorithms and calculations and may also be an easier way for some students to learn than other methods.

## Stakeholders

The stakeholders for my project will be physics teachers, for example the physics teachers at my school. My stakeholders are those specifically teaching younger students as the software will be used to teach more fundamental concepts taught earlier on. The solution will provide my stakeholders with an easy, new, convenient way to teach their subject. It will allow them to have full control over the simulation. They will be able to use this solution to demonstrate a specific scenario or concept (such as friction or terminal velocity perhaps) to the class as opposed to a set of premade simulations. This freedom is essential for the stakeholders as they must be able to customise the simulation in order to effectively use it as a teaching tool which will allow them to teach their class certain concepts more easily. This solution will achieve this by offering them an intuitive way to teach their students the concepts in a digestible and easy to understand way.

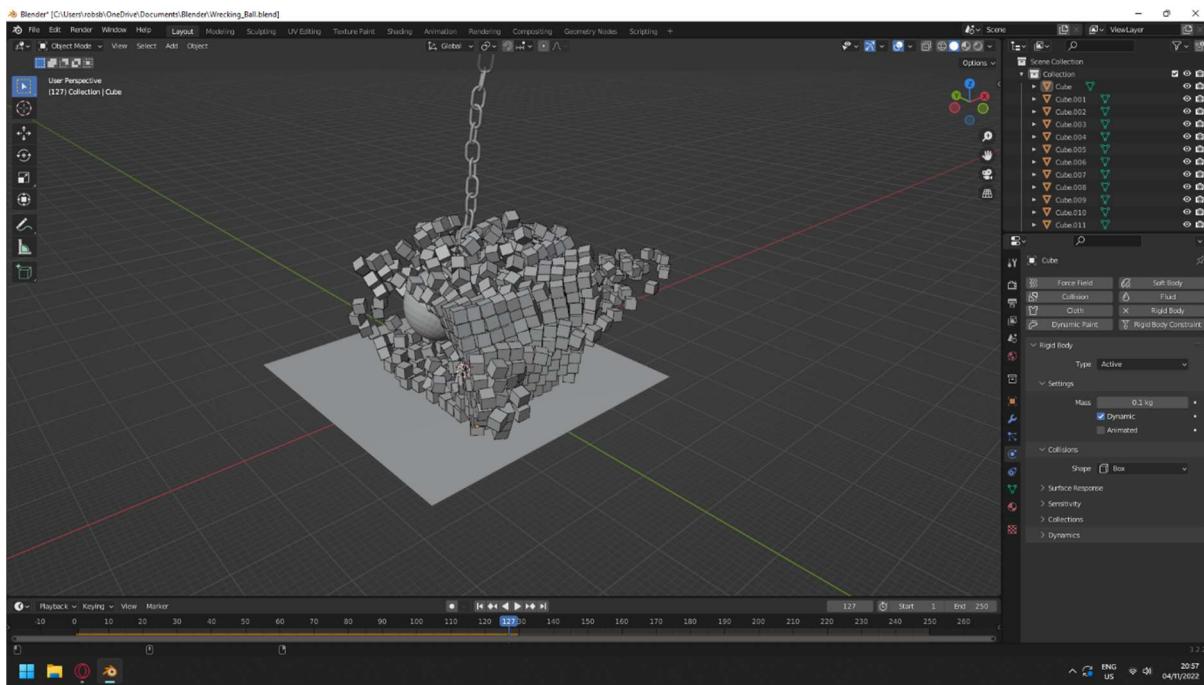
Students could also be stakeholders as they could use the software to experiment and learn about forces for themselves. They would require a piece of software which is easy to use and does not require much time nor effort to be able to use well. They will be able to use the solution by running certain simulations so they can understand more intuitively the physics taking place within them. This will allow them to learn more about the subject without being confined to a classroom and making their education more interactive and fun. This is appropriate for their needs because as students it is important that the resources available to them are accessible and easy to use without prior knowledge. The ease of use of the program is for this reason important as it will allow students to be able to properly utilize the program.

# Research The Problem

## Existing Similar Solutions

### Blender:

#### Overview



Blender has all manner of physics simulation tools such as **rigid body simulation**, soft body simulation, fluid simulation and more.

Blender is primarily a professional rendering software. This means that physics is not the focus meaning that Blender is not **lightweight** and can be very challenging for beginners to grasp. It is expected that one spends time learning how to use the software from third parties, such as internet tutorials because it is not an easy software to use intuitively due to having a variety of shortcuts (referred to as "hotkeys"). This type of software has its advantages, for example it is much more capable than my software is likely to be, but with the downside that it is not very beginner friendly and as mentioned, physics is not Blender's main priority.

#### Parts which can be applied to my solution (1)

Blender has a very capable rigid body simulator, (a rigid body is an object which cannot be deformed.) I hope to create a rigid body simulator with similar capabilities, but more lightweight and intuitive than Blender. Graphics will also not be a focus in my project because that would not be useful to my stakeholders. Specific features I am looking to implement are:

- Collisions, as it is important for object to be able to interact with each other.
- Rotation, because rotational physics is an important part of rigid body simulation.
- The ability to set objects to active (so physics will act on them) or passive (physics will not act on them though objects can still interact with them e.g., a wall or the ground).

This will allow the user more freedom in terms of how objects will interact in the simulation allowing the software to be used in more specific situations.

My software will attempt to improve upon the following:

- Making the software more lightweight and more focused on physics, which will decrease file size and increase usability by making it less cluttered and confusing.
- Better ease of use, because blender is not very beginner friendly and can be very difficult to learn how to use if you are not already familiar because it is important that the user will be able to easily use the program as they may be relatively young (e.g., a student) or may not have the free time to learn how to use it (e.g., a busy teacher).

## Unity:

### Overview



Unity is a game engine designed to make it much easier for developers to create games without needing to worry about the backend such as rendering and physics. Unity has a built-in rigid body simulation system which works in **real-time**. The rigid body simulation engine in unity is very powerful. This video from the YouTube Channel “Brackeys”: ([https://www.youtube.com/watch?v=8zo5a\\_QvJtk](https://www.youtube.com/watch?v=8zo5a_QvJtk) at 1:15) shows a framerate of about 30 fps even when doing physics calculations on 3000 objects at once. Unity is a game engine, not a physics engine. This means that, for the purposes of just a physics engine it is not very **lightweight** which makes it difficult for people to use if they don't want to or can't spend the time to learn how to use it. This also means that it would take up far more storage space than would be necessary for a physics engine alone.

### Parts which can be applied to my solution (2)

Unity, similarly, to Blender, has a great built-in physics engine, but at the end of the day, that is not the software's main purpose. This means that it takes up a lot of storage space and can also be difficult to use. My software will aim to implement the following features as seen in the Unity game engine:

- Real-time simulation so that the user may see how systems behave in real-time as opposed to waiting for a simulation to finish over a long period of time. This is especially important in a classroom as there is a limited amount of time to teach an individual lesson.
- The ability to simulate multiple objects without drastically decreasing performance because seeing how many objects interact with each other at once could be used to demonstrate certain concepts such as chaos theory.

My software will attempt to improve upon the following:

- Usability, because my stakeholders likely do not have the time to learn unity in their spare time, so having the software be immediately easy to use with minimal instruction (though some may still be necessary) is essential.
- More lightweight, as this would reduce file size, reduce clutter and make the program load and run smoother as well as making it overall easier to use due to less clutter and complicated menus.

## My Physics Lab:

### Overview

**Physics Simulations**

Click on one of the physics simulations below... you'll see them animating in real time, and be able to interact with them by dragging objects or changing parameters like gravity.

myPhysicsLab.com  
English previous next

### Customize and Share

There are several ways to reproduce a particular experimental setup. The easiest way is to click the "share" button.

1. Modify the simulation by changing parameters such as gravity, damping, and by dragging objects with your mouse.

2. Click the "share" button. Copy the URL from the dialog.

MyPhysicsLab.com is an online tool with a multitude of physical simulations, including “pendulum”, “newton’s cradle” and “rigid body collisions”.



This tool is very accessible and easy to use as it does not even require a download in order to use. The wide **variety** of demonstrations available is also impressive and certainly makes the tool useful in a wider variety of situations. The tool is made **specifically** for physics simulations which means it is very **focused** on that individual task and is therefore a very **lightweight** tool.

This tool is one already used and recommended by my one of my potential clients (my physics teacher) which shows that it definitely has certain features which would be worth taking inspiration from.

#### Parts which can be applied to my solution (3)

My Physics Lab (MPL) is an excellent online tool which physics teachers can use to demonstrate physics. These are some of the most important features which I hope to implement:

- I like that this software is very focused and lightweight, because this makes it easier to use, avoids unnecessary clutter, such as menus and options which the user will never use, and reduces the overall file size as well as loading times.
- The wide variety of simulations makes the software more useful in a wider variety of scenarios, meaning a variety of simulations is something I should aim to implement as the stakeholder would be able to use it in a wider variety of more specific cases.

My software will attempt to improve upon the following:

- My software will be three-dimensional. I believe this to be important because a 3d simulation is more accurate to the real world and may make it easier to intuitively understand what is going on which is important because one of the main goals of this solution is to be as intuitive and easy to use as possible.

## Features of the Proposed Solution

### Initial Concept

My solution will be a piece of software which will be able to simulate a variety of different physical phenomena in three dimensions. This will include a rigid body simulation in which simulated objects will be able to rotate and collide. The user will have the option to anchor certain objects to prevent them from being able to move, such as a floor or walls like in Blender, which will give the user more control, allowing them to utilise the software in more specific circumstances. My software will be able to simulate multiple objects at once and should still be able to run at a reasonable framerate (30+ fps), like in unity, which makes the program less jarring and makes it easier to understand what is happening in the simulation. My software will be lightweight, like My Physics Lab, meaning there should not be any unnecessary features which would only serve to make the program more difficult to navigate.

There will be a menu from which the user may select a simulation. The user will then be able to change certain parameters to see the effect which will be had. These parameters may be changeable in a side menu (like in My Physics Lab) or perhaps in perhaps in the form of a pause menu which could be accessed by pressing “esc” or some other key.

### Limitations of my solution

One limitation of my solution is that it will not be as accurate to the real world as other software, such Blender. This is because it is necessary to sacrifice precision for performance, as I believe that in this case, the latter is more important because due to the limited time frame for a teacher to teach, there is simply no time to pre-calculate the simulation before running it (this is the method used by Blender). This is justifiable as there is a need for balance between performance and precision, meaning at least one of the two will be limited. The choice to opt for performance is justified as this will allow simulations to be more reasonably run in real-time making them easier to follow and comprehend, and intuitiveness is one of the key goals of the solution.

Another limitation is that my software is unlikely to be as efficient as possible. Due to the nature of the problem, creating a solution which is as efficient as possible would be a very difficult undertaking, even for a larger group of people. I alone will be unable to optimize every aspect of the program, meaning that performance will be limited, at least to an extent. This should not be too important as large-scale simulations are not the aim here, so efficiency does not need to be incredible.

## Specify the Proposed Solution

### Requirements

#### Hardware

A computer with a relatively powerful **CPU** due to the number of calculations per second, though a powerful **GPU** is **not** required as the graphics will be relatively low in quality, although it may be helpful as the graphics are still three-dimensional.

Appropriate **peripherals** such a **mouse** and **keyboard** for inputs and a **display** (likely a smart board in a classroom setting) to display the simulation on.

To run WinForms applications: (512MB RAM minimum, 1GB Recommended)

#### Software

A computer capable of running WinForms applications (Windows 2000 or higher, Visual Studio, .NET Framework, .NET Core)

### Stakeholder Requirements

#### Design

Feature	Justification
Menu screen with a list of options for different simulations.	Allows the user to decide which simulation to demonstrate, e.g., a teacher may want to teach about a specific phenomenon, and so could select the appropriate simulation for it.
Lightweight	Minimise clutter and unnecessary features and options to make the program easier to use and take up less storage space.
Clearly labelled options, e.g., naming each simulation and giving it a relevant image, naming certain settings, etc.	This will make the program easier to use as it will be clearer what each option does.
Simple to do things in the simulation.	It should be relatively easy for the user to understand how they can affect the simulation, e.g., by creating objects or applying forces to existing objects.

## Functionality

Feature	Justification	Why the feature makes the problem solvable by a computational approach
Create a resizable window with contents which will dynamically resize with the window	This feature is important as it allows the teacher to multitask as that the window doesn't need to fill the whole screen, and if it does the image will not be cropped.	This feature would require me to take a given pixel (e.g., coordinates (0.5, 0.5) which would be at the centre of the screen) and work out where in the window it should be displayed based on the current size of the window. This may or may not be necessary as this feature may be present in any libraries I use to create a window.
Rasterization which allows a set of 3d coordinates to be given a point on a 2d screen given the position and direction of the camera, as well as other variables such as the field of view and focal length.	This feature is important as it allows the objects being simulated to be presented in 3d space as you would expect to see them in real life. This is important because having the software mimic real-life (as opposed to having only two dimensions) makes it easier for students to understand how forces act in the real world. Rasterization is more suitable than path tracing as not many objects will be rendered at any given time and the main benefits of path tracing include realistic shadows and light, which is not the main purpose of the project so is not worth the extra computational, or development time.	A rasterization algorithm will require complex calculations to be performed on certain variables. It lends itself to a computational approach because it involves repeating the same set of steps over and over many times per second to create the appearance of three dimensions.
The ability for the user to apply a force to an object to see how it behaves and interacts with other objects.	This feature is essential as the purpose of this software is to allow the teacher to have full control over the simulation. This also allows the students to learn better as they can ask more specific questions, such as "what would happen if you increased the force acting on the object or changed the forces direction?" This feature would allow this to be possible	This feature lends itself to a computational approach as applying a force to an object will change its acceleration, speed and potentially direction. These could all be attributes of the object meaning this feature lends itself especially well to an object-oriented approach. Many repetitive, complex calculations would need to be carried out many times per second which is something very suitable for a computational approach.

	as it will allow the user total freedom.	
The ability for objects to collide and interact realistically.	This feature is important as it would allow Newton's third law of motion to be demonstrated. It would allow the software to do more than just show an object moving, which itself may be difficult to see as there are no other reference objects. Having multiple objects interacts demonstrates what happens when they collide but also makes it easier to understand their motion.	This feature lends itself to a computational approach as each frame the software much run an algorithm on each pair of objects to check if they are overlapping (this algorithm would itself likely be rather complex) and if they are a second algorithm would calculate how they should respond to the collision (another complex algorithm), a final algorithm would then apply this force to the objects, and they would then move appropriately. These algorithms would all likely be quite complex and would all need to work together to solve the problem.

## Hardware and Software

Requirement	Explanation
Relatively powerful CPU.	Due to the large number of calculations being performed each second, a relatively powerful CPU will be necessary to meet the demand.
Suitable Peripherals.	A mouse and keyboard will be necessary to navigate menus, input values, such as when applying forces to objects, and overall using the application. A display of some kind (e.g., a monitor or a smartboard) will be necessary to output the simulation to be viewed on a screen so that students in the classroom can observe the simulation as it takes place.
Minimum 512MB RAM minimum, recommended 1GB RAM.	Required to run WinForms applications.
Windows 2000 or higher, Visual Studio, .NET Framework, .NET Core	These are the requirements to run WinForms applications.

## Success Criteria

Criteria	How to evidence	Justification
Main window which is moveable and resizable.	Video of window being moved and resized.	Having a window which can be moved and resized gives the user more freedom. For example, the teacher may use this feature to show their students multiple things at once side by side.
Menu screen.	Screenshot of menu screen with different options such as “select simulation”, “settings” and “Exit application”.	A menu screen allows everything to be available from one central place. This makes the program more intuitive and easier to use.
Lightweight and easy to navigate.	Screenshot of menu screen not being cluttered.	Having the menu be easy to navigate makes the program much more user friendly which is important as the end user (the teacher) may not have enough free time to learn how to use a new complicated piece of software.
Multiple different options of different simulations.	Screenshot of a menu screen showing these different options.	This would allow the end user to use the program for multiple different purposes. Instead of using it for a single lesson and never again, the program can be reused to teach the students about different topics.
3D rasteriser.	Screenshot of simple objects such as a cube being rendered in 3d space.	This is essential as it allows the objects being simulated to be rendered in three dimensions without too much computation. This is important because the computer being used is unlikely to be very high-end and powerful.

Shading. (Giving different faces of an object different levels of brightness based on how they are oriented.)	Screenshot of an object such as a cube having different levels of lighting for each face based on orientation.	This is important because having objects rendered in 3d space with no way to differentiate between the different faces of each object can lead to the three-dimensional object to look like a single two-dimensional shape which would make it difficult to understand what is happening in the simulation.
Working rigid body motion.	Video of rigid body objects moving.	This is a very important feature as it allows the teacher to use the program to teach about the physics of motion.
Working rigid body collision.	Video of rigid bodies colliding with each other.	This may allow the teacher to demonstrate Newton's third law of motion, a key concept in physics.
Normal reaction force	Image/Video of a cube resting on a stationary plane (despite gravity).	This prevents objects which are intersecting (but not moving so not a collision) from accelerating through each other.
Options next to each simulation such as creating objects, applying forces to objects etc.	Screenshot of simulation running with a list of options next to it.	This feature would allow the user to have more control over the simulation which makes it much easier to ask questions and to use the program as a tool for learning.
Camera control.	Video of the camera being moved and rotated by the user.	This will allow the user to have more control over their perspective allowing them to get a good angle to best view an ongoing simulation.

# Design & Development

## Decompose the Problem

The solution will consist of a few main systems:

- A system for each of the simulations to run.  
Link to success criteria:
  - Multiple different options for different simulations.
  - 3D rasteriser
  - Shading.
  - Working rigid body motion.
  - Working rigid body collision.
- A system to allow the user to select and close each simulation separately.  
Link to success criteria:
  - Main window which is moveable and resizable.
  - Menu screen.
  - Lightweight and easy to navigate.
  - Multiple different options for different simulations.
  - Normal Reaction Force.
- A system to allow the user to input to the simulation.  
Link to success criteria:
  - Lightweight and easy to navigate.
  - Options next to each simulation such as creating objects, applying forces to objects etc.

Each of these systems will need to be further broken down:

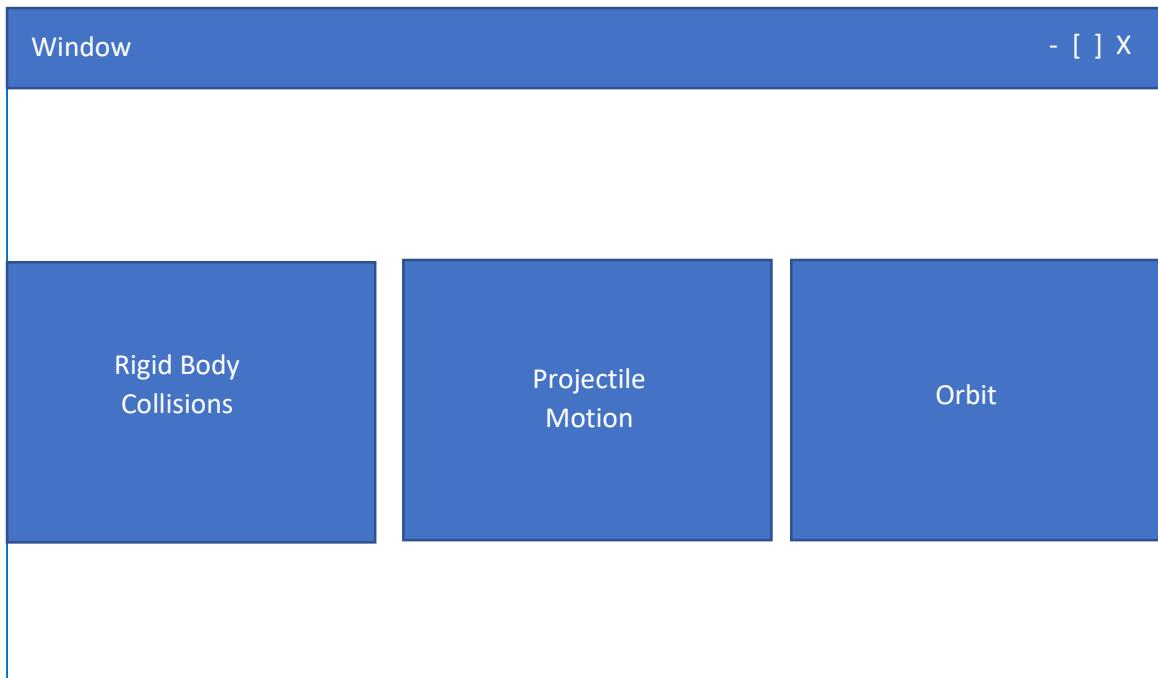
### A system for each of the simulations to run:

- A three-dimensional rasterizer for multiple objects at once
- A rigid body physics engine in three-dimensions (Simulation 1)
- Projectile Motion (Simulation 2)
- One body orbiting another (Simulation 3)

(The specific simulations are subject to change)

### A system to allow the user to select and close each simulation separately:

- A window with a selection menu on it



- The ability to select an option by clicking on it which will then close the menu and launch the simulation
- A button which, when clicked, will close the current simulation and open the menu shown above

A system to allow the user to input to the simulation:

- A series of options and buttons which can be used by the user to influence the current simulation.

Each of these can be broken down further and so on.

A three-dimensional rasterizer for multiple objects at once:

- Objects will be split into faces which will be split into points
- Create a function which takes the inputs of the position and direction of the camera and the position of a point to work out where on the screen that point should be displayed
- Create a function which checks whether or not a point should be rendered
- Create a function to use the above function to work out the positions of all points in a given face and use a separate function to work out the colour and shade of this face
- Do the above for each face, each frame

A rigid body physics engine in three-dimensions:

- Dynamics (movement of objects)
- Collision detection
- Collision response

Projectile motion (Simulation 2):

- Specialized simulation 1

One body orbiting another (Simulation 3):

- Specialized simulation 1

A window with a selection menu on it:

- Create a window with WinForms
- Add UI elements

The ability to select an option by clicking on it which will then close the menu and launch the simulation:

- Make the previously implemented UI elements interactable
- When clicked, the code for the selected simulation should be executed

A button which, when clicked, will close the current simulation and open the menu shown above:

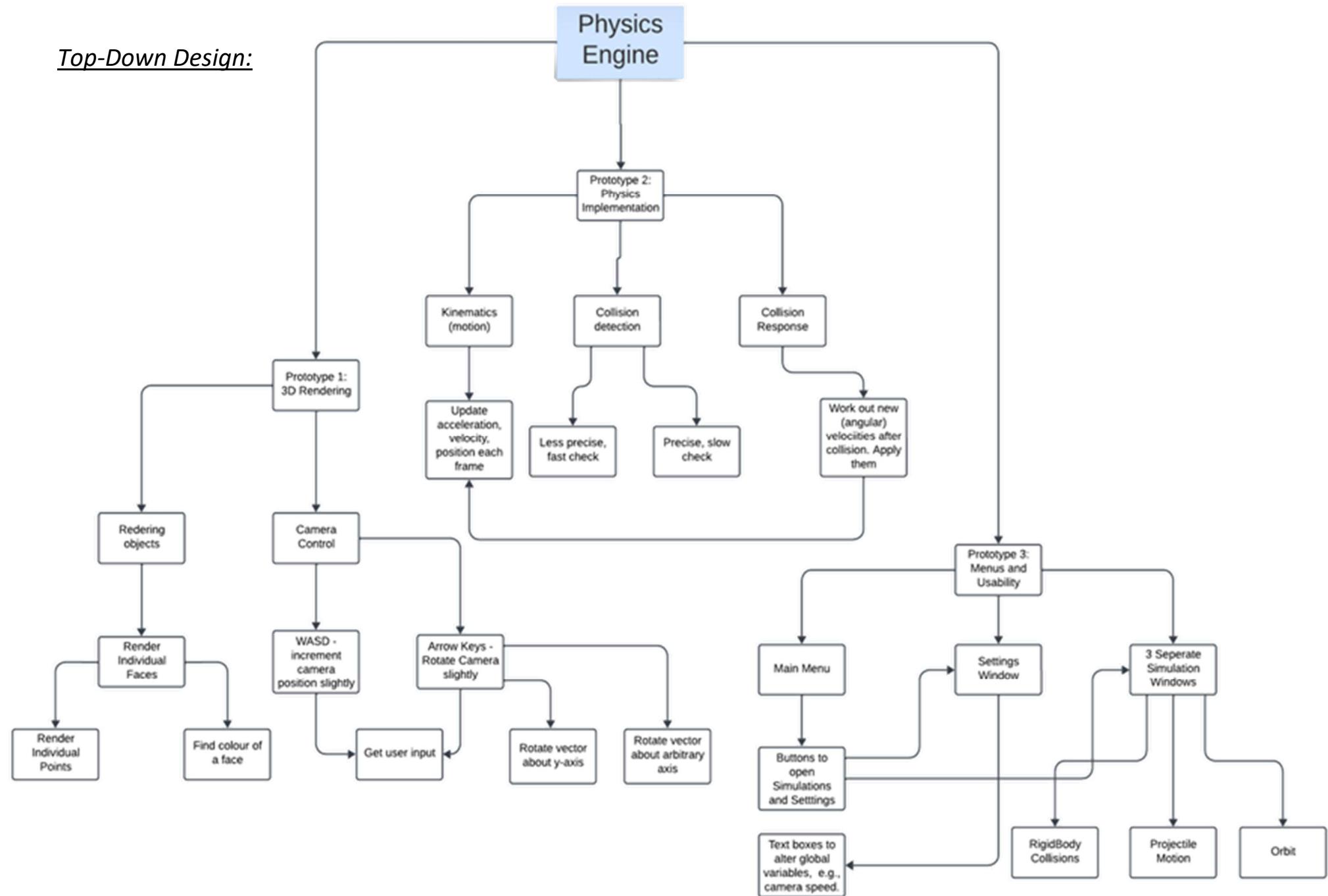
- Create a button UI element
- When clicked the code currently running should be executed and the code which runs when the program is executed should be executed.

A series of options and buttons which can be used by the user to influence the current simulation:

- Add UI elements beside where the running simulation is being shown
- Give these values which can be inputted and these will act as variables in the simulation
- Some may be buttons to execute certain functions for example creating a new object

I will now begin designing the specific algorithms required though more decomposition will likely be required for this process. I will use python for this as it is the language I find easiest to use, though the syntax may not always be correct. This is not important as I will later be rewriting these algorithms in C# during implementation. To be clear:

### Top-Down Design:



# Prototype 1 Design

Objects will be split into faces which will be split into points

For syntax purposes, I will call the class for each physical object “Entity” instead of “Object”.

Python code algorithm:

class Pos:

```
def __init__(self, x, y, z):  
    self.x = x  
    self.y = y  
    self.z = z
```

class Point:

```
def __init__(self, pos): #takes instance of Pos class as an argument  
    self.pos = pos
```

class Face:

```
def __init__(self, points): #takes an array of instances of points as an argument  
    self.points = points
```

class Mesh:

```
def __init__(self, faces): #takes an array of instances of the Face class as an  
    argument  
    self.faces = faces
```

class Entity: #An instance of the entity class will have the mesh associated with that object  
as an attribute. It will also be used later for physics. I do this to keep the rendering and  
physics separate

```
def __init__(self, mesh, ...): #I will add more to this class later  
    self.mesh = mesh
```

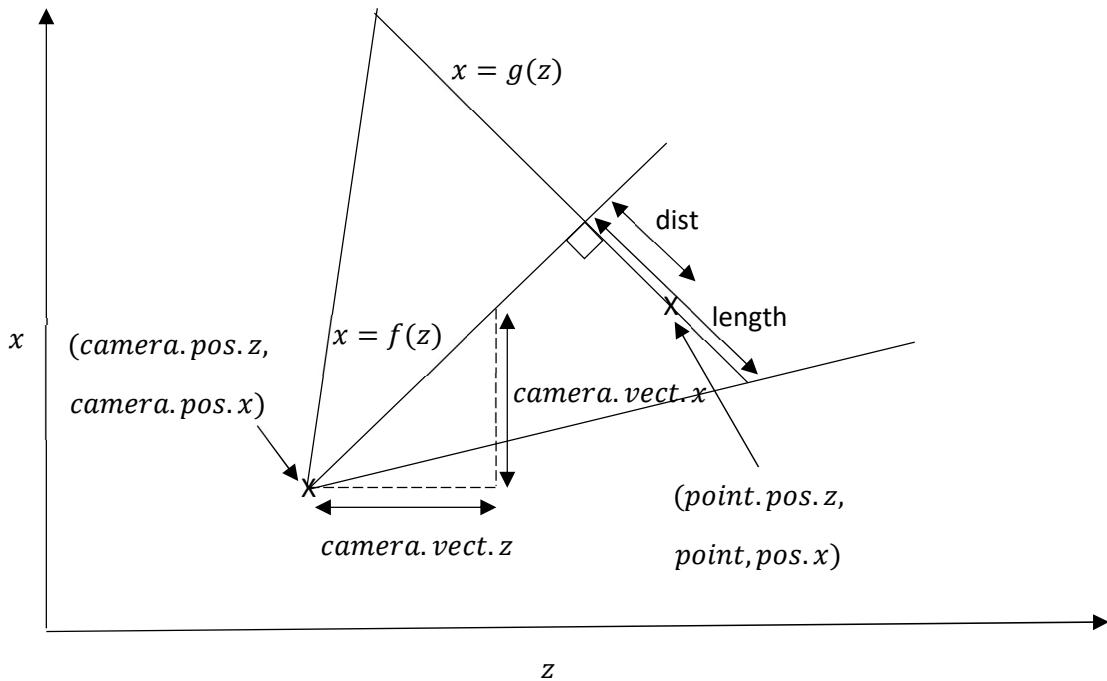
...

Create function which takes the inputs of the position and direction of the camera and the position of a point to work out where on the screen that point should be displayed

There are numerous methods and algorithms to solve this problem. I will use a method which I have not seen elsewhere but is most intuitively to me. I will use co-ordinate geometry to solve this problem.

In the diagram below, the line of sight from the camera is labelled  $x = f(z)$ . The perpendicular distance from this line to the point of interest is labelled “dist” and this line is part of the graph labelled  $x = g(z)$ . The length of this line from the line of sight to the edge of the screen is labelled “length”.  $\frac{dist}{length}$  = the fraction of the way across the screen from the centre that a point should appear. If this value can be calculated this can be used to find the  $x - coordinate$  of where a point should be displayed on the screen. This can be done using coordinate geometry.

Also camera.vect refers to the vector (direction) of the camera.



Finding the equation of the line  $x = f(z)$

in the form:  $x = mz + c$ :

$$m = \frac{\text{camera}.vect.x}{\text{camera}.vect.z}$$

$$x = \frac{\text{camera}.vect.x}{\text{camera}.vect.z} * z + c$$

This line passes through  $(\text{camera}.pos.z, \text{camera}.pos.x)$

$$\text{camera}.pos.x = \frac{\text{camera}.vect.x * \text{camera}.pos.z}{\text{camera}.vect.z} + c$$

$$c = \text{camera}.pos.x - \frac{\text{camera}.vect.x * \text{camera}.pos.z}{\text{camera}.vect.z}$$

$$x = \frac{\text{camera}.vect.x}{\text{camera}.vect.z} * z + \text{camera}.pos.x - \frac{\text{camera}.vect.x * \text{camera}.pos.z}{\text{camera}.vect.z}$$

Next find the equation of  $x = g(z)$

in the form:  $x = mz + c$

Perpendicular to  $x = f(z)$

$$m = -\frac{\text{camera}.vect.z}{\text{camera}.vect.x}$$

$$x = -\frac{\text{camera}.vect.z}{\text{camera}.vect.x} * z + c$$

This line passes through  $(point.pos.z, point.pos.x)$

$$point.pos.x = -\frac{camera.vect.z * point.pos.z}{camera.vect.x} + c$$

$$c = point.pos.x + \frac{camera.vect.z * point.pos.z}{camera.vect.x}$$

$$x = -\frac{camera.vect.z}{camera.vect.x} * z + point.pos.x + \frac{camera.vect.z * point.pos.z}{camera.vect.x}$$

Find the point of intersection:

$$\frac{camera.vect.x}{camera.vect.z} * z + camera.pos.x - \frac{camera.vect.x * camera.pos.z}{camera.vect.z}$$

$$= -\frac{camera.vect.z}{camera.vect.x} * z + point.pos.x + \frac{camera.vect.z * point.pos.z}{camera.vect.x}$$

$$\frac{camera.vect.x}{camera.vect.z} * z + \frac{camera.vect.z}{camera.vect.x} * z = point.pos.x$$

$$+ \frac{camera.vect.x * camera.pos.z}{camera.vect.z} + \frac{camera.vect.z * point.pos.z}{camera.vect.x} - camera.pos.x$$

$$z \left( \frac{camera.vect.x}{camera.vect.z} + \frac{camera.vect.z}{camera.vect.x} \right) = point.pos.x$$

$$+ \frac{camera.vect.x * camera.pos.z}{camera.vect.z} + \frac{camera.vect.z * point.pos.z}{camera.vect.x} - camera.pos.x$$

$$z =$$

$$\frac{\left( point.pos.x + \frac{camera.vect.x * camera.pos.z}{camera.vect.z} + \frac{camera.vect.z * point.pos.z}{camera.vect.x} - camera.pos.x \right)}{\left( \frac{camera.vect.x}{camera.vect.z} + \frac{camera.vect.z}{camera.vect.x} \right)}$$

Find the  $x$ -coordinate of the point of intersection:

$$x = \frac{camera.vect.x}{camera.vect.z} * z + camera.pos.x - \frac{camera.vect.x * camera.pos.z}{camera.vect.z}$$

by using the previously found value of  $z$ .

I will now refer to this set of coordinates as  $(p, q)$

“dist” can now be calculated as the distance between the points  $(p, q)$  and  $(point.pos.z, point.pos.x)$

$$dist = \sqrt{(p - point.pos.z)^2 + (q - point.pos.x)^2}$$

“length” can also be found. The distance between  $(p, q)$  and the camera

$$= \sqrt{(p - camera.pos.z)^2 + (q - camera.pos.x)^2}$$

A right angled triangle is formed of which the opposite is the length, the adjacent is the distance between  $(p, q)$  and the camera and the angle is half the horizontal field of view of the camera. I will use  $70^\circ$  as the field of view  $= \frac{7\pi}{18} rad$ . Half of this is  $35^\circ = \frac{7\pi}{36} rad$ .

$$\tan\left(\frac{7\pi}{36}\right) = \frac{\text{opposite}}{\text{adjacent}} = \frac{\text{length}}{\sqrt{(p - camera.pos.z)^2 + (q - camera.pos.x)^2}}$$

$$\text{length} = \sqrt{(p - camera.pos.z)^2 + (q - camera.pos.x)^2} * \tan\left(\frac{7\pi}{36}\right)$$

Once these two values are obtained the value,  $\frac{dist}{length}$  can be returned.

I will call the above function “calcXCoord” because it calculates the  $x$ -coordinate of the point. [Specific function names may change during development.]

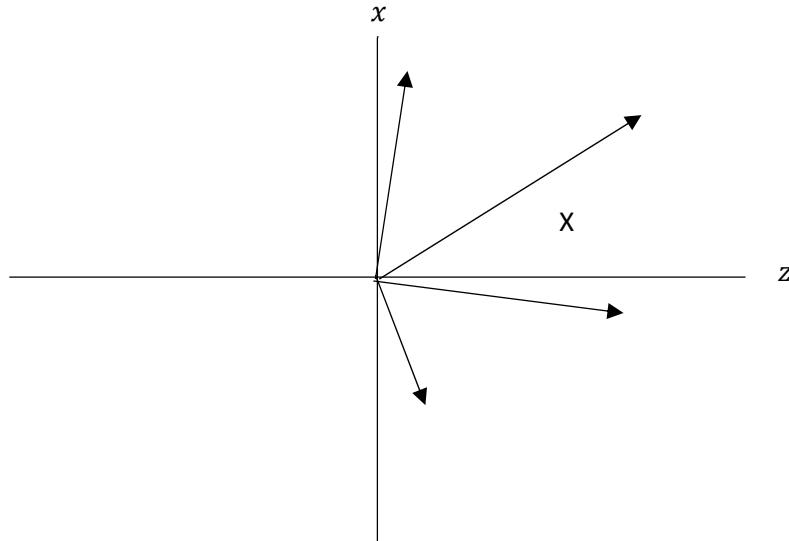
There are, however, a few problems. Firstly, as we are only calculating lengths and distances, the value returned will never be negative, which it should be if the point should be rendered to the left of the centre of the screen. This can be solved by creating an algorithm to work out if the point should be on the left or right of the centre. Also, there is no way to differentiate between vectors facing in opposite directions as they have the same gradient. This means that objects behind the camera would still be rendered when they shouldn't be. An algorithm must be created to work out whether an object will be behind the camera and so should not be rendered. A third issue is that if the camera faces due north (positive  $x$ ), south (negative  $x$ ), east (positive  $z$ ) or west (negative  $z$ ) there will be a division by zero when calculating the gradients of  $f(z)$  and  $g(z)$ . This can be solved by implementing a special case.

First, an algorithm to work out whether a point should be on the left or right side of the screen. To determine this, we need to work out if the point will be on the left or right of the line of sight. There are three cases to review:

- The  $z$ -component of the vector is positive.
- The  $z$ -component of the vector is negative.

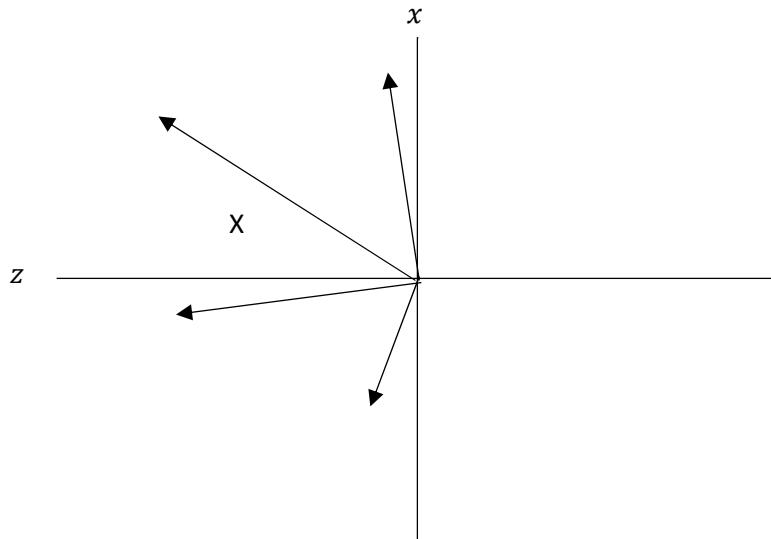
- The z-component of the vector is zero.

Below is a diagram showing examples of vectors with a positive z component.



As you can hopefully see, if the x-coordinate (vertical) of the point is less than the line then it is on the right of the line of sight, if it is greater, it is on the left. If it is on the left, then the previous algorithm should return a negative result, otherwise it should be positive. (Note that if the point lies on the line, then it will be at the centre of the screen so it being positive or negative makes no difference).

Here is a diagram showing examples of vectors with negative z components.

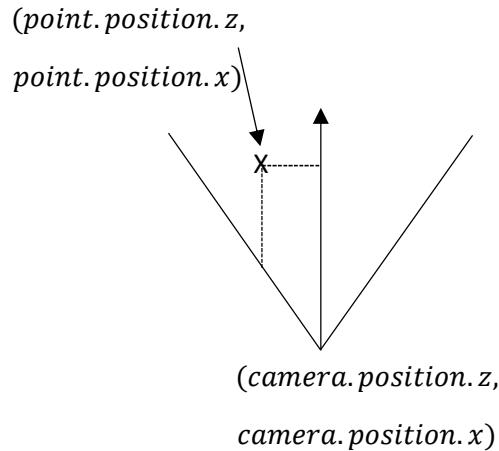


In this case, if the point has an x coordinate above the line, then the point is on the right of the screen, otherwise it is on the left.

The third scenario is where the z component of the vector is zero in which case the vector is facing due north or due east, which is the aforementioned special case, and so this can be resolved there.

Special case where the vector faces due north, east, south or west:

### Due north



In this case,

$$dist = point.position.z - camera.position.z$$

$$\tan\left(\frac{7\pi}{36}\right) = \frac{\text{opposite}}{\text{adjacent}} = \frac{\text{length}}{point.position.x - camera.position.x}$$

$$\text{length} = (point.position.x - camera.position.x) * \tan\left(\frac{7\pi}{36}\right)$$

A benefit of this special case is that it will return a negative value when necessary, which means the check will not be necessary. This will also be the case for the other very similar special cases. I will not draw diagrams for these as they would all be the same but rotated 90°.

### Due south

$$dist = camera.position.z - point.position.z$$

$$\tan\left(\frac{7\pi}{36}\right) = \frac{\text{length}}{(camera.position.x - point.position.x)}$$

$$\text{length} = (camera.position.x - point.position.x) * \tan\left(\frac{7\pi}{36}\right)$$

### Due east

$$dist = camera.position.x - point.position.x$$

$$\tan\left(\frac{7\pi}{36}\right) = \frac{\text{length}}{(\text{point.position.z} - \text{camera.position.z})}$$

$$\text{length} = (\text{point.position.z} - \text{camera.position.z}) * \tan\left(\frac{7\pi}{36}\right)$$

Due west

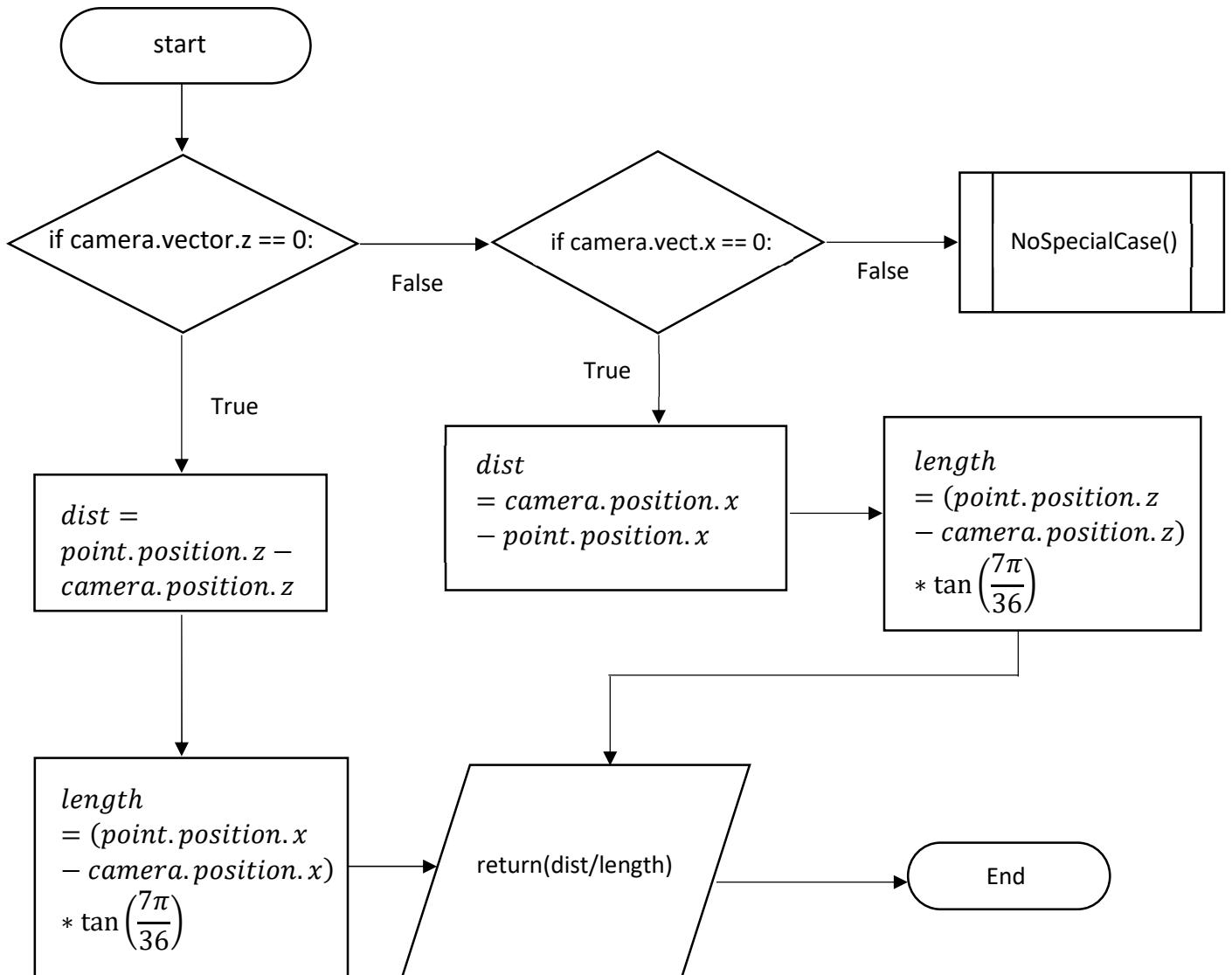
$$\text{dist} = \text{point.position.x} - \text{camera.position.x}$$

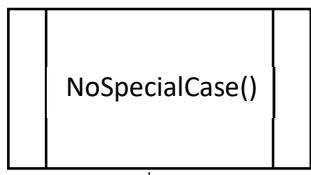
$$\tan\left(\frac{7\pi}{36}\right) = \frac{\text{length}}{\text{camera.position.z} - \text{point.position.z}}$$

$$\text{length} = (\text{camera.position.z} - \text{point.position.z}) * \tan\left(\frac{7\pi}{36}\right)$$

Note that  $\text{dist}_{\text{north}} = -\text{dist}_{\text{south}}$  and  $\text{length}_{\text{nor}} = -\text{length}_{\text{south}}$

This means that  $\frac{\text{dist}_{\text{north}}}{\text{length}_{\text{nor}}} = \frac{-\text{dist}_{\text{south}}}{-\text{length}_{\text{south}}} = \frac{\text{dist}_{\text{south}}}{\text{length}_{\text{south}}}$  which means that we can use just one of the two whenever the z component of the vector is zero. The same is true for east and west, meaning we need only use one of the two when the x component of the vector is zero. Here is the algorithm for that:





$$p = \frac{\left( point.pos.x + \frac{camera.vect.x * camera.pos.z}{camera.vect.z} + \frac{camera.vect.z * point.pos.z}{camera.vect.x} - camera.pos.x \right)}{\left( \frac{camera.vect.x}{camera.vect.z} + \frac{camera.vect.z}{camera.vect.x} \right)}$$

$$q = \frac{camera.vect.x}{camera.vect.z} * p + camera.pos.x - \frac{camera.vect.x * camera.pos.z}{camera.vect.z}$$

$$dist = \sqrt{(p - point.pos.z)^2 + (q - point.pos.x)^2}$$

$$length = \sqrt{(p - camera.pos.z)^2 + (q - camera.pos.x)^2} * \tan\left(\frac{7\pi}{36}\right)$$

return(dist/length)

End

Note that  $\tan\left(\frac{7\pi}{36}\right)$  would be computed beforehand as there would be no reason to recalculate this value each time it needs to be used.

Next, we need to work out for the non-special case whether the point is located on the left (negative) or the right (positive) of the line of sight. This can be done checking an inequality however the inequality we use depends on the direction of the line of sight (because two vectors facing opposite directions have the same linear graph.)

First work out the equation of the line.

$$x = mz + c$$

```

m = gradient = camera.vect.x/camera.vect.z

camera.pos.x = camera.vect.x*camera.pos.z/camera.vect.z + c

c = camera.pos.x - camera.vect.x*camera.pos.z/camera.vect.z

```

So the equation of the line is:

$$x = camera.vect.x*z/camera.vect.z + camera.pos.x - camera.vect.x*camera.pos.z/camera.vect.z$$

Then check whether the point is above or below it.

Return True if the point is on the right and a positive value should be used and return False if the point is on the left and a negative value should be used.

#### Python code algorithm:

```

if camera.vect.z > 0: #vector is facing right

    if point.pos.x < camera.vect.x*point.pos.z/camera.vect.z + camera.pos.x - camera.vect.x*camera.pos.z/camera.vect.z:

        return(True)

    #if True is returned the positive value will be used (the point is on the right)

else:

    return(False)

    #if False is returned the value will be made negative (the point is on the left)

if camera.vect.z < 0: #vector is facing left

    if point.pos.x < camera.vect.x*point.pos.z/camera.vect.z + camera.pos.x - camera.vect.x*camera.pos.z/camera.vect.z:

        return(False)

    else:

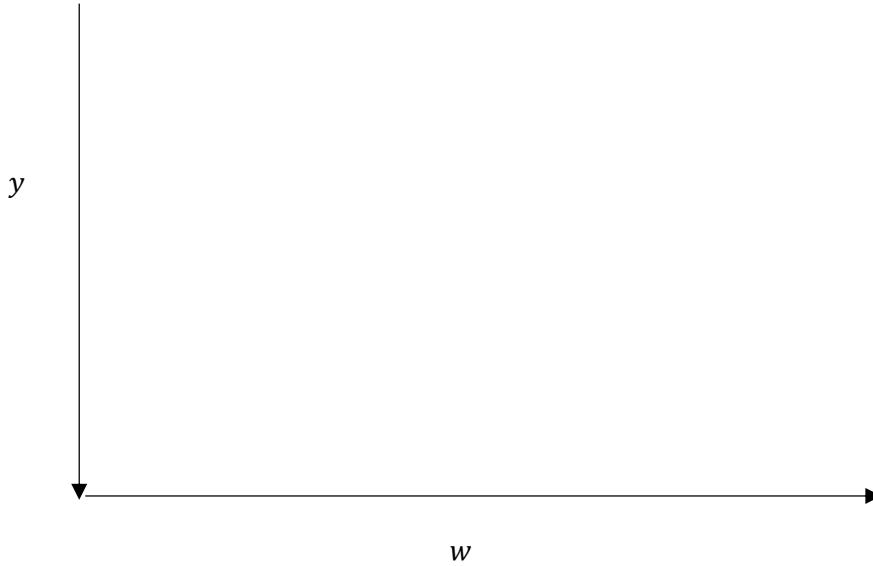
        return(True)

```

This function will be called “LeftOrRight”

So far, with the algorithms created, I will be able to input the camera as well as a point into a function which will return a number describing what fraction of the way from the centre of the screen to the edge it is. This number will be positive if the point is on the right of the screen and will be negative if the point is on the left. My current plan is to create a window

using WinForms which uses pixel coordinates (starting at (0, 0) in the top left) to identify individual pixels. This means that the number which has been returned will need to be translated into this new form. I will do this once I have created the functions to work out the vertical position of the point.



Above I have drawn a side on view, like the top-down view for finding the x-coordinate, this will help us to find the y-coordinate of the point.

If I were to draw diagrams and go through all the mathematics, I would be merely repeating myself as they would all be the same as previously with one key difference.

$x$  has been replaced by  $y$  and  $z$  has been replaced by  $w$ .

As the  $y$  value increases, the vertical position moves downwards, this is convention and also means that, more specifically,

$x$  has been replaced by  $-y$ .

also, the horizontal axis, represents the x-z plane. To find the distance from a point on this plane to the origin the Pythagorean theorem can be used. The distance =  $\sqrt{x^2 + z^2}$ . The distance on the diagram above, of a point on this axis to the origin is the  $w$ -coordinate of that point. In other words,  $w = \sqrt{x^2 + z^2}$

So, to summarise, finding the vertical position of a point on the screen can be done using the same functions as finding the horizontal position of a point on the screen, with just two substitutions.

All occurrences of:

$x$  are to be replaced by  $-y$

$z$  are to be replaced by  $\sqrt{x^2 + z^2}$

This goes for positions, as well as for vectors. This algorithm will be called “calcYCoord”.

Now I will translate the coordinates  $(x, y)$  (as returned by the functions created thus far) into coordinates which can be used by WinForms to draw to the screen.

Firstly, one unit in WinForms is one pixel meaning that if  $x = 1$  a pixel would be drawn one pixel from the left-hand side of the screen. However, in the model which we have so far been working with, one unit means half the width of the screen for the  $x - coordinate$ , and half the height of the screen for the  $y - coordinate$ . I will refer to the pixel width of the screen as screenWidth and the height as screenHeight.

I need to scale the WinForms coordinates to work with our model. One model unit = screenWidth/2 WinForms units when referring to the  $x - coordinate$  and one model unit = screenHeight/2 WinForms units when referring to the  $y - coordinate$ .

So now our coordinates have become  $\left(x * \frac{screenWidth}{2}, y * \frac{screenHeight}{2}\right)$ . These are still centred around the top left corner instead of the centre of the screen. This can be fixed by moving them to the left by half the screen width and then down by half the screen height. Remember, increasing the  $y - coordinate$  is moving down.

Our new coordinates are  $\left(x * \frac{screenWidth}{2} + \frac{screenWidth}{2}, y * \frac{screenHeight}{2} + \frac{screenHeight}{2}\right)$

This can be simplified:  $\left(\frac{screenWidth}{2}(x + 1), \frac{screenHeight}{2}(y + 1)\right)$ .

And so these are the coordinates which will be used, where  $x$  and  $y$  are the values returned by the previous functions, by WinForms to draw to the screen.

I will name this function: “denormalise” as it will convert the normalized coordinates generated by previous algorithms into non-normalised coordinates which can be used by WinForms (or most libraries which do similar things). This function should take an array as an input and should also output an array.

The next thing which must be done is to create a culling algorithm. This is an algorithm which will decide what should and should not be rendered. There are numerous reasons for culling. One such reason is for performance, though in this case because the user is expected to be looking at all objects at once anyway, the program should be able to render all objects at once without performance issues. The main reason why I will be implemented culling is to avoid a graphical error which would otherwise occur.

The positions of points are being rendered using coordinate geometry, the direction of the vector is only being considered when deciding whether the point should be to the left or to the right. This means that objects behind the camera will be rendered as though they are in front but mirrored. The most obvious way I have thought to solve this problem is to check whether a point is behind the camera and if it is, it should not be rendered.

One type of culling is known as the frustum culling algorithm, which prevents objects which are entirely outside of the view of the camera as well as objects which are too close or too far from being rendered. The problem that I have with this is that an object (if particularly large or close to the camera) may have part of itself within the frustum (and so the algorithm will decide to render it) and part of itself behind the camera, leading to the graphical error mentioned earlier, as that part of the object may render even though it should not be visible. I have decided that the best way to deal with this will be to check whether each individual point of a face is behind the camera or not. If any of them are, the face will not be rendered. If they are all in front, the face will be rendered.

The question is now asked, how to find whether a point is behind the camera. To do this I will find the equation of the plane which is perpendicular to the vector of the camera. I will then decide whether the point is behind or in front of the camera, (by using this equation as well as the knowledge of whether the camera is facing up or down). If the camera is facing straight ahead (the y-component of the vector is 0) I will implement a special case to work this out.

I will also find the equation of aforementioned plane once each frame, instead of needing to calculate it each time a new point goes through the algorithm as this would be a waste of resources.

For a vector (a, b, c)

The equation of the plane perpendicular to this vector and passing the point (q, r, s) is:

$$0 = a(x - q) + b(y - r) + c(z - s)$$

Make y the subject:

$$y = -\frac{a(x - q) + c(z - s)}{b} + r$$

Using the values for the direction and position of the camera we have been using so far:

$$y = -\frac{camera.\text{vect}.x(x - camera.\text{pos}.x) + camera.\text{vect}.z(z - camera.\text{pos}.z)}{camera.\text{vect}.y} + camera.\text{pos}.y$$

If the y-component of the vector < 0 (the camera is angled upwards) then the point is behind the camera if its y-coordinate is greater than the above y value (the point is below the plane).

If the y-component of the vector > 0 (the camera is angled downwards) then the point is behind the camera if its y-coordinate is less than the above y value (the point is above the plane).

If the y-component of the vector = 0 then we can look at the situation from a top-down view (the same top-down view which was used earlier to find the perpendicular distance from the point to the line of sight) and view the plane as a two-dimensional line instead.

From this top-down view:

The vector of the camera is (camera.vect.z, camera.vect.x). Its coordinates are (camera.pos.z, camera.pos.x)

The perpendicular line passing through this point has the equation:

$$\begin{aligned}
 x &= mz + c \\
 m &= -\frac{\text{camera. vect. } z}{\text{camera. vect. } x} \\
 \text{camera. pos. } x &= -\frac{\text{camera. vect. } z}{\text{camera. vect. } x} * \text{camera. pos. } z + c \\
 c &= \text{camera. pos. } x + \frac{\text{camera. vect. } z}{\text{camera. vect. } x} * \text{camera. pos. } z \\
 x &= -\frac{\text{camera. vect. } z}{\text{camera. vect. } x} * z + \text{camera. pos. } x + \frac{\text{camera. vect. } z}{\text{camera. vect. } x} * \text{camera. pos. } z \\
 x &= \left( \frac{\text{camera. vect. } z}{\text{camera. vect. } x} \right) (\text{camera. pos. } z - z) + \text{camera. pos. } x
 \end{aligned}$$

This is the equation of the plane as seen from above. The function which was used to determine whether a point is to the left or right of the line of sight can be used again here but using the above equation instead.

This function will be called “checkRender” as it will check whether or not an individual point should be rendered or not.

Next I will create an algorithm which will take a single face as an input. It will then cycle through each vertex (point) of that face. Each of these will be sent through the checkRender function to check if they should be rendered or not.

I will then create another algorithm to send each point through the calcXCoord and calcYCoord functions to obtain their normalised screen coordinates. These will then be sent through the denormalise function to obtain denormalised values of these coordinates. These coordinates can then be used to draw a polygon connecting them (using inbuilt WinForms [or other library] features). The colour of this face will be determined via a later algorithm.

I will then create a third algorithm, taking a mesh as an input, which will check each face and then render them if all of them pass the check.

Python code algorithms:

```
def checkFace(face):
    for x in range(len(face.points)):
        if checkRender(face.points[x]) == False: #Checks if each point should be
            rendered
                return(False)
        else:
            pass
def renderFace(face):
    coords = [] #Array is created for the list of 2D coordinates to be stored
    for x in range(len(face.points)): #runs through each vertex of the face
        coordinates = [calcXCoord(camera, face.points[x]), calcYCoord(camera,
        face.points[x])] #Calculates the x and y coordinates of the point and stores
            them in a list
        coords.append(coordinates) #This list is added to the list of coordinates of all
            the points in this face
    #Denormalise coordinates
    for x in range(len(coords)): #Runs through each pair of coordinates
        coords[x] = [denormalise(coords[x])] #Replaces the coordinates currently in
            that place in the array with the new denormalised version
    #Here the polygon of the vertices stored in cords would be rendered using
    winForms.

def renderMesh(mesh):
    for x in range(len(mesh.faces)):
        if checkFace(mesh.faces[x]) == False: #Checks if each face should be rendered
            return(False) #False is returned meaning the function is terminated
        else:
            pass
    for x in range(len(mesh.faces)):
        renderFace(mesh.faces[x]) #If the function got to this point, that means that
```

False was never returned meaning all the faces passed the render check  
meaning the mesh should be rendered as it is here one face at a time.

One of the success criteria has now been designed. “3D Rasteriser”. I still need to add shading to faces so that different objects can be differentiated between. Doing so will complete another success criteria, “Shading”. I will then design a few algorithms to allow the user to control the camera, completing the success criteria, “Camera Control”. This will conclude the visual aspect of design. I will then move onto development as I create my first prototype. I will then design the physics side of the software and will create the second prototype. Finally, I will design the last few aspects of the project, e.g. GUI and a menu screen, which will then be developed into the final product.

To add shading to each side, my plan is to define a vector to describe the direction light is coming travelling in. Then calculate the vector which is perpendicular to a given face (coming out of it). Then compare the two vectors by finding the angle between them. This angle (in radians) will then be divided by  $\pi$ . If the vectors were the same, the face should not be brightly lit. The angle between the vectors would be  $0 \text{ radians}$  and  $\frac{0}{\pi} = 0$  so the brightness level of this face would be 0. If the faces vector was in the opposite direction of the light vector, then the face should be brightly lit. The angle between them would be  $\pi \text{ radians}$  ( $180^\circ$ ) and  $\frac{\pi}{\pi} = 1$  so the brightness level of this face would be 1. If the vectors were perpendicular the face should be about half way between light and dark. The angle between the vectors would be  $\frac{\pi}{2} \text{ radians}$  ( $90^\circ$ ) and  $\frac{\pi}{2} \div \pi = \frac{1}{2}$  so it would have a brightness level of  $\frac{1}{2}$ .

It may at this point also be worth noting, the reason why I am using radians throughout instead of degrees is because the *sin*, *cos* and *tan* of angles measured in radians can be calculated directly. To calculate the *sin*, *cos* or *tan* of an angle measured in degrees, the angle must first be converted to radians. Using radians saves this extra step, giving the software a small optimisation for free. Even if it is insignificant in the end, there is no reason not to.

\*Calculating the vector perpendicular to a face can be relatively easily done using the cross-product of two vectors which lie on the plane. The challenge comes in working out which direction this vector faces as it is vital that the vector always face out of the plane. I have been unable to come up with a solution to this problem and so have come up instead with a compromise. I will use only polyhedral such that the normal of each face (intersecting the centre of that face) intersects the centre of the polyhedron. A cube (or any cuboid) fits this definition as a vector going from the centre of the cuboid to the centre of a given face will be perpendicular to that vector and will be pointing out of it. Tetrahedra, for example, would also fit this definition. This way, the normal vector can be trivially calculated by finding the centre of a face and then the vector going from the centre of the object to that point.

The centre of the object itself *could* be easily calculated (in basic cases e.g., squares and tetrahedra, the centre is the average of all vertices) however the centre will later need to be directly changed as it will be the centre of mass which will be altered during the physics development stage later on. This attribute, centre, will need to be written to by the “RigidBody” attribute and will need to be read by the “Mesh” attribute. For this reason I will make it its own attribute, separated from the other two. An Instance of the “Entity” class will now have three attributes: “Mesh” which will be responsible for rendering the entity, “RigidBody” which will be responsible for carrying out physics on the object. “CentreOfMass” which will be the coordinates of the objects centre.

The “RigidBody” attribute will write to and read from the “CentreOfMass” e.g., to move the object or see where it is located. The “Mesh” attribute will only read from this attribute as it will never need to change it. There may later be similar attributes involving rotation which will be written to by RigidBody and read by Mesh. As the centre and rotation of the Entity changes, the positions of its points will also change.

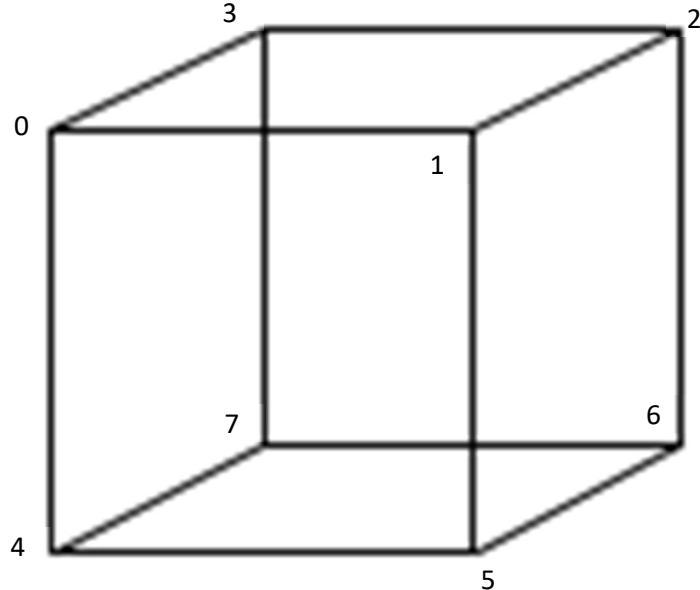
I have had the thought of creating a few more attributes e.g., shape (e.g., cube, cuboid, tetrahedron etc.) & dimensions (to convey its size). With the centre of mass, shape, dimensions and any rotational information, all vertex coordinates could be worked out and sent to the “Mesh” attribute to be rendered. The attributes for the centre and rotational information could be easily changed by the physics engine where necessary (hopefully). This is just a thought, but this could prove useful.

If I am to split up shapes in such a way anyway, it would be wholly unnecessary to calculate the normal to a face in such a way as stated earlier. It would make far more sense to simply take the vector of one the edges connected to it as this (for a cuboid) is perpendicular to the face, which means (unless I later for whatever reason decide it to be essential that the software has a variety of different shapes) the perpendiculars could be easily calculated this way instead, which wastes far less time and effort. This realization also means that the Mesh may never need to interact with the CentreOfMass attribute, and so we could store this, along with the shape, dimensions, rotation, perhaps even colour, inside of a separate attribute which I will call “Characteristics”. \*

The previous section of text (enclosed in asterisks\*) follows my thought process in real time which is why it may seem a bit convoluted. To summarise, “Entity” will now comprise of the attributes: “Mesh”, “RigidBody” and “Characteristics”, the latter of which will contain attributes: “Shape”, “Dimensions”, “Colour”, “Rotation” & “CentreOfMass”. [These may be subject to change]. These characteristics are all the information needed for a separate function to work out the coordinates of each vertex and store them in the Mesh attribute. Calculating the normal to a face can be done by finding the vector going from one vertex to another (more detail on this will be given later). From there, rendering can be done as previously devised.

To create an algorithm using the decided upon method, the vertexes in a mesh will need to be stored in a specific way. This has got me thinking that it may be a better idea to store all

vertices in the Mesh object as opposed to storing them in faces which are then stored in the mesh. If the shape of the object is predetermined (e.g. a cube) there is no reason to store each mesh as a set of faces. The vertices will need to be stored in a specific order so the array of numbers can be easily interpreted. The predetermined order doesn't matter too much as long as it is consistent. Below is a diagram showing one way of storing the points.



This diagram shows the order (starting at zero) in which vertices of a cube (or any cuboid) may be stored in an array. Rotation does not matter as the only reason for ordering them like this is to know how they are arranged *relative to each other*. (Note from future: I ended up changing this order later on.)

When rendering a “Mesh”, instead of cycling through each “Face”, and then using each point from that face to render it, use the following method.

To render the top face (though it may not always be on the top depending on how the cuboid is rotated), using `face.points[0], face.points[1], face.points[2], face.points[3]`, use, `mesh.points[0], mesh.points[1], mesh.points[2], mesh.points[0]`. For the left face use: `mesh.points[0], mesh.points[2], mesh.points[4] & mesh.points[6]`, and so on for the other faces. This is why is it useful to use predetermined shapes such as cubes.

Now to solve the problem which this whole new system was implemented to solve; finding the vector perpendicular to a given face.

For the top face:  $\text{mesh.points}[0] - \text{mesh.points}[4]$

Left:  $\text{mesh.points}[0] - \text{mesh.points}[1]$

Right:  $\text{mesh.points}[1] - \text{mesh.points}[0]$

Back:  $\text{mesh.points}[3] - \text{mesh.points}[0]$

Front:  $\text{mesh.points}[0] - \text{mesh.points}[3]$

Bottom:  $mesh.points[4] - mesh.points[0]$

$mesh.points[x]$  represents a 3d vector which is stored as an array of 3 values. To subtract one from the other is simple:  $\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \end{pmatrix}$ , depending on the feature set of the language IDE used (likely Visual Studio) this vector subtraction may need to be implemented by me or may already be present.

Now that I have a way to find the vector perpendicular to a face I will move on to finding the angle between this vector and the predetermined (by me) vector of the direction of "light".

To do this I will use the dot product of the two vectors. The dot product of  $\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$  &  $\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}$  is the sum of the products of the components of the vector. So

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = (x_1x_2) + (y_1y_2) + (z_1z_2)$$

This will be a scalar quantity (as opposed to a vector). Divide this by the magnitude of the first vector, then by the magnitude of the second. (Due to the way in which we calculated the normal vector earlier, it will have a magnitude equal to the side length it was obtained from. For a unit cube this would always be 1, but other dimensions may also be accounted for later.) The second vector will always be constant so its magnitude can be calculated beforehand. The final step is to take the inverse cosine of this value to obtain the angle in radians.

Python code algorithm:

```
def getBrightnessLevel(vect1, vect1Mag): #vect1Mag is the magnitude of vect1 and  
                                         as this would be more efficient than  
                                         calculating it from the vector.  
  
    angle = vect1[0]*lightVect[0] + vect1[1]*lightVect[1] + vect1[2]*lightVect[2]  
  
    angle = angle/(vect1Mag*lightMag) #lightMag would be a precalculated constant.  
  
    angle = arccos(angle) #using an inverse cosine function from some library.  
  
    return(angle/pi) #The angle is in radians and is divided by π as discussed earlier.
```

When a given face is rendered, this algorithm will be used to determine its colour as an RGB value. Each colour (red, green and blue) can have any integer value between 0 and 255. 255 multiplied by the brightness level calculated would be a number between 0 and 255. This means that some faces may be completely dark (value of 0) or completely bright (value of 255). To remove these extremes instead you could, for example, multiply the brightness value by 205 and add 25 to give a value between 25 and 230.

I would like to now move back to culling as this new system allows us to do something new. When looking at a cuboid you will notice that you can never see more than 3 faces at once. This means that half of the faces being rendered will not even be seen. If we first find which vertex of a mesh is furthest from the camera we will be able to decide which faces should and should not be rendered. e.g., if `mesh.points[6]` is furthest from the camera, the “left”, “back” and “bottom” faces do not need to be rendered. I will add this to the rendering algorithms during the development stage.

Python code algorithm:

```
def closestPoint(mesh):
    for i in range(len(mesh)):
        newDist = ((mesh.points[i].x)**2 + (mesh.points[i].y)**2 +
                   (mesh.points[i].z)**2)**(1/2) #Finds the distance of each vertex of the mesh.

        Also using the new structure of the mesh class
        where the mesh is comprised directly of points
        instead of faces.

        if i == 0:
            index = 0
            dist = newDist #if this is the first cycle through the loop the index is
                           set to 0 meaning the "0th" vertex in the list is the
                           closest to the camera. The distance to this point is also
                           recorded as dist.

        elif newDist < dist:
            index = i
            dist = newDist #Each subsequent cycle through the loop, the new
                           distance is checked against the old shortest distance, if
                           the new one is greater than or equal to, move on, if the
                           new distance is less than the old one, then the new
                           vertex is closer than the old one. Its index is recorded
                           as well as its distance from the camera. After all points
                           have been cycled through, "dist" is no longer relevant
                           to us as we only needed to know which point was the
```

closest. The index is relevant however, and so it will be returned.

mesh.closest = index #I will add the attribute “closest” to the Mesh class, as this will also be used by a later algorithm. It will need to be constantly checked and updated.

mesh.closestDist = dist #The same can be said for this attribute.

```
def renderMesh(mesh): #Updated renderMesh procedure.  
    if mesh.closest == 0:  
        normal = [mesh.points[0][0] - mesh.points[4][0], mesh.points[0][1] -  
                  mesh.points[4][1], mesh.points[0][2] - mesh.points[4][2]]  
        #normal to the face is calculated.  
        renderFace(mesh.points[0], mesh.points[1], mesh.points[2], mesh.points[3],  
                  normal)  
        #The renderFace procedure will need to later be slightly altered as it was  
        #designed to take a face as an input, not 4 individual points and the normal.  
        normal = [mesh.points[0][0] - mesh.points[2][0], mesh.points[0][1] -  
                  mesh.points[2][1], mesh.points[0][2] - mesh.points[2][2]]  
        renderFace(mesh.points[0], mesh.points[1], mesh.points[4], mesh.points[5],  
                  normal)  
        normal = [mesh.points[0][0] - mesh.points[1][0], mesh.points[0][1] -  
                  mesh.points[1][1], mesh.points[0][2] - mesh.points[1][2]]  
        renderFace(mesh.points[0], mesh.points[2], mesh.points[4], mesh.points[6],  
                  normal)  
  
    elif mesh.closest == 1:  
        normal = [mesh.points[0][0] - mesh.points[4][0], mesh.points[0][1] -  
                  mesh.points[4][1], mesh.points[0][2] - mesh.points[4][2]]  
        renderFace(mesh.points[0], mesh.points[1], mesh.points[2], mesh.points[3],  
                  normal)  
        normal = [mesh.points[0][0] - mesh.points[2][0], mesh.points[0][1] -  
                  mesh.points[2][1], mesh.points[0][2] - mesh.points[2][2]]  
        renderFace(mesh.points[0], mesh.points[1], mesh.points[4], mesh.points[5],  
                  normal)
```

```

normal = [mesh.points[1][0] - mesh.points[0][0], mesh.points[1][1] -
mesh.points[0][1], mesh.points[1][2] - mesh.points[0][2]]

renderFace(mesh.points[1], mesh.points[3], mesh.points[5], mesh.points[7],
normal)

elif mesh.closest == 2:

    normal = [mesh.points[0][0] - mesh.points[4][0], mesh.points[0][1] -
mesh.points[4][1], mesh.points[0][2] - mesh.points[4][2]]

    renderFace(mesh.points[0], mesh.points[1], mesh.points[2], mesh.points[3],
normal)

    normal = [mesh.points[2][0] - mesh.points[0][0], mesh.points[2][1] -
mesh.points[0][1], mesh.points[2][2] - mesh.points[0][2]]

    renderFace(mesh.points[2], mesh.points[3], mesh.points[6], mesh.points[7],
normal)

    normal = [mesh.points[0][0] - mesh.points[1][0], mesh.points[0][1] -
mesh.points[1][1], mesh.points[0][2] - mesh.points[1][2]]

    renderFace(mesh.points[0], mesh.points[2], mesh.points[4], mesh.points[6],
normal)

elif mesh.closest == 3:

    normal = [mesh.points[0][0] - mesh.points[4][0], mesh.points[0][1] -
mesh.points[4][1], mesh.points[0][2] - mesh.points[4][2]]

    renderFace(mesh.points[0], mesh.points[1], mesh.points[2], mesh.points[3],
normal)

    normal = [mesh.points[2][0] - mesh.points[0][0], mesh.points[2][1] -
mesh.points[0][1], mesh.points[2][2] - mesh.points[0][2]]

    renderFace(mesh.points[2], mesh.points[3], mesh.points[6], mesh.points[7],
normal)

    normal = [mesh.points[1][0] - mesh.points[0][0], mesh.points[1][1] -
mesh.points[0][1], mesh.points[1][2] - mesh.points[0][2]]

    renderFace(mesh.points[1], mesh.points[3], mesh.points[5], mesh.points[7],
normal)

elif mesh.closest == 4:

    normal = [mesh.points[4][0] - mesh.points[0][0], mesh.points[4][1] -
mesh.points[0][1], mesh.points[4][2] - mesh.points[0][2]]

    renderFace(mesh.points[4], mesh.points[5], mesh.points[6], mesh.points[7],
normal)

```

```

normal = [mesh.points[0][0] - mesh.points[2][0], mesh.points[0][1] -
mesh.points[2][1], mesh.points[0][2] - mesh.points[2][2]]

renderFace(mesh.points[0], mesh.points[1], mesh.points[4], mesh.points[5],
normal)

normal = [mesh.points[0][0] - mesh.points[1][0], mesh.points[0][1] -
mesh.points[1][1], mesh.points[0][2] - mesh.points[1][2]]

renderFace(mesh.points[0], mesh.points[2], mesh.points[4], mesh.points[6],
normal)

elif mesh.closest == 5:

    normal = [mesh.points[4][0] - mesh.points[0][0], mesh.points[4][1] -
mesh.points[0][1], mesh.points[4][2] - mesh.points[0][2]]

    renderFace(mesh.points[4], mesh.points[5], mesh.points[6], mesh.points[7],
normal)

    normal = [mesh.points[0][0] - mesh.points[2][0], mesh.points[0][1] -
mesh.points[2][1], mesh.points[0][2] - mesh.points[2][2]]

    renderFace(mesh.points[0], mesh.points[1], mesh.points[4], mesh.points[5],
normal)

    normal = [mesh.points[1][0] - mesh.points[0][0], mesh.points[1][1] -
mesh.points[0][1], mesh.points[1][2] - mesh.points[0][2]]

    renderFace(mesh.points[1], mesh.points[3], mesh.points[5], mesh.points[7],
normal)

elif mesh.closest == 6:

    normal = [mesh.points[4][0] - mesh.points[0][0], mesh.points[4][1] -
mesh.points[0][1], mesh.points[4][2] - mesh.points[0][2]]

    renderFace(mesh.points[4], mesh.points[5], mesh.points[6], mesh.points[7]
normal)

    normal = [mesh.points[2][0] - mesh.points[0][0], mesh.points[2][1] -
mesh.points[0][1], mesh.points[2][2] - mesh.points[0][2]]

    renderFace(mesh.points[2], mesh.points[3], mesh.points[6], mesh.points[7],
normal)

    normal = [mesh.points[0][0] - mesh.points[1][0], mesh.points[0][1] -
mesh.points[1][1], mesh.points[0][2] - mesh.points[1][2]]

    renderFace(mesh.points[0], mesh.points[2], mesh.points[4], mesh.points[6],
normal)

elif mesh.closest == 7:

```

```

normal = [mesh.points[4][0] - mesh.points[0][0], mesh.points[4][1] -
mesh.points[0][1], mesh.points[4][2] - mesh.points[0][2]]

renderFace(mesh.points[4], mesh.points[5], mesh.points[6], mesh.points[7],
normal)

normal = [mesh.points[2][0] - mesh.points[0][0], mesh.points[2][1] -
mesh.points[0][1], mesh.points[2][2] - mesh.points[0][2]]

renderFace(mesh.points[2], mesh.points[3], mesh.points[6], mesh.points[7],
normal)

normal = [mesh.points[1][0] - mesh.points[0][0], mesh.points[1][1] -
mesh.points[0][1], mesh.points[1][2] - mesh.points[0][2]]

renderFace(mesh.points[1], mesh.points[3], mesh.points[5], mesh.points[7],
normal)

```

This algorithm will render only the faces of a cuboid which will be visible.

The renderFace algorithm must now be changed as it now needs to take inputs like this.

#### Python code algorithm:

```

def renderFace(point1, point2, point3, point4, normal):

    coord1 = [calcXCoord(camera, point1), calcYCoord(camera, point1)]
    coord2 = [calcXCoord(camera, point2), calcYCoord(camera, point2)]
    coord3 = [calcXCoord(camera, point3), calcYCoord(camera, point3)]
    coord4 = [calcXCoord(camera, point4), calcYCoord(camera, point4)]
    coords = [coord1, coord2, coord3, coord4]

    for x in range(4):
        coords[x] = [denormalise(coords[x])]

    normalMag = ((normal[0])**2 + (normal[1])**2 + (normal[2])**2)**(1/2)
    Brightness = getBrightnessLevel(normal, normalMag)

    #The quadrilateral with vertices at "coords" and brightness of "Brightness" (colour
    #will be predetermined, perhaps a user setting) will be rendered using winForms.

```

One last thing must be done in terms of rendering. Working out the order in which objects should be rendered. This is important because if one object is behind another, it should be rendered first so that the object in front is rendered on top of it. To do this I will simply take the closest vertex of each mesh (of those which were not culled) to the camera (I have already created an algorithm for this, it may be a good idea to store this vertex as an attribute of the mesh so it doesn't need to be worked out again for this next algorithm.) and list the meshes in order from largest to smallest and rendered in that order. This algorithm may have certain problems due to its simplicity, but I will test that during development to see if there are actually any problems and if and how they could be addressed.

I will also need an array containing all meshes called "meshes" so that I can order them. Whenever a mesh object is created it will need to be added to this array.

Python code algorithm:

```
def getClosestDist(mesh):
    return(mesh.closestDist)

def renderAll():
    meshes.sort(key=getClosestDist) #Sort meshes with respect to the distance of their
                                    closest vertex to the camera.
    for x in range(len(meshes)):
        renderMesh(meshes[x]) #Render the meshes in this order.
```

I will next need to implement camera control, starting with rotation. The camera vector should be able to rotate freely about the  $y$ -axis but should only be able to go to a certain limit (directly up and directly down) when rotating up and down. My initial plan was to rotate the vector around the correct axis, but this wouldn't work as moving the vector up and down does not rotate it about a specific axis, but rather on a plane. To solve this problem I will rotate the vector about the  $y$ -axis for rotation left and right (*yaw*) and I will need to find the vector perpendicular to that of the camera whilst ignoring both vectors'  $y$ -components (I will need to do this later for camera movement anyway) and I will then need to rotate the camera vector about this new vector to move the camera up and down (*pitch*).

Changing the vector's yaw is relatively straightforward as it requires a rotation about a constant axis, being the  $y$ -axis (this is only because there is no roll).

The matrix for rotating a point (or vector) an angle of  $\theta$  counter-clockwise about the  $y$ -axis when facing in the positive  $y$  direction is  $\begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}$ . (This is assuming that the positive  $y$  direction is up given that the  $x$  and  $z$  axis are orientated as I have done in

previous diagrams. In those diagrams the positive  $y$  axis is downwards meaning I will need to account for this later.)

$$\begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \cos \theta + 0y + z \sin \theta \\ 0x + 1y + 0z \\ -x \sin \theta + 0y + z \cos \theta \end{pmatrix} = \begin{pmatrix} x \cos \theta + z \sin \theta \\ y \\ -x \sin \theta + z \cos \theta \end{pmatrix}$$

So  $\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \cos \theta + z \sin \theta \\ y \\ -x \sin \theta + z \cos \theta \end{pmatrix}$

The  $y$ -coordinate stays the same, as it should, but remember this assumes that the positive  $y$  axis is the other direction. To solve this, I could make  $y$  negative (reflecting the vector in the plane  $y = 0$ ) then rotate it, then change it back. Doing this means that  $y$  would still remain constant after the transformation, but this means that the positive  $y$  direction was the other direction when the vector was rotated meaning the vector is now being rotated counter-clockwise when you are facing the *negative*  $y$  direction (up) meaning **this matrix transformation rotates the vector clockwise when facing the positive  $y$  direction (i.e., down)**.

Python code algorithm:

#Right yaw rotation of “angle” radians.

```
def rotateYaw(angle):
```

```
    camera.vect.x = camera.vect.x*cos(angle) + camera.vect.z*sin(angle)
```

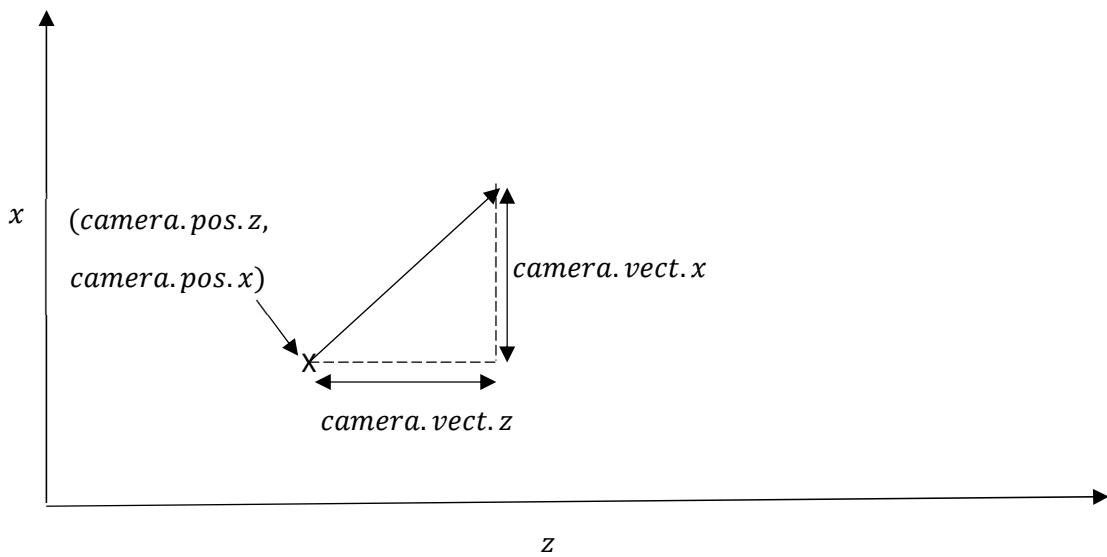
```
    camera.vect.z = -camera.vect.x*sin(angle) + camera.vect.z*cos(angle)
```

This algorithm would require the use of a library with trig functions in order to work, however the C# on visual studio has them built in.

Next, I will need to create an algorithm to find the perpendicular vector and then another algorithm to rotate the main vector about this new one.

The perpendicular vector must be a unit vector as Rodrigues’ Rotation Formula requires this. (This will be explained later).

This vector should lie on the  $x$ - $z$  plane. As such, the  $y$  component of the camera vector should be ignored when finding it.



Above is the top-down view of the camera vector. As this is ignoring the  $y$  component, the length of this vector is unknown. The gradient of the line which this vector lies on has a gradient of  $\frac{camera.vect.x}{camera.vect.z}$  so the perpendicular has a gradient of  $-\frac{camera.vect.z}{camera.vect.x}$ . This means that the vector  $(\frac{-camera.vect.x}{camera.vect.z})$  is perpendicular to our original vector. So is the vector  $(\frac{camera.vect.x}{-camera.vect.z})$ , which points in the opposite direction. I need the vector which is perpendicular and pointing right of the original vector, which in this case is  $(\frac{camera.vect.x}{-camera.vect.z})$ . To make this into a unit vector I need to divide it by its magnitude which  $= \sqrt{(camera.vect.x)^2 + (camera.vect.z)^2}$  which also happens to be the magnitude of the original vector.

#### Python code algorithm:

```
def getCamHorPerpUnitVect(): #Gets the camera's horizontal perpendicular unit vector
    vectMag = ((camera.vect.x)**2+(camera.vect.z)**2)**(1/2) #Calculates vector
                                                               magnitude.

    zComp = (camera.vect.x)/vectMag #Calculates z component of new vector.

    xComp = (-camera.vect.z)/vectMag #Calculates x component of new vector.

    return([xComp, 0, zComp]) #Returns the perpendicular vector as a 3-dimensional
                             vector with y component 0.
```

The main vector can now be rotated about the new perpendicular vector by using the Rodrigues' Rotation Formula:

$$v_{rot} = v \cos \theta + (k \times v) \sin \theta + k(k \cdot v)(1 - \cos \theta)$$

where  $v_{rot}$  is the rotated vector,  $v$  is the original vector,  $k$  is the unit vector about which we rotate the main vector (in this case this is the perpendicular vector),  $\theta$  is the angle (clockwise when facing in the direction of the vector) by which the vector is rotated,  $k \times v$  is

the cross product of  $k$  and  $v$  (I will explain what this is in a moment) and  $k \cdot v$  is the dot product of  $k$  and  $v$  (again I will explain this).

The dot product of two vectors,  $\begin{pmatrix} a \\ b \\ c \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = ax + by + cz$  and is a scalar (meaning a number and not a vector).

The cross product of two vectors,  $\begin{pmatrix} a \\ b \\ c \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix}$  is another 3-dimensional vector. To find it, you work out the determinant of the following matrix:

$$\begin{pmatrix} i & j & k \\ a & b & c \\ x & y & z \end{pmatrix}$$

The determinant of this matrix is denoted by  $\begin{vmatrix} i & j & k \\ a & b & c \\ x & y & z \end{vmatrix}$

$$\begin{vmatrix} i & j & k \\ a & b & c \\ x & y & z \end{vmatrix}$$

$$= i \begin{vmatrix} b & c \\ y & z \end{vmatrix} - j \begin{vmatrix} a & c \\ x & z \end{vmatrix} + k \begin{vmatrix} a & b \\ x & y \end{vmatrix}$$

$$= i(bz - cy) - j(az - cx) + k(ay - bx)$$

$$= i(bz - cy) + j(cx - az) + k(ay - bx)$$

$$= \begin{pmatrix} bz - cy \\ cx - az \\ ay - bx \end{pmatrix}$$

I now need to create two algorithms to find the dot and cross product of two 3d vectors. I will then create an algorithm to find  $v_{rot}$ , the rotated vector, using Rodrigues' Rotation Formula.

Python code algorithm:

```
def dotProd(vect1, vect2):
```

```
    return(vect1[0]*vect2[0]+vect1[1]*vect2[1]+vect1[2]*vect2[2])
```

Python code algorithm:

```
def crossProd(vect1, vect2):
```

```
    xComp = (vect1[1]*vect2[2])-(vect1[2]*vect2[1]) #x-component
```

```
    yComp = (vect1[2]*vect2[0])-(vect1[0]*vect2[2]) #y-component
```

```

zComp = (vect1[0]*vect2[1])-(vect1[1]*vect2[0]) #z-component
return([xComp, yComp, zComp]) #The cross product vector is returned as an array.

```

Rodrigues' rotation formula, like the matrix for rotating a vector about the  $y$ -axis, assumes that the  $y$ -axis faces the other direction. This means that this formula (which is supposed to rotate  $v$  clockwise about  $k$ ), will instead rotate  $v$  counter-clockwise about  $k$ . Because  $k$  is facing to the right relative to  $v$ , this means that  $v$  will be rotating downwards.

Python code algorithm:

```

#Downward pitch rotation of "angle" radians.

def rotatePitch(angle):

    vectK = getCamHorPerpUnitVect():

    vectV = [camera.vect.x, camnera.vect.y, camera.vect.z] #Store k and v as vectors
                                                                making the formula easier to
                                                                follow.

    Cosine = cos(angle)

    Sine = sin(angle) #So sine and cosine don't need to be recalculated each time.

    term1 = [vectV[0]*Cosine, vectV[1]*Cosine, vectV[2]*Cosine] #Each term is
                                                                calculated one by one.

    crossProduct = crossProd(vectK, vectV) #Cross product is another vector.

    term2 = [crossProduct[0]*Sine, crossProduct[1]*Sine, crossProduct[2]*Sine]

    dotProduct = dotProd(vectK, vectV) #Dot product is a scalar (not a vector).

    term3 = [vectK[0]*dotProduct*(1-Cosine), vectK[1]*dotProduct*(1-Cosine),
            vectK[2]*dotProduct*(1-Cosine)] #Each term is a vector, these must now be added
                                                                together.

    xComp = term1[0] + term2[0] + term3[0] #The x components of each term are added
                                                                together to get the x component of the
                                                                rotated vector.

    yComp = term1[1] + term2[1] + term3[1]

    zComp = term1[2] + term2[2] + term3[2]

```

```
return([xComp, yComp, zComp]) #The rotated vector is returned.
```

Now I will introduce a way to move the camera. There are multiple ways of doing this, for example the camera could move relative to the direction the camera is facing, alternatively the camera could move in the same fashion, but ignoring the y-component of the camera's direction and vertical movement could be assigned its own key. There are many other ways in which movement could be implemented but I have decided to go with the latter of the two options presented here.

Moving up and down will be the easiest as this only requires the y coordinate to be changed.

Python code algorithm:

```
def moveUp(distance):
    camera.pos.y = camer.pos.y - distance
def moveDown(distance):
    camera.pos.y = camer.pos.y + distance
```

Moving forward will require the unit vector of [the camera vector without its y-component].

This will be:

$$\frac{vect}{vectMag} = \frac{(camera.vect.z)}{\sqrt{(camera.vect.z)^2 + (camera.vect.x)^2}}$$

Python code algorithm:

```
def getCamHorUnitVect(): #Gets the camera's horizontal unit vector.
    vectMag = ((camera.vect.z)**2 + (camera.vect.x)**2)**(1/2)
    return([(camera.vect.z)/vectMag, (camera.vect.x)/vectMag])
```

This just needs to be added to the camera's vector to move forward and subtracted to move backward.

Python code algorithm:

```
def moveForward(distance):
```

```

camera.pos.x = camera.pos.x + (getCamHorUnitVect()[1]*distance)
camera.pos.z = camera.pos.z + (getCamHorUnitVect()[0]*distance)

def moveBackward(distance):
    camera.pos.x = camera.pos.x - (getCamHorUnitVect()[1]*distance)
    camera.pos.z = camera.pos.z - (getCamHorUnitVect()[0]*distance)

```

Moving left and right requires the getCamHorPerpUnitVect function from before.

Python code algorithm:

```

def moveRight(distance):
    camera.pos.x = camera.pos.x + (getCamHorPerpUnitVect[0]*distance)
    camera.pos.z = camera.pos.z + (getCamHorPerpUnitVect[2]*distance)

def moveLeft(distance):
    camera.pos.x = camera.pos.x - (getCamHorPerpUnitVect[0]*distance)
    camera.pos.z = camera.pos.z - (getCamHorPerpUnitVect[2]*distance)

```

I will now create a procedure to bring all of the camera control algorithms together.

Python code algorithm:

```

def cameraControl(distance, angle):
    #Python cannot detect mouse movements without libraries so for the purposes of
    #this I will just use arrow keys.

    if keyboard.is_pressed("left"):
        rotateYaw(-angle) #When "left" is pressed, rotate left.

    if keyboard.is_pressed("right"):
        rotateYaw(angle)

    if keyboard.is_pressed("up"):
        rotatePitch(-angle)

    if keyboard.is_pressed("down"):
        rotatePitch(angle)

```

```

if keyboard.is_pressed("space"): #When "space" is pressed, move up.

    moveUp(distance)

if keyboard.is_pressed("shift"):

    moveDown(distance)

if keyboard.is_pressed("w"):

    moveForward(distance)

if keyboard.is_pressed("a"):

    moveLeft(distance)

if keyboard.is_pressed("s"):

    moveBackward(distance)

if keyboard.is_pressed("d"):

    moveRight(distance)

```

Finally, I must create a loop which will repeat indefinitely. Each time it runs it will check for and execute camera movements, and then will re-render the screen. It will also later check for physics.

#### Python code algorithm:

GameLoop(): #A loop which runs continuously a set number of times in a given time interval. Python does not support (without certain libraries) these, but C# does so I will ignore this for now.

cameraControl(distance, angle): #These values, affecting the movement and rotation speed can now be easily modified in one place. I might also create a user setting to change them.

renderAll() #Everything is rendered.  
#Physics will be added later.

## Test Plan

Testing Table:

<b><i>Test No.</i></b>	<b><i>Input/Data</i></b>	<b><i>Expected Output/Result</i></b>
1	W key held	Camera should move forward.
2	A key held	Camera should move left.
3	S key held	Camera should move backward.
4	D key held	Camera should move right.
5	Up arrow key held	Camera should tilt up.
6	Left arrow key held	Camera should rotate to the left.
7	Down arrow key held	Camera should tilt down.
8	Right arrow key held	Camera should rotate to the right.

Each of these tests will be done at the end of the development of prototype 1, each being supported with evidence. These tests are to confirm that camera control is fully functional and usable.

## Key Classes, Subroutines and Variables

Classes, functions/procedures and variables for prototype 1 (after refinement).

Class name	Attributes	Description
Face	array points	Stores a face by referring to its vertices/corners
Vector	double x, double y, double z	Stores a vector, position, or anything else represented by three numbers.
Mesh	array points, array faces, Vector dimensions, Vector colour	Stores the data of an object's shape and position, is used for rendering.
Entity	Mesh mesh, RigidBody rigidBody	Will later be used to link the mesh used for rendering, to the rigid body used for physics interactions, e.g., motion and collisions.
Camera	Vector position, Vector direction, double fov	Stores data about the camera which the user will use to view the scene. This includes the camera's position, direction, and field of view.
Global	None (static class)	This is a class which is accessible by the rest of the program, it is used to hold global variables and methods which the rest of the program can then make use of.

I also used encapsulation because this was the only way I was able to access certain objects from outside the Global class:

Class Name	Methods
Entities	get(), set()
Camera	get(), set()

Function/Procedure Name	Inputs	Outputs	Description
calcXCoord	Camera camera, Vector point, double tanfov	Proportion along the screen from the centre, where a point should appear.	This function takes a camera and point as an input. It calculates horizontally along the screen,

			where the point should appear.
calcYCoord	Camera camera, Vector point, double tanfov	Same as above, but vertical.	Same as above, but vertical.
denormalizeX	double x, double width	pixel x-coordinate.	Takes the proportion along the screen where a point appears, along with the width of the screen in pixels, to find the horizontal pixel coordinate of the point.
denormalizeY	double y, double height	pixel y-coord	Same as above, but vertical.
leftOrRight	Vector point, Camera camera	boolean	Find whether the point should be on the left or right side of the screen.
upOrDown	Vector point, Camera camera	boolean	Same as above, but vertical.
renderFace	Face face, Vector colour, Camera camera	void	Renders the given face with the given rotation, taking into account the faces rotation.
renderMesh	Mesh mesh, Camera camera	void	Renders a mesh, face by face. Only renders the 3 faces which are visible.
renderFacesInOrder	Face face1, Face face2, Face face3, Camera camera, Mesh mesh	void	Renders the three given faces in the correct order such that they are drawn furthest to closest.
setMeshFaces	Mesh mesh	void	Uses the points which make up a mesh, to determine what

			its faces and their normals should be.
renderAll	List entities, Camera camera	void	Renders the mesh of each entity in a given list.
gameLoop	void	void	Runs each frame, calls the renderAll procedure and will later be used to call physics procedures.
dotProduct	Two Vectors	double (number)	computes the dot product of two vectors
crossProduct	Two Vectors	Vector	computes the cross product of two vectors
getCamHorPerpUnitVect	Camera camera	Vector (unit vector which lies on the x-z plane which is perpendicular to the camera's direction).	Gets the vector about which the camera vector will be rotated for a pitch rotation.
getCamHorUnitVect	Camera camera	Vector	Same as above, but is the component acting parallel to the camera's direction, not perpendicular.
checkRender	Camera camera, Vector point, double vertFov	boolean	Works out whether a given point would be in front of the camera (true) or not (false).
moveUp	double displacement	void	Moves camera up by some small amount.
moveDown	double displacement	void	Moves camera down by some small amount.

<code>moveForward</code>	<code>double displacement</code>	<code>void</code>	Moves camera forward by some small amount.
<code>moveBackward</code>	<code>double displacement</code>	<code>void</code>	Moves camera backward by some small amount.
<code>moveLeft</code>	<code>double displacement</code>	<code>void</code>	Moves camera left by some small amount.
<code>moveRight</code>	<code>double displacement</code>	<code>void</code>	Moves camera right by some small amount.
<code>rotateYaw</code>	<code>double angle, Vector vect</code>	<code>Vector</code>	Rotates a vector about the y-axis (mainly used to rotate the camera left and right).
<code>rotatePitch</code>	<code>double angle, Vector vect</code>	<code>Vector</code>	rotates the input vector up by the specified angle (down if negative). This is used to rotate the camera up and down.
<code>cameraControl</code>	<code>double displacement, double angle, Camera camera</code>	<code>void</code>	Applies the above operations to the camera depending on which keys the user currently has held down.
<code>getBrightnessLevel</code>	<code>Vector vect, double vectMag</code>	brightness level	The input vector is the normal to a face being drawn. The vectMag is the magnitude (length) of this vector. The returned value is how bright the face being drawn needs to be.

distBetweenPoints	Vector point1, Vector point2	double distance	Returns the distance between two given points.
averagePoint	list of points	Vector averagePoint	Returns the average of a list of given points.
vectBetweenPoints	Vector point1, Vector point2	Vector	Returns the vector going from the first point to the second.
closestPoint	Mesh mesh, Camera camera	int	Returns the index of whichever vertex of a mesh is closest to the camera.
closestPointCoords	Mesh mesh, Camera camera	Vector	Returns the coordinates of the above.

The vast majority of the above functions/procedures are methods of the Global class, these are highlighted in green above.

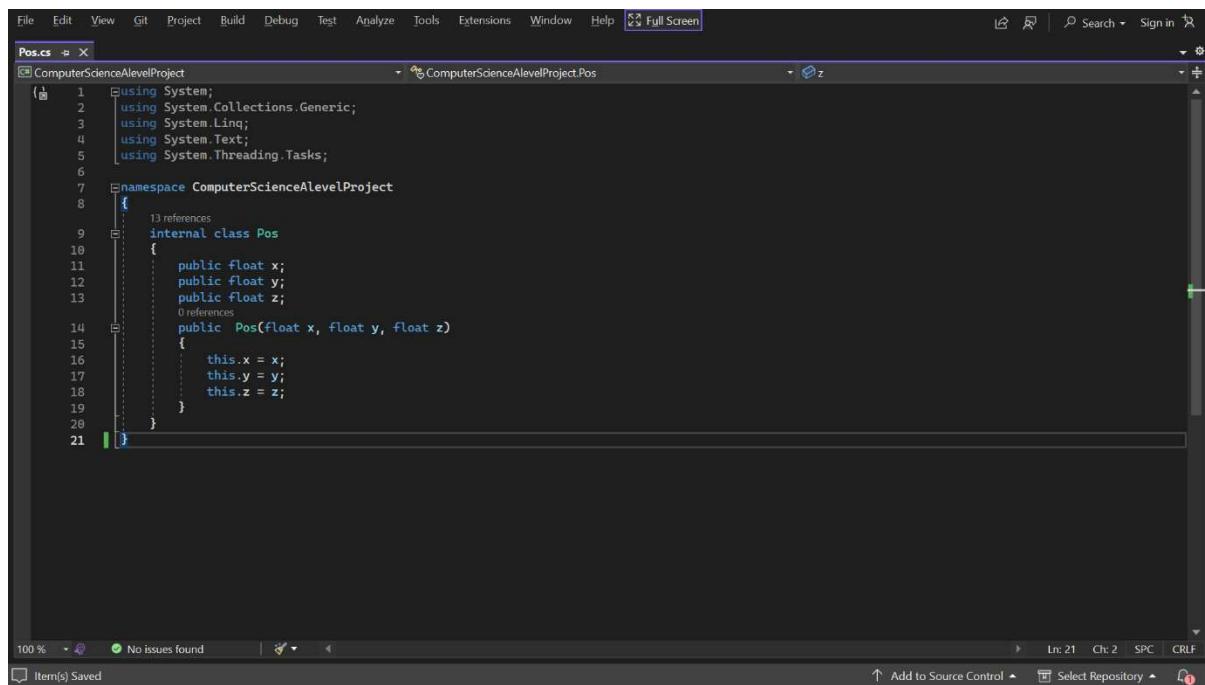
Variable Name	Data Type/Class	Description
lightVector	Vector	The vector which describes the direction which light is coming from. This is used to determine how faces should be coloured.
lightMag	double	Magnitude of the above.
entities	list	Holds a list of all entities which currently “exist”.
camera	Camera	instance of the camera class, is a camera.
camSpeed	double	movement speed of the camera
camSensitivity	double	rotation speed of the camera

This concludes the design of prototype 1, I will now move on to implementation.

# Prototype 1 Development

The first stage of implementation is to create a WinForms project in visual studio. I will then create all the necessary classes as previously outlined.

Position class:



```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen

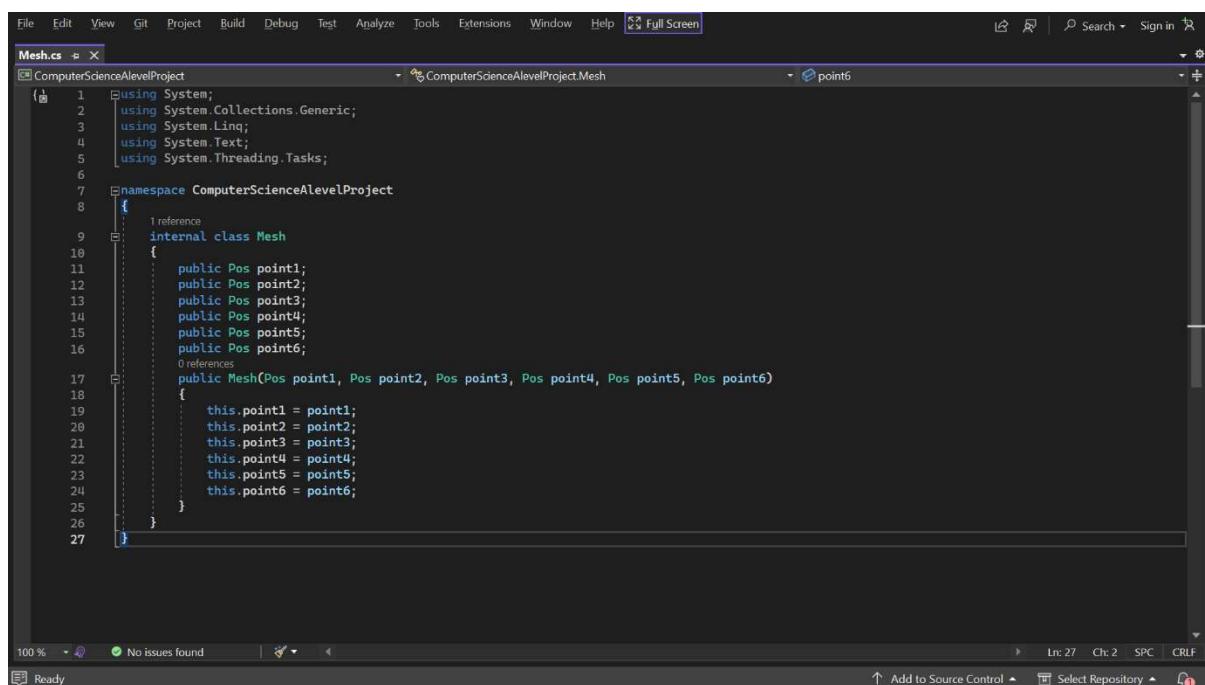
Pos.cs ComputerScienceAlevelProject ComputerScienceAlevelProject.Pos

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ComputerScienceAlevelProject
8 {
9     internal class Pos
10    {
11        public float x;
12        public float y;
13        public float z;
14
15        public Pos(float x, float y, float z)
16        {
17            this.x = x;
18            this.y = y;
19            this.z = z;
20        }
21    }
}

100 % No issues found | L: 21 C: 2 SPC CRLF
Item(s) Saved Add to Source Control Select Repository Ready
```

There will be no Point class as it is redundant because it does the same thing as the position class. There is also no face class as meshes are now comprised directly of points.

Mesh class:



```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen

Mesh.cs ComputerScienceAlevelProject ComputerScienceAlevelProject.Mesh

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ComputerScienceAlevelProject
8 {
9     internal class Mesh
10    {
11        public Pos point1;
12        public Pos point2;
13        public Pos point3;
14        public Pos point4;
15        public Pos point5;
16        public Pos point6;
17
18        public Mesh(Pos point1, Pos point2, Pos point3, Pos point4, Pos point5, Pos point6)
19        {
20            this.point1 = point1;
21            this.point2 = point2;
22            this.point3 = point3;
23            this.point4 = point4;
24            this.point5 = point5;
25            this.point6 = point6;
26        }
27    }
}

100 % No issues found | L: 27 C: 2 SPC CRLF
Item(s) Saved Add to Source Control Select Repository Ready
```

## Vector class:

The screenshot shows a code editor window with the following details:

- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Full Screen, Search, Sign in.
- Code Editor:** The file "Vector.cs" is open, showing the following code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ComputerScienceAlevelProject
8  {
9      1 reference
10     internal class Vector
11     {
12         public float x;
13         public float y;
14         public float z;
15
16         0 references
17         public Vector(float x, float y, float z)
18         {
19             this.x = x;
20             this.y = y;
21             this.z = z;
22         }
23     }
}
```
- Status Bar:** 100%, No issues found, Item(s) Saved, Add to Source Control, Select Repository, CRLF.

Strictly speaking, the Vector class is also redundant as it is identical to the Position class, but I think that creating it is still useful as it will allow me to more easily distinguish between vectors and positions. Also there are certain operations (such as the dot and cross products) which should only be performed on vectors and not positions. For this reason, I think it makes sense to create a separate vector class.

## Entity class:

The screenshot shows a code editor window with the following details:

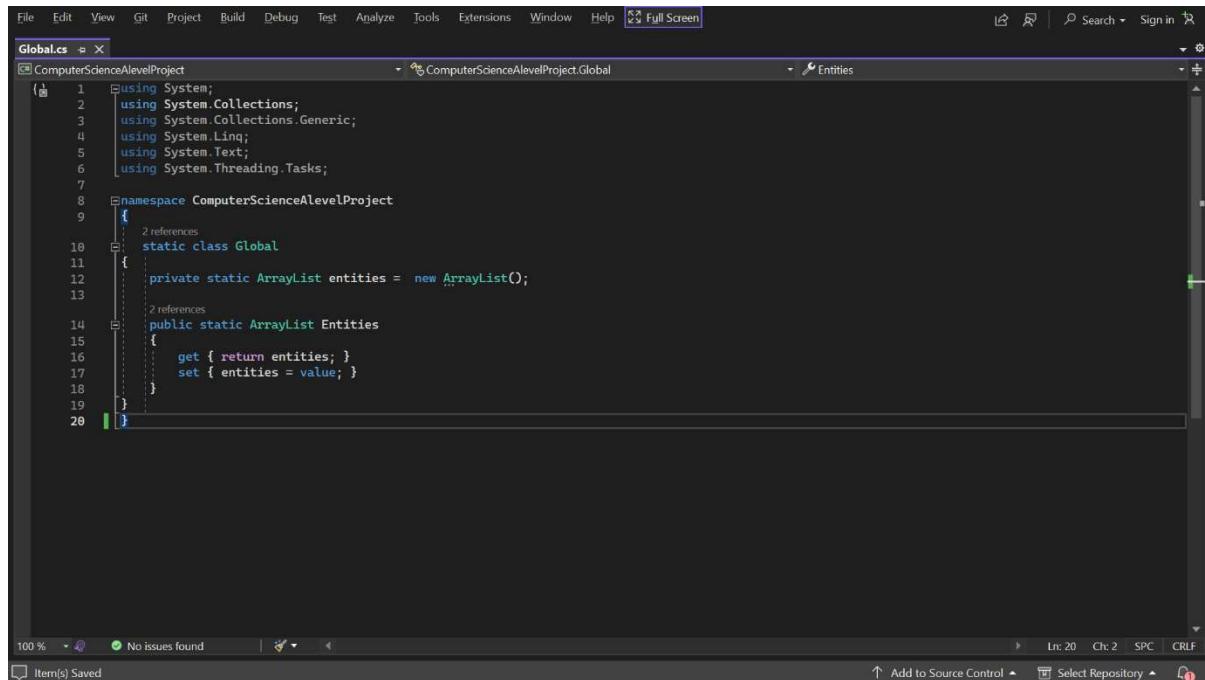
- File Menu:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Full Screen, Search, Sign in.
- Code Editor:** The file "Entity.cs" is open, showing the following code:

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace ComputerScienceAlevelProject
9  {
10     1 reference
11     internal class Entity
12     {
13         public Mesh mesh;
14         public Pos centreOfMass;
15
16         0 references
17         public Entity(Mesh mesh, Pos centreOfMass)
18         {
19             this.mesh = mesh;
20             this.centreOfMass = centreOfMass;
21             ArrayList entities = Global.Entities; //retrieves the list of all entities
22             entities.Add(this); //adds the newly created entity to this list
23             Global.Entities = entities; //saves the new list
24         }
25     }
}
```
- Status Bar:** 100%, No issues found, Ready, Add to Source Control, Select Repository, CRLF.

I will not be using the entity class much for now, as its main purpose is to connect the rigid body to the mesh. For now though, there are no rigid bodies.

I created the ArrayList (array which can change size) using a global class which I created which will allow me to create global variables which can be accessed from anywhere in the program.

Global Class:

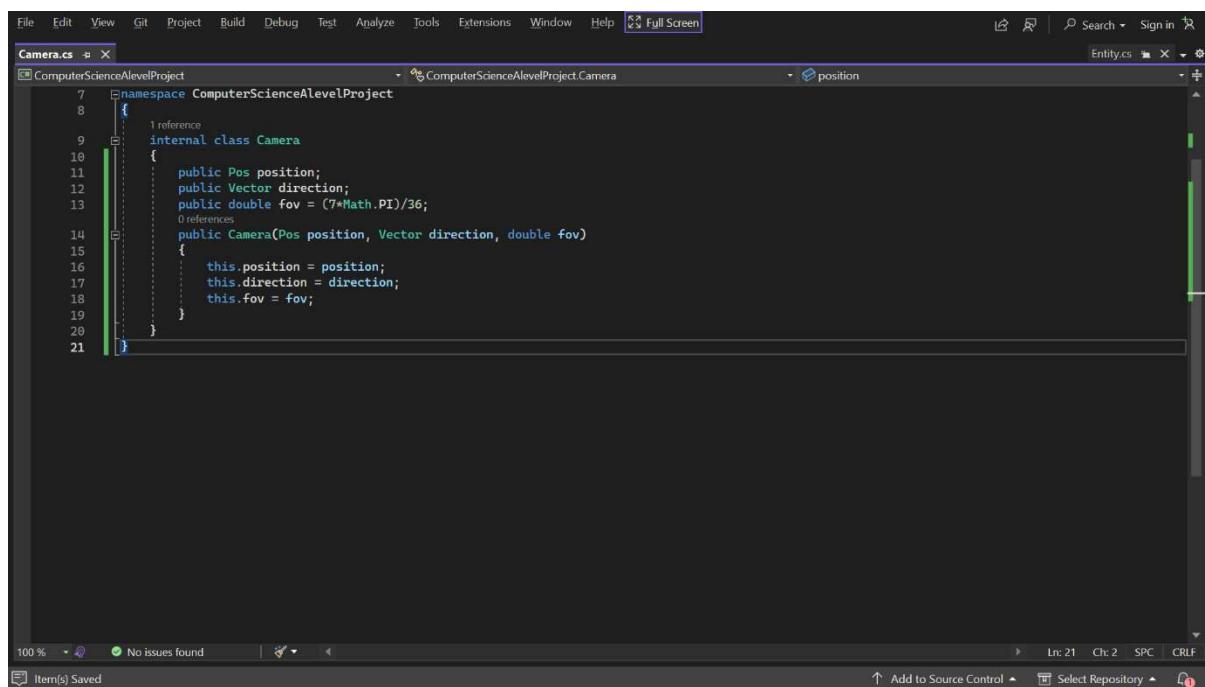


A screenshot of the Visual Studio IDE showing the Global.cs file. The code defines a static class Global with a public static ArrayList Entities. The code is as follows:

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
Global.cs ComputerScienceAlevelProject ComputerScienceAlevelProject.Global Entities
1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace ComputerScienceAlevelProject
9 {
10     static class Global
11     {
12         private static ArrayList entities = new ArrayList();
13
14         public static ArrayList Entities
15         {
16             get { return entities; }
17             set { entities = value; }
18         }
19     }
20 }
```

The code editor shows the file path as Global.cs, the project as ComputerScienceAlevelProject, and the current tab as Global. The status bar at the bottom indicates "Item(s) Saved".

Finally (for now), the camera class:



A screenshot of the Visual Studio IDE showing the Camera.cs file. The code defines an internal class Camera with a constructor that takes Pos position, Vector direction, and double fov. The code is as follows:

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
Camera.cs ComputerScienceAlevelProject ComputerScienceAlevelProject.Camera Entity.cs position
7 namespace ComputerScienceAlevelProject
8 {
9     internal class Camera
10    {
11        public Pos position;
12        public Vector direction;
13        public double fov = (7 * Math.PI) / 36;
14
15        public Camera(Pos position, Vector direction, double fov)
16        {
17            this.position = position;
18            this.direction = direction;
19            this.fov = fov;
20        }
21    }
}
```

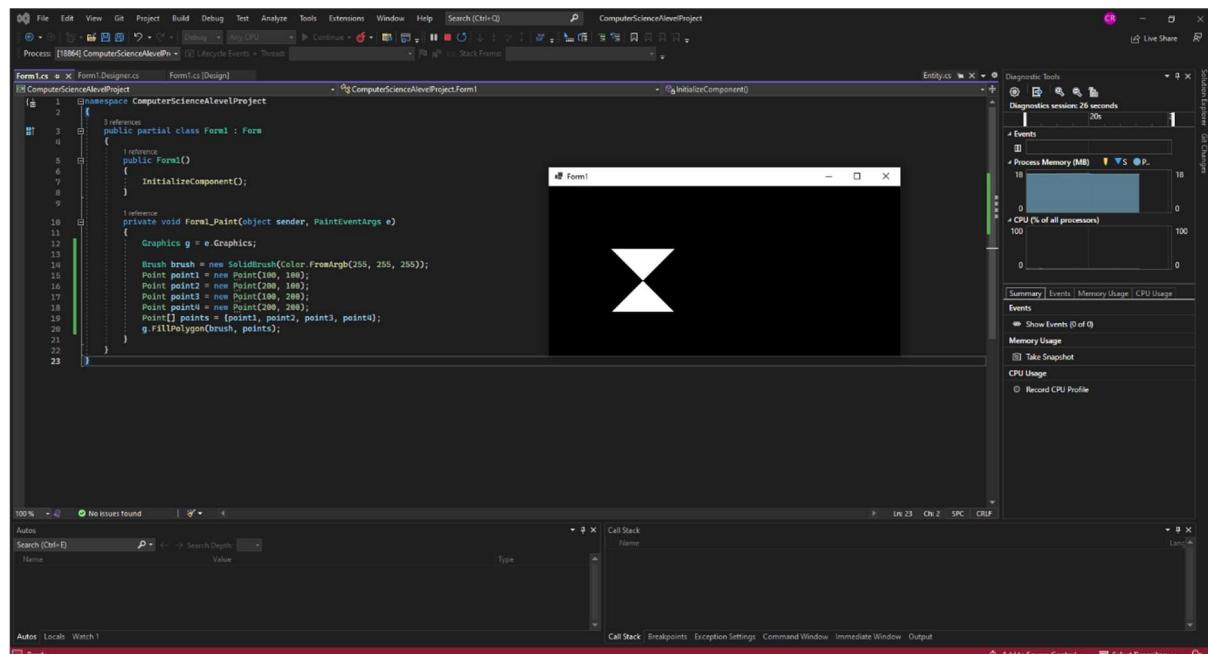
The code editor shows the file path as Camera.cs, the project as ComputerScienceAlevelProject, and the current tab as Camera. The status bar at the bottom indicates "Item(s) Saved".

Whilst making the camera class I discovered that C# uses doubles for the “Math” library which I will be using a lot of, so at this stage I went back and changed any usage of the “float” data type to “double”.

I next changed the background of the form to black:

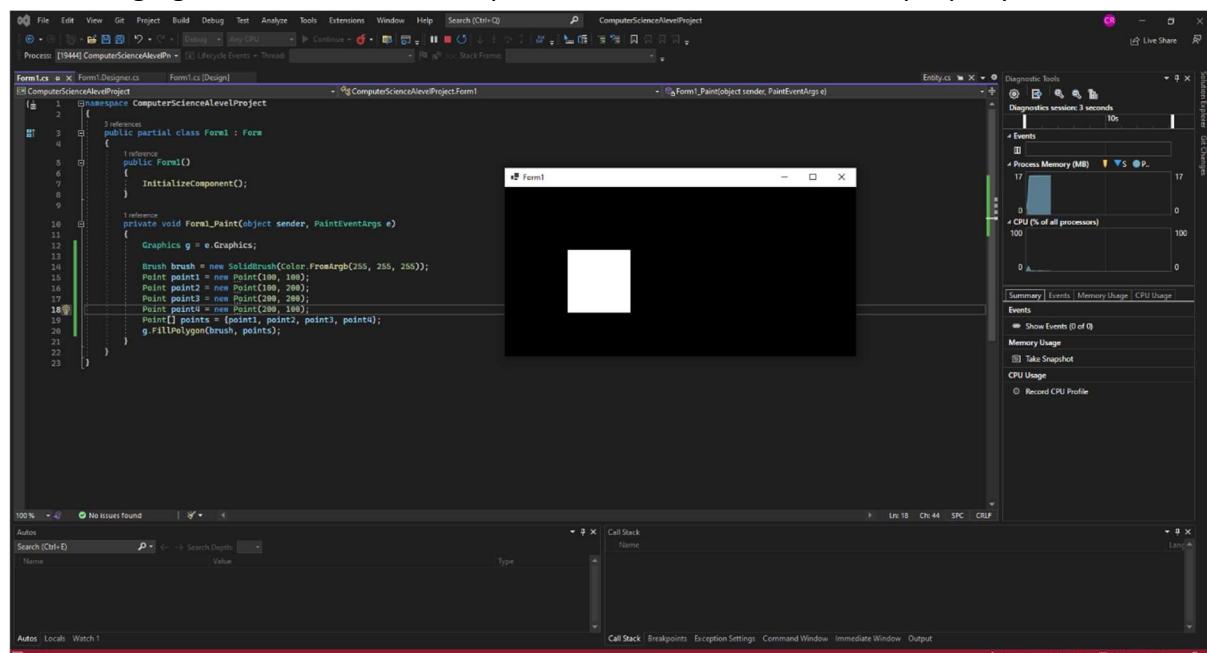
I might later make the background colour a setting which the user can change.

Before creating any complicated functions and procedures, I will need to be able to draw things into the window (form) which is created when I run the program. This can be done using the `DrawPolygon()` method.



Here, I attempted to draw a square by using the “`DrawPolygon()`” method, which instead gave me this. This shows that order is important when using this method. (I cannot use the `DrawRectangle()` method because usually the shape will be non-rectangular).

After changing the order in which the points were drawn, it rendered properly:



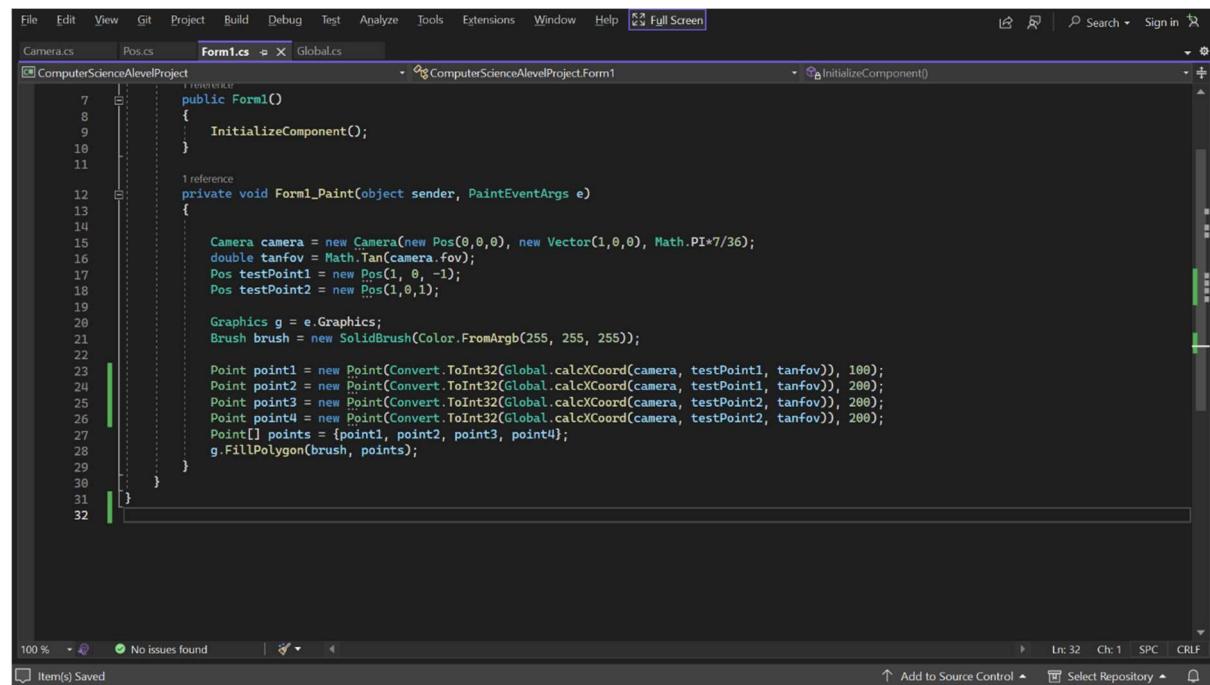
I will need to bear the fact that order matters in mind later.

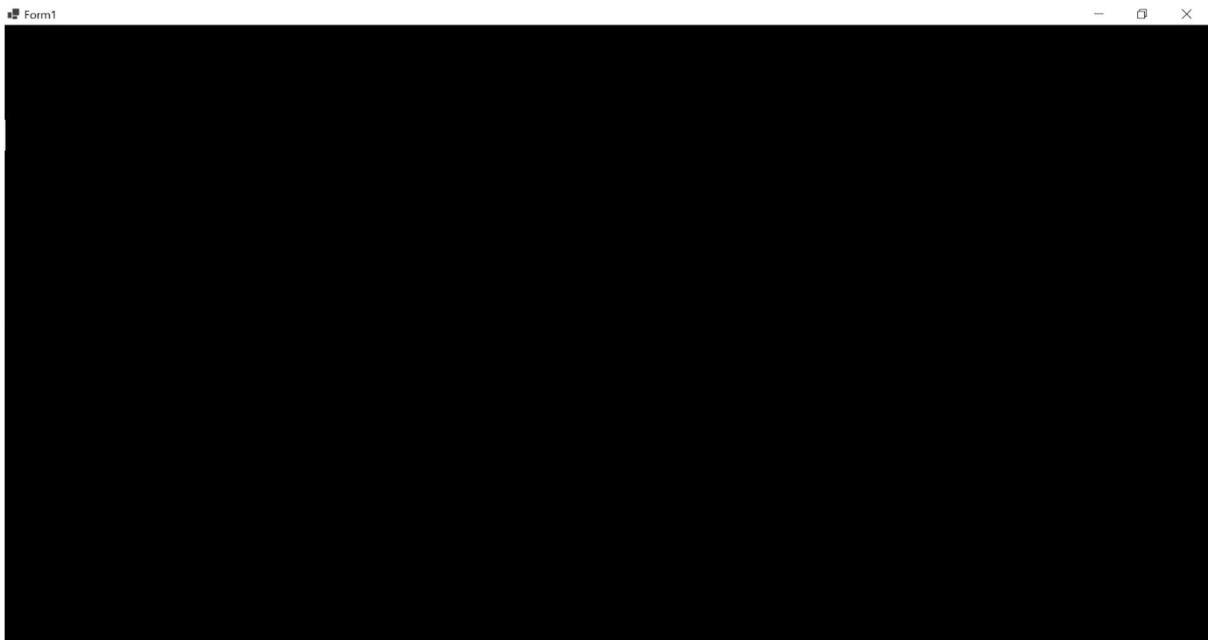
I will next need to create the first proper algorithm in C#. Because these functions often need to access and be accessed by a variety of different classes, I will make them methods of the Global class so that they can be accessed from anywhere.

Here I have recreated the calcXCoord function in C#:

```
public static double calcXCoord(Camera camera, Pos point, double tanfov) //tanfov is tan(field of view)
{
    if(camera.direction.z == 0) //special case 1
    {
        double dist = point.z - camera.position.z;
        double length = (point.x - camera.position.x)*tanfov;
        return(dist/length);
    }
    else if (camera.direction.x == 0) //special case 2
    {
        double dist = camera.position.x - point.x;
        double length = (point.z - camera.position.z) * tanfov;
        return(dist/length);
    }
    else //no special case
    {
        double p = (point.x*(camera.direction.x*camera.position.z/camera.direction.z)+(camera.direction.z*point.z*camera.direction.x)-camera.position.x) / ((camera.direction.x/camera.direction.z)+(camera.direction.z/camera.direction.x));
        double q = camera.direction.x * p / camera.direction.z + camera.position.x - camera.direction.x * camera.position.z / camera.direction.z;
        double dist = Math.Sqrt(Math.Pow(p-point.z, 2)+Math.Pow(q-point.x, 2));
        double length = Math.Sqrt(Math.Pow(p-camera.position.z, 2)+Math.Pow(q-camera.position.x, 2))*tanfov;
        return(dist/length);
    }
}
```

I next created an instance of the camera class and two instances of the point class to test whether the points (and hence the line) appear as they should.





They did not, instead appearing as a single-pixel-wide strip which is barely visible at the side. This is because I forgot to denormalise the coordinates. I will now rewrite the denormalise function in the Global class to allow this to work properly.

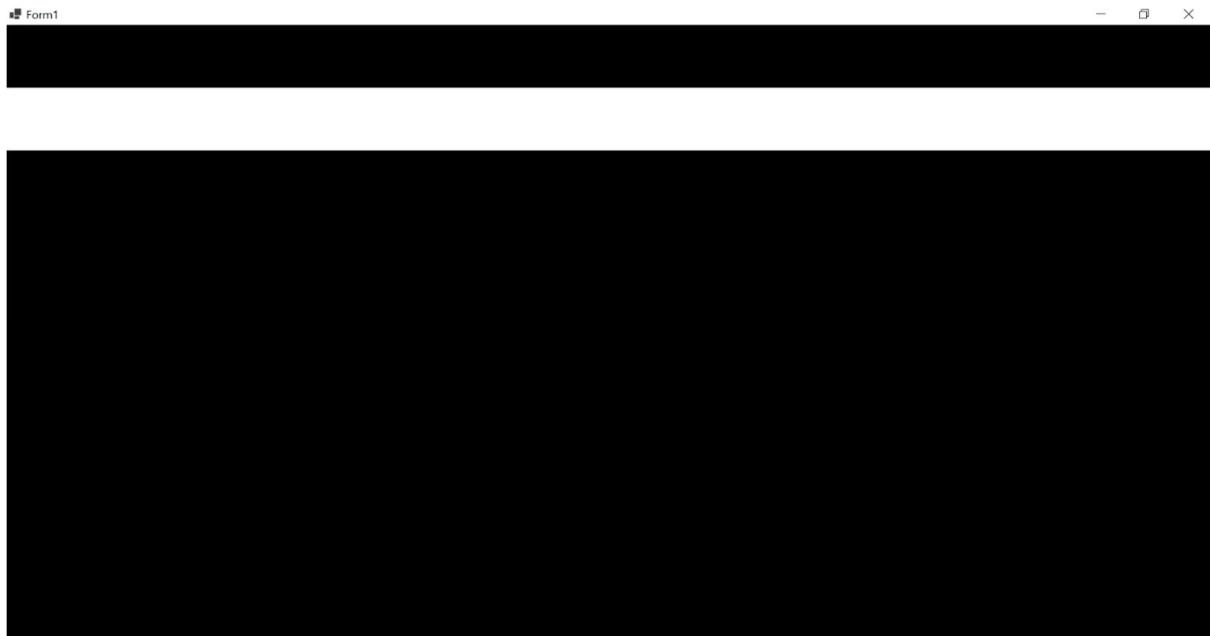
This is the function which will denormalise the x coordinate. This function is a method of the Global class.

```
public static int denormaliseX(double x, double width)
{
    return Convert.ToInt32((width / 2 * (x + 1)));
}
```

This is now being used when rendering the points.

```
int formWidth = this.Width;
Point point1 = new Point(Global.denormaliseX(Global.calcXCoord(camera, testPoint1, tanfov), formWidth), 100);
Point point2 = new Point(Global.denormaliseX(Global.calcXCoord(camera, testPoint1, tanfov), formWidth), 200);
Point point3 = new Point(Global.denormaliseX(Global.calcXCoord(camera, testPoint2, tanfov), formWidth), 200);
Point point4 = new Point(Global.denormaliseX(Global.calcXCoord(camera, testPoint2, tanfov), formWidth), 100);
Point[] points = {point1, point2, point3, point4};
g.FillPolygon(brush, points);
```

This is the result:



This is certainly more reasonable of a result, but it may or may not be correct. I have tried using various camera positions and directions, which seem to give the expected results. The only way to test this fully will be to implement the camera control features now. (I only need to do this for movement on the  $x$ - $z$  plane and rotation about the  $y$ -axis because I am currently only concerned with the horizontal rasterization of the points.)

I found that the form has a method called `Update()` which recalls the paint event allowing the contents of the window to be updated. This does not clear the current contents of the window, which is a problem. Another method called `Refresh()` does the same thing but does clear the window. This method, however, makes the window unresponsive. Another method called `Invalidate()` solves both of these problems however there is flickering. This flickering is solved by setting “`DoubleBuffered`” to true.

I moved all variables to the Global class because otherwise they get reassigned to the default values every time that the form is repainted.

At this point everything is working as it should, as in the window can be moved and resized and what is being displayed in the window is rescaled accordingly in real time, meaning success criteria “Main window which is moveable and resizable.” has been met.

After doing this and setting the camera to move backwards by 0.1 every cycle of a while loop, the bar on the screen changes size in an appropriate manner. This can be seen in [video1](#). The code is here:

```

1  File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help F1 Full Screen
2  ComputerScienceAlevelProject ComputerScienceAlevelProject
3  ComputerScienceAlevelProject
4  1 reference
5  public partial class Form1 : Form
6  {
7      public Form1()
8      {
9          InitializeComponent();
10     }
11
12     private async void Form1_Paint(object sender, PaintEventArgs e)
13     {
14         double tanfov = Math.Tan(Global.Camera.fov);
15         Pos testPoint1 = new Pos(1, 0, -1);
16         Pos testPoint2 = new Pos(-1, 0, 1);
17
18         Graphics g = e.Graphics;
19         Brush brush = new SolidBrush(Color.FromArgb(255, 255, 255));
20         int formWidth = this.Width;
21         int formHeight = this.Height;
22         this.DoubleBuffered = true;
23
24         Point point1 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint1, tanfov), formWidth), 100);
25         Point point2 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint1, tanfov), formWidth), 200);
26         Point point3 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint2, tanfov), formWidth), 200);
27         Point point4 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint2, tanfov), formWidth), 100);
28         Point[] points = { point1, point2, point3, point4 };
29         g.FillPolygon(brush, points);
30
31         Global.Camera.position.x = Global.Camera.position.x - 0.001;
32         Invalidate();
33     }
34 }

```

I next created the rotateYaw() procedure in the Global class to test rotation of the camera. I then rotated the camera a small amount each time the window is redrawn. The code for this is here:

```

52     public static void rotateYaw(double angle) //Rotates the camera right by some angle
53     {
54         camera.direction.x = camera.direction.x * Math.Cos(angle) + camera.direction.z * Math.Sin(angle);
55         camera.direction.z = -camera.direction.x * Math.Sin(angle) + camera.direction.z * Math.Cos(angle);
56     }

```

**video2** shows the camera being rotated anticlockwise, though something is clearly wrong because there is never a point in the video at which the bar takes up the entire screen. This could be a problem with the rotateYaw() procedure or with the NoSpecialCase of the calcXCoord() function.

I realised that the rotateYaw procedure calculates a new *x* component and then it uses this new *x* component to calculate the new *z* component. To prevent this from happening, I created two new variables newX and newZ which will both be calculated before being assigned to the actual vector components of the camera.

This was not the cause of the problem however. The cause of the problem, I have realised, is that I have not yet implemented the “leftOrRight()” function which checks for just this.

```

61     public static bool leftOrRight(Pos point)
62     {
63         if(camera.direction.z>0) //camera is facing right
64         {
65             if(point.x < camera.direction.x*point.z / camera.direction.z + camera.position.x - camera.direction.x*camera.position.z / camera.direction.z)
66             {
67                 return (true);
68             }
69             else
70             {
71                 return (false);
72             }
73         }
74         else //vector is facing left (facing due north or south was already covered by special cases)
75         {
76             if(point.x < camera.direction.x*point.z / camera.direction.z + camera.position.x - camera.direction.x*camera.position.z / camera.direction.z)
77             {
78                 return (false);
79             }
80             else
81             {
82                 return (true);
83             }
84         }
85     }

```

To use this function, I have added this to the end of the NoSpecialCase:

```

if (leftOrRight(point) == true)
{
    return (dist / length);
}
else
{
    return (-dist / length);
}

```

The rotation can be seen working as intended in [video3](#). You'll notice that the bar is now going left and right and back again. It should only be going in one direction (because the camera is only being rotated in one direction and so it is not changing its rotation). The reason this *seems* to happen is because it is still being rendered when behind the camera. This should be solved when I implement culling.

I will first implement vertical rasterisation. First I have created the calcYCoord(), denormaliseY() and upOrDown():

```

57     public static double calcYCoord(Camera camera, Pos point, double tanfov) //tanfov is tan((field of view)/2)
58     {
59         double pointW = Math.Sqrt(Math.Pow(point.x,2)+Math.Pow(point.z,2));
60         double dirW = Math.Sqrt(Math.Pow(camera.direction.x,2)+Math.Pow(camera.direction.z,2));
61         double camPosW = Math.Sqrt(Math.Pow(camera.position.x,2)+Math.Pow(camera.position.z,2)); //W values all calculated
62         if (dirW == 0) //special case 1
63         {
64             double dist = pointW - camPosW;
65             double length = (-point.y + camera.position.y) * tanfov;
66             return (dist / length);
67         }
68         else if (-camera.direction.y == 0) //special case 2
69         {
70             double dist = -camera.position.y + point.y;
71             double length = (pointW - camPosW) * tanfov;
72             return (dist / length);
73         }
74         else //no special case
75         {
76             double p = (-point.y + (-camera.direction.y * camPosW) / dirW) + ((dirW * pointW) / -camera.direction.y) + camera.position.y / ((-camera.direction.y / dirW) + (dirW / -camera.direction.y));
77             double q = ((-camera.direction.y * p) / dirW) - camera.position.y - ((-camera.direction.y * camPosW) / dirW);
78             double dist = Math.Sqrt(Math.Pow(p - pointW, 2) + Math.Pow(q + point.y, 2));
79             double length = Math.Sqrt(Math.Pow(p - camPosW, 2) + Math.Pow(q + camera.position.y, 2)) * tanfov;
80             if (upOrDown(point, pointW, dirW, camPosW) == true)
81             {
82                 return (dist / length);
83             }
84             else
85             {
86                 return (-dist / length);
87             }
88         }
89     }

```

I also corrected the comment here as technically tanfov is  $\tan\left(\frac{fov}{2}\right)$  not  $\tan(fov)$ .

```

95     public static int denormaliseY(double y, double height)
96     {
97         return Convert.ToInt32((height / 2 * (y + 1)));
98     }

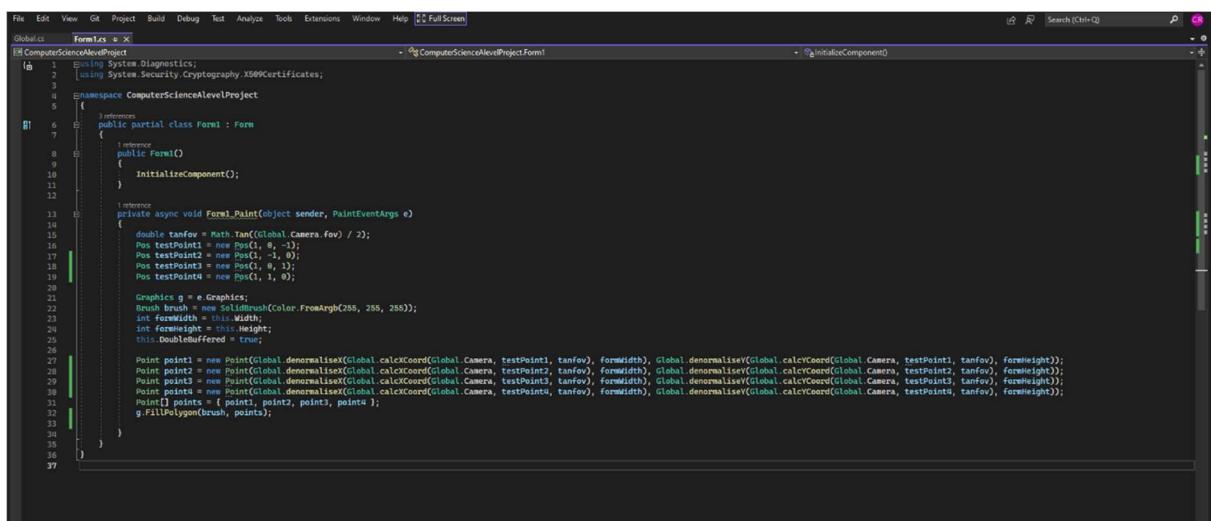
```

```

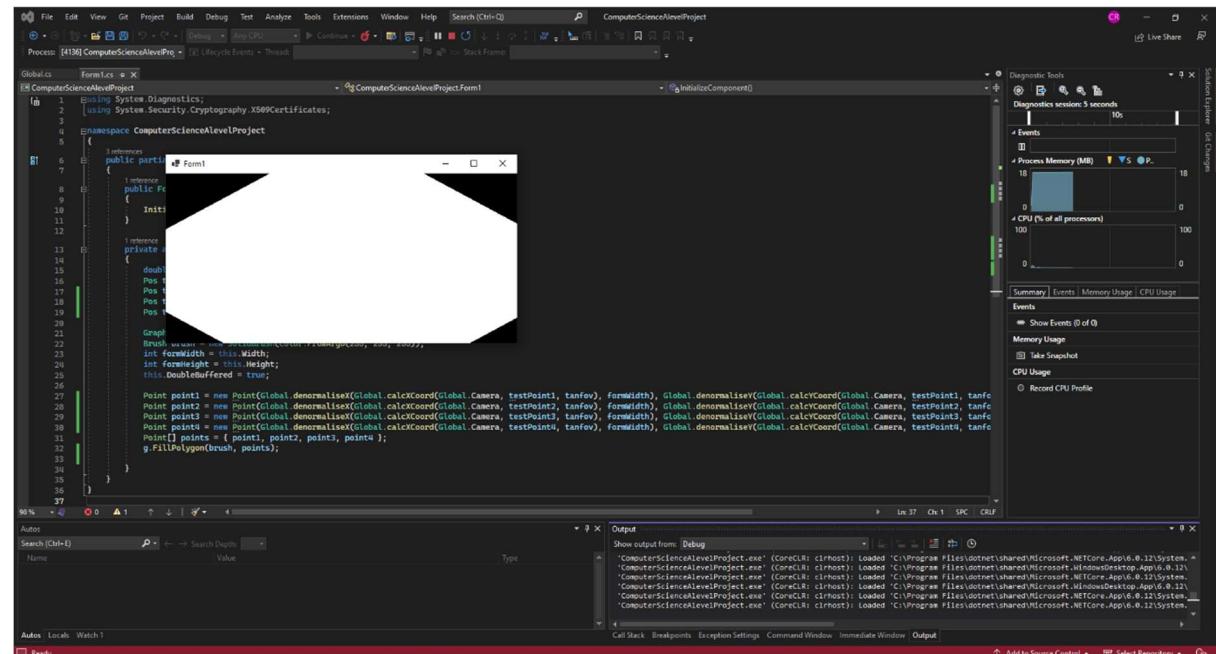
131     public static bool upOrDown(Pos point, double pointW, double dirW, double camPosW)
132     {
133         if (dirW > 0)
134         {
135             if (-point.y < -camera.direction.y * pointW / dirW - camera.position.y + camera.direction.y * camPosW / dirW)
136             {
137                 return (true);
138             }
139             else
140             {
141                 return (false);
142             }
143         }
144         else
145         {
146             if (-point.y < -camera.direction.y * pointW / dirW - camera.position.y + camera.direction.y * camPosW / dirW)
147             {
148                 return (false);
149             }
150             else
151             {
152                 return (true);
153             }
154         }
155     }

```

Here I created four test points and attempted to fully render the polygon they would make:



This was the result (as expected):



By moving the camera backwards as was done before, what happens is shown in [video4](#). As the camera moves back, the left and right points (with  $y$ -coordinate 0) move inwards horizontally as they should and as they were doing before. The top and bottom points (with  $x$ -coordinate 0) which both start off-screen, remain off-screen as they do not move inwards. The application then crashes due to some error. It is clear that there is an issue with the vertical rasterisation of points. (I made the camera move extra slowly here to make it clearer what is happening in the video).

I think this happens because when calculating the  $w$ -coordinates it assumes that the camera and the point are in the same  $xz$  quadrant. This means that when the  $x$ -coordinate of the camera is  $-1$  the program thinks that the camera is inside the plane being rendered and this leads to a division by zero. I will add a check for whether the camera and point are in the same  $xz$  quadrant or not (if not then `camPosW` will be made negative) and will later check if the point is inside the plane (if “length” is zero) in which case it will not be rendered.

Checking  $w$ -coordinate quadrant:

```

57     public static double calcYCoord(Camera camera, Pos point, double tanfov) //tanfov is tan(field of view)/2
58     {
59         double pointW;
60         if (camera.position.x > 0 && point.x < 0 | camera.position.x < 0 && point.x > 0 | camera.position.z > 0 && point.z < 0 | camera.position.z < 0 && point.z > 0)
61         {
62             pointW = -Math.Sqrt(Math.Pow(point.x, 2) + Math.Pow(point.z, 2));
63         }
64         else
65         {
66             pointW = Math.Sqrt(Math.Pow(point.x, 2) + Math.Pow(point.z, 2));
67         }

```

After implementing this, it now works correctly as seen in [video5](#).

I will implement the other check later when I get to culling.

Now I will implement pitch rotation (vertical rotation).

Using the new Vector class (as opposed to an array), the cross product function would be easier to write by using the matrix than the python code which I made before:

$$\begin{aligned}
 & \begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} \\
 &= i \begin{vmatrix} y_1 & z_1 \\ y_2 & z_2 \end{vmatrix} - j \begin{vmatrix} x_1 & z_1 \\ x_2 & z_2 \end{vmatrix} + k \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} \\
 &= i(y_1z_2 - z_1y_2) - j(x_1z_2 - z_1x_2) + k(x_1y_2 - y_1x_2) \\
 &= i(y_1z_2 - z_1y_2) + j(z_1x_2 - x_1z_2) + k(x_1y_2 - y_1x_2) \\
 &\quad \begin{pmatrix} y_1z_2 - z_1y_2 \\ z_1x_2 - x_1z_2 \\ x_1y_2 - y_1x_2 \end{pmatrix}
 \end{aligned}$$

dotProduct function and cross product functions:

```
166     public static double dotProduct(Vector vect1, Vector vect2)
167     {
168         return (vect1.x*vect2.x + vect1.y*vect2.y + vect1.z*vect2.z);
169     }
170     public static Vector crossProduct(Vector vect1, Vector vect2)
171     {
172         double xComp = vect1.y * vect2.z - vect1.z * vect2.y;
173         double yComp = vect1.z * vect2.x - vect1.x * vect2.z;
174         double zComp = vect1.x * vect2.y - vect1.y * vect2.x;
175         return (new Vector(xComp, yComp, zComp));
176     }
```

I now need to write the getCamHorPerpUnitVect() function.

```
177     public static Vector getCamHorPerpUnitVect(Camera camera) //gets the vector about which the camera vector will be rotated
178     {
179         double vectMag = Math.Sqrt(Math.Pow(camera.direction.x,2)+Math.Pow(camera.direction.z,2));
180         double zComp = camera.direction.x / vectMag;
181         double xComp = -camera.direction.z / vectMag;
182         return (new Vector(xComp, 0, zComp));
183     }
```

Everything needed for the rotatePitch() procedure is now in place:

```
116     public static void rotatePitch(double angle) //Rotates the camera down by some angle
117     {
118         Vector k = getCamHorPerpUnitVect(camera);
119         Vector v = camera.direction;
120         double cosine = Math.Cos(angle);
121         double sine = Math.Sin(angle);
122
123         Vector term1 = new Vector(v.x*cosine, v.y*cosine, v.z*cosine);
124         Vector crossProd = crossProduct(k, v);
125         Vector term2 = new Vector(crossProd.x*sine, crossProd.y*sine, crossProd.z*sine);
126         double dotProd = dotProduct(k, v);
127         Vector term3 = new Vector(k.x*dotProd*(1-cosine), k.y*dotProd*(1-cosine), k.z*dotProd*(1-cosine));
128         camera.direction = new Vector(term1.x + term2.x + term3.x, term1.y + term2.y + term3.y, term1.z + term2.z + term3.z); //the cameras direction is changed to this new vector
129     }
```

Using this, I get the result shown in [video6](#). The camera seems to rotate up (it should rotate down) and then the object never comes back into view (which it should do once the camera has done a full revolution). I found that rotating the camera once by  $\pi$  radians so that the plane is directly behind it, the plane is rendered in (because culling has still not been implemented). I then found that rotating it by 3.14 does not have the same effect as rotating it by 3 and then by 0.14 which means that something is definitely wrong with the rotatePitch() procedure.

Whilst trying to find the cause of the problem, I discovered that when checking w-coordinate quadrant I set both of the pointWs to negative when the first should have been negative and the second positive.

After fixing this, it now rotates down as intended. The other problem persists. I believe this has something to do with the w-coordinates because I did not consider all possible cases, for example the w-coordinate assumes that the line going from the point to the camera goes through the y-axis. To solve the other problem, I will change my approach. Instead of using the w-coordinate to find the perpendicular distance from the line of sight to the point, I will instead find the perpendicular distance from the point to the plane with maximum slope in the direction of the line of sight and a slope in that direction equal to the vertical slope of the camera vector. Using partial derivatives:

$\nabla y$  is the vector of partial derivatives of  $y$ , its magnitude is the magnitude of maximum slope and it point in the direction of maximum slope.

$$\nabla y = \begin{pmatrix} \frac{\partial y}{\partial x} \\ \frac{\partial y}{\partial y} \\ \frac{\partial y}{\partial z} \end{pmatrix} = \lambda \begin{pmatrix} camera.vect.x \\ camera.vect.z \end{pmatrix}$$

$$\Rightarrow \frac{\partial y}{\partial x} \div \frac{\partial y}{\partial z} = \frac{camera.vect.x}{camera.vect.z}$$

$$\Rightarrow \frac{\partial y}{\partial x} = \frac{camera.vect.x}{camera.vect.z} \times \frac{\partial y}{\partial z}$$

and:

$$\sqrt{\left(\frac{\partial y}{\partial x}\right)^2 + \left(\frac{\partial y}{\partial z}\right)^2} = \frac{camera.vect.y}{\sqrt{(camera.vect.x)^2 + (camera.vect.z)^2}}$$

$$\Rightarrow \left(\frac{\partial y}{\partial x}\right)^2 + \left(\frac{\partial y}{\partial z}\right)^2 = \frac{(camera.vect.y)^2}{(camera.vect.x)^2 + (camera.vect.z)^2}$$

so:

$$\left(\frac{camera.vect.x}{camera.vect.z} \times \frac{\partial y}{\partial z}\right)^2 + \left(\frac{\partial y}{\partial z}\right)^2 = \frac{(camera.vect.y)^2}{(camera.vect.x)^2 + (camera.vect.z)^2}$$

$$\left(\frac{\partial y}{\partial z}\right)^2 \left(\left(\frac{camera.vect.x}{camera.vect.z}\right)^2 + 1\right) = \frac{(camera.vect.y)^2}{(camera.vect.x)^2 + (camera.vect.z)^2}$$

$$\left(\frac{\partial y}{\partial z}\right)^2 = \frac{(camera.vect.y)^2}{\left(\left(\frac{camera.vect.x}{camera.vect.z}\right)^2 + 1\right)((camera.vect.x)^2 + (camera.vect.z)^2)}$$

$$\left(\frac{\partial y}{\partial z}\right)^2 = \frac{(camera.vect.y)^2}{\frac{(camera.vect.x)^4}{(camera.vect.z)^2} + 2(camera.vect.x)^2 + (camera.vect.z)^2}$$

$$\frac{\partial y}{\partial z} = \frac{camera.vect.y}{\sqrt{\frac{(camera.vect.x)^4}{(camera.vect.z)^2} + 2(camera.vect.x)^2 + (camera.vect.z)^2}}$$

$$Let D = \frac{camera.vect.y}{\sqrt{\frac{(camera.vect.x)^4}{(camera.vect.z)^2} + 2(camera.vect.x)^2 + (camera.vect.z)^2}}$$

$$\frac{\partial y}{\partial x} = \frac{camera.vect.x}{camera.vect.z} \times \frac{\partial y}{\partial z}$$

$$\frac{\partial y}{\partial x} = D \times \frac{camera.vect.x}{camera.vect.z} \quad \& \quad \frac{\partial y}{\partial z} = D$$

Where  $D$  is a known constant.

This means that for a unit change in  $x$  there is a change in  $y$  of  $D \times \frac{\text{camera.vect.x}}{\text{camera.vect.z}}$  and for a unit change in  $z$ , there is a change in  $y$  of  $D$ . The plane can therefore be expressed using the following vectors which lie on the plane:

$$v_1 = \begin{pmatrix} 1 \\ D \times \frac{\text{camera.vect.x}}{\text{camera.vect.z}} \\ 0 \end{pmatrix}$$

$$v_2 = \begin{pmatrix} 0 \\ D \\ 1 \end{pmatrix}$$

I now need to find the vector perpendicular to these two (perpendicular to the plane) which I can then use to find the equation of said plane. Luckily, the cross product (for which an algorithm has already been produced) performs precisely this function.

$$\text{If } v_1 \times v_2 = \text{crossProd}(v_1, v_2) = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

then the equation of the plane is  $ax + by + cz = d$  where  $a, b$  and  $c$  are known constants and  $d$  is a constant to be determined. As the plane passes through the camera:

$$d = a \times \text{camera.position.x} + b \times \text{camera.position.y} + c \times \text{camera.position.z}$$

which can then be put back into the above equation of the plane.

The perpendicular distance between this plane and the point  $P = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$  is given by:

$$\frac{|ap_1 + bp_2 + cp_3 - d|}{\sqrt{a^2 + b^2 + c^2}}$$

This is equivalent to the value  $dist$  in the old algorithm.

I next must find the distance from the camera to the point where the line which passes through  $P$  and is perpendicular to the plane, intersects the plane. This distance can then be used with trigonometry to find the equivalent of the value  $length$  in the original algorithm. These values can then be divided as originally intended, completing the new algorithm.

To find this distance I will first find the point of intersection. For this I will the equation of the line perpendicular to the plane, passing through  $P$  which in vector form is:

$$\mathbf{r} = \mathbf{p} + \lambda \mathbf{t}$$

where  $\mathbf{r}$  is the general position vector of the points on the line and so is the general vector  $\begin{pmatrix} x \\ y \\ z \\ a \\ b \\ c \end{pmatrix}$ ,  $\mathbf{p}$  is the position vector of  $P$  and  $\mathbf{t}$  is the perpendicular vector which we already know is

$\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ .

Separating into individual vector components and making  $\lambda$  the subject:

$$x = p_1 + \lambda a$$

$$\lambda = \frac{x - p_1}{a}$$

$$y = p_2 + \lambda b$$

$$\lambda = \frac{y - p_2}{b}$$

$$z = p_3 + \lambda c$$

$$\lambda = \frac{z - p_3}{c}$$

Eliminating  $\lambda$ :

$$\frac{x - p_1}{a} = \frac{y - p_2}{b} = \frac{z - p_3}{c}$$

Which is the cartesian equation of our line.

I now have 3 equations  $\frac{x - p_1}{a} = \frac{y - p_2}{b} = \frac{z - p_3}{c}$ ,  $ax + by + cz = d$  and three unknowns  $x$ ,  $y$ ,  $z$ . ( $\frac{x - p_1}{a} = \frac{z - p_3}{c}$  does not count as a fourth equation as it can be derived from the first two and so adds no new information).

$$ax + by + cz = d$$

$$y = \frac{d - ax - cz}{b}, (1)$$

$$\frac{x - p_1}{a} = \frac{y - p_2}{b}$$

$$y = \frac{b(x - p_1)}{a} + p_2, (2)$$

$$\frac{y - p_2}{b} = \frac{z - p_3}{c}$$

$$y = \frac{b(z - p_3)}{c} + p_2, (3)$$

$$\frac{b(x - p_1)}{a} + p_2 = \frac{b(z - p_3)}{c} + p_2, (2) = (3)$$

$$\frac{x - p_1}{a} = \frac{z - p_3}{c}$$

As mentioned before, this can be derived from the other two,

$$x = \frac{a(z - p_3)}{c} + p_1$$

$$\frac{d - ax - cz}{b} = \frac{b(x - p_1)}{a} + p_2, (1) = (2)$$

$$ad - a^2x - acz = b^2x - b^2p_1 + abp_2$$

$$ad - acz + b^2p_1 - abp_2 = b^2x + a^2x$$

$$x(a^2 + b^2) = ad - acz + b^2p_1 - abp_2$$

$$x = \frac{ad - acz + b^2p_1 - abp_2}{a^2 + b^2}$$

$$\frac{ad - acz + b^2p_1 - abp_2}{a^2 + b^2} = \frac{a(z - p_3)}{c} + p_1$$

$$\frac{ad - acz + b^2p_1 - abp_2}{a^2 + b^2} = \frac{az - ap_3}{c} + p_1$$

$$acd - ac^2z + b^2cp_1 - abcp_2 = (a^2 + b^2)(az - ap_3 + cp_1)$$

$$acd - ac^2z + b^2cp_1 - abcp_2 = a^3z - a^3p_3 + a^2cp_1 + ab^2z - ab^2p_3 + b^2cp_1$$

$$ac^2z + a^3z + ab^2z = a^3p_3 - a^2cp_1 + ab^2p_3 - b^2cp_1 + acd + b^2cp_1 - abcp_2$$

$$z(ac^2 + a^3 + ab^2) = a^3p_3 - a^2cp_1 + ab^2p_3 + acd - abcp_2$$

$$z = \frac{a^3p_3 - a^2cp_1 + ab^2p_3 + acd - abcp_2}{ac^2 + a^3 + ab^2}$$

$$z = \frac{a^2p_3 - acp_1 + b^2p_3 + cd - bcp_2}{a^2 + b^2 + c^2}$$

Using this value for  $z$ , find  $x$ :

$$x = \frac{a(z - p_3)}{c} + p_1$$

Then find  $y$ :

$$y = \frac{d - ax - cz}{b}$$

Using these  $x$ ,  $y$  and  $z$  values find the distance between the camera and the intersection point using:

$$\sqrt{(camera.position.x - x)^2 + (camera.position.y - y)^2 + (camera.position.z - z)^2}$$

Using this as the adjacent find the value of "length" by using  $\tan \theta = \frac{\text{opposite}}{\text{adjacent}} = \frac{\text{length}}{\text{adjacent}}$

$$\text{length} = \text{adjacent} \times \tan \theta$$

Where  $\theta$  is half the vertical FOV. This has so far been assumed to be the same as the horizontal field of view, however this assumes that the screen being used by the user is a square, which it is likely not. This has the result of stretching the image to fit the rectangular screen. I will work out the vertical FOV later on, for now I will keep it the same as the horizontal. Finally, the function upOrDown() must be rewritten to check if the point is above or below the plane as opposed to the line. Equation of the plane:

$$ax + by + cz = d$$

$$y = \frac{d - ax - cz}{b}$$

Checking if the point is above or below this line:

$$\text{If } \text{point}.position.y < \frac{d - a(\text{point}.position.x) - c(\text{point}.position.z)}{b}$$

(The camera is above the line) return true. Otherwise, false should be returned.

The above algorithm yields a few problems. First:

$$\frac{\partial y}{\partial x} \div \frac{\partial y}{\partial z} = \frac{\text{camera}.vect.x}{\text{camera}.vect.z}$$

When  $\text{camera}.vect.z = 0$  this is undefined. In this case we can apply a special case.  $\frac{\partial y}{\partial z} = 0$  since the direction of maximum slope has a  $z$  component of 0. This leaves only one unknown,  $\frac{\partial y}{\partial x}$  which equals the maximum slope which (when  $\text{camera}.vect.z = 0$ ) is simply  $\frac{\text{camera}.vect.y}{\text{camera}.vect.x}$  and so

$$\frac{\partial y}{\partial x} = \frac{dy}{dx} = \frac{\text{camera}.vect.y}{\text{camera}.vect.x}$$

$$y = \int \frac{\text{camera}.vect.y}{\text{camera}.vect.x} dx$$

$$y = \frac{\text{camera}.vect.y}{\text{camera}.vect.x} \times x + c$$

$$\text{camera}.position.y = \frac{\text{camera}.vect.y}{\text{camera}.vect.x} \times \text{camera}.position.x + c$$

$$c = \text{camera}.position.y - \frac{\text{camera}.vect.y}{\text{camera}.vect.x} \times \text{camera}.position.x$$

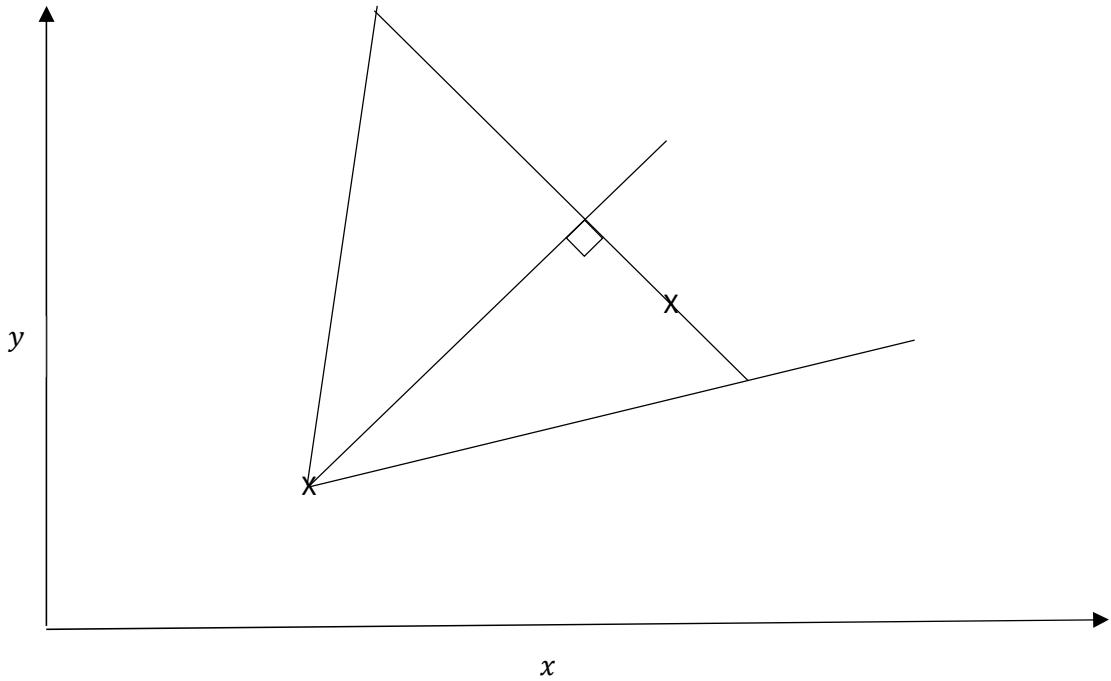
$$y = \frac{\text{camera}.vect.y}{\text{camera}.vect.x} \times x + \text{camera}.position.y - \frac{\text{camera}.vect.y}{\text{camera}.vect.x} \times \text{camera}.position.x$$

$$y - \frac{\text{camera}.vect.y}{\text{camera}.vect.x} x = \text{camera}.position.y - \frac{\text{camera}.vect.y}{\text{camera}.vect.x} \times \text{camera}.position.x$$

[I have used *camera.vect* instead of *camera.direction* so that each equation can remain on a single line.]

Which is the equation of the plane in this special case.

To find “*dist*” and “*length*” in this case, a side on view could be used, ignoring the *z*-axis as every point on the plane has the same *z*-coordinate (*camera.position.z*). For this I can use the exact same algorithm as for calcXCoord except using *y* instead of *x*, and *x* instead of *z*.



(I will fix any issues with the *y* axis being the wrong way around later.)

After implementing this, I still encounter errors for cases where *c* = 0 (plane is horizontal in the *z* direction) or *b* = 0, (camera is pointing directly up or down).

If *b* = 0 then the *y*-component of the vector perpendicular to the plane is 0. This means the plane is completely vertical (camera points directly up or down). When the camera looks directly up or down, the calcXCoord function ceases to work as the *x* and *z* components of the camera’s direction are both 0. I will simply prevent the camera from looking directly up or down. The original plan was to allow the camera to rotate its pitch until being completely vertical (up or down) but no more. I will change this to allow it to get to the point where is

very close to being completely vertical but not get all the way there. Making the case where  $b = 0$  impossible.

If  $c = 0$  then the  $z$  component of the vector perpendicular to the plane is 0. This means either, the plane is horizontal, or  $camera.direction.z = 0$ . As there is already an if statement for  $camera.direction.z = 0$ , I will add the new code in there. As for the plane being horizontal,  $dist = point.position.y - camera.position.y$  and  $adjacent = \sqrt{(point.position.x - camera.position.x)^2 + (point.position.z - camera.position.z)^2}$

I will not need to use the upOrDown function in this case as the sign will be determined anyway.

The vertical FOV is currently the same as the horizontal FOV. This assumes that the monitor is square (which it will likely not be for the end user) and so the image appears stretched. To fix this I must determine the correct vertical FOV for a given horizontal FOV.

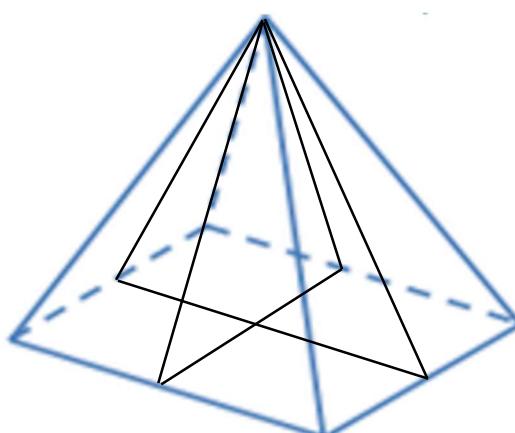
$depth$  is the height of the pyramid, width and height are the width and height of the screen and  $fov_h$  and  $fov_v$  are the horizontal and vertical fields of view respectively.

```
public static double calcCoord(Camera camera, Pos point, double tanfov) //tanfov is tan((field of view)/2)
{
    double D = (camera.direction.y) / Math.Sqrt((Math.Pow(camera.direction.x, 4) / Math.Pow(camera.direction.z, 2)) + 2 * Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2));
    double a;
    double b;
    double c;
    double d;
    double x;
    double y;
    double z;
    double dist;
    double length;
    double adjacent;
    //Plane is horizontal
    if (camera.direction.y == 0)
    {
        dist = point.y - camera.position.y;
        adjacent = Math.Sqrt((point.x - camera.position.x), 2) + Math.Pow(point.z - camera.position.z, 2));
        length = adjacent + tanfov;
        return dist / length;
    }
    //y/xz=0
    else if (camera.direction.z == 0)
    {
        //Equation of plane is: y = [camera.direction.y/camera.direction.x]*x + camera.position.y - (camera.direction.y/camera.direction.x)*camera.position.x
        //In the form ax+by+cz=d:
        a = -camera.direction.y / camera.direction.x;
        b = 1;
        c = 0;
        d = camera.position.y - (camera.direction.y / camera.direction.x) * camera.position.x;
        //since c = 0 in this case, I find dist and length in here.
        Pos dummyPos = new Pos(-camera.position.y, 0, camera.position.x);
        Vector dummyVector = new Vector(-camera.direction.y, 0, camera.direction.x);
        Camera dummyCamera = new Camera(dummyPos, dummyVector, tanfov); //Replaced x by -y and replaced z by x so that the calcXCoord function can be reused with the parameters switched around.
        return calcXCoord(dummyCamera, point, tanfov);
    }
    //No special case:
    else
    {
        Vector v1 = new Vector(1, D * (camera.direction.x / camera.direction.z), 0);
        Vector v2 = new Vector(0, D, 1);
        Vector perpVect = crossProduct(v1, v2);
        a = perpVect.x;
        b = perpVect.y;
        c = perpVect.z;
        d = a * camera.position.x + b * camera.position.y + c * camera.position.z;

        dist = Math.Abs(a * point.x + b * point.y + c * point.z - d) / Math.Sqrt(a * a + b * b + c * c);
        z = (Math.Pow(a, 2) * point.z - a * c * point.x + Math.Pow(b, 2) * point.z + c * d - b * c * point.y) / (Math.Pow(a, 2) + Math.Pow(b, 2) + Math.Pow(c, 2));
        x = a * (z - point.z) / c + point.x;
        y = (d - a * x - c * z) / b;
        adjacent = Math.Sqrt(Math.Pow(camera.position.x - x, 2) + Math.Pow(camera.position.y - y, 2) + Math.Pow(camera.position.z - z, 2));
        length = adjacent * tanfov;
    }
}
```

$$\tan\left(\frac{fov_h}{2}\right) = \frac{width}{2} \div depth$$

$$depth = \frac{width}{2} \div \tan\left(\frac{fov_h}{2}\right)$$



$$\tan\left(\frac{fov_v}{2}\right) = \frac{height}{2} \div depth$$

$$depth = \frac{height}{2} \div \tan\left(\frac{fov_v}{2}\right)$$

$$\frac{width}{2} \div \tan\left(\frac{fov_h}{2}\right) = \frac{height}{2} \div \tan\left(\frac{fov_v}{2}\right)$$

$$\frac{width}{2} \times \tan\left(\frac{fov_v}{2}\right) = \frac{height}{2} \times \tan\left(\frac{fov_h}{2}\right)$$

$$\tan\left(\frac{fov_v}{2}\right) = \frac{height}{width} \times \tan\left(\frac{fov_h}{2}\right)$$

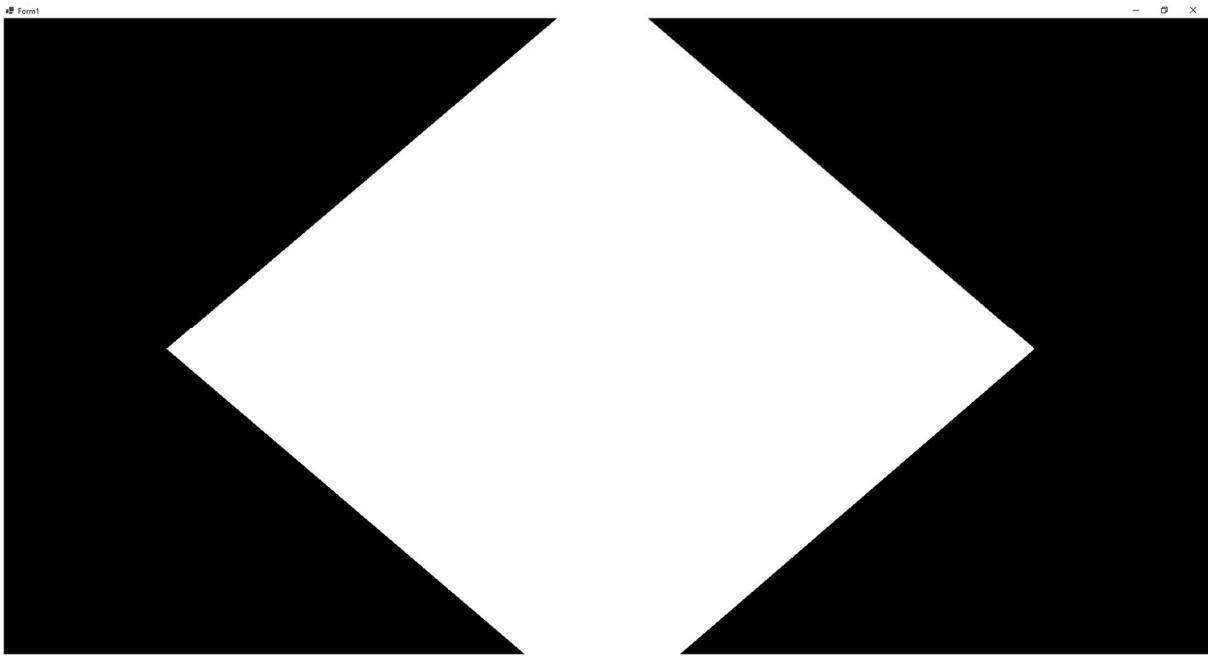
$$\frac{fov_v}{2} = \tan^{-1}\left(\frac{height}{width} \times \tan\left(\frac{fov_h}{2}\right)\right)$$

$$fov_v = 2 \tan^{-1} \left( \frac{height}{width} \times \tan \left( \frac{fov_h}{2} \right) \right)$$

After this, the shape appears as a tilted square instead of a stretched one as it was before.

```
int formWidth = this.Width;
this.DoubleBuffered = true;
double vertfov = 2 * Math.Atan((formHeight * Math.Tan(Global.Camera.fov / 2)) / formWidth);
double verttanfov = Math.Tan(vertfov / 2);

Point point1 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint1, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, testPoint1, tanfov), formWidth));
Point point2 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint2, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, testPoint2, tanfov), formWidth));
Point point3 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint3, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, testPoint3, tanfov), formWidth));
Point point4 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, testPoint4, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, testPoint4, tanfov), formWidth));
points = new Point[] { point1, point2, point3, point4 };
g.FillPolygon(brush, points);
//Global.renderBatch();
Invalidate();
```



Just after doing this, I believe I have worked out what the problem with pitch rotation was. The camera rotates about the perpendicular vector to the right, but as the direction of the camera rotates beyond verticality, this perpendicular vector (still on the right) switches sides, meaning that the camera rotates in the opposite direction. This continues keeping it perpetually facing (roughly) upwards hence the image does not come back when the camera continues rotating. I will now create an algorithm (as previously mentioned) to prevent the camera from rotating beyond this point. I will do this by creating a dummy vector inside the rotatePitch function. This will hold the vector of the camera's direction which has been calculated. It will then be determined if the camera should be rotated or not. I will do this by checking whether the new direction (stored in the dummy vector) will (ignoring the y-component) be facing the same direction as before (the camera currently). If they face the same direction, update the camera's direction to the new one. If not, then don't. After implementing this, I tested it by increasing the height of the tilted square so it could be seen when looking from a higher angle and after [videos 7 and 8](#) show this before and after the implementation of the above. Before it was very jittery (and would likely be near impossible to control by the user) after, it is now much better.

New rotatePitch function improvements:

```
public static void rotatePitch(double angle) //Rotates the camera up by some angle
{
    Vector k = getCamHorPerpUnitVect(camera);
    Vector v = camera.direction;
    double cosine = Math.Cos(angle);
    double sine = Math.Sin(angle);

    Vector term1 = new Vector(v.x*cosine, v.y*cosine, v.z*cosine);
    Vector crossProd = crossProduct(k, v);
    Vector term2 = new Vector(crossProd.x*sine, crossProd.y*sine, crossProd.z*sine);
    double dotProd = dotProduct(k, v);
    Vector term3 = new Vector(k.x*dotProd*(1-cosine), k.y*dotProd*(1-cosine), k.z*dotProd*(1-cosine));
    Vector dummyVector = new Vector(term1.x + term2.x + term3.x, term1.y + term2.y + term3.y, term1.z + term2.z + term3.z);
    if (dummyVector.x > 0 && Global.Camera.direction.x < 0 || dummyVector.x < 0 && Global.Camera.direction.x > 0 || dummyVector.z > 0 && Global.Camera.direction.z < 0 || dummyVector.z < 0 && Global.Camera.direction.z > 0)
        //camera is not in the same quadrant before or after
    {
        //Do nothing
    }
    else
    {
        Global.Camera.direction = dummyVector; //the cameras direction is changed to this new vector
    }
}
```

I will next implement camera control to complete another success criteria.

I would implement the movement functions as methods of the camera class, but I will instead use the global class for the same reason as why I used it for the rotation. This is necessary because I need to access these methods from “Form1.cs” and “Global.cs” is the only class which can be accessed from outside of itself as it is the bridge connecting all other code files.

```
//User input:  
  
//Creating bools for whether each important key is pressed or not which can be changed and viewed anywhere else in the program.  
private static bool wHeld = false;  
3 references  
public static bool WHeld  
{  
    get { return wHeld; }  
    set { wHeld = value; }  
}  
private static bool aHeld = false;  
2 references  
public static bool AHeld  
{  
    get { return aHeld; }  
    set { aHeld = value; }  
}  
private static bool sHeld = false;  
2 references  
public static bool SHeld  
{  
    get { return sHeld; }  
    set { sHeld = value; }  
}  
private static bool dHeld = false;  
2 references  
public static bool DHeld  
{  
    get { return dHeld; }  
    set { dHeld = value; }  
}  
private static bool upHeld = false;  
0 references  
public static bool UpHeld  
{  
    get { return upHeld; }  
    set { upHeld = value; }  
}  
private static bool downHeld = false;  
public static bool DownHeld  
{  
    get { return downHeld; }  
    set { downHeld = value; }  
}  
private static bool leftHeld = false;  
0 references  
public static bool LeftHeld  
{  
    get { return leftHeld; }  
    set { leftHeld = value; }  
}  
private static bool rightHeld = false;  
0 references  
public static bool RightHeld  
{  
    get { return rightHeld; }  
    set { rightHeld = value; }  
}  
private static bool spaceHeld = false;  
2 references  
public static bool SpaceHeld  
{  
    get { return spaceHeld; }  
    set { spaceHeld = value; }  
}  
private static bool eHeld = false;  
2 references  
public static bool EHeld  
{  
    get { return eHeld; }  
    set { eHeld = value; }  
}  
private static bool shiftHeld = false;  
2 references  
public static bool ShiftHeld  
{  
    get { return shiftHeld; }  
    set { shiftHeld = value; }  
}
```

```

private static bool qHeld = false;
2 references
public static bool QHeld
{
    get { return qHeld; }
    set { qHeld = value; }
}
0 references
public static void cameraControl(double speed, double angle)
{
    if(wHeld)
    {
        moveForward(speed);
    }
    if(aHeld)
    {
        moveLeft(speed);
    }
    if(sHeld)
    {
        moveBackward(speed);
    }
    if(dHeld)
    {
        moveRight(speed);
    }
    if(spaceHeld || eHeld)
    {
        moveUp(speed);
    }
    if(shiftHeld || qHeld)
    {
        moveDown(speed);
    }
    if(upHeld)
    {
        rotatePitch(angle);
    }
    if(downHeld)
    {
        rotatePitch(-angle);
    }
    if(leftHeld)
    {
        rotateYaw(angle);
    }
    if(rightHeld)
    {
        rotateYaw(-angle);
    }
}
//-----

```

After creating this Boolean values and functions I implemented a way to change these values when the corresponding keys are pressed using WinForms' built in events:

```

private void Form1_KeyDown(object sender, KeyEventArgs e)    private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = true;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = true;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = true;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = true;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = true;
    }
    if (e.KeyCode == Keys.Shift)
    {
        Global.ShiftHeld = true;
    }
    if (e.KeyCode == Keys.E)
    {
        Global.EHeld = true;
    }
    if (e.KeyCode == Keys.Q)
    {
        Global.QHeld = true;
    }
    Invalidate();
}

private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = false;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = false;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = false;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = false;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = false;
    }
    if (e.KeyCode == Keys.Shift)
    {
        Global.ShiftHeld = false;
    }
    if (e.KeyCode == Keys.E)
    {
        Global.EHeld = false;
    }
    if (e.KeyCode == Keys.Q)
    {
        Global.QHeld = false;
    }
    Invalidate();
}

```

I forgot the up, down, left and right before taking the screenshots, but those have now been added. Also “Shift”, should have been “ShiftKey” to be recognized as the correct input by the API.

Finally I need to call the CameraControl function repeatedly which can be done easily by using Invalidate().

After all this, basic camera control has been implemented as seen in [video 9](#) and camera control can be checked off the success criteria list.

I will next implement culling so that objects behind the camera are not rendered.

```
public static bool checkRender(Camera camera, Pos point)
{
    if (camera.direction.y == 0)
    {
        //Same code as "leftOrRight()" function but with different equations used
        if(camera.direction.z>0)
        {
            if(point.x < (camera.direction.z / camera.direction.x)*(camera.position.z-point.z)+camera.position.x)
            {
                return (true);
            }
            else
            {
                return (false);
            }
        }
        else
        {
            if(point.x > (camera.direction.z / camera.direction.x) *(camera.position.z - point.z) + camera.position.x)
            {
                return (false);
            }
            else
            {
                return (true);
            }
        }
    }

    else if (camera.direction.y < 0)
    {
        if (point.y < -(camera.direction.x*(point.x-camera.position.x) + camera.direction.z*(point.z-camera.position.z))/camera.direction.y + camera.position.y)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        if (point.y > -(camera.direction.x*(point.x-camera.position.x) + camera.direction.z*(point.z-camera.position.z))/camera.direction.y + camera.position.y)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

This now works (the plane does not render when it is behind the camera) but a more full implementation will be possible when I add 3D objects (cubes).

To do this I will use the “Mesh” class which I created at the beginning. (I corrected a mistake in the class, increasing from 6 points to 8 points.)

I have added a new “Face” class consisting of a list of points.

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ComputerScienceAlevelProject
{
    2 references
    internal class Face
    {
        public ArrayList points;
        0 references
        public Face(ArrayList points)
        {
            this.points = points;
        }
    }
}

```

Next, I created an algorithm to render a single face and then one to render a mesh by using these faces.

```

//Rendering individual face:
void renderFace(Face face, Mesh mesh)
{
    Point point1 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex1, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex1, vertTanFov), formHeight));
    Point point2 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex2, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex2, vertTanFov), formHeight));
    Point point3 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex3, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex3, vertTanFov), formHeight));
    Point point4 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex4, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex4, vertTanFov), formHeight));

    //Render mesh
    void renderMesh(Mesh mesh)
    {
        renderFace(new Face(mesh.point1, mesh.point2, mesh.point3, mesh.point4), mesh); //Top face
        renderFace(new Face(mesh.point5, mesh.point6, mesh.point7, mesh.point8), mesh); //Bottom face
        renderFace(new Face(mesh.point1, mesh.point4, mesh.point8, mesh.point5), mesh); //Left face
        renderFace(new Face(mesh.point2, mesh.point3, mesh.point7, mesh.point6), mesh); //Right face
        renderFace(new Face(mesh.point1, mesh.point2, mesh.point6, mesh.point5), mesh); //Front face
        renderFace(new Face(mesh.point3, mesh.point4, mesh.point8, mesh.point7), mesh); //Back face
    }
}

```

The renderFace procedure takes the mesh as an input so it can be used to determine the colour of the face which I will implement now by using the getBrightnessLevel function.

First, the light vector was defined:

```

private static Vector lightVector = new Vector(-1,1,-1);
private static double lightMag = Math.Sqrt(Math.Pow(lightVector.x, 2) + Math.Pow(lightVector.y, 2) + Math.Pow(lightVector.z, 2));

```

Next the function to determine brightness was created given the normal vector and its magnitude:

```

public static double getBrightnessLevel(Vector vect, double vectMag)
{
    double angle = vect.x*lightVector.x + vect.y*lightVector.y + vect.z*lightVector.z;
    angle = angle/(vectMag*lightMag);
    angle = Math.Acos(angle);
    return (angle/Math.PI);
}

```

This is then added to the face renderer to determine colour:

```

//Rendering individual face:
void renderFace(Face face, Mesh mesh)
{
    //Colour determination
    Pos faceCentre = new Pos((face.vertex1.x + face.vertex2.x + face.vertex3.x + face.vertex4.x) / 4, (face.vertex1.y + face.vertex2.y + face.vertex3.y + face.vertex4.y) / 4, (face.vertex1.z + face.vertex2.z + face.vertex3.z + face.vertex4.z) / 4);
    Pos meshCentre = new Pos((mesh.point1.x + mesh.point2.x + mesh.point3.x + mesh.point4.x + mesh.point5.x + mesh.point6.x + mesh.point7.x + mesh.point8.x) / 8, (mesh.point1.y + mesh.point2.y + mesh.point3.y + mesh.point4.y + mesh.point5.y + mesh.point6.y + mesh.point7.y + mesh.point8.y) / 8, (mesh.point1.z + mesh.point2.z + mesh.point3.z + mesh.point4.z + mesh.point5.z + mesh.point6.z + mesh.point7.z + mesh.point8.z) / 8);
    Vector perpVect = new Vector(faceCentre.x - meshCentre.x, faceCentre.y - meshCentre.y, faceCentre.z - meshCentre.z);
    double vectMag = Math.Sqrt(Math.Pow(perpVect.x, 2) + Math.Pow(perpVect.y, 2) + Math.Pow(perpVect.z, 2));
    double shade = Global.getBrightnessLevel(perpVect, vectMag);
    Brush brush = new SolidBrush(Color.FromArgb(Convert.ToInt16(255*shade), Convert.ToInt16(255*shade), Convert.ToInt16(255*shade)));

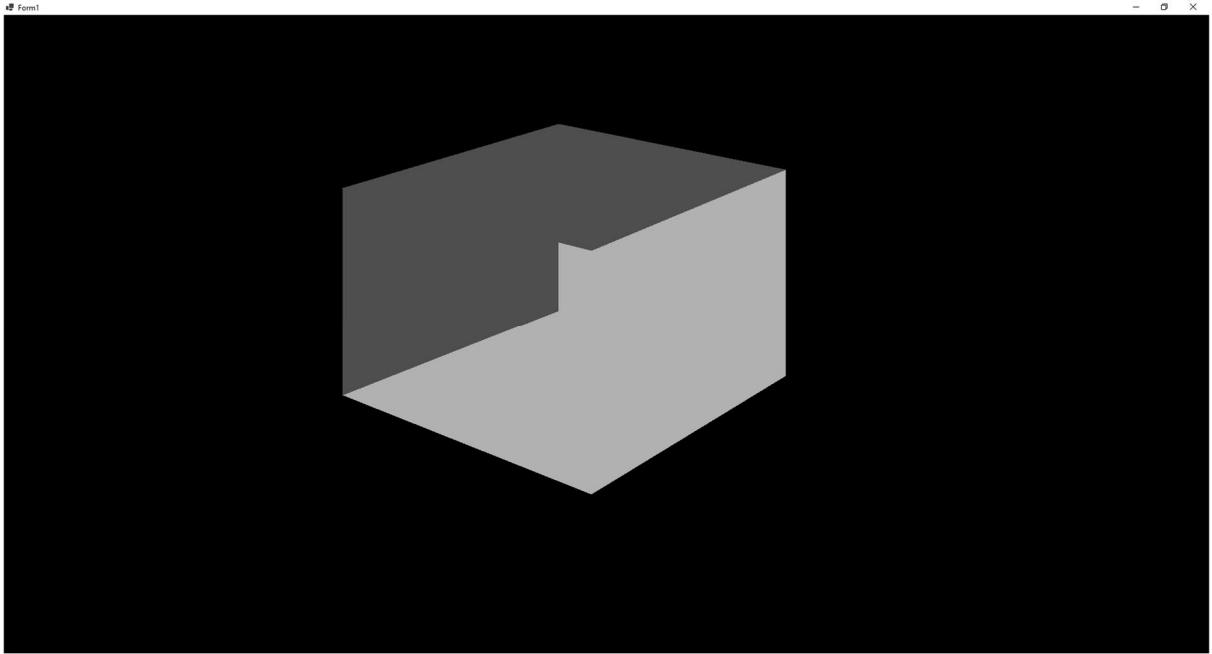
    Point point1 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex1, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex1, vertTanFov), formHeight));
    Point point2 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex2, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex2, vertTanFov), formHeight));
    Point point3 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex3, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex3, vertTanFov), formHeight));
    Point point4 = new Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertex4, tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera, face.vertex4, vertTanFov), formHeight));
    Point[] points = { point1, point2, point3, point4 };
    g.FillPolygon(brush, points);
}

```

(This code is cut off, but the rest was just calculating the average of the vertices to find the central point.)

This algorithm is only a few lines as opposed to the multiple pages which I wrote in the plan because the original plan turned out to be much simpler to implement and will also allow shapes other than strictly cuboids to be rendered.

This is the result so far:



The cube is drawn! The problem is that its faces are being rendered in the wrong order. This is a problem to be addressed by the algorithms which find which sides will be visible and render only those. Once this has been implemented, the cube should appear as normal.

### Algorithms:

```
public static int closestPoint(Mesh mesh, Camera camera)
{
    double dist1 = Math.Sqrt(Math.Pow(mesh.point1.x - camera.position.x, 2) + Math.Pow(mesh.point1.y - camera.position.y, 2) + Math.Pow(mesh.point1.z - camera.position.z, 2));
    double dist2 = Math.Sqrt(Math.Pow(mesh.point2.x - camera.position.x, 2) + Math.Pow(mesh.point2.y - camera.position.y, 2) + Math.Pow(mesh.point2.z - camera.position.z, 2));
    double dist3 = Math.Sqrt(Math.Pow(mesh.point3.x - camera.position.x, 2) + Math.Pow(mesh.point3.y - camera.position.y, 2) + Math.Pow(mesh.point3.z - camera.position.z, 2));
    double dist4 = Math.Sqrt(Math.Pow(mesh.point4.x - camera.position.x, 2) + Math.Pow(mesh.point4.y - camera.position.y, 2) + Math.Pow(mesh.point4.z - camera.position.z, 2));
    double dist5 = Math.Sqrt(Math.Pow(mesh.point5.x - camera.position.x, 2) + Math.Pow(mesh.point5.y - camera.position.y, 2) + Math.Pow(mesh.point5.z - camera.position.z, 2));
    double dist6 = Math.Sqrt(Math.Pow(mesh.point6.x - camera.position.x, 2) + Math.Pow(mesh.point6.y - camera.position.y, 2) + Math.Pow(mesh.point6.z - camera.position.z, 2));
    double dist7 = Math.Sqrt(Math.Pow(mesh.point7.x - camera.position.x, 2) + Math.Pow(mesh.point7.y - camera.position.y, 2) + Math.Pow(mesh.point7.z - camera.position.z, 2));
    double dist8 = Math.Sqrt(Math.Pow(mesh.point8.x - camera.position.x, 2) + Math.Pow(mesh.point8.y - camera.position.y, 2) + Math.Pow(mesh.point8.z - camera.position.z, 2));
    double shortestDist = Math.Min(dist1,dist2);
    shortestDist = Math.Min(shortestDist,dist3);
    shortestDist = Math.Min(shortestDist,dist4);
    shortestDist = Math.Min(shortestDist,dist5);
    shortestDist = Math.Min(shortestDist,dist6);
    shortestDist = Math.Min(shortestDist,dist7);
    shortestDist = Math.Min(shortestDist,dist8);
    if (shortestDist == dist1)
    {return 1;}
    else if (shortestDist == dist2)
    {return 2;}
    else if (shortestDist == dist3)
    {return 3;}
    else if (shortestDist == dist4)
    {return 4;}
    else if (shortestDist == dist5)
    {return 5;}
    else if (shortestDist == dist6)
    {return 6;}
    else if (shortestDist == dist7)
    {return 7;}
    else
    {return 8;}
}
```

```

//Render mesh
void renderMesh(Mesh mesh)
{
    int closestPoint = Global.closestPoint(mesh, Global.Camera);
    switch (closestPoint)
    {
        case 1:
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point3, mesh.point4), mesh); //Top face
            renderFace(new Face(mesh.point1, mesh.point4, mesh.point8, mesh.point5), mesh); //Left face
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point6, mesh.point5), mesh); //Front face
            break;
        case 2:
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point3, mesh.point4), mesh); //Top face
            renderFace(new Face(mesh.point2, mesh.point3, mesh.point7, mesh.point6), mesh); //Right face
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point6, mesh.point5), mesh); //Front face
            break;
        case 3:
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point3, mesh.point4), mesh); //Top face
            renderFace(new Face(mesh.point2, mesh.point3, mesh.point7, mesh.point6), mesh); //Right face
            renderFace(new Face(mesh.point3, mesh.point4, mesh.point8, mesh.point7), mesh); //Back face
            break;
        case 4:
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point3, mesh.point4), mesh); //Top face
            renderFace(new Face(mesh.point1, mesh.point4, mesh.point8, mesh.point5), mesh); //Left face
            renderFace(new Face(mesh.point3, mesh.point4, mesh.point8, mesh.point7), mesh); //Back face
            break;
        case 5:
            renderFace(new Face(mesh.point5, mesh.point6, mesh.point7, mesh.point8), mesh); //Bottom face
            renderFace(new Face(mesh.point1, mesh.point4, mesh.point8, mesh.point5), mesh); //Left face
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point6, mesh.point5), mesh); //Front face
            break;
        case 6:
            renderFace(new Face(mesh.point5, mesh.point6, mesh.point7, mesh.point8), mesh); //Bottom face
            renderFace(new Face(mesh.point2, mesh.point3, mesh.point7, mesh.point6), mesh); //Right face
            renderFace(new Face(mesh.point1, mesh.point2, mesh.point6, mesh.point5), mesh); //Front face
            break;
        case 7:
            renderFace(new Face(mesh.point5, mesh.point6, mesh.point7, mesh.point8), mesh); //Bottom face
            renderFace(new Face(mesh.point2, mesh.point3, mesh.point7, mesh.point6), mesh); //Right face
            renderFace(new Face(mesh.point3, mesh.point4, mesh.point8, mesh.point7), mesh); //Back face
            break;
        case 8:
            renderFace(new Face(mesh.point5, mesh.point6, mesh.point7, mesh.point8), mesh); //Bottom face
            renderFace(new Face(mesh.point1, mesh.point4, mesh.point8, mesh.point5), mesh); //Left face
            renderFace(new Face(mesh.point3, mesh.point4, mesh.point8, mesh.point7), mesh); //Back face
            break;
    }
}

```

The result is seen in [video 10](#) (ignore the colour change). There are two problems, the first is that the logic (the only visible faces will be those adjacent to the closest vertex) is usually correct but not always, the other is that moving up and down does not cause the cube to scale correctly horizontally, this is because calcXCoord does not take vertical distance into account.

The latter problem could be solved by using a plane (like with calcYCoord) but this is unnecessary. I will use the same system I currently have to calculate dist but will take vertical distance into account to calculate length.

$$\text{new length} = \sqrt{(\text{old length})^2 + (\text{vertical distance})^2}$$

This should (hopefully) work.

This worked on the first try as shown in [video 11](#).

Code: (with the red arrow pointing to the new line)

```

public static double calcXCoord(Camera camera, Pos point, double tanfov) //tanfov is tan((field of view)/2)
{
    if(camera.direction.z == 0) //special case 1
    {
        double dist = point.z - camera.position.z;
        double length = (point.x - camera.position.x)*tanfov;
        return (dist / length);
    }
    else if (camera.direction.x == 0) //special case 2
    {
        double dist = camera.position.x - point.x;
        double length = (point.z - camera.position.z) * tanfov;
        return(dist/length);
    }
    else //no special case
    {
        double p = (point.x+((camera.direction.x*camera.position.z)/camera.direction.z)+((camera.direction.y*camera.position.y)/camera.direction.z));
        double q = ((camera.direction.x * p) / camera.direction.z) + camera.position.x - ((camera.direction.y * p) / camera.direction.z);
        double dist = Math.Sqrt(Math.Pow(p - point.z, 2)+Math.Pow(q - point.x,2));
        double length = Math.Sqrt(Math.Pow(p - camera.position.z, 2) + Math.Pow(q-camera.position.x, 2));
        length = Math.Sqrt(Math.Pow(length, 2) + Math.Pow(camera.position.y - point.y, 2));
        if (leftOrRight(point) == true)
        {
            return (dist / length);
        }
        else
        {
            return (-dist / length);
        }
    }
}

```

The other problem though still persists.

I have an idea of how to fix this. This happens only when only two faces are visible, meaning the problem would be solved if I changed the order in which the faces were rendered. To do this I will render them in order of how close they are to the camera.

I first created some new functions to make the new algorithms easier to write, and make the “closestPoint” algorithm slightly more memory efficient:

```

public static double distBetweenPoints(Pos point1, Pos point2)
{
    return Math.Sqrt(Math.Pow(point1.x - point2.x, 2) + Math.Pow(point1.y - point2.y, 2) + Math.Pow(point1.z - point2.z, 2));
}

0 references
public static Pos averagePoint(params Pos[] points) //returns the average of a set of points (e.g., centre of a face or cube)
{
    double xTotal = 0;
    double yTotal = 0;
    double zTotal = 0;
    int len = points.Length;
    for (int i = 0; i < len; i++)
    {
        xTotal = xTotal + points[i].x;
        yTotal = xTotal + points[i].y;
        zTotal = xTotal + points[i].z;
    }
    return new Pos(xTotal / len, yTotal / len, zTotal / len);
}

0 references
public static Vector vectBetweenPoints(Pos point1, Pos point2) //returns the vector from point1 to point2
{
    double xComp = point2.x - point1.x;
    double yComp = point2.y - point1.y;
    double zComp = point2.z - point1.z;
    return new Vector(xComp, yComp, zComp);
}

```

[Note from future: this code had a typo inside the for loop as mentioned later.]

I then created this next algorithm to render faces in the correct order by first finding the distance of the midpoints of each face to the camera and adding the faces to a list in order based on distance from the camera.

```

void renderFacesInOrder(Face face1, Face face2, Face face3, Camera camera, Mesh mesh)
{
    Pos point1 = Global.averagePoint(face1.vertex1, face1.vertex2, face1.vertex3, face1.vertex4);
    Pos point2 = Global.averagePoint(face2.vertex1, face2.vertex2, face2.vertex3, face2.vertex4);
    Pos point3 = Global.averagePoint(face3.vertex1, face3.vertex2, face3.vertex3, face3.vertex4);
    double dist1 = Global.distBetweenPoints(point1, camera.position);
    double dist2 = Global.distBetweenPoints(point2, camera.position);
    double dist3 = Global.distBetweenPoints(point3, camera.position);
    //distance of each face from the camera

    List<Face> faces = new List<Face>();

    if (dist1 < dist2 && dist1 < dist3) //dist1 is the smallest
    {
        faces.Add(face1);
        if (dist2 < dist3) //dist2 is the next smallest
        {
            faces.Add(face2);
            faces.Add(face3); //Add face2 then face3
        }
        else //dist3 > dist2 so face3 is added, then face2
        {
            faces.Add(face3);
            faces.Add(face2);
        }
    }
    else if (dist2 < dist1 && dist2 < dist3) //dist2 is the smallest
    {
        faces.Add(face2);
        if (dist1 < dist3)
        {
            faces.Add(face1);
            faces.Add(face3);
        }
        else
        {
            faces.Add(face3);
            faces.Add(face1);
        }
    }
    else //dist3 is the smallest
    {
        faces.Add(face3);
        if (dist1 < dist2)
        {
            faces.Add(face1);
            faces.Add(face2);
        }
        else
        {
            faces.Add(face2);
            faces.Add(face1);
        }
    }
    //Render faces in order:
    for (int i = 2; i > -1; i--) //render from furthest to closest
    {
        renderFace(faces[i], mesh);
    }
}

```

I then used a for loop to render the faces in the correct order (descending order as the furthest face should be rendered first):

```

//Render mesh
void renderMesh(Mesh mesh, Camera camera)
{
    int closestPoint = Global.closestPoint(mesh, camera);
    Face topFace = new Face(mesh.point1, mesh.point2, mesh.point3, mesh.point4);
    Face bottomFace = new Face(mesh.point5, mesh.point6, mesh.point7, mesh.point8);
    Face leftFace = new Face(mesh.point1, mesh.point4, mesh.point8, mesh.point5);
    Face rightFace = new Face(mesh.point2, mesh.point3, mesh.point7, mesh.point6);
    Face frontFace = new Face(mesh.point1, mesh.point2, mesh.point6, mesh.point5);
    Face backFace = new Face(mesh.point3, mesh.point4, mesh.point8, mesh.point7);
    switch (closestPoint)
    {
        case 1:
            renderFacesInOrder(topFace, leftFace, frontFace, camera, mesh);
            break;
        case 2:
            renderFacesInOrder(topFace, rightFace, frontFace, camera, mesh);
            break;
        case 3:
            renderFacesInOrder(topFace, rightFace, backFace, camera, mesh);
            break;
        case 4:
            renderFacesInOrder(topFace, leftFace, backFace, camera, mesh);
            break;
        case 5:
            renderFacesInOrder(bottomFace, leftFace, frontFace, camera, mesh);
            break;
        case 6:
            renderFacesInOrder(bottomFace, rightFace, frontFace, camera, mesh);
            break;
        case 7:
            renderFacesInOrder(bottomFace, rightFace, backFace, camera, mesh);
            break;
        case 8:
            renderFacesInOrder(bottomFace, leftFace, backFace, camera, mesh);
            break;
    }
}

```

The result can be seen in [Video 12](#). The left and right sides not are no longer visible through the front side.

The next thing I will do is allow multiple objects to be rendered at once. This will require me to use the “closestPoint” algorithm to decide which object to render first.

Sidenote: at this stage, I changed the entities ArrayList from the beginning to a regular list.

I created an instance of the entity class using the cube mesh as its mesh and using the origin as its centre of mass (as centre of mass is not important right now). I then added this to the “Entities” list. I then created another procedure “renderAll” which orders the list of entities according to the distance of the closest point of the mesh to the camera and renders them in this order (furthest from the camera first, closest last).

This is the code for that:

```

//RENDER ALL:
void renderAll(List<Entity> entities, Camera camera)
{
    entities = entities.OrderByDescending(x => Global.distBetweenPoints(Global.closestPointCoords(x.mesh, camera), camera.position)).ToList();
    for (int i = 0; i < entities.Count; i++)
    {
        renderMesh(entities[i].mesh, camera);
    }
}

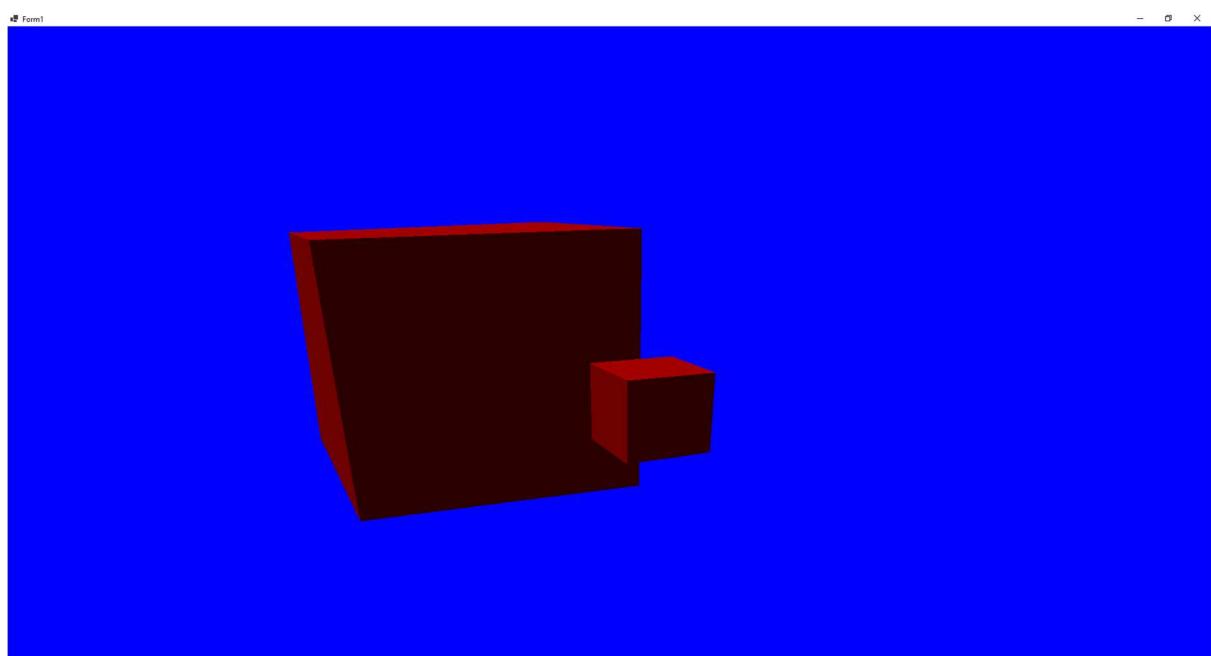
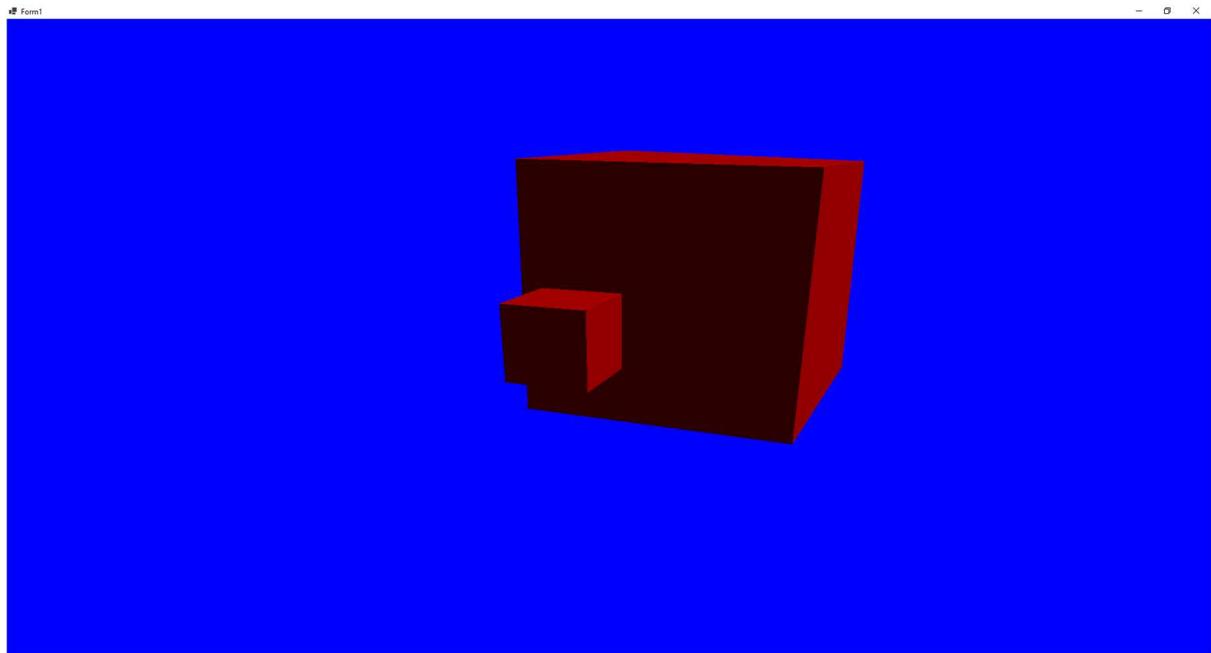
void gameLoop()
{
    Global.cameraControl(0.5, 0.1);
    renderAll(Global.Entities, Global.Camera);
}

gameLoop();
Invalidate();

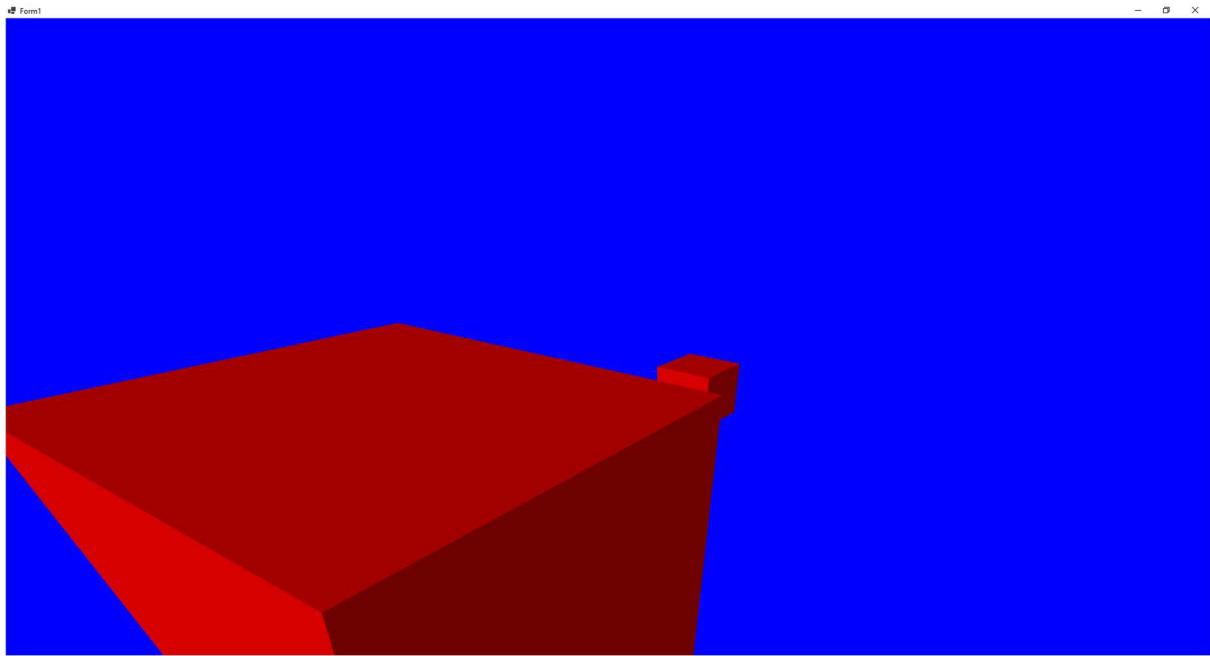
```

These are some screenshots of the result:

(I made one cube larger than the other to demonstrate the distinction between them).



Here is an example of the larger cube in front of the smaller cube:



The program is now, however, incredibly poor in performance with a lot of stuttering. I found out that the reason for this is that it runs through the for loop from the previous algorithm many times instead of 2. This is because each time the screen is refreshed, the code to add a new cube to the list is run again, this has the effect of adding hundreds of the same cube to the list and then trying to render them all which leads to an extreme drop in performance.

To fix this I will simply move the code which defines the cubes and adds them to the list outside of the paint event (which is called each time the screen is refreshed) into the Form\_Load event (which is called when the program is first loaded).

```
private void Form1_Load(object sender, EventArgs e)
{
    //Define cube 1
    Pos cubeVertex1 = new Pos(0, -1, 0);
    Pos cubeVertex2 = new Pos(0, -1, 1);
    Pos cubeVertex3 = new Pos(1, -1, 1);
    Pos cubeVertex4 = new Pos(1, -1, 0);
    Pos cubeVertex5 = new Pos(0, 0, 0);
    Pos cubeVertex6 = new Pos(0, 0, 1);
    Pos cubeVertex7 = new Pos(1, 0, 1);
    Pos cubeVertex8 = new Pos(1, 0, 0);

    Mesh cube1Mesh = new Mesh(cubeVertex1, cubeVertex2, cubeVertex3, cubeVertex4, cubeVertex5, cubeVertex6, cubeVertex7, cubeVertex8);
    Entity cube1 = new Entity(cube1Mesh, new Pos(0, 0, 0)); //centre of mass is not important for now, so I will leave it at the origin

    //Define cube 2
    cubeVertex1 = new Pos(5, -2.5, -2.5);
    cubeVertex2 = new Pos(5, -2.5, 2.5);
    cubeVertex3 = new Pos(10, -2.5, 2.5);
    cubeVertex4 = new Pos(10, -2.5, -2.5);
    cubeVertex5 = new Pos(5, 2.5, -2.5);
    cubeVertex6 = new Pos(5, 2.5, 2.5);
    cubeVertex7 = new Pos(10, 2.5, 2.5);
    cubeVertex8 = new Pos(10, 2.5, -2.5);

    Mesh cube2Mesh = new Mesh(cubeVertex1, cubeVertex2, cubeVertex3, cubeVertex4, cubeVertex5, cubeVertex6, cubeVertex7, cubeVertex8);
    Entity cube2 = new Entity(cube2Mesh, new Pos(0, 0, 0));

    Global.Entities.Add(cube1); //Whenever a new physics object is created add it to this list
    Global.Entities.Add(cube2);
}
```

Now the program runs smoothly. [Video13](#).

There is one more huge issue, which was not addressed in any of the videos so far: when the camera rotates over the z axis, the camera's rotation is reversed (the up and down keys

for rotating the camera up and down are switched) and the cube appears near the bottom of the screen instead the top and vice versa. [Video14](#).

I started by adding this to the beginning of the rotatePitch function so that it now the up and down keys work correctly when the camera crosses the z-axis.

```
public static Vector rotatePitch(double angle, Vector vect) //Rotates a vector up by some angle
{
    if (camera.direction.z > 0)
    {
        angle = -angle;
    }
}
```

I then added this to the cameraControl function to flip the camera's rotation when it crosses the z-axis.

```
if (leftHeld)
{
    Vector dummyVector = rotateYaw(angle, camera.direction); //preview" the vector before the rotation takes place
    if (Math.Sign(dummyVector.z) == Math.Sign(camera.direction.z))
    { //The vector is on the same size of the x axis before and after, rotate the same as usual
        camera.direction = rotateYaw(angle, camera.direction); //rotate yaw the same as usual
    }
    else
    { //For some reason (I don't know why) the vertical angle of the camera's direction seems to change sign when the z-component of the camera's direction switches
        camera.direction = rotateYaw(angle, camera.direction);
        camera.direction.y = -camera.direction.y;
    }
}
if (rightHeld)
{
    Vector dummyVector = rotateYaw(-angle, camera.direction);
    if (Math.Sign(dummyVector.z) == Math.Sign(camera.direction.z))
    {
        camera.direction = rotateYaw(-angle, camera.direction);
    }
    else
    {
        camera.direction = rotateYaw(-angle, camera.direction);
        camera.direction.y = -camera.direction.y;
    }
}
```

This fixed the problem, as shown in [Video 15](#).

Whilst reviewing the videos I noticed that there were still occurrences of faces being visible through faces which should be in front of them. This was simply due to a typo which has now been corrected.

There is still another issue I need to resolve, that is that the culling algorithm I implemented earlier has ceased working since the implementation of meshes (the cubes). I hadn't noticed this as I hadn't tried turning the camera around until now (since I thought it already worked as intended).

I believe the equation of the plane used is incorrect. The equation of the plane

perpendicular to the vector  $\begin{pmatrix} vect.x \\ vect.y \\ vect.z \end{pmatrix}$  is:

$$vect.x \times x + vect.y \times y + vect.z \times z = d$$

Passing through  $(pos.x, pos.y, pos.z)$ :

$$d = vect.x \times pos.x + vect.y \times pos.y + vect.z \times pos.z$$

Put this back in to the original:

$$\begin{aligned}
& vect.x \times x + vect.y \times y + vect.z \times z \\
& = vect.x \times pos.x + vect.y \times pos.y + vect.z \times pos.z \\
y(vect.y) & = (vect.x)(pos.x - x) + vect.y \times pos.y + vect.z(pos.z - z) \\
y & = \frac{(vect.x)(pos.x - x) + vect.y \times pos.y + vect.z(pos.z - z)}{vect.y} \\
y & = \frac{(vect.x)(pos.x - x) + vect.z(pos.z - z)}{vect.y} + pos.y
\end{aligned}$$

This new equation made no difference. I do, however, have another idea as to how this problem could be solved. Instead of concerning myself with the equation of the plane itself, I may consider only the normal vector (*camera.direction*) and the point it passes through (*camera.position*). For some point  $P$ ,  $P - camera.position$  is the vector going from the camera to the point. If the angle between this vector and the camera's direction is  $< 90^\circ$  then the point is "in front" of the camera so should be rendered. Using the dot product:

$$\begin{aligned}
& (camera.direction) \cdot (P - camera.position) \\
& = |camera.direction| |P - camera.position| \cos \theta \\
& = distBetweenPoints(P, camera.position) \times \cos \theta
\end{aligned}$$

(*camera.direction* has a magnitude of 1.)

$$\theta = \cos^{-1} \left( \frac{(camera.direction) \cdot (P - camera.position)}{distBetweenPoints(P, camera.position)} \right)$$

This angle should be checked against  $\frac{\pi}{2}$ .

After implementing this, it mostly works, the cubes no longer render when behind the camera, but occasionally they still disappear briefly. After debugging, using `Debug.WriteLine()`, I found that the reason for this wasn't that the angle was actually  $> \frac{\pi}{2}$  but rather was NaN (undefined) so wasn't considered to be  $\leq \frac{\pi}{2}$ . To fix this I changed the if statement to check that  $\theta \geq \frac{\pi}{2}$  and to return false if yes, and return true if no. This seems to have fixed the problem. One small optimization, if  $\theta \geq \frac{\pi}{2}$  then  $\cos \theta \leq 0$  so use that condition instead, as then  $\arccos(\cos^{-1})$  does not need to be calculated.

```
//Culling
1 reference
public static bool checkRender(Camera camera, Pos point, double vertFov)
{
    Vector dummyCameraDir;
    if (camera.direction.z < 0)
    {
        dummyCameraDir = new Vector(camera.direction.x, -camera.direction.y, camera.direction.z);
    }
    else
    {
        dummyCameraDir = camera.direction;
    }
    double cosTheta = dotProduct(dummyCameraDir, vectBetweenPoints(camera.position, point)) / distBetweenPoints(camera.position, point);
    if (cosTheta <= 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

This function finds  $\cos \theta$  and then checks it using the aforementioned inequality  $\cos \theta \leq 0$ .

The reason for the dummyCameraDir variable is to correct for the reversing of the  $y$  component of the camera when crossing the  $z$ -axis. This is shown working in [Video 16](#). This concludes the rendering and the development and implementation of prototype 1. Before moving on, I will revisit the success criteria.

## Testing Prototype 1

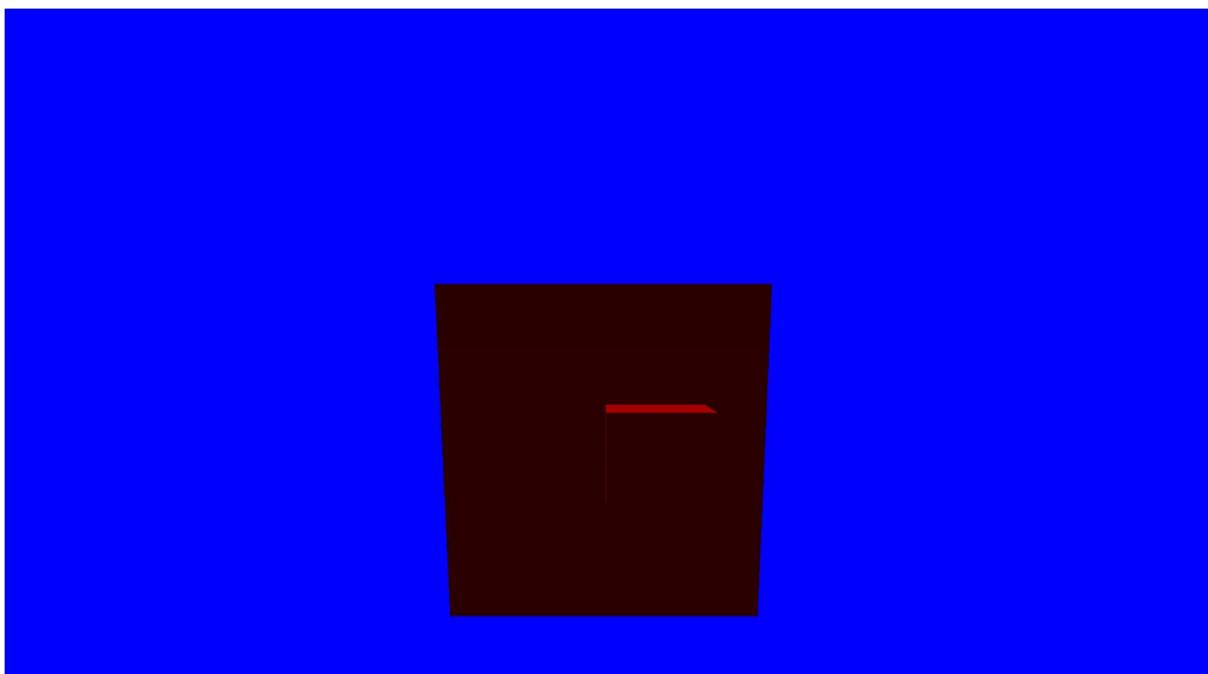
The Final result of this prototype should be a pair of cubes which render in the correct order, which can be viewed by a camera which the user can move using keyboard keys. I have tested each feature individually immediately after development, but this section is to demonstrate that everything implemented in this prototype still works.

Test Table:

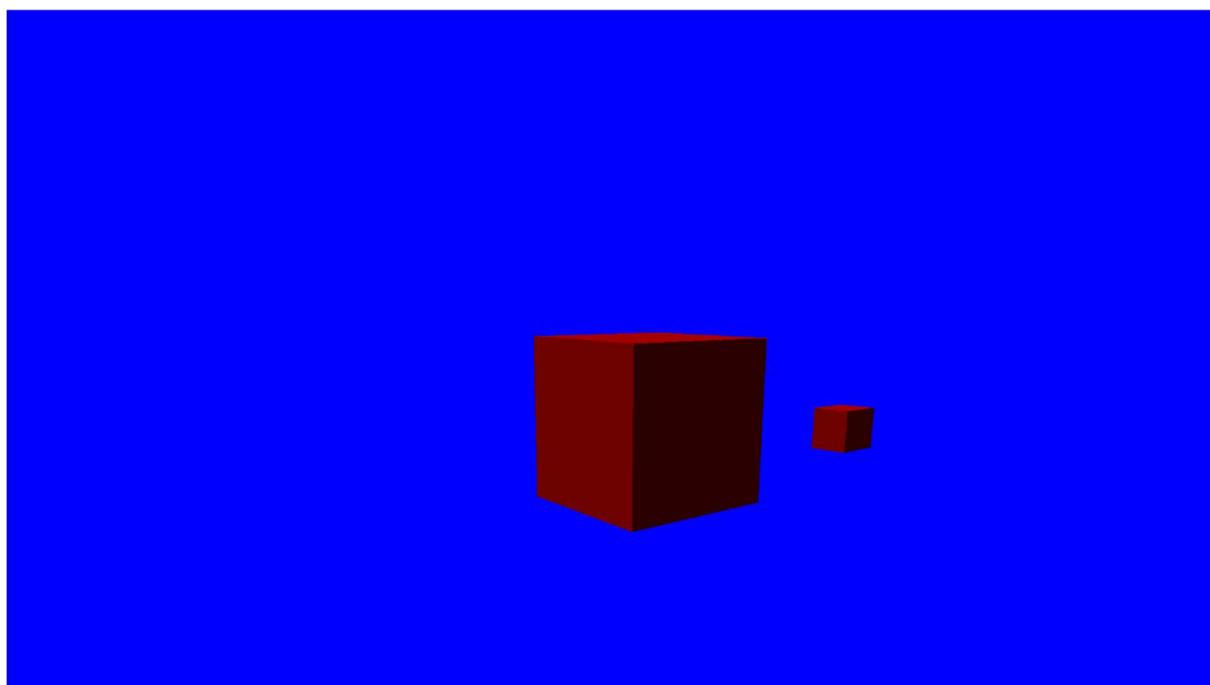
Test No.	Input/Data	Expected Output/Result	Actual Output/Result	Success?
1	W key held	Camera should move forward.	Camera moves forward.	Yes
2	A key held	Camera should move left.	Camera moves left.	Yes
3	S key held	Camera should move backward.	Camera moves backward.	Yes
4	D key held	Camera should move right.	Camera moves right.	Yes
5	Up arrow key held	Camera should tilt up.	Camera tilts up.	Yes
6	Left arrow key held	Camera should rotate to the left.	Camera rotates left.	Yes
7	Down arrow key held	Camera should tilt down.	Camera tilts down.	Yes
8	Right arrow key held	Camera should rotate to the right.	Camera rotates right.	Yes

### Evidence for each test:

Before Movement:

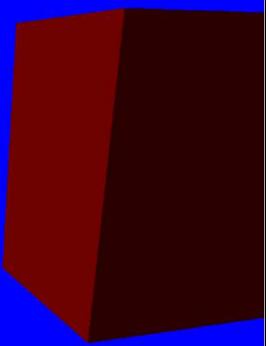


Before Rotation:

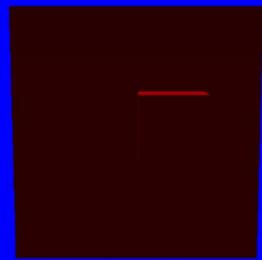


Test - No.	Evidence
1	A 3D rendering of two cubes against a solid blue background. The larger cube from the previous image has been rotated 90 degrees clockwise, so its red front face is now vertical and oriented downwards. The smaller cube has also been rotated 90 degrees clockwise, with its red front face now vertical and oriented upwards. They are both positioned to the right of the original cube's location.

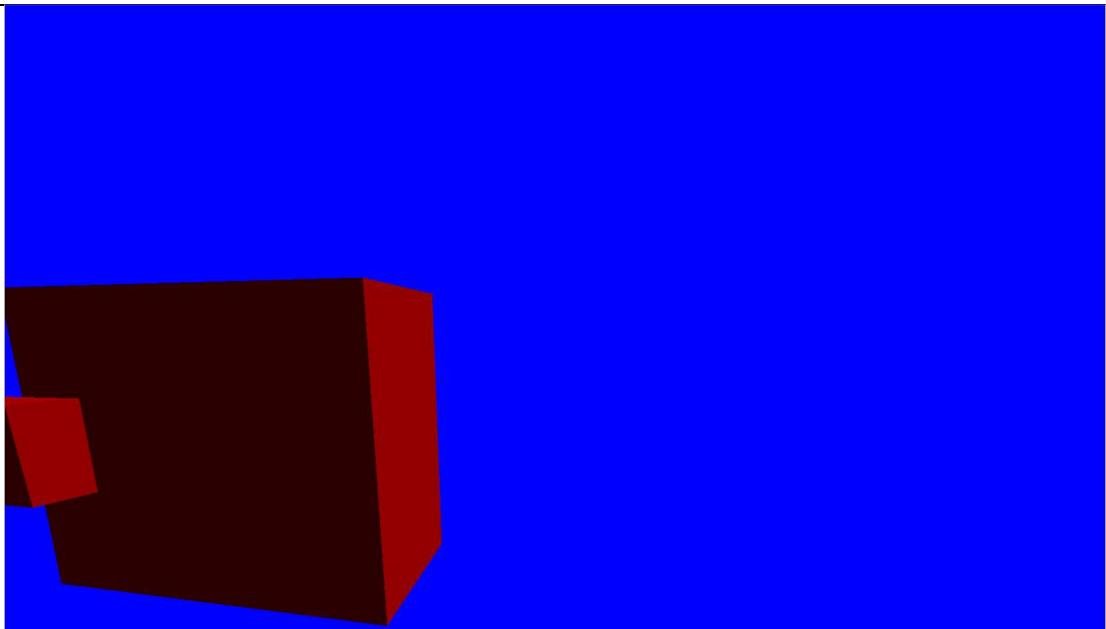
2



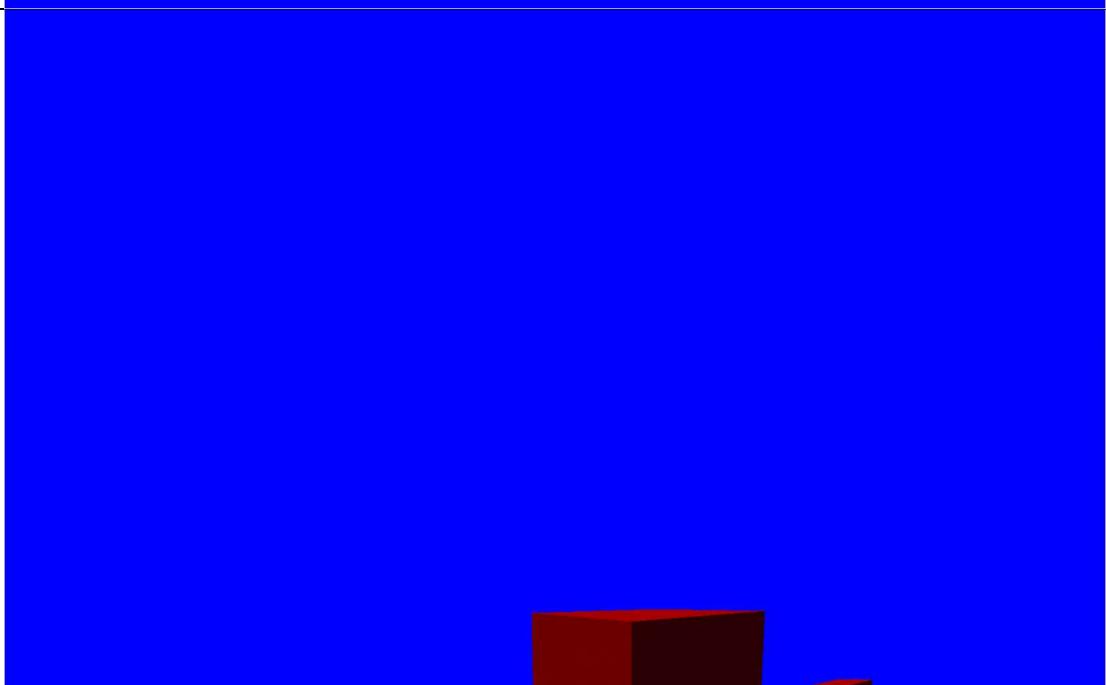
3



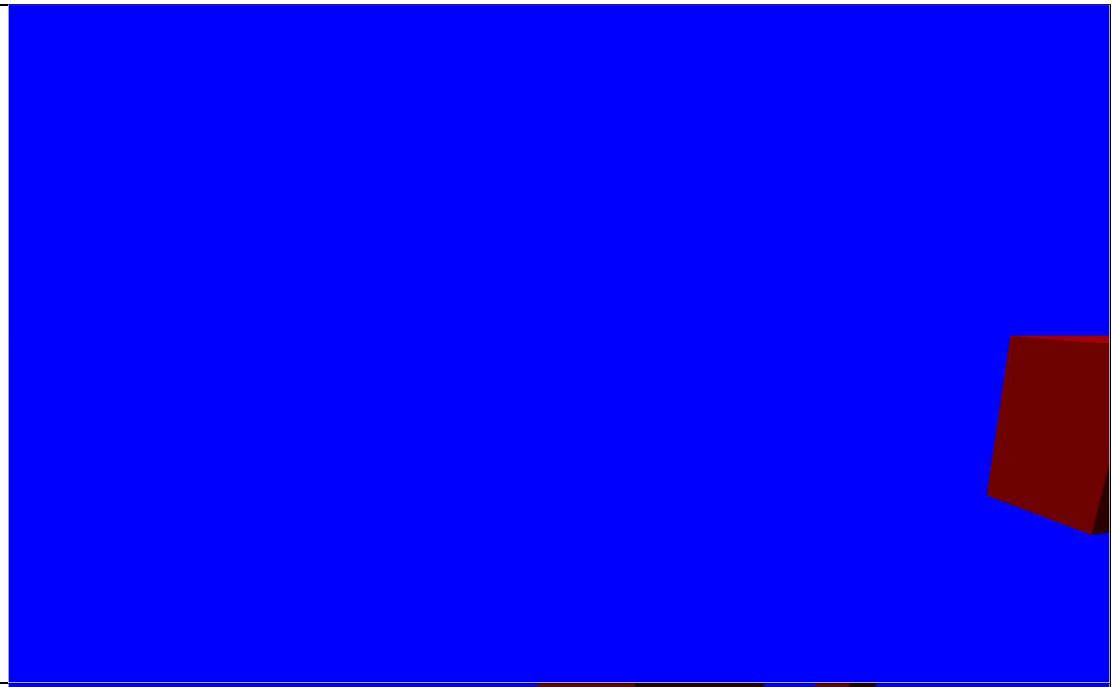
4



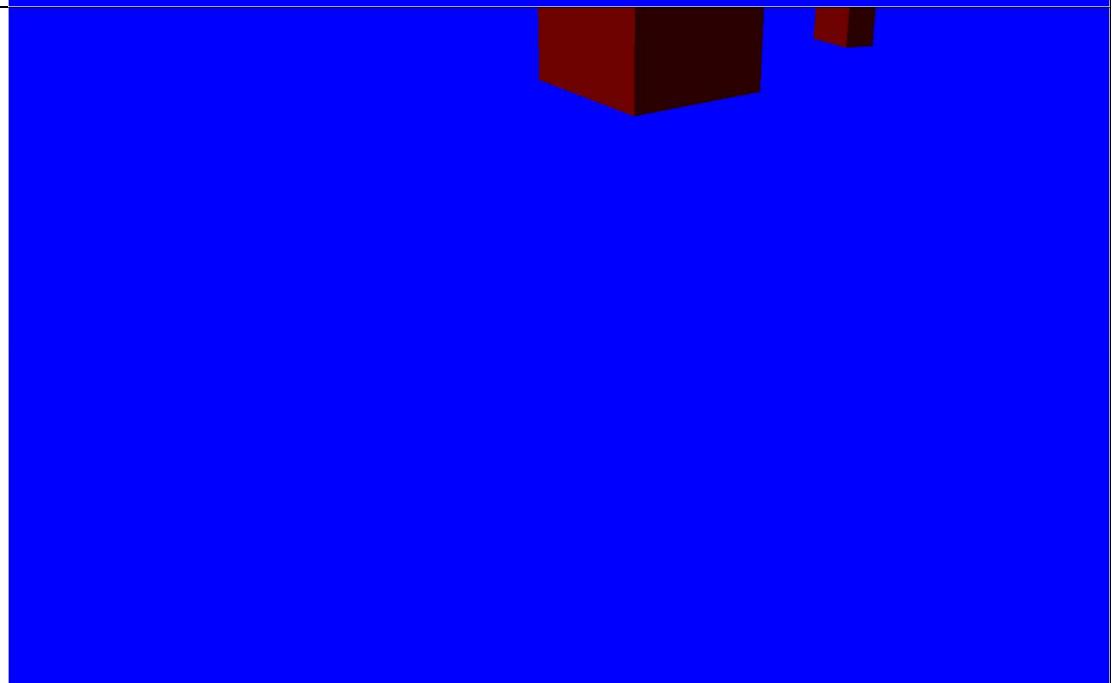
5



6



7



8



Each of these tests specifically addresses success criteria number 11, but 5 and 6 are also self-evident here.

## Success Criteria Review

Main window which is moveable and resizable. ✓

Menu screen. X

Lightweight and easy to navigate [subjective, judge at the end]. X

Multiple different options of different simulations. X

3D rasteriser. ✓

Shading (giving different faces of an object different levels of brightness based on how they are oriented). ✓

Working rigid body motion. X

Working rigid body collision. X

Normal reaction force. X

Options next to each simulation such as creating objects, applying forces to objects etc.,. X

Camera control. ✓

4/11 success criteria complete so far. I will now move on to the physics part of development with prototype 2.

## Prototype 2 Design

Each instance of the entity class has a mesh attribute which handles the object's rendering. Each entity also has a rigid body attribute which will handle the physics which will be implemented in this section.

This section will aim to design algorithms for the following:

- Dynamics (objects moving).
- Rotation of objects from applied forces.
- Collision detection (detecting when two objects are intersecting).
- Collision response (pushing the objects further apart once they have been found to intersect).

### Dynamics (objects moving)

Objects need to be able to move. This will be based upon the objects centre of mass which will be able to be manipulated. This could be found by calculating the average position of the corners of the cube but since the centre of mass is the thing which needs to be directly manipulated, I will simply set it as some position and will calculate the coordinate of the corners for rendering separately. The positions of these vertices will depend on the coordinates of the centre of mass, the shape of the object, and how the object is rotated. I will design rotation later. For now, I will just focus on how to manipulate the centre of mass i.e., change the position of the object. The object has a position, that is the centre of mass, a velocity (which will be a vector), an acceleration (which will also be a vector), a mass and a resultant force (which is the sum of all forces acting on the object). Assuming the program runs at 60 frames per second, the time between each time when these attributes are updated is  $\frac{1}{60}$  seconds, this number can of course be changed for any framerate, and I may even be able to make it dynamic (allow it to change if the framerate changes as the program is being run). Using  $v = \frac{s}{t}$  ( $velocity = \frac{displacement}{time}$ ), we may find the displacement based on only the objects current velocity.

$$s = vt$$

where  $v$  is the velocity of the object, (in arbitrary units per second) and  $t = \frac{1}{60}$  seconds. This displacement can be added to the objects current position to calculate the objects new position which can then be set. The velocity of the object on the next frame can be determined by its current velocity and its acceleration.  $a = \frac{v}{t}$  or  $acceleration = \frac{velocity}{time}$  so  $v = at$  where  $t = \frac{1}{60}$  and  $a$  is the objects acceleration.

Finally, determining the objects acceleration will be done by using  $F = ma \rightarrow a = \frac{F}{m}$  where  $a$  is the object's acceleration to be determined,  $F$  is the object's resultant force and  $m$  is the object's mass. Once each of these attributes and appropriate algorithms have been implemented, an object will be able to realistically move by simply specifying its mass and any forces acting upon it.

I will test this implementation by applying a constant downward force of gravity and then seeing what happens when I apply a force to push the object. The object should move with a parabolic trajectory (the path an object follows when you throw it).

For the following code, position (which is the centre of mass), velocity, acceleration, mass and resultant force are all attributes of the rigidBody class and “RB” is an instance of this class. vectorXScalar is a function which I will create to multiply a vector by a scalar to make code easier to write. Similarly, vectorAddVector adds two vectors together. fps is the frames per second.

#### Python code algorithm:

```
def updateAcceleration(RB):
    RB.acceleration.x = vectorXScalar(RB.force.x, 1/RB.mass)

def updateVelocity(RB):
    RB.velocity = vectorAddVector(RB.velocity, vectorXScalar(RB.acceleration, 1/fps))

def updatePosition(RB):
    RB.position = vectorAddVector(RB.position, vectorXScalar(RB.velocity, 1/fps))
```

Each of these will be run once each frame.

#### Object rotation

Objects need a way to be rotated. First, I will need a way to describe a rotated object and then a way for its corners to be found so it can be rendered. To describe a rotated object, I will need its centre of mass, and three angles to describe how it is rotated about each axis. I could use other methods, for example a vector to keep track of how an object is rotated but using angles makes the physics much easier to implement later as this approach lends itself to the circular motion formulae.

The next concern will be working out how to find the coordinates of the corners given the centre of mass, and the angles of rotation. I will do this one corner at a time.

To find the new coordinates of a given corner, I will subtract the centre of mass from its current position so that it is now centred at the origin and can be rotated using the rotation matrices. I will call the three angles of rotation  $\theta_x$  (about  $x$  axis),  $\theta_y$  (about the  $y$  axis) and  $\theta_z$  (about the  $z$  axis). These angles represent the change in angle since the previous frame, i.e., how much further the object needs to be rotated during this frame. The order in which 3D rotations are performed does matter in general, this is negligible when rotating by small amounts. Since the change in angles between each frame will be very small, this is fine.

Since these angles represent the change in angle per frame, they can be interpreted as the angular velocity of the object's rotation divided by the number of frames per second.

I will rotate the point (now centred at the origin) using the following matrices.

$$\text{about the } x \text{ axis} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x \\ 0 & \sin \theta_x & \cos \theta_x \end{pmatrix}$$

$$\text{about the } y \text{ axis} \begin{pmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{pmatrix}$$

$$\text{about the } z \text{ axis} \begin{pmatrix} \cos \theta_z & -\sin \theta_z & 0 \\ \sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

These are assuming that the  $y$  axis is facing the other direction meaning that all rotations will be in the opposite direction as expected (e.g., clockwise instead of anticlockwise). To fix this I will replace  $\theta$  by  $-\theta$  giving.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & \sin \theta_x \\ 0 & -\sin \theta_x & \cos \theta_x \end{pmatrix}, \begin{pmatrix} \cos \theta_y & 0 & -\sin \theta_y \\ 0 & 1 & 0 \\ \sin \theta_y & 0 & \cos \theta_y \end{pmatrix}, \begin{pmatrix} \cos \theta_z & \sin \theta_z & 0 \\ -\sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Finally, I will combine these rotation transformations into one by multiplying the matrices together.

$$\begin{aligned} & \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & \sin \theta_x \\ 0 & -\sin \theta_x & \cos \theta_x \end{pmatrix} \begin{pmatrix} \cos \theta_y & 0 & -\sin \theta_y \\ 0 & 1 & 0 \\ \sin \theta_y & 0 & \cos \theta_y \end{pmatrix} \begin{pmatrix} \cos \theta_z & \sin \theta_z & 0 \\ -\sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & \sin \theta_x \\ 0 & -\sin \theta_x & \cos \theta_x \end{pmatrix} \begin{pmatrix} \cos \theta_y \cos \theta_z & \cos \theta_y \sin \theta_z & -\sin \theta_y \\ -\sin \theta_z & \cos \theta_z & 0 \\ \sin \theta_y \cos \theta_z & \sin \theta_y \sin \theta_z & \cos \theta_y \end{pmatrix} \\ &= \begin{pmatrix} \cos \theta_y \cos \theta_z & \cos \theta_y \sin \theta_z & -\sin \theta_y \\ -\cos \theta_x \sin \theta_z + \sin \theta_x \sin \theta_y \cos \theta_z & \cos \theta_x \cos \theta_z + \sin \theta_x \sin \theta_y \sin \theta_z & \sin \theta_x \cos \theta_y \\ \sin \theta_x \sin \theta_z + \cos \theta_x \sin \theta_y \cos \theta_z & -\sin \theta_x \cos \theta_z + \cos \theta_x \sin \theta_y \sin \theta_z & \cos \theta_x \cos \theta_y \end{pmatrix} \end{aligned}$$

Applying this transformation to a point with coordinates  $(x, y, z)$  gives the result:

$$\begin{aligned} & \begin{pmatrix} \cos \theta_y \cos \theta_z & \cos \theta_y \sin \theta_z & -\sin \theta_y \\ -\cos \theta_x \sin \theta_z + \sin \theta_x \sin \theta_y \cos \theta_z & \cos \theta_x \cos \theta_z + \sin \theta_x \sin \theta_y \sin \theta_z & \sin \theta_x \cos \theta_y \\ \sin \theta_x \sin \theta_z + \cos \theta_x \sin \theta_y \cos \theta_z & -\sin \theta_x \cos \theta_z + \cos \theta_x \sin \theta_y \sin \theta_z & \cos \theta_x \cos \theta_y \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\ &= \begin{pmatrix} x \cos \theta_y \cos \theta_z + y \cos \theta_y \sin \theta_z - z \sin \theta_y \\ x(\sin \theta_x \sin \theta_y \cos \theta_z - \cos \theta_x \sin \theta_z) + y(\sin \theta_x \sin \theta_y \sin \theta_z + \cos \theta_x \cos \theta_z) + z \sin \theta_x \cos \theta_y \\ x(\sin \theta_x \sin \theta_z + \cos \theta_x \sin \theta_y \cos \theta_z) + y(\cos \theta_x \sin \theta_y \sin \theta_z - \sin \theta_x \cos \theta_z) + z \cos \theta_x \cos \theta_y \end{pmatrix} \end{aligned}$$

Which are the coordinates of the point after being rotated.

Finally, I will add the centre of mass (coordinates) to these coordinates to translate this point to where it should be. This will then be the final coordinates of the point for that frame, ready to be put into the Mesh attribute to be rendered.

Python code algorithm:

```
def rotateObject(object, aX, aY, aZ): #Takes the entity and the three angles of rotation.  
    for i in range(vertices): #for each vertex of the object.  
        #Point is recentred at the origin.  
        object.Mesh.Points[i] = object.Mesh.Points[i] – object.RigidBody.CentreOfMass  
  
    #I reference the object (instance of entity class) as this contains the mesh which contains the vertices.  
  
    #I will create a function to subtract positions from each other to make the above easier.  
  
    x = object.Mesh.Points[i].x  
    y = object.Mesh.Points[i].y  
    z = object.Mesh.Points[i].z #To make the next lines easier to write  
  
    #Point is rotated about the origin  
    object.mesh.Points[i].x = x*cos(aY)*cos(aZ) + y*cos(aY)*sin(aZ) - z*sin(aY)  
    object.mesh.Points[i].y = x*(sin(aX)*sin(aY)*cos(aZ) - cos(aX)*sin(aZ)) +  
    y*(sin(aX)*sin(aY)*sin(aZ) + cos(aX)*cos(aZ)) +  
    z*(sin(aX)*cos(aY))  
    object.mesh.Points[i].z = x*(sin(aX)*sin(aZ) + cos(aX)*sin(aY)*cos(aZ)) +  
    y*(cos(aX)*sin(aY)*sin(aZ) - sin(aX)*cos(aZ)) +  
    z*(cos(aX)*cos(aY))  
  
    #Centre of mass is added back on  
    object.mesh.Points[i] = object.mesh.point[i] + object.rigidBody.centreOfMass
```

## Finding angular velocity from applied forces

So far, I have only considered forces which act on (or in the line of) the centre of mass of the object. Such forces cannot cause the object to rotate. I will next consider forces acting from some general point in space and see how this force would affect the object in terms of its position and its rotation.

For this, imagine a rigid (non-bending) lightweight (no mass) rod from the centre of mass of the object to the point where the force is being applied. How would this force affect the position and rotation of the object. If the force acts parallel to the rod (directly towards and through the centre of mass) then the object will move but not rotate. If the force acts perpendicular to the rod, the object will rotate about its centre of mass, but the centre of mass will not move. If the force is neither perfectly in line nor perfectly perpendicular to the rod, then the force can be decomposed into two vectors which are. These vectors can then be separately used to handle movement and rotation. Note that if the force is perpendicular to the rod, the component which is parallel to it and vice-versa meaning that this general case does also apply to the two special cases.

The parallel force will handle object movement using the algorithm described earlier in dynamics. The perpendicular force could be used to handle rotation. Torque  $\tau$  is the distance from the centre of mass multiplied by the magnitude of the component of the force which is perpendicular to this distance. If the force is not perpendicular, the magnitude of the perpendicular force =  $|F| \sin \theta$  where  $\theta$  is the angle between the force and the measured distance. Using this,  $\tau = |F||r| \sin \theta$  where  $r$  is the vector going from the centre of mass to the point where the force is acting. This is very similar to the cross product, for which an algorithm has been created.  $r \times F = |F||r| \sin \theta \hat{n}$  where  $\hat{n}$  is the perpendicular unit force to  $F$  and  $r$ . Using the right-hand rule for torque, the torque acts in the direction in which your fingers curl when you point your thumb in the direction of the vector  $\tau$  (although in this case it is the left-hand rule due to unconventional axes, but this makes no difference as the same is true for the cross product). I will hence define torque by the following formula:

$$\tau = r \times F$$

Torque  $\tau$  is linked to angular acceleration  $\alpha$  by the following formula:

$$\tau = md^2\alpha$$

where  $d = |r|$  and is the distance from the centre of mass to the point where the force is acting.

The angular velocity will be increase by  $\frac{\alpha}{\text{frames per second}}$  each frame and the angular velocity will rotate the object as previously outlined.

Torque of the original force can be found by using  $\tau = r \times F$  where  $F$  is either the original force or the perpendicular force, it makes no difference since these two vectors have the same torque, I will use the original force to save me from needing to calculate the perpendicular one. I still need to find the parallel component of the original force to deal with motion. The parallel component should be facing in the direction of the vector going

from the point on which the force is acting to the centre of mass. This requires a process known as vector projection which finds what component of one vector acts in the direction of a second. I will project the resultant force  $\mathbf{F}$  in the direction of the parallel vector  $\mathbf{v}$ . I will call the component of  $\mathbf{F}$  in the direction of  $\mathbf{v}$   $\mathbf{F}_{para}$ . This is done using the formula:

$$\mathbf{F}_{para} = \frac{(\mathbf{F} \cdot \mathbf{v})\mathbf{v}}{|\mathbf{v}|^2}$$

Once this has been done for all forces acting on an object, I can add all each of the parallel forces together to get a resultant force acting on the centre of mass (for motion) and I can add the angular velocities together to get a resultant angular velocity,  $\alpha$  (for rotation).

#### Python code algorithm:

```
#takes the rigidBody, the force and the point where the force is acting as parameters.
#Returns the angular acceleration caused by this force.

def getAngularAcceleration (rigidBody, force):
    rodVector = vectorBetweenPoints(rigidBody.centreOfMass, force.position)

    #Finds the vector from centre of mass to point where force is applied.

    torque = crossProduct(rodVector, force.vector)

    return (torque/(rigidBody.mass*(vectorMag(rodVector)**2)) #I will need to create a
function to multiply/divide a vector by a scalar.
```

A function will add the results of the above function for every force acting on an object to return the resultant angular acceleration.

#### Python code algorithm:

```
def updateAngularVelocity(rigidBody):
    rigidBody.angularVelocity = rigidBody.angularVelocity + angularAcceleration/fps
```

This procedure will update the angular velocity vector each frame. Its three components will be passed into the rotateObject procedure which will actually rotate the object each frame.

### Python code algorithm

```
def projectVector(v1, v2): #Projects v1 in the direction of v2.  
    return(VectorTimesScalar(v2,dotProduct(v1, v2))/(vectorMag(v2))**2)
```

The rigid body will class have a resultant force attribute which will be the sum of all forces after being projected by the above function. It will also have an angular acceleration attribute which will be the sum of all angular accelerations. To keep track of each force individually, the rigid body will have an attribute, “forces” which will be an array containing each force. Finally, force will be made into its own class with a vector and position attribute.

### Python code algorithm:

```
def forceActingToCentre(rigidBody, force): #finds the component of the force acting towards  
    #the centre of the object.  
  
    rodVector = vectorBetweenPoints(rigidBody.centreOfMass, force.position)  
  
    return(projectVector(force.vector, rodVector))
```

## Collision Detection

Detecting collisions requires a lot of calculations as collision needs to be checked for between every possible pair of rigid bodies. To simplify the problem A simpler less precise algorithm can be used to determine if two objects are close enough that there is a possibility of intersection. To do this I will use a bounding sphere. The centre of this sphere will be the objects centre of mass, and the radius will be the distance of the furthest vertex to the centre of mass (since all objects in this program are regular polyhedra or cuboids this distance is the same for each vertex, so it does not matter which is used). If the sum of two objects’ radii is greater than the distance between the centres of mass, then the bounding spheres are intersecting meaning the rigid bodies may or may not be colliding, a more precise check would be used at this point. If the sum of the radii is less than the distance between the centres of mass, then the bounding spheres are not intersecting so the rigid bodies are not intersecting, and no further checks need to be done.

### Python code algorithm:

```
def boundingSpheresCheck(object1, object2): #Returns true if there is a collision, false if not.  
  
radius1 = distanceBetweenPoints(object1.RigidBody.CentreOfMass, object1.Mesh.points[0])  
  
radius2 = distanceBetweenPoints(object2.RigidBody.CentreOfMass, object2.Mesh.points[0])  
  
distance = distance BetweenPoints(object1.RigidBody.CentreOfMass,  
object2.RigidBody.CentreOfMass)
```

```

if radius1 + radius2 < distance:
    return(false)

else: #If they are equal then the spheres are touching, I will count this as a collision.

return(true)

```

I next need to create an algorithm for the more precise case to check whether the objects intersect after having already passed the previous test.

To check if object 1 and object 2 are intersecting, I will check each vertex of object 1 against the faces of object 2 and each vertex of object 2 against the faces of object 1. I will use the dot product and the normal to the face to determine whether a given point is on the correct side of that face. I already have an algorithm to find the normal to the face sticking outwards (from design section 1 when determining the colour of a face) so I will reuse that. I will find the angle between the normal and the vector going from a point the face (I can just use one of the vertices) to the given point using the dot product, if  $0^\circ \leq \theta < 90^\circ$  then the object is on the outside. If  $90^\circ \leq \theta \leq 180^\circ$  then the object is on the correct side of the face. If the point passes this check for every face, then it is inside, if it fails just one, then the algorithm can stop as it is outside. This check must be done for each vertex of the first object against each face of the second and then for each vertex of the second against each face of the first. If just one check for a vertex intersection passes, then there is an intersection, and the algorithm can stop.

#### Python code algorithm:

```

def checkPointAgainstFace(point, face):
    pointVect = vectorBetweenPoints(face.vertices[0], point)
    angle = angleBetweenVectors(pointVect, face.normal)
    if (0<=angle<pi/2):
        return(false)
    else:
        return(true)

def checkPointAgainstMesh(point, mesh):
    for i in range(len(mesh.faces)):
        if checkPointAgainstFace(point, mesh.faces[i]) == false:
            return(false) #If the point is on the outside of any face, the point is on the outside.
    return(true) #If the function has not returned false by this time, it is on the inside.

```

```

def checkMeshAgainstMesh(mesh1, mesh2):
    for i in range(len(mesh1.points)):
        if checkPointAgainstMesh(mesh1.points[i], mesh2) == true:
            return([true, mesh1.points[i]], mesh1, mesh2) #if one vertex intersects, then there is a collision. I am returning an array so I can also obtain the vertex which caused the collision.
    return([false])

def checkCollisionPrecise(mesh1, mesh2):
    check1 = checkMeshAgainstMesh(mesh1, mesh2)
    if check1[0] == true:
        return([true, check1[1], check1[2], check1[3]])
    check2 = checkMeshAgainstMesh(mesh2, mesh1)
    if check2[0] == true:
        return([true, check2[1], check2[2], check2[3]]) #If there is a collision I need to also return the vertex which caused the collision (check2[1]) as this can be used for the position where the collision occurred.
    else:
        return([false])

```

I now need to combine the precise and less precise algorithms together in order to have a complete collision detection system.

#### Python code algorithm:

```

def checkCollision(object1, object2): #Checks if two objects are colliding.

    if boundingSpheresCheck(object1, object2) == true
        preciseCheck = checkCollisionPrecise(object1.mesh, object2.mesh)
        if preciseCheck[0] == true:
            return([true, preciseCheck[1], preciseCheck[2], preciseCheck[3]])
        else:
            return([false])

```

```
def checkAllCollisions():
```

for i in range(len(Entities)-1): #Runs through each entity object except the last one as this will be checked by the rest anyway.

    for j in range(len(Entities)-i-1)):

        check = checkCollision(Entities[i], Entities[j+i+1])

    if check[0] == true:

        #Checks through every combination of two entities without repeats.

        return([true, check[1], check[2], check[3]])

        #returns true, signalling that there was a collision, the position of the collision, and the meshes involved with the second being the one with the normal of interest.

    return([false]) #If by this point, true has not been returned, there was no collision.

## Collision Response

I now have an algorithm for detecting collisions and where they occur, I now need to design the response to these collisions. I want to do this by using forces because I have designed motion and rotation using forces. When an object collides with a face, only the force/velocity/momentum in the direction of the normal to the face which the object collided with will be affected. Since objects will typically only be intersecting for a single frame, the force applied must be very large to have any noticeable effect in such a small amount of time. I will do this by using impulse (change in momentum). *Momentum* = *mass × velocity* and so *impulse* =  $\Delta(mv)$  ( $\Delta$  means change in). Impulse is related to force by the equation:

$$\Delta(mv) = F\Delta t$$

Meaning  $F = \frac{\Delta(mv)}{\Delta t}$  and  $\Delta t$  is the time over which these calculations take place (which should be once each frame) meaning  $\Delta t = \frac{1}{fps}$ . If I can find the impulse (change in momentum) then I can find the force. To find the momentum before I must multiply the mass of the object by its velocity (remember this is the component of velocity acting in the direction of the normal). The momentum after the collision is the mass multiplied by the velocity afterwards. The mass remains the same however the velocity changes. To find the velocity afterwards (still only considering the component in the direction of the normal) I need to use the coefficient of restitution (*e*) between the two objects. The coefficient of restitution determines how bouncy (for lack of a better term) an object is. It is always between 0 and 1. If it is 1, the collision is perfectly elastic meaning kinetic energy is conserved. If it is 0, the collision is perfectly inelastic meaning all kinetic energy (contributing to the velocity in the direction of the normal) is lost. In real life it is typically somewhere in between. Each object will have its own coefficient of restitution, chosen either by myself or

the end user. The problem is finding the coefficient between two objects given it for each object individually.

The COR is technically a property between two objects. The COR of an individual object technically refers to the COR between two identical copies of the object or the COR between this object and a rigid wall.

Finding the coefficient of restitution between two objects is not a trivial task. I will instead use a simple approximation by taking the average of the CORs of both objects. This unprecise to reality but should still look mostly realistic.

Once the coefficient of restitution,  $e$ , between the two objects has been determined, the following (definition of the COR) can be used.

$$e = \left| \frac{v_1 - v_2}{u_1 - u_2} \right|$$

Where  $u_1$  &  $u_2$  are the initial velocity of objects 1 & 2 respectively and  $v_1$  &  $v_2$  are the final velocities of objects 1 & 2 respectively, again these only refer to the components of the velocities which act in the direction of the normal.  $u_1$  and  $u_2$  are both known, and I wish to find  $v_1$  and  $v_2$  as this will allow me to work out the change in momentum and hence the force required to act upon each object to produce the required change in velocities. I can create a second equation by using the conservation of momentum. Momentum before equals momentum after. This is only true for the whole system (which is why the change in momentum for each individual object is not always 0) but the combined change in momentum for each object is zero.

$$\text{Momentum before} = m_1 u_1 + m_2 u_2$$

$$\text{Momentum after} = m_1 v_1 + m_2 v_2$$

$$m_1 u_1 + m_2 u_2 = m_1 v_1 + m_2 v_2$$

Where  $m_1$  and  $m_2$  are the masses of object 1 and object 2 respectively.  $u_1$ ,  $u_2$ ,  $m_1$  &  $m_2$  are all known.

I want to solve these equations simultaneously, but the modulus signs in the first equation make this difficult. Currently the first equation does not make sense as vectors cannot be divided by each other. Rewriting the first equation:

$$e = \frac{|v_1 - v_2|}{|u_1 - u_2|}$$

This now makes sense as the modulus sings convert the vector into a length, these lengths are then divided by each other to give a value for  $e$ .

$$e|u_1 - u_2| = |v_1 - v_2|$$

The modulus symbols exist to ensure that the value of  $e$  will be positive. If it is true that vectors  $(u_1 - u_2)$  and  $(v_1 - v_2)$  always face the same direction, then  $e$  will be positive anyway and so I can remove the modulus signs. For this I will refer to the magnitudes of the

vectors and let  $u_1$  be the positive direction (you may picture this as  $u_1$  is facing towards the right).

If  $u_1 > u_2$  then  $u_2$  may be negative (object 1 is moving left) in which case, there is a head on collision (where object 1 is on the left of object 2). If  $u_2$  is positive (objects is moving right but slower than object 1) then object 1 is to the left of object 2. So, if  $u_1 > u_2$ , object 1 is on the left of object 2. After the collision then it would be impossible for object 2 to be moving to the left relative to object 1 as this would mean they would pass through each other which is impossible. In other words,  $v_2$  cannot be  $< v_1$  meaning  $v_1 \leq v_2$ . A similar argument can be made the other way around by just swapping all mentions of 1 and 2.

This means that vectors  $u_1 - u_2$  and  $v_1 - v_2$  always face opposite directions, meaning  $u_1 - u_2$  and  $v_2 - v_1$  always face the same direction. The modulus signs can now be removed.

$$e(u_1 - u_2) = v_2 - v_1$$

Solving simultaneously:

$$\begin{aligned} v_2 &= v_1 + e(u_1 - u_2) \\ m_1 u_1 + m_2 u_2 &= m_1 v_1 + m_2 v_2 \\ m_1 u_1 + m_2 u_2 &= m_1 v_1 + m_2(v_1 + e(u_1 - u_2)) \\ m_1 u_1 + m_2 u_2 &= m_1 v_1 + m_2 v_1 + m_2 e(u_1 - u_2) \\ m_1 u_1 + m_2 u_2 - m_2 e(u_1 - u_2) &= v_1(m_1 + m_2) \\ v_1 &= \frac{m_1 u_1 + m_2 u_2 - m_2 e(u_1 - u_2)}{m_1 + m_2} \\ v_1 &= \frac{m_1 u_1 + m_2 u_2 + m_2 e(u_2 - u_1)}{m_1 + m_2} \end{aligned}$$

Since the initial equations were symmetrical, I can switch all 1s and 2s:

$$v_2 = \frac{m_1 u_1 + m_2 u_2 + m_1 e(u_1 - u_2)}{m_1 + m_2}$$

To write the Pseudocode for collision response I will first need a way to work out the normal to the face at which the objects collided. I will get to this later but will for now assume that I already have this information.

Python code algorithm:

```
def collisionResponse(object1, object2, position, normal):
    #The algorithm needs to know which objects collided, where and at which face.
    u1 = projectVector(object1.RigidBody.Velocity, normal)
```

```

u2 = projectVector(object2.RigidBody.Velocity, normal)

m1 = object1.RigidBody.Mass

m2 = object2.RigidBody.Mass

e = (object1.CoR+object2.CoR)/2

v1 = (m1*u1+m2*u2+m2*e*(u2-u1))/(m1+m2)

v2 = (m1*u1+m2*u2+m1*e*(u1-u2))/(m1+m2)

F1 = new Force(m1*(v1-u1)*fps, position)

F2 = new Force(m2*(v2-u2)*fps, position)

```

**#F1 will act on object1 at the position of the collision, and F2 will act on object2.**

```

a1 = getAngularAcceleration(object1.RigidBody, F1)

a2 = getAngularAcceleration(object2.RigidBody, F2)

#Finds the angular acceleration contributed by each force on each object.

```

```

object1.RigidBody.ResultantAngularAcceleration =
addVectors(object1.RigidBody.ResultantAngularAcceleration, a1)

object2.RigidBody.ResultantAngularAcceleration =
addVectors(object2.RigidBody.ResultantAngularAcceleration, a2)

F1 = forceActingToCentre(object1.RigidBody, F1)

F2 = forceActingToCentre(object2.RigidBody, F2)

object1.RigidBody.ResultantForce =
addVectors(object1.RigidBody.ResultantForce, F1.vector)

object2.RigidBody.ResultantForce =
addVectors(object2.RigidBody.ResultantForce, F2.vector)

```

This procedure works out the correct force to be applied. It then works out how this force will affect angular acceleration as well as how it will contribute to the resultant force acting on the centre of mass. All forces and angular accelerations will be reset to zero and recalculated each frame so that the object does not just continue to accelerate forever.

I still need an algorithm to find the normal at which the objects collided.

My first thought is to revisit the collision detection algorithm which detects whether a point is within the object. It does this by checking each face individually. What I need to do is, on the check where the point is inside of the object (meaning it wasn't during the previous

frame), see which face was it on the wrong side of during the previous frame. This would give a fully accurate detection however would be complicated to implement.

I could take a heuristic approach with a slightly less accurate but easier solution. I will take advantage of the fact that all objects are going to move very small distances each frame, so immediately after a collision, the point will be only slightly in the face. If I find the perpendicular distance between this point and each face and take the shortest one, then I am done.

### Python code Algorithm

```
def getCollisionNormal(point, mesh):
    distances = []
    normals = []
    for i in range(len(mesh.faces)): #runs through each face.
        face = mesh.faces[i]
        faceNormal = face.normal #I will reuse the code from the rendering stage to
        set the outgoing normal as an attribute of the face.
        normals.append(faceNormal)
        a = faceNormal.x
        b = faceNormal.y
        c = faceNormal.z
        d = crossProduct(positionVector(face.point1), normalVector)
        #The equation of the plane is then ax+by+cz=d.
        distance = abs(a*point.x + b*point.y + c*point.z + d)/sqrt(a**2 + b**2 + c**2)
        #This is the perpendicular distance from the point to this face.
        distances.append(distance)
        minDist = min(distances) #This determines the minimum distance.
        index = distance.index(minDist)
        #This determines which face has the minimum distance.
    return (normals(index))
```

After consideration, I have realised that the above approach will not work correctly. I realised that when the impulse force acts perpendicular to the normal it could cause the object to accelerate in the wrong direction. This happens because when deriving the

equation for impulse I only considered an object which does not rotate meaning when I applied that same vector to an object which does rotate I get an incorrect outcome. I will quickly derive the equation for impulse again (in a more efficient way) for an objects which do not rotate, I will then try to adapt this approach to objects which do rotate.

This is my second attempt at the following derivation. The reason why my first attempt (and indeed the above) does not work as intended is that I had a misunderstanding of the coefficient of restitution formula. In the formula:

$$v_1 - v_2 = -e(u_1 - u_2)$$

as derived earlier, I thought that the velocities were in reference to the velocities of the centres of masses of the objects. It turns out, however, that the velocities actually represent the initial velocities of a particle  $P$  which at the point where the objects collided. It is important to note that even though these particles on objects 1 and 2 are at the same position at the instance of the collision, they do not necessarily have the same velocity. It is still true that these are the components of velocity which are in the direction of the normal.

I start by creating a “simpler” equation for a scenario where objects cannot rotate, I will then adapt this to a scenario where they can.

First, I will define the velocity of the point  $P$  on object 1 before and after as  $u_{p_1}$  and  $v_{p_1}$  respectively. I will then similarly define  $u_{p_2}$  and  $v_{p_2}$  for object 2. To use the coefficient of restitution formula, I will need the component of these which act in the direction of the normal,  $n$ . Although I already have a function for this, I will use the mathematical formula for now for easy rearrangement. Also note that I will take object 1 as being the one which had the vertex which intersected object 2. This means that the normal  $n$  is of one of the faces of object 2 and the outgoing normal is the direction in which the impulse should act on object 1. The projected versions of the above velocities are:

$$\begin{aligned} & (u_{p_1} \cdot n) n \\ & (v_{p_1} \cdot n) n \\ & (u_{p_2} \cdot n) n \\ & (v_{p_2} \cdot n) n \end{aligned}$$

Using these in the coefficient of restitution equation:

$$\begin{aligned} (v_{p_1} \cdot n) n - (v_{p_2} \cdot n) n &= -e ((u_{p_1} \cdot n) n - (u_{p_2} \cdot n) n) \\ (v_{p_1} \cdot n - v_{p_2} \cdot n) n &= -e (u_{p_1} \cdot n - u_{p_2} \cdot n) n \\ ((v_{p_1} - v_{p_2}) \cdot n) n &= -e ((u_{p_1} - u_{p_2}) \cdot n) n \end{aligned}$$

I can now simplify this by using the fact that object (in this simpler scenario) do not rotate. This means that all points on the rigid body have the same velocity, including the centre of mass. This means that the velocities of the centres of mass  $v_1, v_2, u_1, u_2$  equal the velocities of the points  $v_{p_1}, v_{p_2}, u_{p_1}, u_{p_2}$  respectively.

Next I will create an equation to find the final velocity of the centre of mass of object 1,  $v_1$  in terms of the impulse,  $j$ , where  $j$  is the linear change in momentum of object 1. ( $j = \Delta(m_1 v_1)$ )

$$v_1 = u_1 + (v_1 - u_1)$$

$$v_1 = u_1 + \Delta(v_1)$$

$$v_1 = u_1 + \frac{\Delta(m_1 v_1)}{m_1}$$

$$v_1 = u_1 + \frac{j}{m_1}$$

I can also use the conservation of momentum. I know that the change in momentum for the entire system is zero  $\Delta(m_1 v_1 + m_2 v_2) = \Delta(m_1 v_1) + \Delta(m_2 v_2) = 0$  meaning  $\Delta(m_2 v_2) = -\Delta(m_1 v_1) = -j$ . This means the impulse for object 2 is negative  $j$ . Usinfg this:

$$\begin{aligned} v_2 &= u_2 + (v_2 - u_2) = u_2 + \Delta(v_2) = u_2 + \frac{\Delta(m_2 v_2)}{m_2} \\ v_2 &= u_2 - \frac{j}{m_2} \end{aligned}$$

Finally, I can find an equation for  $j$  by incorporating both of these equations into the coefficient of restitution equation. This won't be too bad as  $v_1 = v_{p_1}$ ,  $v_2 = v_{p_2}$ ,  $u_1 = u_{p_1}$  and  $u_2 = u_{p_2}$ .

$$\begin{aligned} ((v_1 - v_2) \cdot n)n &= \left( \left( u_1 + \frac{j}{m_1} - u_2 - \frac{j}{m_2} \right) \cdot n \right) n = -e((u_1 - u_2) \cdot n)n \\ \left( \left( u_1 - u_2 + \frac{j}{m_1} - \frac{j}{m_2} \right) \cdot n \right) n &= -e((u_1 - u_2) \cdot n)n \end{aligned}$$

Strictly speaking I can't divide both sides by  $n$  since  $n$  is a vector. However, I know that the vectors on each side of this equation are equal. The left side is  $n$  multiplied by some scalar,  $\lambda$ , the right is  $n$  multiplied by some scalar,  $\mu$ . This is only possible if  $\lambda = \mu$  so I can get away with "dividing" both sides in this case.

$$\left( u_1 - u_2 + \frac{j}{m_1} - \frac{j}{m_2} \right) \cdot n = -e(u_1 - u_2) \cdot n$$

$$(u_1 - u_2) \cdot n + \left( \frac{1}{m_1} - \frac{1}{m_2} \right) j \cdot n = -e(u_1 - u_2) \cdot n$$

$$\left(\frac{1}{m_1} - \frac{1}{m_2}\right) j \cdot n = -e(u_1 - u_2) \cdot n - (u_1 - u_2) \cdot n$$

$$j \cdot n = \frac{-(e+1)(u_1 - u_2)}{\left(\frac{1}{m_1} - \frac{1}{m_2}\right)} \cdot n$$

$j$  is the the change in momentum and due to the coefficient of restitution equation involving components of vectors in the direction of the normal I know that the change in momentum is in the same direction as the normal. This means that the angle,  $\theta$  between  $j$  and  $n$  is zero and so  $j \cdot n = |j||n| \cos \theta = |j|1 \cos 0 = |j|$  meaning that:

$$|j| = \frac{-(e+1)(u_1 - u_2)}{\left(\frac{1}{m_1} - \frac{1}{m_2}\right)} \cdot n$$

and since  $j$  acts in the direction of  $n$ , and  $n$  is a unit vector,  $j = |j|n$

$$j = \left( \frac{-(e+1)(u_1 - u_2)}{\left(\frac{1}{m_1} - \frac{1}{m_2}\right)} \cdot n \right) n$$

This vector for  $j$  can then be put back into the equations for  $v_1$  and  $v_2$ . This is essentially another way of doing the simultaneous equations I had earlier. I now need to do a similar thing, but this time for objects with rotation.

For this scenario, I will define the values the same as I did in the above. There are two main differences. The first is that the velocity of a particle  $P$  is not equal to the velocity of the rigid it is a part of ( $v_{p_1} \neq v_p$ ) unless the object is not rotating. The second difference is that I also need an equation for  $\omega_{final_1}$  and  $\omega_{final_2}$ , these are the angular velocities of objects 1 and 2 respectively after the collision. First I will find an equation linking  $v_p$  to  $v$ .

$v_p$  has two components. The velocity of the centre of mass (the component caused by the motion of the whole body) and the velocity caused by the rigid body's rotation. For this rotational component,  $v_{rot}$  I will use the formula  $v = \omega r$ . This equation is do with the magnitudes of the vectors in question. That is  $|v_{rot}| = |\omega||r_p| \sin \theta$  where  $r_p$  is the vector from the centre of mass to the point  $P$  and  $\theta$  is the angle between  $\omega$  and  $r_p$ . I know that  $v_{rot}$  is perpendicular to both  $r_p$  and  $\omega$ . Using the left hand rule (not right hand due to unconventional axes, although it doesn't end up making a difference here),  $v_{rot}$  is in the direction of  $\omega \times r_p$ . This means  $\omega \times r_p = \lambda v_{rot}$  where  $\lambda$  is some scalar. Using the definition of the cross product:

$$\omega \times r_p = |\omega||r_p| \sin \theta \hat{n} = \lambda v_{rot}$$

Where  $\hat{n}$  is the normal unit vector. Since I know that this is in the same direction as  $v_{rot}$ ,  $\hat{n} = \frac{v_{rot}}{|v_{rot}|}$ .

$$\frac{|\omega||r_p| \sin \theta v_{rot}}{|v_{rot}|} = \lambda v_{rot}$$

Using  $|v_{rot}| = |\omega||r_p| \sin \theta$ :

$$v_{rot} = \lambda v_{rot}$$

So  $\lambda = 1$ . This all means that:

$$v_{rot} = \omega \times r_p$$

This all allows me to find a formula for  $v_p$ . This formula is only valid when  $\omega$  and  $r_p$  are not facing the same direction as in this case  $v_{rot} = \frac{0}{0}$  which is undefined. In this case  $v_{rot}$  should be 0 as the rotational velocity is not affecting the position of  $P$ . I will later add a check for this.

$$v_p = v + \omega \times r_p$$

This formula can be used to find equations for the velocities in the coefficient of restitution equation.

In this rotational scenario, I have chosen to define  $j$  the same way as I did before. Let  $j = \Delta(m_1 v_1)$ .

$$v_1 = u_1 + \Delta(v_1) = u_1 + \frac{\Delta(m_1 v_1)}{m_1}$$

$$v_1 = u_1 + \frac{j}{m_1}$$

Similarly:

$$v_2 = u_2 - \frac{j}{m_2}$$

These are exactly the same as I had before, except I cannot use them as is in the coefficient of restitution equation since I need  $v_{p_1}$  instead of  $v_1$  etc. Writing  $u_1$  and  $u_2$  using the previously defined formula for  $u_{p_1}$  and  $u_{p_2}$  is easy.

$$u_{p_1} = u_1 + \omega_{initial1} \times r_{p_1}$$

$$u_{p_2} = u_2 + \omega_{initial2} \times r_{p_2}$$

However, writing  $v_1$  and  $v_2$  is more difficult, as I do not have an expression for the angular velocities after the collision,  $\omega_{final1}$  and  $\omega_{final2}$ :

$$v_{p_1} = v_1 + \omega_{final1} \times r_{p_1}$$

$$v_{p_2} = v_2 + \omega_{final2} \times r_{p_2}$$

I now need to find an expression for  $\omega_{final}$  in terms of  $j$ . This is very similar to what I did for  $v$ .

$$\omega_{final} = \omega_{initial} + \Delta\omega$$

Using  $I\Delta\omega = \tau\Delta t$ , where  $I$  is moment of inertia and  $\tau$  is torque (as in the torque cause by the impulse,  $j$ ).

$$\omega_{final} = \omega_{initial} + \frac{\tau\Delta t}{I}$$

Using  $\tau = r \times F$  where  $F$  is the force responsible for the change in angular velocity (this is the force from the collision).

$$\begin{aligned}\omega_{final} &= \omega_{initial} + \frac{(r_p \times F)\Delta t}{I} = \omega_{initial} + \frac{\left(r_p \times \frac{\Delta(mv)}{\Delta t}\right)\Delta t}{I} \\ \omega_{final} &= \omega_{initial} + \frac{r_p \times j}{I}\end{aligned}$$

$I$  is the moment of inertia of the object. This is a rotational analogue to mass (like how torque is a rotational analogue of force and angular velocity is a rotational analogue to (linear) velocity). It depends on the axis of rotation about which it is being considered. In two dimensions, since rotation is always about the centre of mass, the moment of inertia is a constant scalar (number), in three dimensions however, the axis of rotation can change and therefore the moment of inertia as a scalar can change. A perspective is to view rotational inertia as a tensor (or matrix) which takes into account the axis of rotation. In general:

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

The specific components of this matrix are not too important for our purposes. The elements of the leading diagonal (top left, centre, bottom right) are called the moments of inertia, whereas the other elements are the products of inertia.

The matrix you get may be different depending on the axis which you take to be  $x$ ,  $y$ , and  $z$ . One of the potential solutions is a diagonalised matrix (meaning all elements except the leading diagonal are zero) which makes it very easy to multiply by a vector (just multiple each component of the vector by the corresponding matrix component) and easy to find an inverse (take the reciprocal of each element on the leading diagonal and leave the zeros the same) which is much easier to work with. For this to happen, the products of inertia must all be zero, which happens when the axis we take are axis of symmetry for the rigid body, for example for a cuboid, if we take its centre of mass to be the origin and assume is to be axis aligned, then the products of inertia will be zero. This is how I will find the inertia tensor for a cuboid. The diagonalised inertia tensor looks like this:

$$\begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix}$$

Where  $I_{xx}$ ,  $I_{yy}$ , and  $I_{zz}$  are the moments of inertia about the  $x$ ,  $y$ , and  $z$  axis respectively.

To find the moment of inertia of a rigid body about an axis, you need to sum the moments of inertia of each individual particle of the rigid body about that same axis. The moment of inertia of a particle is given by:

$$I = mr^2$$

where  $m$  is the mass of the particle and  $r$  is the perpendicular distance from that particle to the axis of rotation. For a rigid body with  $N$  particles, the  $k^{th}$  particle has a distance of  $r_k$  and a mass of  $m_k = \frac{m}{N}$  where  $m$  is the mass of the rigid body. The moment of inertia of the rigid body is:

$$I = \sum_{k=1}^N (r_k)^2 \times \frac{m}{N}$$

Since we want the rigid body to be made of an infinite number of points, we need to see what happens when  $N$  approaches infinity. When this happens the mass of each particle, and hence its moment of inertia approaches zero. This means we will be summing an infinite number of infinitely small quantities, that is, an integral.

$$I = \int r^2 dm$$

This integral cannot be evaluated directly as  $r$  is not a function of  $m$  (given the mass of a particle I could not tell you its distance as each particle has the same mass). To solve this problem, I will use a substitution. Assuming that density,  $\rho$  is uniform across the rigid body (it is the same everywhere and so is a constant) I know that  $\rho = \frac{m}{v}$ , which I can use to turn the integral from one over mass into one over volume.

$$m = \rho v$$

$$dm = \rho dv$$

$$I = \int r^2 \rho dv$$

$$I = \rho \int r^2 dv$$

I still cannot evaluate this directly.  $dv$  represents the volume of each particle. Each particle is a cuboid with a length, width, and height. Since the cuboid is axis aligned, the particle are as well meaning the length of each particle is an infinitesimal change in  $x$ , the height an infinitesimal change in  $y$ , and the width an infinitesimal change in  $z$ . Using this I can say that the volume of each particle is  $dxdydz$ . If I integrate each particle over the length of the cuboid (with centre at the origin and axis aligned faces) I would be integrating over  $x$  from

$-\frac{l}{2}$  to  $\frac{l}{2}$ . This would account for each particle connect in a line over the width of the cuboid.

If I then integrate with  $y$  from  $-\frac{h}{2}$  to  $\frac{h}{2}$ , I get all of the particle on one face of the cuboid.

Finally, with  $z$  from  $-\frac{w}{2}$  to  $\frac{w}{2}$  will have considered each particle precisely once. This is the total moment of inertia.

$$I = \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \int_{y=-\frac{h}{2}}^{\frac{h}{2}} \int_{x=-\frac{l}{2}}^{\frac{l}{2}} r^2 dx dy dz$$

I now need to write  $r^2$  as an explicit function of  $x$ ,  $y$ , and  $z$ . For rotation about the  $x$  axis:

$$r^2 = z^2 + y^2$$

by the pythagorean theorem.

$$\begin{aligned} I_{xx} &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \int_{y=-\frac{h}{2}}^{\frac{h}{2}} \int_{x=-\frac{l}{2}}^{\frac{l}{2}} (z^2 + y^2) dx dy dz \\ &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \int_{y=-\frac{h}{2}}^{\frac{h}{2}} [(xz^2 + xy^2)]_{x=-\frac{l}{2}}^{\frac{l}{2}} dy dz \\ &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \int_{y=-\frac{h}{2}}^{\frac{h}{2}} (lz^2 + ly^2) dy dz \\ &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \left[ lyz^2 + \frac{l}{3}y^3 \right]_{y=-\frac{h}{2}}^{\frac{h}{2}} dz \\ &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \left( l\left(\frac{h}{2}\right)z^2 - l\left(-\frac{h}{2}\right)z^2 + \frac{l}{3}\left(\frac{h}{2}\right)^3 - \frac{l}{3}\left(-\frac{h}{2}\right)^3 \right) dz \\ &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \left( lhz^2 + \frac{l}{3}\frac{h^3}{8} + \frac{l}{3}\frac{h^3}{8} \right) dz \\ &= \rho \int_{z=-\frac{w}{2}}^{\frac{w}{2}} \left( lhz^2 + \frac{lh^3}{12} \right) dz \\ &= \rho \left[ \frac{lh}{3}z^3 + \frac{lh^3z}{12} \right]_{z=-\frac{w}{2}}^{\frac{w}{2}} \\ &= \rho \left( \frac{lh}{3}\left(\left(\frac{w}{2}\right)^3 - \left(-\frac{w}{2}\right)^3\right) + \frac{lh^3}{12}\left(\frac{w}{2} - -\frac{w}{2}\right) \right) \\ &= \rho \left( \frac{lh}{3}\left(\frac{w^3}{4}\right) + \frac{lh^3w}{12} \right) \end{aligned}$$

$$\begin{aligned}
&= \rho \left( \frac{lw^3h}{12} + \frac{lwh^3}{12} \right) \\
&= \frac{\rho lwh}{12} (w^2 + h^2)
\end{aligned}$$

Using  $m = \rho v$  and  $v = lwh$ :

$$\begin{aligned}
I_{xx} &= \frac{\rho v}{12} (w^2 + h^2) \\
I_{xx} &= \frac{m}{12} (w^2 + h^2)
\end{aligned}$$

Similarly:

$$\begin{aligned}
I_{yy} &= \frac{m}{12} (l^2 + w^2) \\
I_{zz} &= \frac{m}{12} (l^2 + h^2)
\end{aligned}$$

Putting this all into the tensor for moment of inertia:

$$I = \begin{pmatrix} \frac{m}{12} (w^2 + h^2) & 0 & 0 \\ 0 & \frac{m}{12} (l^2 + w^2) & 0 \\ 0 & 0 & \frac{m}{12} (l^2 + h^2) \end{pmatrix}$$

This matrix can have the  $\frac{m}{12}$  factored out giving:

$$I = \frac{m}{12} \begin{pmatrix} w^2 + h^2 & 0 & 0 \\ 0 & l^2 + w^2 & 0 \\ 0 & 0 & l^2 + h^2 \end{pmatrix}$$

This matrix is specifically for a cuboid. Others would need to be derived for different shapes. Even though the above derivation assumed that the cuboid was axis aligned and centred at the origin, this matrix should work for all positions and rotations of the cuboid.

One problem is that in the equation  $\omega_{final} = \omega_{initial} + \frac{r_p \times j}{I}$  I am dividing by  $I$  and you cannot divide by a matrix. If I look at where this division originated, I rearranged  $I\Delta\omega = \tau\Delta t$  into  $\Delta\omega = \frac{\tau\Delta t}{I}$  which is not allowed if  $I$  is a matrix. What I should instead do is multiply (to the left) the inverse of  $I$  on both sides:  $\Delta\omega = I^{-1}(\tau\Delta t)$ . This leads the final result to become:

$$\omega_{final} = \omega_{initial} + I^{-1}(r_p \times j)$$

Using this to complete the equations for  $v_{p_1}$  and  $v_{p_2}$ :

$$v_{p_1} = v_1 + \omega_{final_1} \times r_{p_1}$$

$$v_{p_1} = v_1 + \left( \omega_{initial_1} + I_1^{-1}(r_{p_1} \times j) \right) \times r_{p_1}$$

$$\begin{aligned}
v_{p_1} &= v_1 + \omega_{initial1} \times r_{p_1} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} \\
v_{p_1} &= u_1 + \frac{j}{m_1} + \omega_{initial1} \times r_{p_1} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} \\
v_{p_1} &= u_{p_1} - \omega_{initial1} \times r_{p_1} + \frac{j}{m_1} + \omega_{initial1} \times r_{p_1} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} \\
v_{p_1} &= u_{p_1} + \frac{j}{m_1} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1}
\end{aligned}$$

$$\begin{aligned}
v_{p_2} &= v_2 + \omega_{final2} \times r_{p_2} \\
v_{p_2} &= v_2 + \left( \omega_{initial2} + I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2} \\
v_{p_2} &= v_2 + \omega_{initial2} \times r_{p_2} + \left( I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2} \\
v_{p_2} &= u_2 - \frac{j}{m_2} + \omega_{initial2} \times r_{p_2} + \left( I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2} \\
v_{p_2} &= u_{p_2} - \omega_{initial2} \times r_{p_2} - \frac{j}{m_2} + \omega_{initial2} \times r_{p_2} + \left( I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2} \\
v_{p_2} &= u_{p_2} - \frac{j}{m_2} + \left( I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2}
\end{aligned}$$

I am now ready to introduce the coefficient of restitution equation.

$$\begin{aligned}
\left( (v_{p_1} - v_{p_2}) \cdot n \right) n &= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) n \\
\left( \left( \left( u_{p_1} + \frac{j}{m_1} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} \right) - \left( u_{p_2} - \frac{j}{m_2} + \left( I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2} \right) \right) \cdot n \right) n \\
&= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) n
\end{aligned}$$

Now all I need to do is solve for  $j$ .

$$\begin{aligned}
\left( \left( u_{p_1} + \frac{j}{m_1} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} - u_{p_2} + \frac{j}{m_2} - \left( I_2^{-1} (r_{p_2} \times -j) \right) \times r_{p_2} \right) \cdot n \right) n \\
&= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) n
\end{aligned}$$

$$\begin{aligned} & \left( j \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + u_{p_1} - u_{p_2} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times j) \right) \times r_{p_2} \right) \cdot n \\ &= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) n \end{aligned}$$

It may seem like I am about to divide by  $n$  which is not allowed as it is a vector, however in reality I am using the same trick as I used before to say that the scalars are equal.

$$\begin{aligned} & \left( j \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + u_{p_1} - u_{p_2} + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times j) \right) \times r_{p_2} \right) \cdot n \\ &= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) \end{aligned}$$

$$\begin{aligned} & \left( j \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times j) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times j) \right) \times r_{p_2} \right) \cdot n + (u_{p_1} - u_{p_2}) \cdot n \\ &= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) \end{aligned}$$

I am again going to use the same trick I used before, I will find the magnitude of  $j$  by using the fact that  $j$  is in the direction of the normal vector.  $j = |j|n$

$$\begin{aligned} & |j|n \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times |j|n) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times |j|n) \right) \times r_{p_2} \cdot n + (u_{p_1} - u_{p_2}) \\ & \cdot n = -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) \end{aligned}$$

$$\begin{aligned} & |j| \left( n \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times n) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times n) \right) \times r_{p_2} \right) \cdot n + (u_{p_1} - u_{p_2}) \cdot n \\ &= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) \end{aligned}$$

$$\begin{aligned} & |j| \left( n \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times n) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times n) \right) \times r_{p_2} \right) \cdot n \\ &= -e \left( (u_{p_1} - u_{p_2}) \cdot n \right) - (u_{p_1} - u_{p_2}) \cdot n \end{aligned}$$

$$\begin{aligned} & |j| \left( n \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times n) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times n) \right) \times r_{p_2} \right) \cdot n \\ &= -((e+1)(u_{p_1} - u_{p_2}) \cdot n) \end{aligned}$$

$$|j| = \frac{-((e+1)(u_{p_1} - u_{p_2}) \cdot n)}{\left( n \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times n) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times n) \right) \times r_{p_2} \right) \cdot n}$$

Finally using  $j = |j|n$

$$j = \frac{-((e+1)(u_{p_1} - u_{p_2}) \cdot n)}{\left( n \left( \frac{1}{m_1} + \frac{1}{m_2} \right) + \left( I_1^{-1} (r_{p_1} \times n) \right) \times r_{p_1} + \left( I_2^{-1} (r_{p_2} \times n) \right) \times r_{p_2} \right) \cdot n} n$$

This vector,  $j$  can then be put back into the equations for  $v_1$ ,  $v_2$ ,  $\omega_{final_1}$ ,  $\omega_{final_2}$ .

$$v_1 = u_1 + \frac{j}{m_1}$$

$$v_2 = u_2 - \frac{j}{m_2}$$

$$\omega_{final_1} = \omega_{initial_1} + I_1^{-1} (r_{p_1} \times j)$$

$$\omega_{final_2} = \omega_{initial_2} + I_2^{-1} (j \times r_{p_2})$$

The above derivation was heavily adapted from that given by Chris Hecker on his website: <https://www.chrishecker.com/images/e/e7/Gdmphys3.pdf> His derivation was only for two dimensions (hence his lack of cross products), whereas mine is for three dimensions. Also he used  $j$  to represent the magnitude of impulse, whereas I used it to represent the vector. His being in two dimensions also means that his moment of inertia was a scalar (whereas mine is a tensor).

The above equations which I have just derived will be very useful to me, I will be able to directly calculate the velocities and angular velocities of objects after a collision without needing to worry about forces. This means that the algorithms I created earlier to use a force to cause a rotation have been made obsolete. The algorithms involved in applying linear forces are not, since there are still other important forces, namely weight, which may be modelled by these algorithms.

### Note on Post-Development Testing

During post development, once creating objects is easier, I will test the program with many objects at once. This will be done to test how well my optimisation has worked. This will be done during evaluation.

## Test Plan

Since this prototype has no usability features, it is all backend functionality which will be made usable by prototype 3, there is little I can do in the way of a usability test plan. What I could do however is test scenarios by altering the source code and seeing if the actual result matches the expected.

Test No.	Input Data	Expected Output
9	A single cube, centred at the origin. No resultant force set.	Cube should accelerate down due to gravity at $9.81 \text{ ms}^{-2}$
10	A single cube centred at the origin, with an upward force of 9.81	Cube should remain stationary (since upward force balances with gravity).
11	A single cube centred at the origin, with an upward force of 9.81 and angular velocity of (1,1,1)	Cube should rotate about the line $y = z = x$ with a uniform angular speed.
12	A single cube centred at the origin, with an upward force of 9.81 and angular acceleration of (1,1,1)	Cube should rotate about the line $y = x = z$ with an increasing angular speed.
13	Two cubes, centred at (0,0,2) and (0,0,-2) moving with respective velocities (0,0,-1) and (0,0,1) (towards each other). (Each will have a coefficient of restitution of 1 so the collision is elastic).	Cubes should move towards each other, and instantaneously separate away with the same speed as they came together.
14	Two cubes, centred at (0,0,2) and (0,0,-2) moving with respective velocities (0,0,-1) and (0,0,1) (towards each	Cubes should move towards each other, and instantaneously separate away with half the speed as they came together.

	other). (Each will have a coefficient of restitution of 0.5 so the collision is inelastic).	
15	Two cubes, centred at (0,0,2) and (0,0,-2) moving with respective accelerations (0,0,-1) and (0,0,1) (towards each other). (Each will have a coefficient of restitution of 1)	Cubes should accelerate towards each other. They should collide and rebound away from each other, slowing down until they move towards each other again and rebound again to the same point, this repeats indefinitely.
16	Same as above with Coefficient of Restitution of 0.5	Same as above, but after each collision, they lose speed and don't make it as far away.
17	Two cubes centred at (0,0,2) and (0.2,0.2,-2) and velocities (0,0,-1) and (0,0,1), Coefficient of Restitution of 1.	The cubes should move towards each other with constant speed. When they collide, they will each rebound and rotate (because they are not aligned to the line of centres).

These tests are to ensure that the physics side works as expected, which is essential before progressing to prototype 3.

## Key Classes, Subroutines and Variables

Classes, functions/procedures and variables for prototype 2 (after refinement).

Classes designed for this prototype:

Class Name	Attributes	Description
RigidBody	double mass, Vector centreOfMass, Vector velocity, Vector acceleration, Vector angularVelocity, Vector angularAcceleration, Vector force, double CoR, Vector inertialInv	Holds data about an object which is used an updated for physics interactions such as collisions and applying a force.

Function/Procedure Name	Inputs	Output	Description
vectorTimesScalar	Vector v, double s	Vector	Multiplies a vector by a number.
vectorAddVector	Two Vectors	Vector	Adds two vectors.
vectMag	Vector	double magnitude	Returns the magnitude (length) of a given vector.
angleBetweenVectors	Two Vectors	double angle	Returns the angle between two vectors.
distBetweenPointAndFace	Vector point, Face face	double distance	Returns the shortest distance between a point and a face.
normalOfClosestFace	Vector point, List of faces	Vector normal	Returns the normal vector of whichever face was closest to the given point.
projectVector	Two vectors	Vector	Returns the component of the first vector, acting in the direction of the second.
faceNormal	Face and Mesh	Vector	Returns the normal vector to a face, given also the mesh it came from.
updateAcceleration	Entity entity	void	Updates an object's acceleration.
updateVelocity	Entity entity	void	updates an object's velocity.

displaceObject	Entity, double displacement	void	Moves an object by a specified amount.
updateAngularVelocity	Entity entity	void	updates an object's angular velocity.
rotateObject	Entity entity, Vector angle	void	rotates an object by a specified amount.
boundingSpheresCheck	Two Entities	boolean	checks if the "bounding spheres" are intersecting (true) or not (false). This is done for collision detection optimisation.
checkPointAgainstFace	Vector point, Face face	boolean	checks if a point is on the "inside" of the given face (true) or not (false). "Inside" here means, on the same side of that face as it would be if the point were inside the mesh from which the face came.
checkPointAgainstMesh	Vector point, Mesh mesh	boolean	Checks if a point is inside the mesh.
checkMeshAgainstMesh	Two Entities	tuple containing the following: boolean, Vector (position of intersection), Both Entities	Checks if the meshes of two entities intersect (only checks one of two cases).
checkCollisionPrecise	Two Entities	Same tuple as above	Checks if the meshes of two entities intersect (checks both cases by using the above both ways round).
checkCollision	Two entities	Same tuple as above	Checks if they intersect, first using the bounded spheres check, and then the more precise one (if the first one succeeds).

			This is done for optimisation.
checkAllCollisions	void	list of tuples with the above format	Returns a list containing collision data for all collisions which occurred during that frame (stored within the tuples).
collisionResponse	2 Entities, Vector point (point of intersection), Vector normal	void	Performs the collision response on the given entities.
getNormalReactions	2 Entities, Vector normal	Tuple containing a pair of Vectors	Returns the pair of normal reaction force (vectors) given two entities which are intersecting and the normal at which this happens.
doCollisionResponse	Tuple with format from getCollision	void	Finds the appropriate normal and calls the collision response function.
physics	void	void	performs collision detection, calls doCollisionResponse if necessary. Calls the procedures to update each objects' position, velocity etc.,

Again, most of these are methods of the Global class, these are highlighted in green.

Variable Name	Data Type	Description
gravity	double	acceleration due to gravity.
collisionFlags	list of Tuples, each with a bool and two Entities.	Represents a specific pair of entities and whether they had a collision registered in the previous frame. Prevents collision response from being executed multiple times for a single collision.

This concludes the design of the second prototype.

## Prototype 2 Development

The first prototype has now been fully implemented, meaning I can move on to the implementation of the second as outlined in my design section. The first thing I will do is something which I meant to do at the end of the first prototype, but I forgot to. This is a framerate counter/limiter. I was hoping that there would be an in built feature of WinForms to handle this for me, but there does not appear to be. My idea is to create a timer which we measure how long a frame takes to be drawn. If this is below the desired frametime, the program will wait until the correct amount of time has elapsed. In this case, the fps will be set as the desired fps. If it is above the desired frametime, the program will not wait and the fps will be set as the reciprocal of the measured frametime ( $\frac{1}{measured\ frametime}$ ).

The purpose of this is to ensure that more resources than necessary are not being used (there is no point in drawing 200 frames per second if the user only wants 60), and to ensure that a lower framerate does not alter things like the camera speed or the actual physics calculations. I will change camera's speed and rotation to be in terms of the framerate to ensure consistency across different devices and on the same device when the program slows down or when the resolution of the window is changed (the program runs faster when the resolution is smaller).

I first need a way to create this timer in C#. I can use the built in Stopwatch class for this.

I first created a timer (as an instance of the stopwatch class). I then started the timer using its Start() method:

```
Stopwatch timer = new Stopwatch();
timer.Start();
```

The timer is then stopped at the end of the frame and the reciprocal (frames per second) is printed to the debug menu.

```
Debug.WriteLine((1/timer.Elapsed.TotalSeconds).ToString());
```

In doing this, I found that my average framerate seems to be around 10000-20000 fps on my desktop and around 1000-2000 fps on my laptop. This demonstrates the importance of the framerate limiter/counter, as currently the camera moves 10 times faster on the desktop as it does on the laptop, and I want the camera speed to the same regardless of the device which my end user is using.

Since I want the frameTime to be accessed by other parts of the program, I made it a public variable in the Global class:

```
public static double frameTime;
public static int fps = 165;
```

As you can see, I also created a variable for the desired fps, which I have here set to 165, but I will later give the user the option to change this.

I created an if statement to check if the frametime is less than the desired frametime ( $\frac{1}{desired\ fps}$ ). If yes, (i.e., the frame was drawn quicker than expected) then the extra time needed to wait is calculated. The program then waits for this amount of time, then the framerate is set to the desired one so that it can be used in calculations. I then went to the where the cameraControl() function is called and changed the inputed displacement and angle of the camera to being some constant (representing velocity or angular velocity) multiplied by the frameTime (since  $displacement = velocity \times time$ ).

After implementing this, the fps can be changed, and speed and angular speed of the camera is the same regardless of window size or framerate chosen. This is shown in [video17](#).

```
Global.frameTime = timer.Elapsed.TotalSeconds;
if (Global.frameTime < 1.0/Global.fps)
{
    Thread.Sleep(Convert.ToInt32(((1.0 / Global.fps - Global.frameTime) * 1000)));
    Global.frameTime = 1.0 / Global.fps;
}

Global.cameraControl(50*Global.frameTime, 5*Global.frameTime, Global.Camera);
```

I will now begin work on the physics, starting with the motion of objects. I first created the RigidBody class and added some attributers to it. I may end up removing angular acceleration, as the only time when angular velocity changes is during collisions which are instantaneous and so they won't use this attribute.

```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
RigidBody.cs  ComputerScienceAlevelProject
ComputerScienceAlevelProject
RigidBody.cs
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

namespace ComputerScienceAlevelProject
{
    internal class RigidBody
    {
        public double mass;
        public Pos centreOfMass;
        public Vector velocity;
        public Vector acceleration;
        public Vector angularVelocity;
        public Vector angularAcceleration;
        public Vector force;

        public RigidBody(double mass, Pos centreOfMass, Vector velocity, Vector acceleration, Vector angularVelocity, Vector angularAcceleration, Vector force)
        {
            this.mass = mass;
            this.centreOfMass = centreOfMass;
            this.velocity = velocity;
            this.acceleration = acceleration;
            this.angularVelocity = angularVelocity;
            this.angularAcceleration = angularAcceleration;
            this.force = force;
        }
    }
}
```

Now I need to make these attributes do something. I also changed centreOfMass to be a vector as this makes calculations involving it easier to write code for. I also replaced every instance of the position class with the vector class as they both do the same thing. I then also removed the position class. I changed the mesh class to store its vertices in an array (as opposed to as individual attributes). This allowed me to loop through them in a for loop. I created the following function to update acceleration, velocity and position:

I then called each of these functions in the Form1 class during the paint event (so they will be called once each frame). Finally I set the velocity of one of the cubes I have created to  $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ . The expected result is for the cube to move to the right, which it does as shown in

[video18](#). As shown in the video, as I change the velocity vector, the direction and speed of the cube changes accordingly. I then tried setting the velocity to zero and setting the acceleration to 9.81 down. This had no effect. The reason for this is that the program detected that there was no resultant force and so it set the acceleration back to zero again. I instead tried setting the force vector of the small cube to be 9.81 down and for the larger cube to be  $9.81 \times 125$  down (since I set the mass of the second cube as 125 and  $W = mg$ ). This caused them both to accelerate down as shown in [video19](#). The video shows that the objects do certainly accelerate down, but it feels far too slow (bearing in mind the the smaller cube has side length 1 unit). This seems to be caused by the framerate. If I set Global.fps to a very large number (like 1000) then the cubes accelerate very slowly. If I set it to a small number (like 30) they accelerate more quickly. The same applies to camera movement. I believe this is is caused by the Sleep method or the timer being inaccurate. I created a second timer which starts, the program then waits for  $\frac{1000}{165}$  milliseconds, the second timer then stops, and the time in second is printed. I expected the number  $\sim 0.00606$ , I instead got  $\sim 0.016$ . This means that either the timer is wrong or the Sleep method is wrong. To test the timer, I created one in the global class (so it wouldn't be reset each frame). I set it to begin when I press space and to stop when I release space and print the time taken. I set a timer on my phone for 20 seconds as I held down the space key. After closing the application, the debugger had printed 20 meaning the timer is accurate and the sleep method is what is causing the inaccuracies. The problem is that the sleep method waits too long before continuing the program.

My solution to this problem was to use another timer and a while loop. This while loop would only exit once the new timer reached the wanted amount of time. Now the amount of time the program waits for is much closer to the expected value. Objects now accerate due to gravity at a much more reasonable rate as shown in [Video20](#).

```
if (Global.frameTime < 1.0 / Global.fps)
{
    Stopwatch timer2 = new Stopwatch();
    timer2.Start();

    while (timer2.Elapsed.TotalSeconds < 1.0/Global.fps - Global.frameTime)
    {
        ...
    }

    timer2.Stop();
    Global.frameTime = 1.0 / Global.fps;
}
```

Objects are now able to realistically move due to forces, but they cannot yet rotate. I will next implement the algorithm outlined for rotation in the design section.

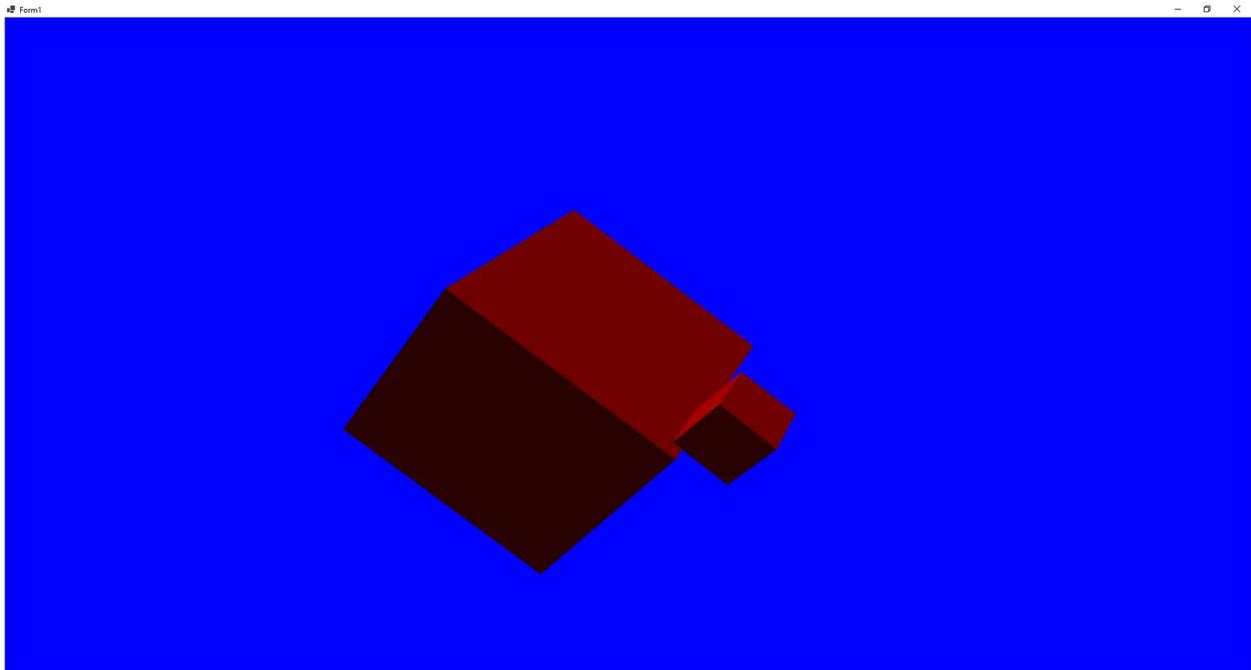
The code has now been written:

```
//Rotation
reference
public static void rotateObject(Entity obj, Vector angle)
{
    double aX = angle.x;
    double aY = angle.y;
    double aZ = angle.z;
    for (int i = 0; i < obj.mesh.vertices.Length; i++)
    {
        obj.mesh.vertices[i] = vectBetweenPoints(obj.rigidBody.centreOfMass, obj.mesh.vertices[i]);
        double x = obj.mesh.vertices[i].x;
        double y = obj.mesh.vertices[i].y;
        double z = obj.mesh.vertices[i].z;
        obj.mesh.vertices[i].x = x * Math.Cos(aY) * Math.Cos(aZ) + y * Math.Cos(aY) * Math.Sin(aZ) - z * Math.Sin(aY);
        obj.mesh.vertices[i].y = x * (Math.Sin(aX) * Math.Sin(aY) * Math.Cos(aZ) - Math.Sin(aX) * Math.Cos(aY) * Math.Sin(aZ)) + y * (Math.Sin(aX) * Math.Sin(aY) * Math.Sin(aZ) + Math.Cos(aX) * Math.Cos(aY) * Math.Sin(aZ)) + z * Math.Sin(aX) * Math.Cos(aY);
        obj.mesh.vertices[i].z = x * (Math.Sin(aX) * Math.Sin(aZ) + Math.Cos(aX) * Math.Sin(aY) * Math.Cos(aZ)) + y * (Math.Cos(aX) * Math.Sin(aY) * Math.Sin(aZ) - Math.Sin(aX) * Math.Cos(aY) * Math.Sin(aZ)) + z * Math.Cos(aX) * Math.Cos(aY);
        obj.mesh.vertices[i] = vectorAddVector(obj.mesh.vertices[i], obj.rigidBody.centreOfMass);
    }
}
```

I then call this function once per frame on each object with the vector  $(0.01, 0, 0)$  which should cause it to rotate slowly, clockwise about the  $x$ -axis, which it does. Similarly,  $(0, 0.01, 0)$  rotates about the  $y$ -axis and  $(0, 0, 0.01)$  about the  $z$ -axis. I then set each of them to equal Global.frameTime (the time taken for a frame to elapse) which should result in them rotating about each axis by an equal amount. To test this properly, I stopped the objects from moving by removing the update position procedure call.

```
void physics()
{
    for (int i = 0; i < Global.Entities.Count; i++)
    {
        Global.updateAcceleration(Global.Entities[i]);
        Global.updateVelocity(Global.Entities[i]);
        //Global.updatePosition(Global.Entities[i]);
        Global.rotateObject(Global.Entities[i], new Vector(Global.frameTime, Global.frameTime, Global.frameTime));
    }
}
```

This works as intended, as shown here:



And in [Video21](#).

I then repeated the same code for updating velocity for angular velocity.

```
public static void updateAngularVelocity(Entity entity)
{
    entity.rigidBody.angularVelocity = vectorAddVector(entity.rigidBody.angularVelocity, vectorTimesScalar(entity.rigidBody.angularAcceleration, frameTime));
}
```

Rotation is now implemented and works fully as expected. I can move on to collision detection.

I first create the less precise “bounding spheres” algorithm.

```
public static bool boundingSpheresCheck(Entity obj1, Entity obj2)
{
    double radius1 = distBetweenPoints(obj1.rigidBody.centreOfMass, obj1.mesh.vertices[0]);
    double radius2 = distBetweenPoints(obj2.rigidBody.centreOfMass, obj2.mesh.vertices[0]);
    double distance = distBetweenPoints(obj1.rigidBody.centreOfMass, obj2.rigidBody.centreOfMass);
    if (radius1 + radius2 < distance)
    {
        return (false);
    }
    else
    {
        return (true);
    }
}
```

I then create the more precise algorithm starting with the function to check whether a point is on the correct side of a given face by using that face’s outgoing normal.

```
public static bool checkPointAgainstFace(Vector point, Face face)
{
    Vector pointVect = vectBetweenPoints(face.vertices[0], point);
    double angle = angleBetweenVectors(pointVect, face.normal);
    if (0 <= angle && angle < Math.PI / 2)
    {
        return (false);
    }
    else
    {
        return (true);
    }
}
```

For this to work, I needed to create the function to find the angle between two vectors.

```
public static double vectMag(Vector v)
{
    return (Math.Sqrt(Math.Pow(v.x, 2) + Math.Pow(v.y, 2) + Math.Pow(v.z, 2)));
}

1 reference
public static double angleBetweenVectors(Vector v1, Vector v2)
{
    return (Math.Acos(dotProduct(v1, v2)/(vectMag(v1) * vectMag(v2))));
```

As you can see, I also created a function to find the magnitude of a vector.

Next, I create the function to check the point against each face in the mesh.

```

public static bool checkPointAgainstMesh(Vector point, Mesh mesh)
{
    for(int i = 0; i < mesh.faces.Length; i++)
    {
        if (checkPointAgainstFace(point, mesh.faces[i]) == false)
        {
            return (false);
        }
    }
    return (true);
}

```

Next, I create the function to check if two meshes intersect.

```

public static Tuple<bool, Vector, Mesh, Mesh> checkMeshAgainstMesh(Mesh mesh1, Mesh mesh2)
{
    for (int i = 0; i < mesh1.vertices.Length; i++)
    {
        if (checkPointAgainstMesh(mesh1.vertices[i], mesh2) == true)
        {
            var t1 = new Tuple<bool, Vector, Mesh, Mesh>(true, mesh1.vertices[i], mesh1, mesh2);
            return t1;
        }
    }
    var t2 = new Tuple<bool, Vector, Mesh, Mesh>(false, new Vector(0,0,0), mesh1, mesh2);
    return (t2); //The only one of these returned values to be used is the bool, the vector and meshes are only here to avoid errors.
}

```

This function returns a tuple because it needs to return multiple values of different types.

Finally, the last function for the precise detection algorithm brings these all together.

```

public static Tuple<bool, Vector, Mesh, Mesh> checkCollisionPrecise(Mesh mesh1, Mesh mesh2)
{
    Tuple<bool, Vector, Mesh, Mesh> check1 = checkMeshAgainstMesh(mesh1, mesh2);
    if (check1.Item1 == true)
    {
        return (check1);
    }
    Tuple<bool, Vector, Mesh, Mesh> check2 = checkMeshAgainstMesh(mesh2, mesh1);
    if (check2.Item1 == true)
    {
        return (check2);
    }
    else
    {
        var t = new Tuple<bool, Vector, Mesh, Mesh>(false, new Vector(0, 0, 0), mesh1, mesh2);
        return (t);
    }
}

```

The last algorithm in collision detection, brings together to precise and non-precise algorithms which we have so far created:

```

public static Tuple<bool, Vector, Mesh, Mesh> checkCollision(Entity obj1, Entity obj2)
{
    if (boundingSpheresCheck(obj1, obj2) == true)
    {
        Tuple<bool, Vector, Mesh, Mesh> preciseCheck = checkCollisionPrecise(obj1.mesh, obj2.mesh);
        if (preciseCheck.Item1 == true)
        {
            return (preciseCheck);
        }
    }
    var t = new Tuple<bool, Vector, Mesh, Mesh> (false, new Vector(0, 0, 0), obj1.mesh, obj2.mesh);
    return t;
}
0 references
public static Tuple<bool, Vector, Mesh, Mesh> checkAllCollisions()
{
    for (int i = 0; i < Entities.Count - 1; i++)
    {
        for (int j = 0; j < Entities.Count - i - 1; j++)
        {
            Tuple<bool, Vector, Mesh, Mesh> check = checkCollision(Entities[i], Entities[i + j + 1]);
            if (check.Item1 == true)
            {
                return (check);
            }
        }
    }
    Vector[] v = { new Vector(0, 0, 0) };
    Mesh m = new Mesh(v);
    var t = new Tuple<bool, Vector, Mesh, Mesh>(false, new Vector(0, 0, 0), m, m);
    return t;
}

```

The last few lines are not important, they are only included so that the appropriate tuple is returned to prevent errors. The last function here, checkAllCollision should be run each frame so that the program can be informed when two objects have collided, as well as which two and where this happened.

To test this collision detection algorithm, I added this line of code to print “false” each frame if there are no collisions, and “true” if there is a collision.

```

//Collision Detection
Tuple<bool, Vector, Mesh, Mesh> check = Global.checkAllCollisions();
Debug.WriteLine(check.Item1);

```

I then ran the program, allowing the cubes to accelerate down and the program printed “false” repeatedly as expected (as there were no collisions). I then set the force acting on the larger cube to 0, so it would not move, and the force on the smaller cube to (1, 0, 0) so it would move towards the larger one. Before they collided with each other however, I got the following error:

```

3 references
public static double dotProduct(Vector vect1, Vector vect2) //computes the dot product of two vectors
{
    return (vect1.x * vect2.x + vect1.y * vect2.y + vect1.z * vect2.z);
}
2 references
public static Vector crossProduct(Vector vect1, Vector vect2) //computes
{
    double xComp = vect1.y * vect2.z - vect1.z * vect2.y;
    double yComp = vect1.z * vect2.x - vect1.x * vect2.z;
    double zComp = vect1.x * vect2.y - vect1.y * vect2.x;
    return (new Vector(xComp, yComp, zComp));
}
3 references
public static Vector getCamHorPerpUnitVect(Camera camera) //gets the vec
{
    double vectMag = Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Po
    double zComp = camera.direction.x / vectMag;
    double xComp = -camera.direction.z / vectMag;
    return (new Vector(xComp, 0, zComp));
}

```

After commenting the code to call the checkAllCollisions function, this error went away, so I know that it is caused by my collision detection code. I first removed the Debug.WriteLine() above to remove confusion. I instead added a similar one for the boundingSpheresCheck function, which prints true before returning true and prints false before returning false. I ran the code again and saw this after the error.

```
Output
Show output from: Debug
false
true
Exception thrown: 'System.NullReferenceException' in ComputerScienceAlevelProject.dll
Object reference not set to an instance of an object.
```

As you can see, the error occurs just after the bounding spheres intersect, giving me a clue as to what went wrong. Also the error message before highlighted the dot product function. The only time where the dot product is used in the collision detection is to find the angle between two vectors meaning the problem is probably inside that function. I added this line of code to tell me what parameters were being sent into the function (prints the components of the vectors).

```
public static double angleBetweenVectors(Vector v1, Vector v2)
{
    Debug.WriteLine(v1.x.ToString() + " " + v1.y.ToString() + " " + v1.z.ToString() + " " + v2.x.ToString() + " " + v2.y.ToString() + " " + v2.z.ToString());
    return (Math.Acos(dotProduct(v1, v2)/(vectMag(v1) * vectMag(v2))));
}
```

The result after the crash was:

```
-3.137259871688456 1.5 2.5 0 -2.5 0
**v2** was null.
```

The important thing the second line, as this is where the problem occurred. The second vector being entered to the function is null.

This vector is the normal to the face. This is because I am currently only calculating face normals via the rendering from prototype 1, which means if a face is not visible, it will not have had its normal calculated, which leads to this error. I created a new function which finds the normal of a given face, also given its mesh.

```
//Face normal
6 references
public static Vector faceNormal(Face face, Mesh mesh)
{
    Vector faceCentre = Global.averagePoint(face.vertices[0], face.vertices[1], face.vertices[2], face.vertices[3]);
    Vector meshCentre = Global.averagePoint(mesh.vertices[0], mesh.vertices[1], mesh.vertices[2], mesh.vertices[3], mesh.vertices[4], mesh.vertices[5], mesh.vertices[6], mesh.vertices[7]);
    Vector perpVect = new Vector(faceCentre.x - meshCentre.x, faceCentre.y - meshCentre.y, faceCentre.z - meshCentre.z);
    return perpVect;
}
```

I used this to create a second function to run each frame on each face. This function will define the faces and normals of each mesh.

```
//Define mesh faces
void setMeshFaces(Mesh mesh)
{
    Vector v = new Vector(0, 0, 0);
    Face face1 = new Face(new Vector[] { mesh.vertices[0], mesh.vertices[1], mesh.vertices[2], mesh.vertices[3] }, v);
    Face face2 = new Face(new Vector[] { mesh.vertices[4], mesh.vertices[5], mesh.vertices[6], mesh.vertices[7] }, v);
    Face face3 = new Face(new Vector[] { mesh.vertices[0], mesh.vertices[3], mesh.vertices[7], mesh.vertices[4] }, v);
    Face face4 = new Face(new Vector[] { mesh.vertices[1], mesh.vertices[2], mesh.vertices[6], mesh.vertices[5] }, v);
    Face face5 = new Face(new Vector[] { mesh.vertices[0], mesh.vertices[1], mesh.vertices[5], mesh.vertices[4] }, v);
    Face face6 = new Face(new Vector[] { mesh.vertices[2], mesh.vertices[3], mesh.vertices[7], mesh.vertices[6] }, v);
    face1.normal = Global.faceNormal(face1, mesh);
    face2.normal = Global.faceNormal(face2, mesh);
    face3.normal = Global.faceNormal(face3, mesh);
    face4.normal = Global.faceNormal(face4, mesh);
    face5.normal = Global.faceNormal(face5, mesh);
    face6.normal = Global.faceNormal(face6, mesh);
    Face[] faces = { face1, face2, face3, face4, face5, face6 };
    mesh.faces = faces;
}
```

These last two screenshots took much longer than expected.

After these fixes, the program no longer crashes, the smaller cube moves through the first (as it would have before) and the debugger displays “false” over and over again, followed by a period of printing “true” and then back to false, with the period showing “true” representing the time for which the objects were intersecting.

I then had the program close when the objects intersected so I could have an idea of exactly when the collision was detected in real time and it seems as though the detection is relatively accurate. I will now implement collision response, which shouldn’t be difficult as this algorithm was captured in just 5 equations which I must now put into code. As a reminder, these were the equations:

$$j = \frac{-((e+1)(u_{p_1} - u_{p_2}) \cdot n)}{\left(n\left(\frac{1}{m_1} + \frac{1}{m_2}\right) + (I_1^{-1}(r_{p_1} \times n)) \times r_{p_1} + (I_2^{-1}(r_{p_2} \times n)) \times r_{p_2}\right) \cdot n} n$$

$$v_1 = u_1 + \frac{j}{m_1}$$

$$v_2 = u_2 - \frac{j}{m_2}$$

$$\omega_{final_1} = \omega_{initial_1} + I_1^{-1}(r_{p_1} \times j)$$

$$\omega_{final_2} = \omega_{initial_2} + I_2^{-1}(j \times r_{p_2})$$

Also

$$u_{p_1} = u_1 + \omega_{initial_1} \times r_{p_1}$$

$$u_{p_2} = u_2 + \omega_{initial_2} \times r_{p_2}$$

Where  $j$  is the change in momentum (impulse). For this I will need the normal to the face with which the object collided ( $n$ ), the moment of inertia tensors ( $I_1^{-1}$  and  $I_2^{-1}$ ), and the coefficient of restitution between the objects ( $e$ ). From our algorithm, it is impossible to determine with 100% accuracy which face the object collided with, but assuming the objects are moving at reasonable speeds, I can just work out which face is closest to the point of intersection and use that.

Every function above which returns a tuple with four items, the first is a bool to determine whether a collision occurred, the second is a vector to describe the location of the collision, the third and fourth are the meshes which collided. These meshes though, are in a particular order. The one with the vertex which intersects the other mesh is returned first, (except in one case which I have now changed). Meaning I know that the face where the collision happened is located on the second mesh to be returned. I need to create an algorithm to determine from a list of faces, which is closest to a given point. For this, I will first need a function to determine what the distance between a given point and face is using

$$distance = \frac{|an_1 + bn_2 + cn_3 + d|}{\sqrt{(n_1)^2 + (n_2)^2 + (n_3)^2}}$$

where  $(a, b, c)$  are the coordinates of the point and  $n_1x + n_2y + n_3z + d = 0$  is the cartesian equation of the plane.

```
public static double distBetweenPointAndFace(Vector point, Face face)
{
    Vector normal = face.normal;
    double d = -dotProduct(normal, face.vertices[0]);
    double distance = Math.Abs(dotProduct(normal, point) + d) / vectMag(normal);
    return distance;
}
```

I simplified it a bit by reusing functions which I have already created. I then need a function to find this value for a single point and a set of different faces and return only the normal who's face had the shortest distance.

I created a list of distances, added the perpendicular distance from each face to the point to the list, I then found the index of the shortest distance, finally I returned the normal of the face with this index.

```
public static Vector normalOfClosestFace(Vector point, Face[] faces)
{
    List<double> distances = new List<double>();
    for (int i = 0; i < faces.Length; i++)
    {
        double distance = distBetweenPointAndFace(point, faces[i]);
        distances.Add(distance);
    }
    int indexOfShortest = distances.IndexOf(distances.Min());
    Vector normal = faces[indexOfShortest].normal;
    return normal;
}
```

I will now be able to find the normal vector when creating the collision response algorithm. Next, I need the inertia tensors and coefficients of restitution. These should each be attributes of the rigid bodies (except the coefficient of restitution we use in the algorithm will be the average of those of our two rigid bodies). Since the equations never actually use the moment of inertia, only its inverse, I can make that an attribute of the rigid body instead.

Finding the inverse shouldn't be difficult because the matrix is diagonal. For the inverse of a diagonal  $3 \times 3$  matrix:

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{a} & 0 & 0 \\ 0 & \frac{1}{b} & 0 \\ 0 & 0 & \frac{1}{c} \end{pmatrix}$$

In our particular case:

$$\begin{aligned} I^{-1} &= \begin{pmatrix} \frac{m}{12}(w^2 + h^2) & 0 & 0 \\ 0 & \frac{m}{12}(l^2 + w^2) & 0 \\ 0 & 0 & \frac{m}{12}(l^2 + h^2) \end{pmatrix}^{-1} \\ &= \begin{pmatrix} \frac{12}{m(w^2 + h^2)} & 0 & 0 \\ 0 & \frac{12}{m(l^2 + w^2)} & 0 \\ 0 & 0 & \frac{12}{m(l^2 + h^2)} \end{pmatrix} \\ &= \frac{12}{m} \begin{pmatrix} \frac{1}{w^2 + h^2} & 0 & 0 \\ 0 & \frac{1}{l^2 + w^2} & 0 \\ 0 & 0 & \frac{1}{l^2 + h^2} \end{pmatrix} \end{aligned}$$

Where  $m$  is the cuboids mass and  $w, h$  and  $l$  are its dimensions.

Instead of creating a new class to store matrices, I will just use a vector because this matrix is

diagonal. I will represent  $\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$  as the vector  $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$  but it is important to understand

that this is not actually a vector and multiplying it with another vector  $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$  will result in  $\begin{pmatrix} ax \\ by \\ cz \end{pmatrix}$  because this is really matrix multiplication, I am just representing it in a simpler way as that is what the matrix being diagonal allows me to do.

The coefficient of restitution will just be arbitrarily chosen (the user will later have the option to set these themselves).

I updated the rigid body class to include coefficient of restitution (cOr) and the inverse of the moment of inertia tensor (inertiaInv).

```
internal class RigidBody
{
    public double mass;
    public Vector centreOfMass;
    public Vector velocity;
    public Vector acceleration;
    public Vector angularVelocity;
    public Vector angularAcceleration;
    public Vector force;
    public double CoR;
    public Vector inertiaInv;
    2 references
    public RigidBody(double mass, Vector centreOfMass, Vector velocity, Vector acceleration, Vector angularVelocity, Vector angularAcceleration, Vector force, double CoR, Vector inertiaInv)
    {
        this.mass = mass;
        this.centreOfMass = centreOfMass;
        this.velocity = velocity;
        this.acceleration = acceleration;
        this.angularVelocity = angularVelocity;
        this.angularAcceleration = angularAcceleration;
        this.force = force;
        this.CoR = CoR;
        this.inertiaInv = inertiaInv;
    }
}
```

I then added dimensions to the Mesh class. I later want the user to be able to just enter the dimensions and have the coordinates of the vertices be calculated, so I will allow the dimensions to just be added manually.

```
internal class Mesh
{
    public Vector[] vertices;
    public Face[] faces;
    public Vector dimensions;
    3 references
    public Mesh(Vector[] vertices, Vector dimensions)
    {
        this.vertices = vertices;
        this.dimensions = dimensions;
    }
}
```

The code to create the objects has been altered accordingly.

```
private void Form1_Load(object sender, EventArgs e)
{
    Vector v = new Vector(0, 0, 0);
    //Define cube 1
    Vector cubeVertex1 = new Vector(0, -1, 0);
    Vector cubeVertex2 = new Vector(0, -1, 1);
    Vector cubeVertex3 = new Vector(1, -1, 1);
    Vector cubeVertex4 = new Vector(1, -1, 0);
    Vector cubeVertex5 = new Vector(0, 0, 0);
    Vector cubeVertex6 = new Vector(0, 0, 1);
    Vector cubeVertex7 = new Vector(1, 0, 1);
    Vector cubeVertex8 = new Vector(1, 0, 0);

    Vector[] vertices1 = { cubeVertex1, cubeVertex2, cubeVertex3, cubeVertex4, cubeVertex5, cubeVertex6, cubeVertex7, cubeVertex8 };
    Vector dimensions1 = new Vector(1, 1, 1);
    Mesh cube1Mesh = new Mesh(vertices1, dimensions1);
    double m1 = 1;
    double l1 = dimensions1.x; double w1 = dimensions1.y; double h1 = dimensions1.z;
    Vector tensor1 = Global.vectorTimesScalar(new Vector(1 / (m1 * w1 + h1 * h1), 1 / (l1 * l1 + w1 * w1), 1 / (l1 * l1 + h1 * h1)), 12 / m1);
    Entity cube1 = new Entity(cube1Mesh, new RigidBody(m1, Global.averagePoint(vertices1), v, v, v, new Vector(1, 0 * m1, 0), 0.5, tensor1));

    //Define cube 2
    cubeVertex1 = new Vector(5, -2.5, -2.5);
    cubeVertex2 = new Vector(5, -2.5, 2.5);
    cubeVertex3 = new Vector(10, -2.5, 2.5);
    cubeVertex4 = new Vector(10, -2.5, -2.5);
    cubeVertex5 = new Vector(5, 2.5, -2.5);
    cubeVertex6 = new Vector(5, 2.5, 2.5);
    cubeVertex7 = new Vector(10, 2.5, 2.5);
    cubeVertex8 = new Vector(10, 2.5, -2.5);

    Vector[] vertices2 = { cubeVertex1, cubeVertex2, cubeVertex3, cubeVertex4, cubeVertex5, cubeVertex6, cubeVertex7, cubeVertex8 };
    Vector dimensions2 = new Vector(5, 5, 5);
    Mesh cube2Mesh = new Mesh(vertices2, dimensions2);
    double m2 = 125;
    double l2 = dimensions2.x; double w2 = dimensions2.y; double h2 = dimensions2.z;
    Vector tensor2 = Global.vectorTimesScalar(new Vector(1 / (m2 * w2 + h2 * h2), 1 / (l2 * l2 + w2 * w2), 1 / (l2 * l2 + h2 * h2)), 12 / m2);
    Entity cube2 = new Entity(cube2Mesh, new RigidBody(m2, Global.averagePoint(vertices2), v, v, v, new Vector(0, 0 * m2, 0), 0.5, tensor2));

    Global.Entities.Add(cube1); //Whenever a new physics object is created add it to this list
    Global.Entities.Add(cube2);
}
```

This section of code is more messy than it needs to be, with all the repeated code, but this will later be addressed in prototype 3 when I allow the user to create their own objects.

I am now ready to implement the actual collision response algorithm using the previously derived equations. Because the first equation is so large, I will break it down into smaller parts to make it more readable.

$$j = \frac{-((e + 1)(u_{p_1} - u_{p_2}) \cdot n)}{\left(n\left(\frac{1}{m_1} + \frac{1}{m_2}\right) + \left(I_1^{-1}(r_{p_1} \times n)\right) \times r_{p_1} + \left(I_2^{-1}(r_{p_2} \times n)\right) \times r_{p_2}\right) \cdot n}$$

Where

$$u_{p_1} = u_1 + \omega_{initial1} \times r_{p_1}$$

$$u_{p_2} = u_2 + \omega_{initial2} \times r_{p_2}$$

Then:

$$v_1 = u_1 + \frac{j}{m_1}$$

$$v_2 = u_2 - \frac{j}{m_2}$$

$$\omega_{final1} = \omega_{initial1} + I_1^{-1}(r_{p_1} \times j)$$

$$\omega_{final2} = \omega_{initial2} + I_2^{-1}(j \times r_{p_2})$$

```
//Collision Response
1 reference
public static void collisionResponse(Entity obj1, Entity obj2, Vector point, Vector n)
{
    double e = (obj1.rigidBody.CoR + obj2.rigidBody.CoR)/2;
    double m1 = obj1.rigidBody.mass; double m2 = obj2.rigidBody.mass;
    Vector u1 = obj1.rigidBody.velocity; Vector u2 = obj2.rigidBody.velocity;
    Vector omegaInitial1 = obj1.rigidBody.angularVelocity; Vector omegaInitial2 = obj2.rigidBody.angularVelocity;
    Vector rp1 = vectBetweenPoints(obj1.rigidBody.centreOfMass, point); Vector rp2 = vectBetweenPoints(obj2.rigidBody.centreOfMass, point);
    Vector up1 = vectorAddVector(u1, crossProduct(omegaInitial1, rp1)); Vector up2 = vectorAddVector(u2, crossProduct(omegaInitial2, rp2));
    double part1 = -((e + 1)*dotProduct(vectBetweenPoints(up1, up2), n));
    Vector part2 = vectorTimesScalar(n, 1 / m1 + 1 / m2);
    Vector v = crossProduct(rp1, n); //v is to be multiplied by the tensor
    Vector tensor1 = obj1.rigidBody.inertiaInv;
    Vector part3 = crossProduct(new Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z), rp1);
    Vector tensor2 = obj2.rigidBody.inertiaInv;
    v = crossProduct(rp2, n);
    Vector part4 = crossProduct(new Vector(tensor2.x * v.x, tensor2.y * v.y, tensor2.z * v.z), rp2);
    Vector j = vectorTimesScalar(n, part1 / dotProduct(vectorAddVector(part2, part3), part4), n));
    Vector v1 = vectorAddVector(u1, vectorTimesScalar(j, 1 / m1));
    Vector v2 = vectorAddVector(u2, vectorTimesScalar(j, -1 / m2));
    v = crossProduct(rp1, j);
    Vector omegaFinal1 = vectorAddVector(omegaInitial1, new Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z));
    v = crossProduct(j, rp2);
    Vector omegaFinal2 = vectorAddVector(omegaInitial2, new Vector(tensor2.x * v.x, tensor2.y * v.y, tensor2.z * v.z));

    obj1.rigidBody.velocity = v1;
    obj2.rigidBody.velocity = v2;
    obj1.rigidBody.angularVelocity = omegaFinal1;
    obj2.rigidBody.angularVelocity = omegaFinal2;
}
```

To test that this procedure actually works I will call it each time a collision is detected. Also the order in which the objects are passed in is important, object 2 is assumed to be the one

who's face has the normal for the collision, in other words, object 1 had a vertex which intersected object 2.

For this to work, I had to change all of the collision detection algorithms to return the colliding entities instead of meshes, though this was just as simple as changing a few key words.

```
//Collision Detection
Tuple<bool, Vector, Entity, Entity> check = Global.checkAllCollisions();
if (check.Item1 == true)
{
    //Collision Response
    Vector normal = Global.normalOfClosestFace(check.Item2, check.Item4.mesh.faces);
    Global.collisionResponse(check.Item3, check.Item4, check.Item2, normal);
}
```

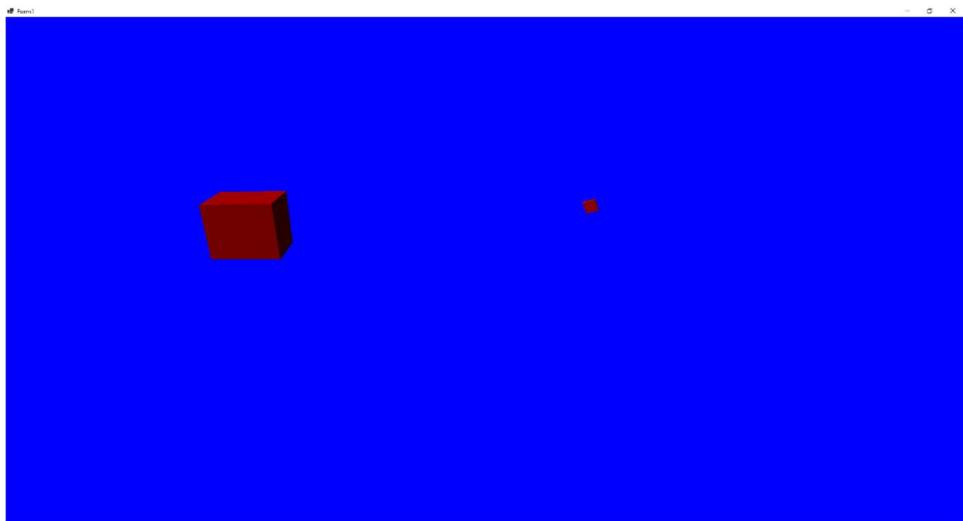
After implementing this, the objects still just move through each as they did before.

**Video22.** I noticed that I was not using the unit normal vector (its length was not 1), after changing this, the issue still persisted. I used Debug.WriteLine to see what the collision response function thought was the normal. I printed  $(-1, 0, 0)$  which was expected. I then printed the vector  $j$ , which gave me this:

The magnitude seems reasonable, but it keeps changing direction. To avoid this, I took the absolute value of the magnitude of  $j$  before multiplying it by  $n$ . The code now looks like this:

```
//Collision Response
1 reference
public static void collisionResponse(Entity obj1, Entity obj2, Vector point, Vector normal)
{
    Vector n = vectorTimesScalar(normal, 1 / vectMag(normal));
    double e = (obj1.rigidBody.CoR + obj2.rigidBody.CoR)/2;
    double m1 = obj1.rigidBody.mass; double m2 = obj2.rigidBody.mass;
    Vector u1 = obj1.rigidBody.velocity; Vector u2 = obj2.rigidBody.velocity;
    Vector omegaInitial1 = obj1.rigidBody.angularVelocity; Vector omegaInitial2 = obj2.rigidBody.angularVelocity;
    Vector rpl = vectBetweenPoints(obj1.rigidBody.centreOfMass, point); Vector rp2 = vectBetweenPoints(obj2.rigidBody.centreOfMass, point);
    Vector up1 = vectorAddVector(u1, crossProduct(omegaInitial1, rpl)); Vector up2 = vectorAddVector(u2, crossProduct(omegaInitial2, rp2));
    double part1 = -((e + 1)*dotProduct(vectBetweenPoints(up2, up1), n));
    Vector part2 = vectorTimesScalar(n, 1 / m1 + 1 / m2);
    Vector v = crossProduct(rpl, n); //v is to be multiplied by the tensor
    Vector tensor1 = obj1.rigidBody.inertiaInv;
    Vector part3 = crossProduct(new Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z), rp1);
    Vector tensor2 = obj2.rigidBody.inertiaInv;
    v = crossProduct(rp2, n);
    Vector part4 = crossProduct(new Vector(tensor2.x * v.x, tensor2.y * v.y, tensor2.z * v.z), rp2);
    Vector j = vectorTimesScalar(n, Math.Abs(part1 / dotProduct(vectorAddVector(vectorAddVector(part2, part3), part4), n)));
    Vector v1 = vectorAddVector(u1, vectorTimesScalar(j, 1 / m1));
    Vector v2 = vectorAddVector(u2, vectorTimesScalar(j, -1 / m2));
    v = crossProduct(rpl, j);
    Vector omegaFinal1 = vectorAddVector(omegaInitial1, new Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z));
    v = crossProduct(j, rp2);
    Vector omegaFinal2 = vectorAddVector(omegaInitial2, new Vector(tensor2.x * v.x, tensor2.y * v.y, tensor2.z * v.z));
    Debug.WriteLine(j.x + " " + j.y + " " + j.z);
    Debug.WriteLine("\n");
    obj1.rigidBody.velocity = v1;
    obj2.rigidBody.velocity = v2;
    obj1.rigidBody.angularVelocity = omegaFinal1;
    obj2.rigidBody.angularVelocity = omegaFinal2;
}
```

The result of this change is shown in [video23](#) and here:



The collision response is certainly able to push the object back now, but it is doing so too quickly. The smaller cube moves faster at the end than at the beginning. This is not what was expected. Looking at the debugger (still printing  $j$  (change in momentum)), we see:

```
-0.4985243678711015 0 0

-0.4985243678711014 0 0

-1.5102329813279989 8.760877874980172E-06 -8.76087787477897E-06

-4.530698943982822 2.6282633624933704E-05 -2.6282633624330094E-05

-15.114115367779888 0.0005539653715464574 -0.0005539133640231309

-45.34234610278447 0.001661896114619023 -0.0016617400920490456
```

The problem seems, when the impulse is applied, the objects are still briefly intersecting, causing it to be applied again, and so on, meaning it compounds until the objects are moving much more quickly than they should be. To fix this, I must ensure that the impulse is being applied only once per collision.

```
public static List<Tuple<Vector, Entity, Entity>> checkAllCollisions()
{
    List<Tuple<Vector, Entity, Entity>> collisions = new List<Tuple<Vector, Entity, Entity>>();
    for (int i = 0; i < Entities.Count - 1; i++)
    {
        for (int j = 0; j < Entities.Count - i - 1; j++)
        {
            Tuple<bool, Vector, Entity, Entity> check = checkCollision(Entities[i], Entities[i + j + 1]);
            if (check.Item1 == true)
            {
                Tuple<Vector, Entity, Entity> collision = new Tuple<Vector, Entity, Entity>(check.Item2, check.Item3, check.Item4);
                collisions.Add(collision);
                Debug.WriteLine(collisions[0].Item1.ToString());
            }
        }
    }
    return collisions;
}
```

I noticed that using the current system, only one collision is allowed to occur each frame, I allowed the `checkAllCollisions` function to return an array of Tuples (each describing a

collision) instead of just one. I also think this may make it easier to keep track of collisions in order to solve this problem.

I will create a list of Tuples called “collisionFlags”. If a collision occurs and it does not have a corresponding flag, I will add the flag to the list and perform the collision response. If the flag is in the list but is set to false, I will set it to true and then perform the collision response. If it is in the list and is set to true, I will not perform the collision response. Each frame I will iterate through the list. If I find any flags which have not had a collision during this frame, I will set the flag to false. This is the list of flags:

```
public static List<Tuple<bool, Entity, Entity>> collisionFlags = new List<Tuple<bool, Entity, Entity>>();
```

This is the code to check/update them:

```
//Checking collision Flags
for (int j = 0; j < check.Count(); j++)
{
    bool found = false;
    for (int k = 0; k < Global.collisionFlags.Count(); k++)
    {
        if ((Global.collisionFlags[k].Item2 == check[j].Item2 && Global.collisionFlags[k].Item3 == check[j].Item3)
            || (Global.collisionFlags[k].Item2 == check[k].Item3 && Global.collisionFlags[k].Item3 == check[j].Item2))
        {
            //There is a match
            found = true;
            if (Global.collisionFlags[k].Item1 == false)
            {
                Global.collisionFlags[k] = new Tuple<bool, Entity, Entity> (true, Global.collisionFlags[k].Item2, Global.collisionFlags[k].Item3);
                //Set the flag to true, do collision response.
            }//else, do nothing
        }
    }
    if (found == false)
    {
        //check[j] is in check, but does not exist as a flag.
        Global.collisionFlags.Add(new Tuple<bool, Entity, Entity> (true, check[j].Item2, check[j].Item3));
        //Do collision response
    }
}

//Checking if there are any flags which need to be updated from true to false.
for (int k = 0; k < Global.collisionFlags.Count(); k++)
{
    if (Global.collisionFlags[k].Item1)
    {
        bool found = false;
        for (int j = 0; j < check.Count(); j++)
        {
            if ((Global.collisionFlags[k].Item2 == check[j].Item2 && Global.collisionFlags[k].Item3 == check[j].Item3)
                || (Global.collisionFlags[k].Item2 == check[k].Item3 && Global.collisionFlags[k].Item3 == check[j].Item2))
            {
                found = true;
            }
        }
        if (found == false)
        {
            //There was a flag set to true, for which there was no collision this frame, set the flag to false.
            Global.collisionFlags[k] = new Tuple<bool, Entity, Entity> (false, Global.collisionFlags[k].Item2, Global.collisionFlags[k].Item3);
            //Do nothing
        }
    }
}
```

Looking at this, it would be a good idea to create a procedure called `doCollisionResponse` to be called whenever collision response here is needed.

```
void doCollisionResponse(Tuple<Vector, Entity, Entity> check)
{
    Vector normal = Global.normalOfClosestFace(check.Item1, check.Item3.mesh.faces);
    Global.collisionResponse(check.Item2, check.Item3, check.Item1, normal);
}
```

I then changed the comments into calls to this function. This now seems to succeed at the task of ensuring that impulse is applied only once for a single collision. As shown here:

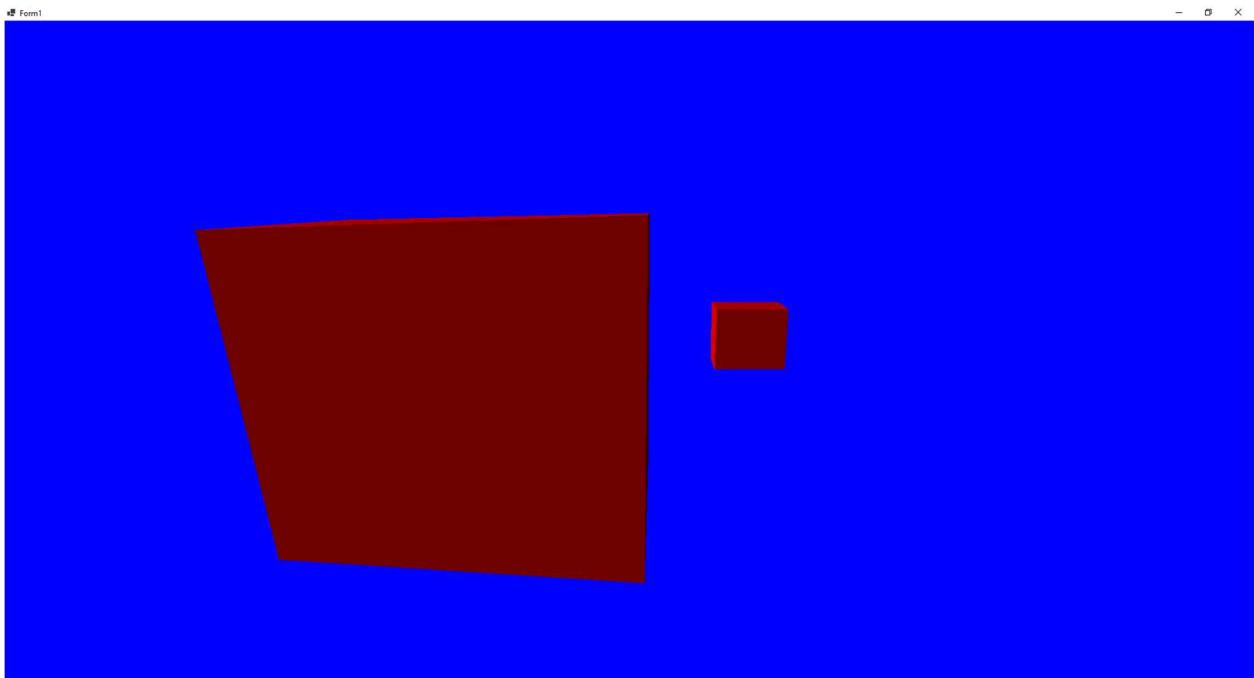


and in [video24](#), the cube no longer moves to the right at great speeds, impulse is now being applied only one time. It is now too weak however as the small cube now moves through the large cube.

I thought this may have had something to do with the collision acting on only one of the four corners (hence the rotation), to check this I manually set the coordinates of the collision to be at the centre of the face:

```
public static void collisionResponse(Entity obj1, Entity obj2, Vector point, Vector normal)
{
    Vector point = new Vector(5, -0.5, 0.5);
```

and this actually worked!



This is also shown in [Video25](#). Of course, I now need to implement this generally, so that if there is every a collision involving multiple contact points, the average of these is used as the point of collision.

I updated the “checkMeshAgainstMesh” function to, instead of returning the point, add the point to a list “points” and to then return the average of these points at the end.

```
public static Tuple<bool, Vector, Entity, Entity> checkMeshAgainstMesh(Entity obj1, Entity obj2)
{
    List<Vector> points = new List<Vector>();
    for (int i = 0; i < obj1.mesh.vertices.Length; i++)
    {
        if (checkPointAgainstMesh(obj1.mesh.vertices[i], obj2) == true)
        {
            points.Add(obj1.mesh.vertices[i]);
        }
    }

    if (points.Count == 0)
    {
        return new Tuple<bool, Vector, Entity, Entity>(false, new Vector(0, 0, 0), obj1, obj2);
    }
    else
    {
        return new Tuple<bool, Vector, Entity, Entity>(true, averagePoint(points.ToArray()), obj1, obj2);
    }
}
```

This now works exactly the same as shown in the previous image/video, except it now works in general. If you watched the video, you will have noticed that there are many instances where the smaller cube renderes behind the larger one, even though it should be in front the whole time. This is due to the algorithm I used to determine the order in which objects are rendered. I did this in order form distance to the camera, which takes into account the distances left, right, up and down. This issue had not shown up before as the objects were sufficiently far apart that this made no difference. To resolve this, I will alter this algorithm from before, so instead of finding the distance from the camera to the point, it will instead find the vector, which will then be projected in the direction of the camera’s direction vector. The length of this new vector is what will then be used.

I created a function to project a vector,  $u$ , in the direction of a unit vector,  $v$ .

```
public static Vector projectVector(Vector u, Vector v) //v is a unit vector
{
    return vectorTimesScalar(v, dotProduct(u, v));
```

I then used this to alter the renderAll function from prototype 1.

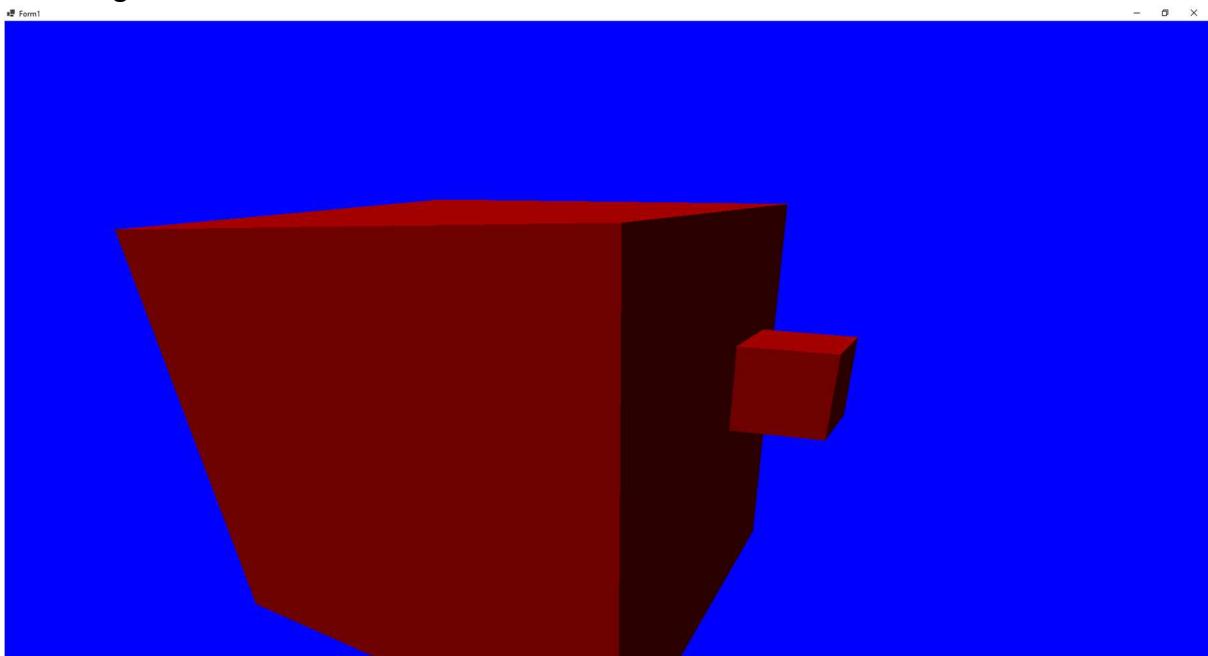
```
void renderAll(List<Entity> entities, Camera camera)
{
    entities = entities.OrderByDescending(x => Global.vectMag(Global.projectVector(Global.vectBetweenPoints(Global.closestPointCoords(x.mesh, camera), camera.position), camera.direction))).ToList();
    for (int i = 0; i < entities.Count; i++)
    {
        setMeshFaces(entities[i].mesh);
        renderMesh(entities[i].mesh, camera);
    }
}
```

This still didn’t make much difference since we are only considering the closest point of each object. I decided to instead use the centre of each object, by replacing `Global.closestPointCoords(x.mesh, camera)` by `x.rigidBody.centreOfMass`

```
entities = entities.OrderByDescending(x => Global.vectMag(Global.projectVector(Global.vectBetweenPoints(x.rigidBody.centreOfMass, camera.position), camera.direction))).ToList();
```

This has now fixed the problem, and objects render in the correct order. The “closestPoint” and “closestPointCoords” functions are still being used elsewhere so I cannot remove them from the code.

This image



shows that close together objects of different sizes now render correctly.

I will now test whether rotated objects experience collisions as expected. The smaller cube should bounce away from the larger cube, the larger cube will also move away but too slowly to notice. Both will have rotational effects applied (though this again will hardly be noticeable on the larger cube). [Video26](#) demonstrates this working as intended, as also shown in this image here:



The last thing I need to implement in prototype is a normal reaction force. All this means is that when two objects are intersecting, the component of any resultant forces acting in the direction of the normal is zero (this allows an object to stay at rest against another object, e.g., on the ground).

The following code finds “reaction1” and “reaction2” which are the normal reaction forces, these are then added to the resultant forces of objects 1 and 2 respectively.

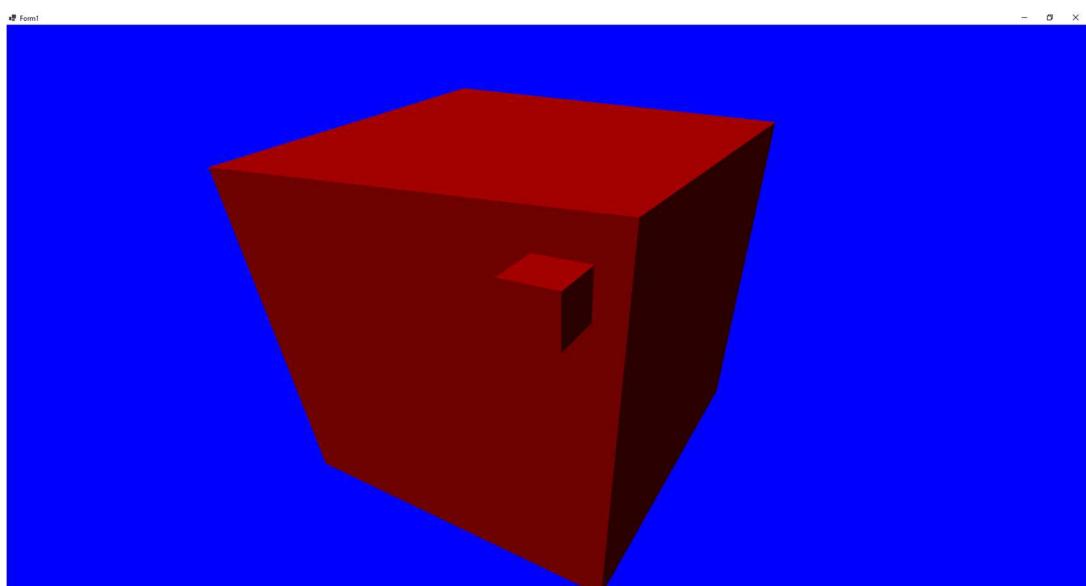
```
//Normal Reaction Force:  
Vector reaction1 = vectorTimesScalar(projectVector(obj1.rigidBody.force, n), -1);  
obj1.rigidBody.force = vectorAddVector(obj1.rigidBody.force, reaction1);  
Vector reaction2 = vectorTimesScalar(projectVector(obj2.rigidBody.force, n), -1);  
obj2.rigidBody.force = vectorAddVector(obj2.rigidBody.force, reaction2);
```

There are currently no forces involved so there is no observed difference. To change this, instead of setting the smaller cubes velocity to (1,0,0), I instead set its resultant force to (1,0,0). The small cube now accelerates towards the larger cube, it then bounces off the larger cube and continues moving away at a constant speed.

It should still be accelerating towards the larger cube, the reason why it isn’t is because the normal reaction force is applied, but it is never removed. I want to add the reaction force when they are intersecting and remove it when they stop intersecting. It would be even better to remove and add it back each frame allowing it to change over time. To simplify matters, I will create a new function to return the values for reaction1 and reaction2.

```
public static Tuple<Vector, Vector> getNormalReactions(Entity obj1, Entity obj2, Vector normal)  
{  
    Vector n = vectorTimesScalar(normal, 1 / vectMag(normal));  
    Vector reaction1 = vectorTimesScalar(projectVector(obj1.rigidBody.force, n), -1);  
    Vector reaction2 = vectorTimesScalar(projectVector(obj2.rigidBody.force, n), -1);  
    return (new Tuple<Vector, Vector>(reaction1, reaction2));  
}
```

This function has now been created but I am not yet using it anywhere, before I do, I will reposition the cubes so that their centres were aligned, this will mean that neither cube will rotate after the collisions meaning the smaller cube can “rest” on the larger one, this will reveal why the normal reaction force is necessary. I also decreased the coefficient of restitution to 0.2 so kinetic energy is lost between collisions. [Video27](#) shows the smaller cube bouncing from the larger one repeatedly until it reaches rest. At this point, it begins to move through the larger cube. This is also shown here:

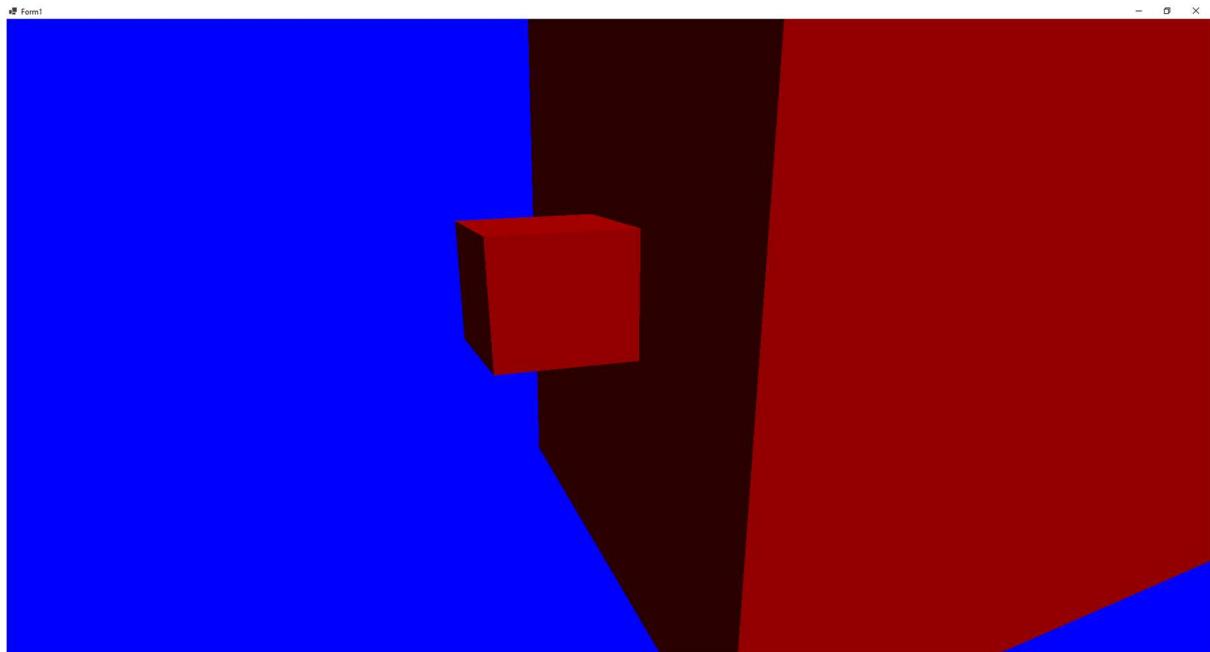


Inside the “physics()” function, I need to add the normal reaction force before the acceleration is determined and then take it away immediately after (so that it does compound over multiple frames). First though, I need to determine what this reaction force should be. The vector “reaction” is set to (0,0,0) by default, meaning if there should be no normal reaction force, then nothing will happen. I check if the current entity being studied “Global.Entities[i]” is equal to any of the entries in the “check” list, this is done by iterating using a for loop. If it is, the appropriate normal reaction is found using the function just created. After the loop, this vector is added to the force, the acceleration is calculated, and the vector is subtracted again.

The code:

```
//Find with which object and normal (if any) this object collided with each frame and apply normal reaction.
Vector reaction = new Vector(0, 0, 0);
for(int j = 0; j < check.Count; j++)
{
    if (Global.Entities[i] == check[j].Item2)
    {
        reaction = Global.getNormalReactions(check[j].Item2, check[j].Item3, Global.normalOfClosestFace(check[j].Item1, check[j].Item3.mesh.faces)).Item1;
    }
    if (Global.Entities[i] == check[j].Item3)
    {
        reaction = Global.getNormalReactions(check[j].Item2, check[j].Item3, Global.normalOfClosestFace(check[j].Item1, check[j].Item3.mesh.faces)).Item2;
    }
}
Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, reaction);
Global.updateAcceleration(Global.Entities[i]);
Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, Global.vectorTimesScalar(reaction, -1));
```

After Implementing this, the smaller cube now rests on the larger one since the force applied to it is cancelled out by the normal reaction as shown here.



This is also shown by [Video28](#).

## Testing Prototype 2

As discussed in the test plan, I will alter the source code to create different objects with different attributes to test different scenarios.

Test No.	Input Data	Expected Output	Actual Output	Success?
9	A single cube, centred at the origin. No resultant force set.	Cube should accelerate down due to gravity at $9.81\text{ ms}^{-2}$	Accelerates down due to gravity at what seems to be $\sim 9.81\text{ms}^{-2}$ by eye, but I have no way to precisely measure this. Seems to be working as intended.	Yes
10	A single cube centred at the origin, with an upward force of 9.81	Cube should remain stationary (since upward force balances with gravity).	Cube remains stationary.	Yes
11	A single cube centred at the origin, with an upward force of 9.81 and angular velocity of (1,1,1)	Cube should rotate about the line $y = z = x$ with a uniform angular speed.	Cube rotates about the line $y = z = x$ with a uniform angular speed (by eye).	Yes
12	A single cube centred at the origin, with an upward force of 9.81 and angular acceleration of (1,1,1)	Cube should rotate about the line $y = x = z$ with an increasing angular speed.	Cube rotates about the line $y = x = z$ with an increasing angular speed.	Yes
13	Two cubes, centred at (0,0,2) and (0,0,-2) moving with respective velocities (0,0,-1) and (0,0,1) (towards each other). (Each will have a coefficient of restitution of 1 so the collision is elastic).	Cubes should move towards each other, and instantaneously separate away with the same speed as they came together.	Cubes move towards each other, and instantaneously separate away with the same speed as they came together.	Yes
14	Two cubes, centred at (0,0,2) and (0,0,-2) moving with respective	Cubes should move towards each other, and instantaneously separate away	Cubes move towards each other, and instantaneously separate away	Yes

	velocities (0,0,-1) and (0,0,1) (towards each other). (Each will have a coefficient of restitution of 0.5 so the collision is inelastic).	with half the speed as they came together.	with half the speed as they came together.	
15	Two cubes, centred at (0,0,2) and (0,0,-2) moving with respective accelerations (0,0,-1) and (0,0,1) (towards each other). (Each will have a coefficient of restitution of 1)	Cubes should accelerate towards each other. They should collide and rebound away from each other, slowing down until they move towards each other again and rebound again to the same point, this repeats indefinitely.	Cubes accelerate towards each other, rebound to the same distance, move towards each other again, and so on.	Yes
16	Same as above with Coefficient of Restitution of 0.5	Same as above, but after each collision, they lose speed and don't make it as far away.	Cubes accelerate towards each other, rebound to a shorter distance, move towards each other again, and so on.	Yes
17	Two cubes centred at (0,0,2) and (0.2,0.2,-2) and velocities (0,0,-1) and (0,0,1), Coefficient of Restitution of 1.	The cubes should move towards each other with constant speed. When they collide, they will each rebound and rotate (because they are not aligned to the line of centres).	The cubes move towards each other, they rebound and rotate when they collide.	Yes

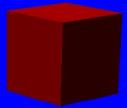
Evidence for each test:

Test -No.	Evidence

9



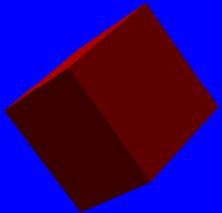
10

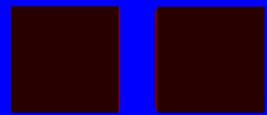


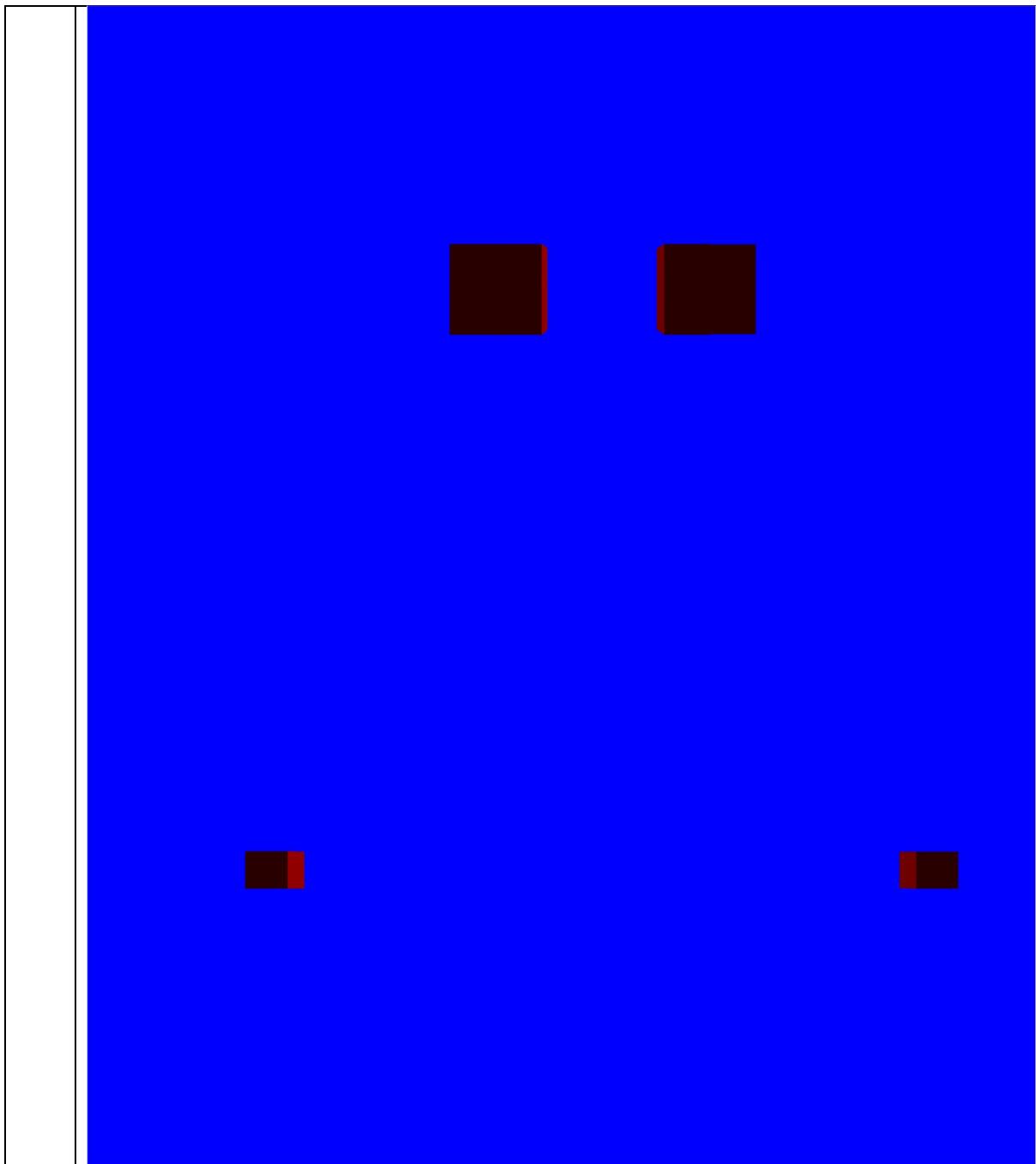
11

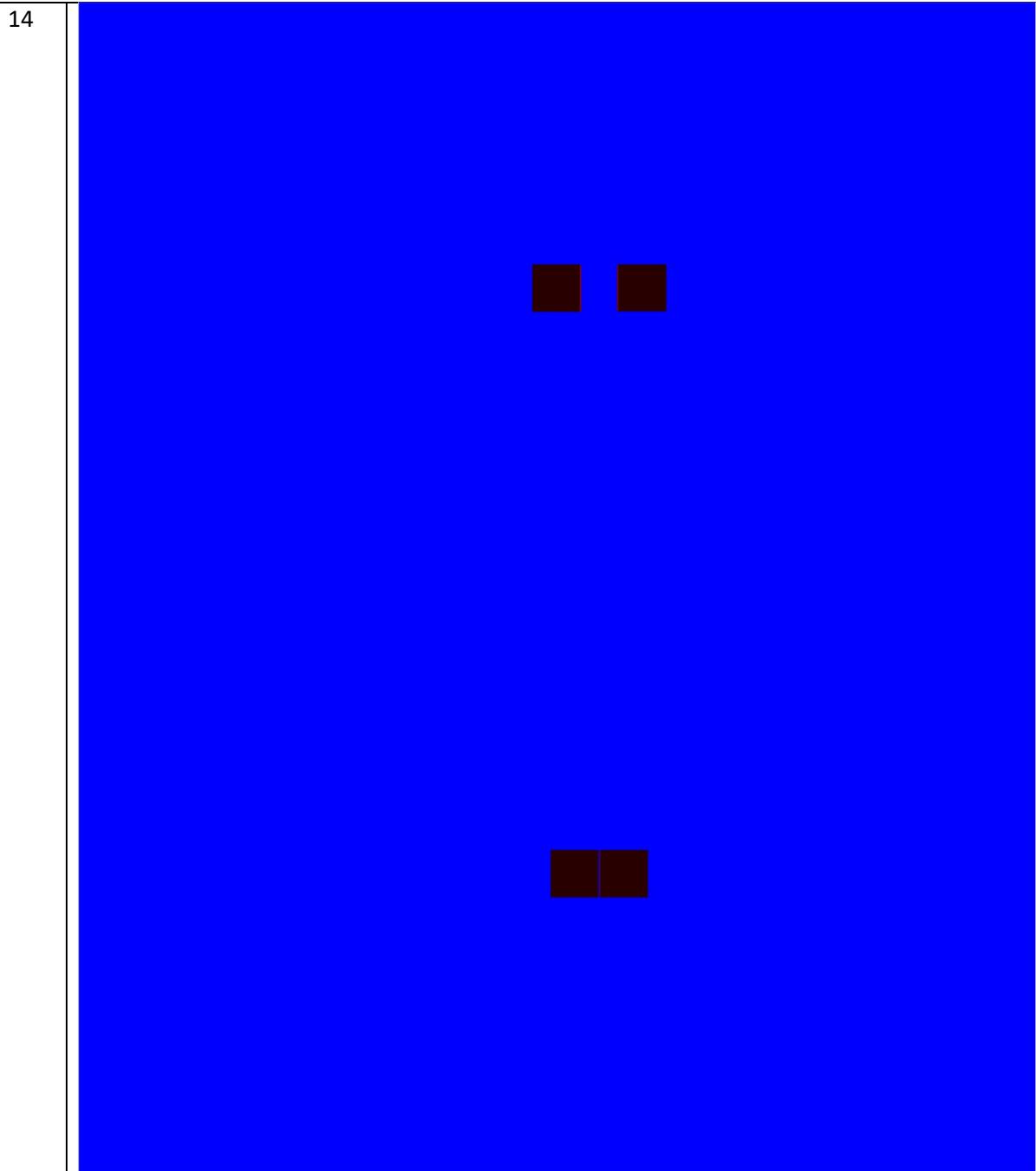


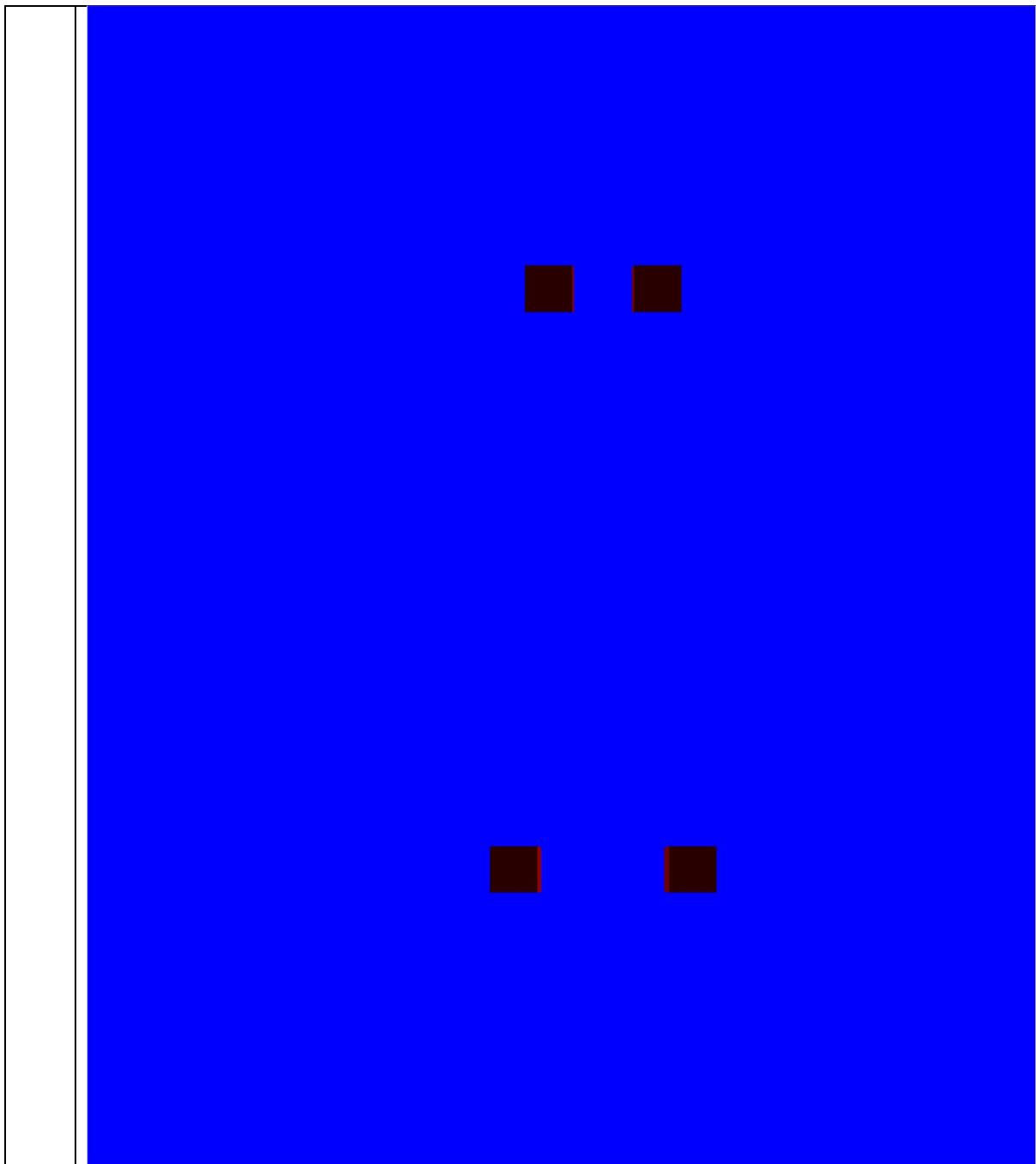
12

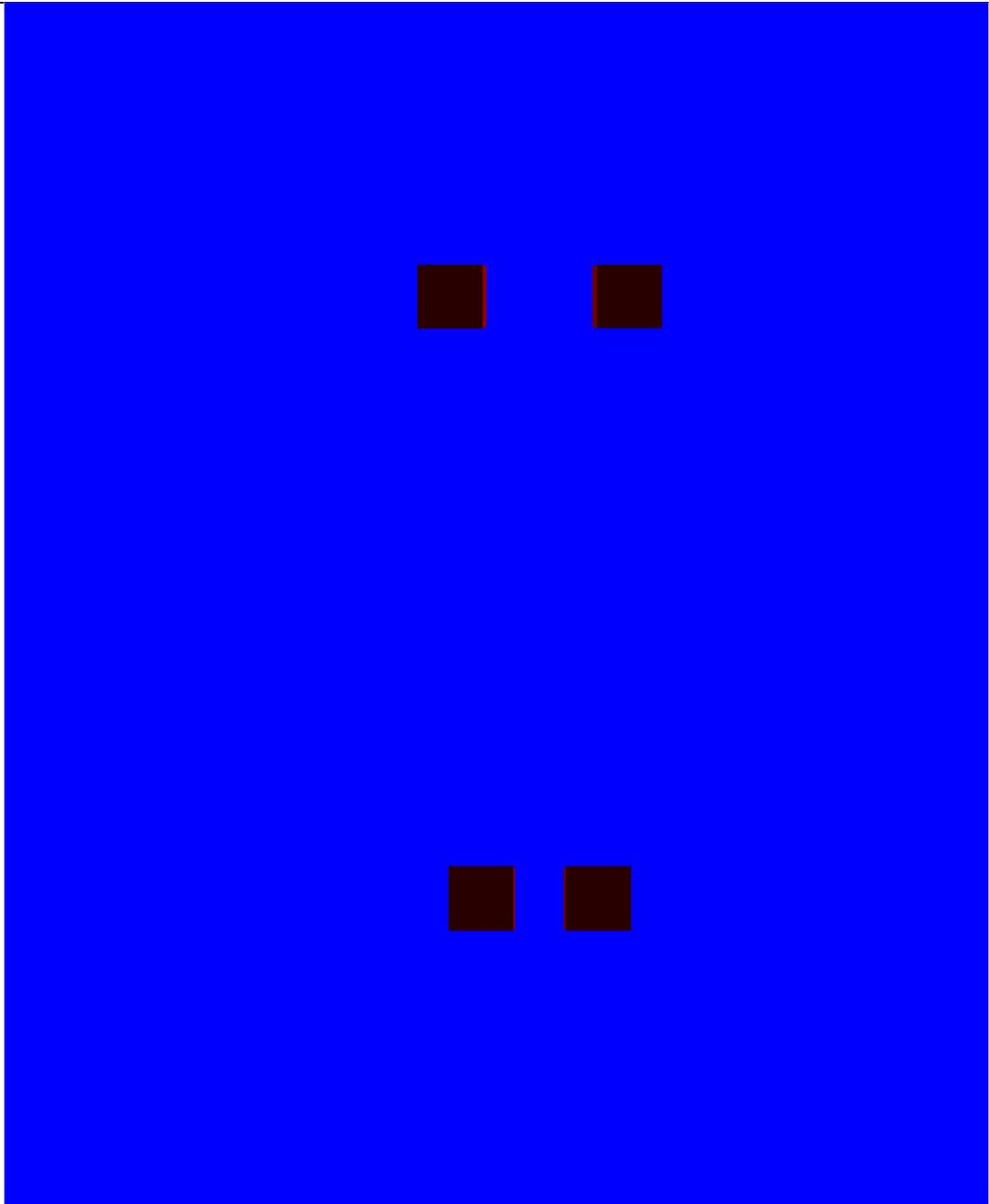


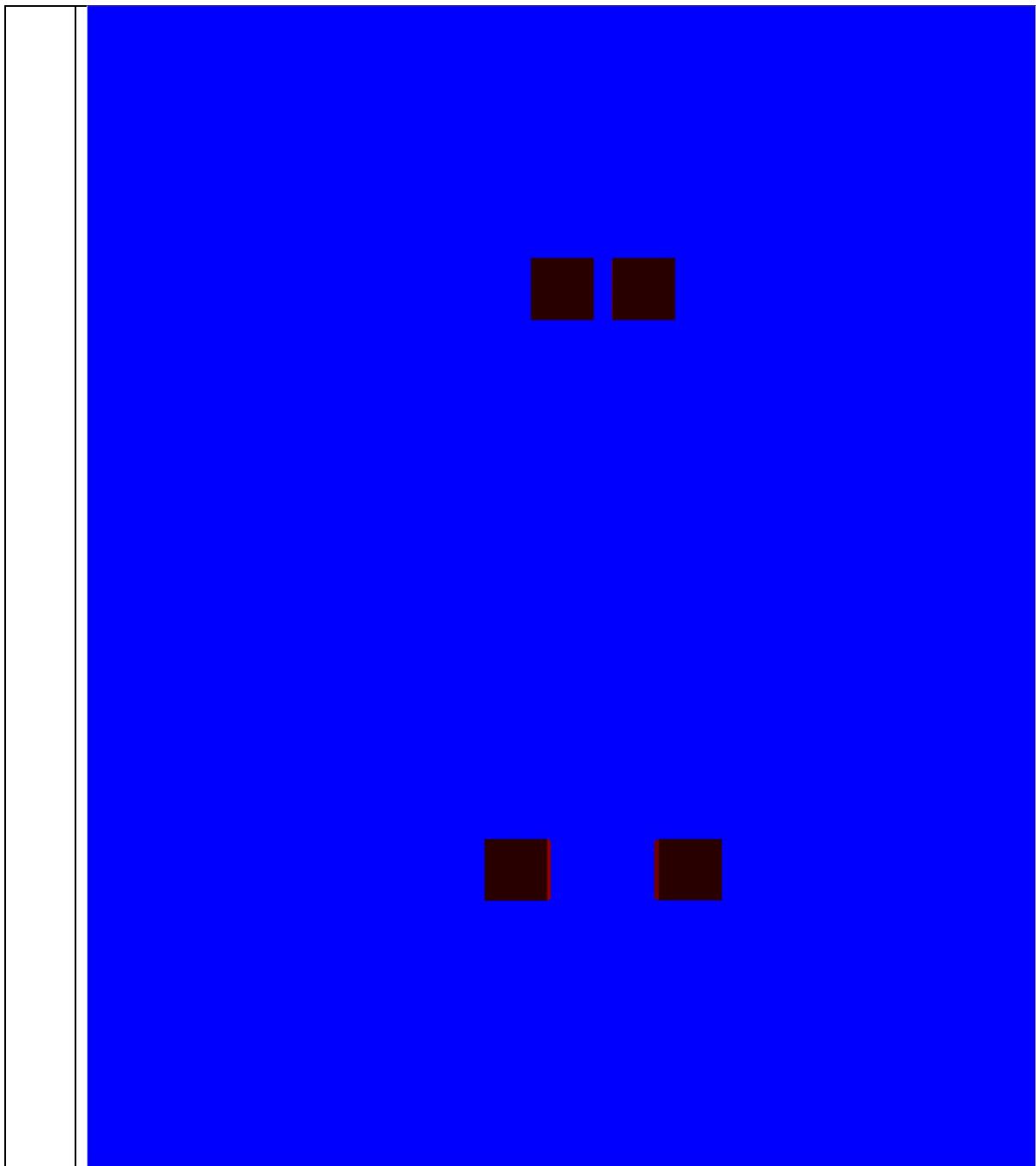


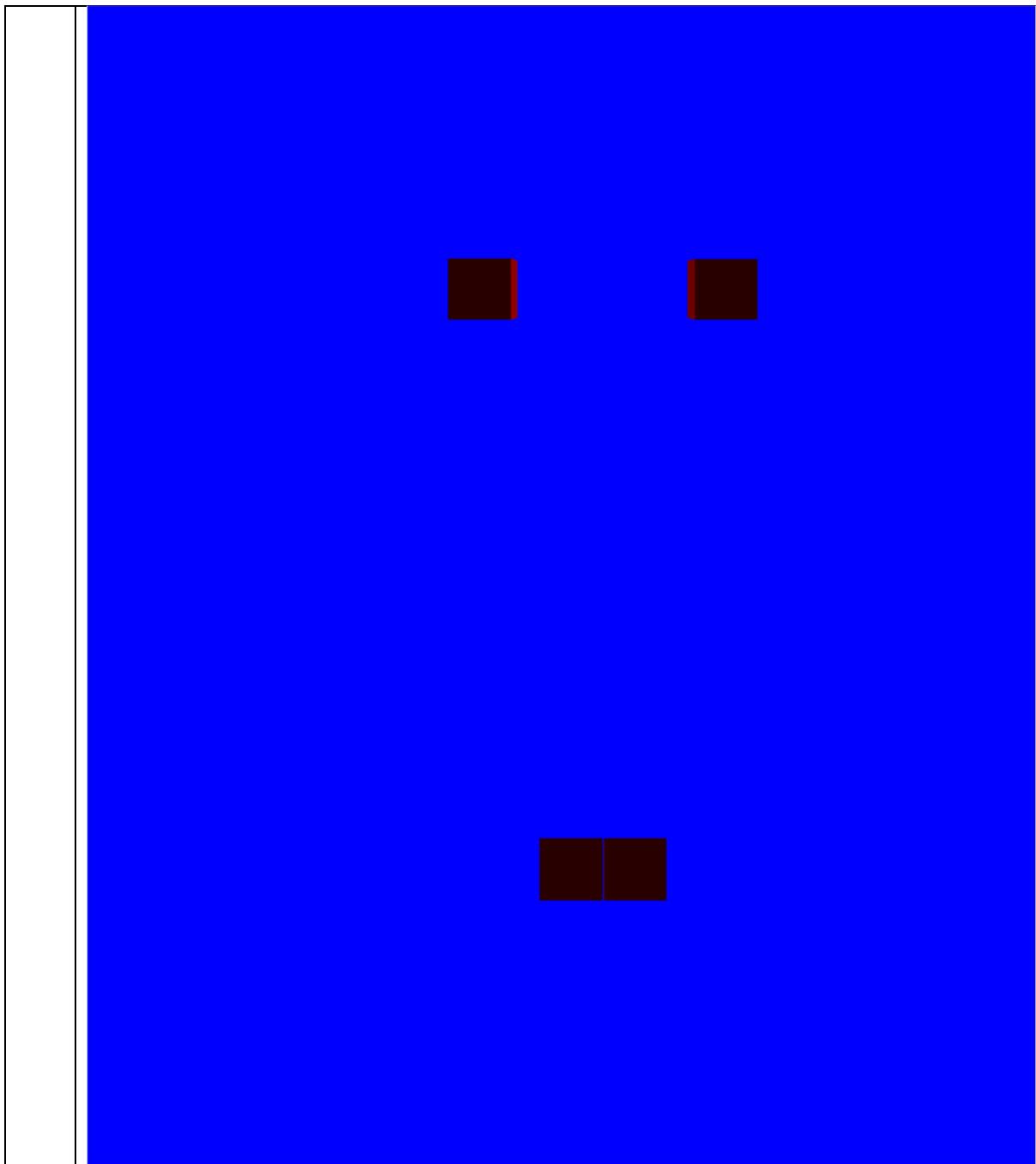


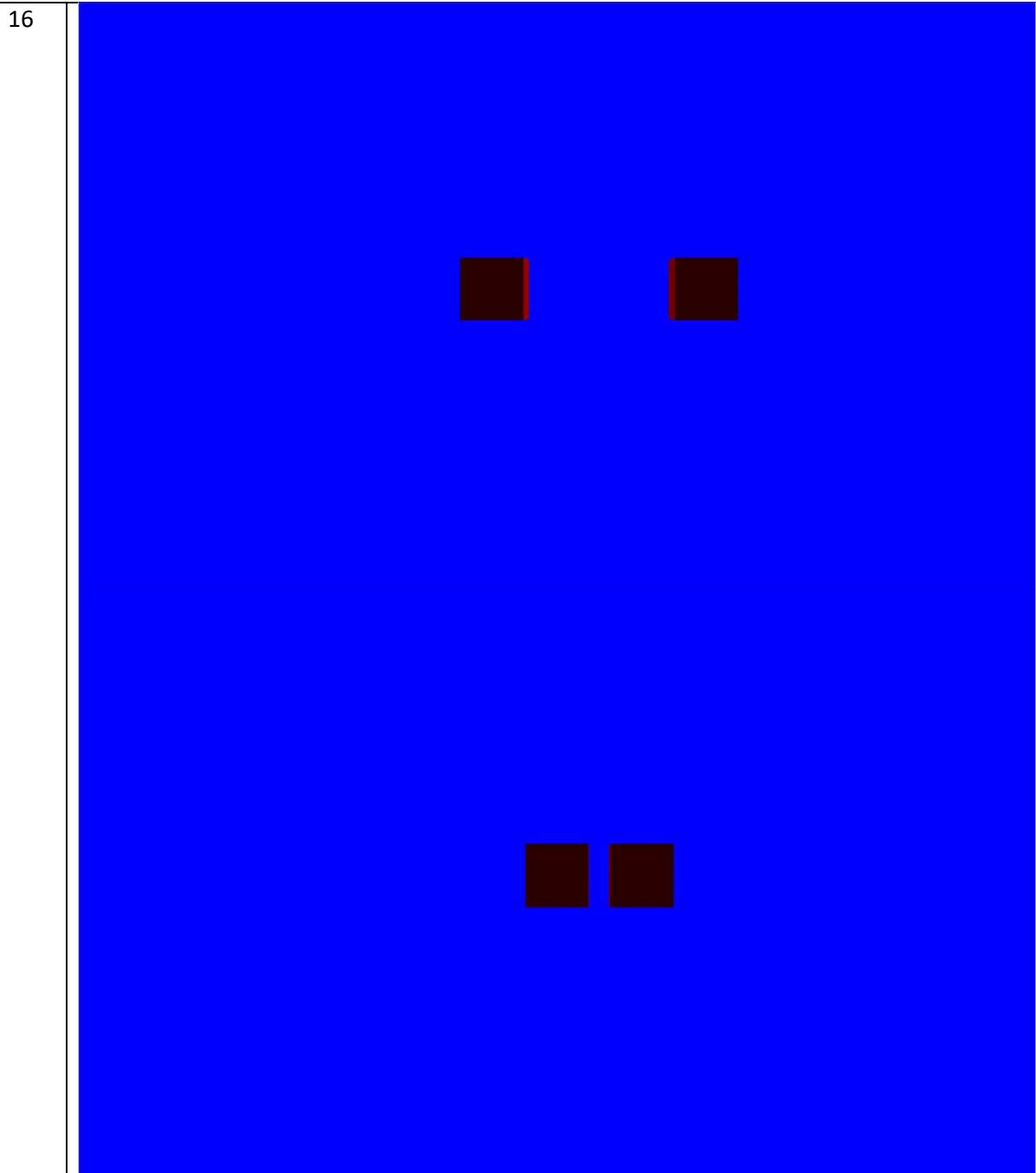


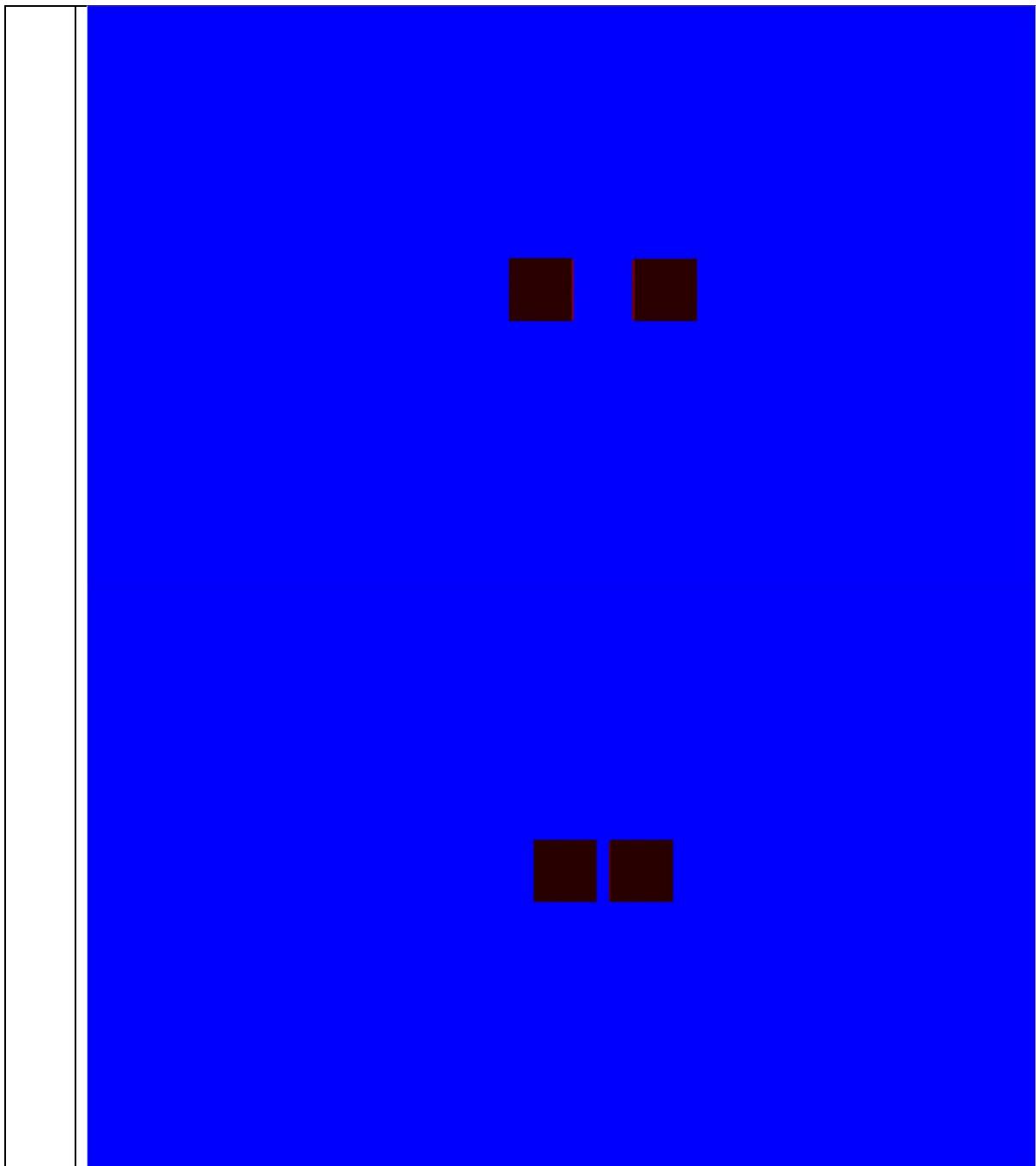


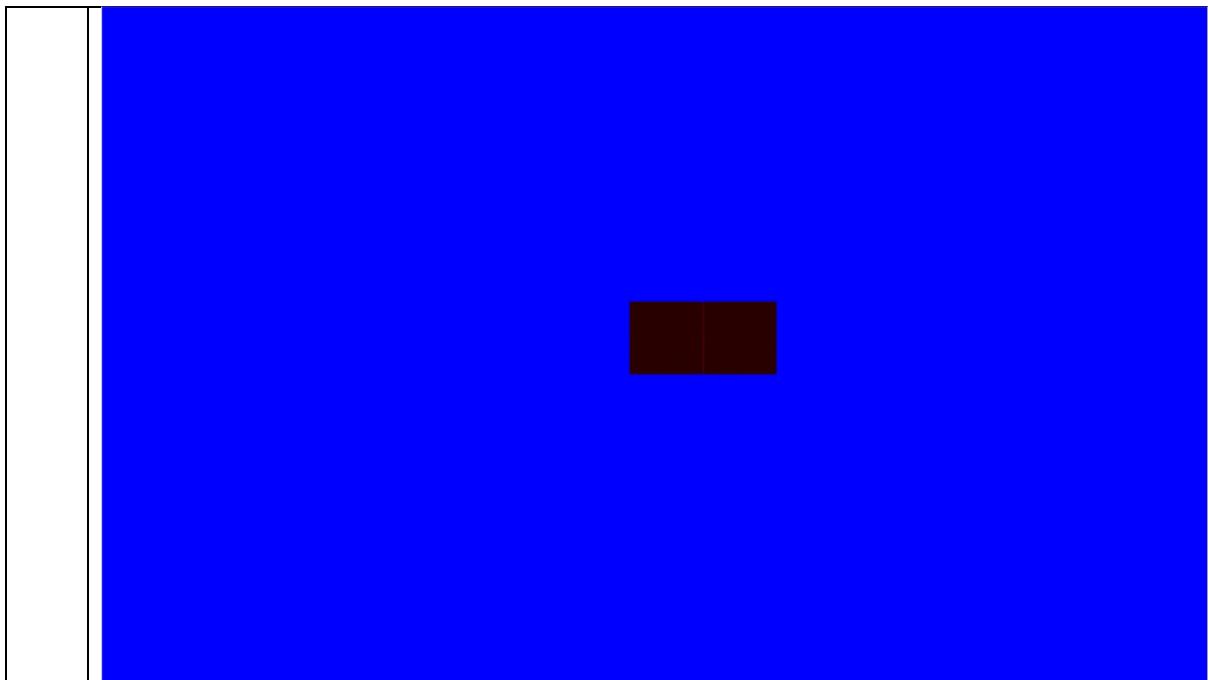


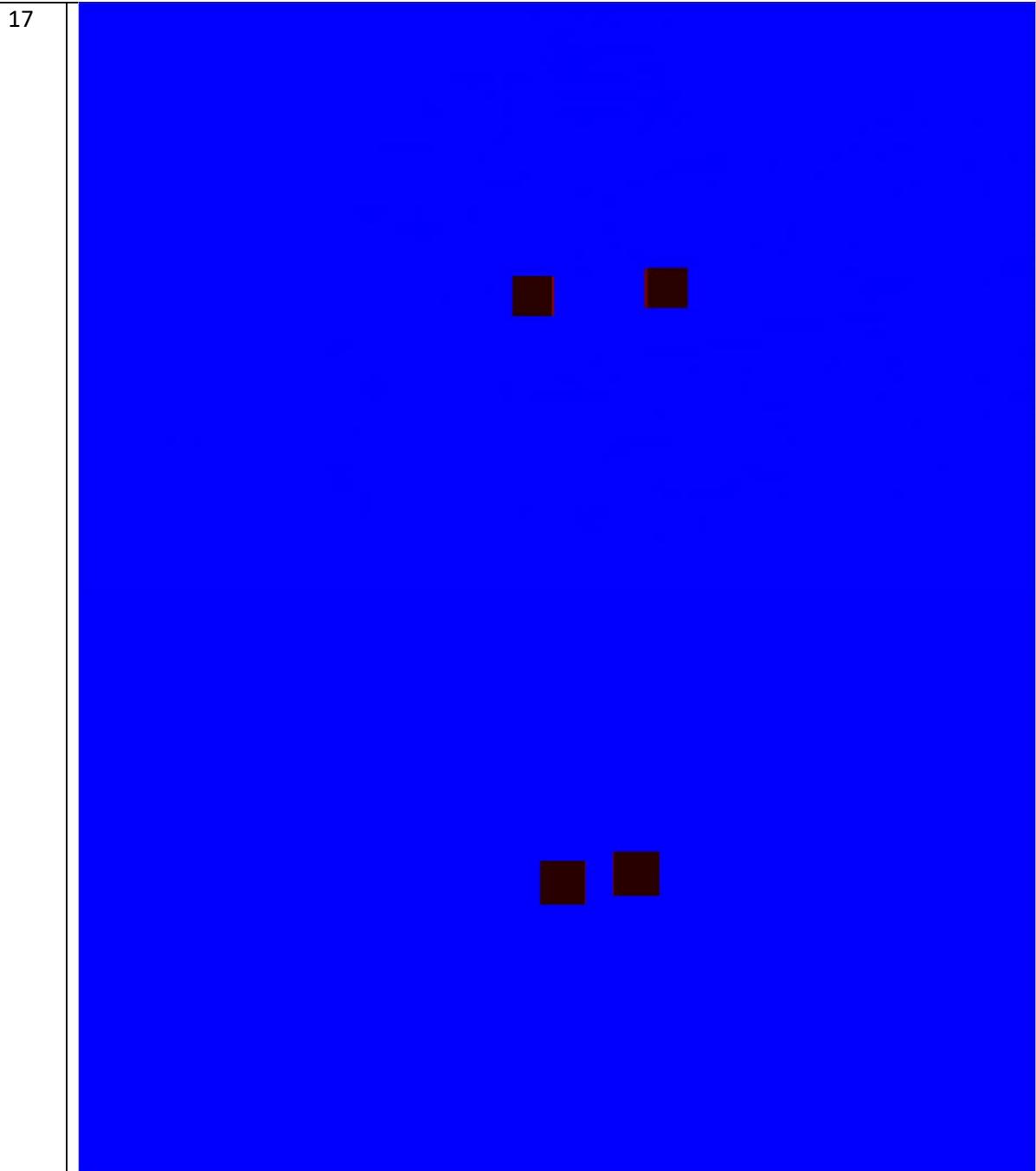


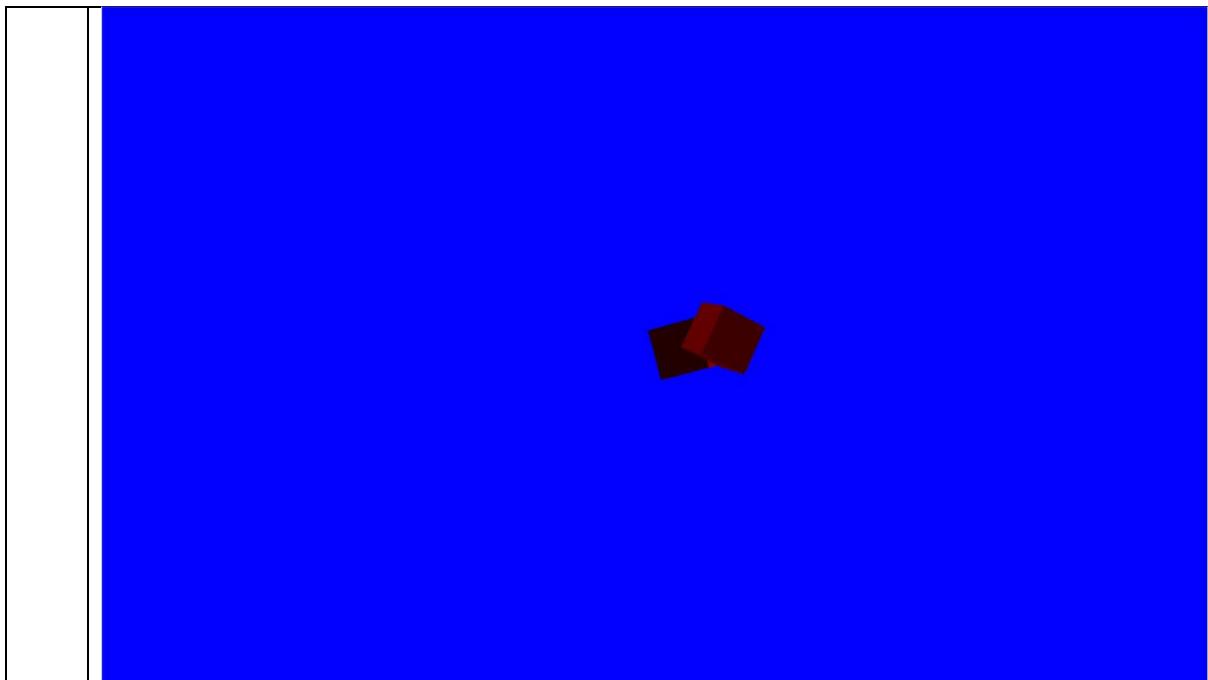












These tests address Success Criteria 7, 8 and 9.

## Success Criteria Review

To review the success criteria so far:

Main window which is moveable and resizable. ✓

Menu screen. X

Lightweight and easy to navigate [subjective, judge at the end]. X

Multiple different options of different simulations. X

3D rasteriser. ✓

Shading. ✓

Working rigid body motion. ✓

Working rigid body collision. ✓

Normal reaction force. ✓

Options next to each simulation such as creating objects, applying forces to objects etc.. X

Camera control. ✓

7/11 success criteria complete so far. I will now move on to the usability feature part of development in prototype 3.

# Prototype 3 Design

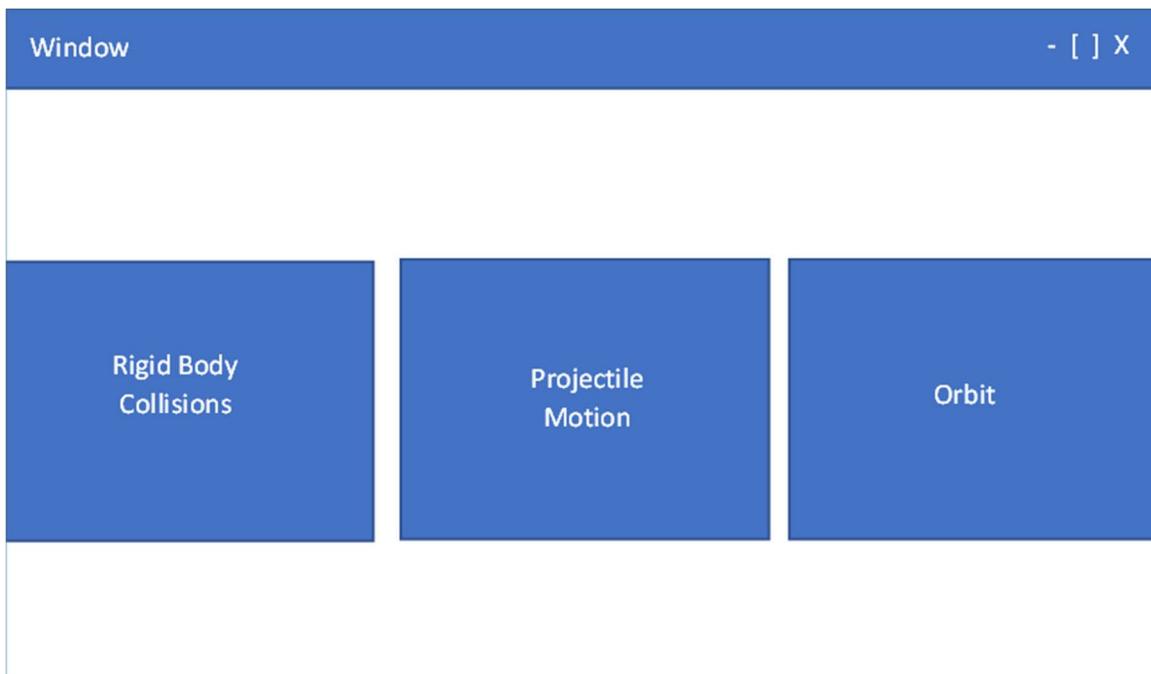
Prototype 3 will aim to implement usability features including the remaining success criteria:

- Menu screen.
- Lightweight and easy to navigate.
- Multiple different options of different simulations.
- Options next to each simulation such as creating objects, applying forces to objects etc.,

I will use the design section to go into more detail.

## Menus

I did provide a menu screen idea near the beginning of this project:



I want this to be the window which launches when the application starts. I want the current starting window which I have already created (its default name is "form1" but I will rename it) to open when I click the associated option. I will modify this form (as it uses many reusable components) to create the other two simulations and also to add new features. For the time being, whilst I add features which will be used in all three simulations, I will leave them all as a single form, and they will split off later on. Once I have created the "Menu" form I will need to add some options to it. This includes those above, but also a settings menu option somewhere in which the user will be able to change settings such as camera speed, sensitivity, field of view, and framerate. I will use a gear icon to represent the settings page.

Window

- [ ] X

Rigid Body  
Collisions

Projectile  
Motion

Orbit



I am also thinking I should replace the blue boxes behind each name with an image to give the user an idea of what the simulation is, also to make the menu screen more interesting to look at.

Menu

- [ ] X

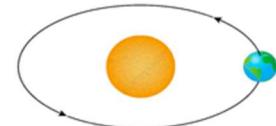
Rigid Body Collisions



Projectile Motion



Orbit



I also moved the text above the images so it is still readable. I also renamed this form to "Menu" as that describes its purpose. When the corresponding simulation button is clicked, that simulation should be launched (each will have its own form, which I will design and implement later). When the gear icon in the bottom right is clicked, a setting form called "Settings" will be launched, I will design this settings form now.

The settings available will be:

- Camera speed
- Camera sensitivity
- Field of view
- Framerate Limit

The form will look something like this:

The form is a standard Windows-style dialog box. It has a blue header bar with the word "Settings" in white. In the top right corner of the header bar are three small icons: a minus sign, a square, and an "X". The main body of the dialog box is white. On the left side, there are two buttons: "Exit" at the top and "Settings" below it. To the right of these buttons are four settings, each consisting of a label and an input field. The first setting is "Camera Speed" with an input field labeled "Enter Number...". The second setting is "Camera Sensitivity" with an input field labeled "Enter Number...". The third setting is "Field of View (degrees)" with an input field labeled "Enter Number...". The fourth setting is "Framerate Limit" with an input field labeled "Enter Number...".

Input validation will be used to determine whether the input from the user is appropriate or not (it should be a positive number), if it isn't or if the box is left blank then a default value will be used. These default values will simply be the values which I am currently using (except I will set the framerate limit to 60 as opposed to 165 as 60 is the standard). I specified that the field of view be entered in degrees, this is because most people are more familiar with degrees than with radians however this does mean that I will need to convert them into radians once they have been inputted. The settings entered will only be updated into the actual program when the settings are closed using the button in the top left corner, the reason for this is that it saves me from needing to implement a "confirm" box for each setting individually.

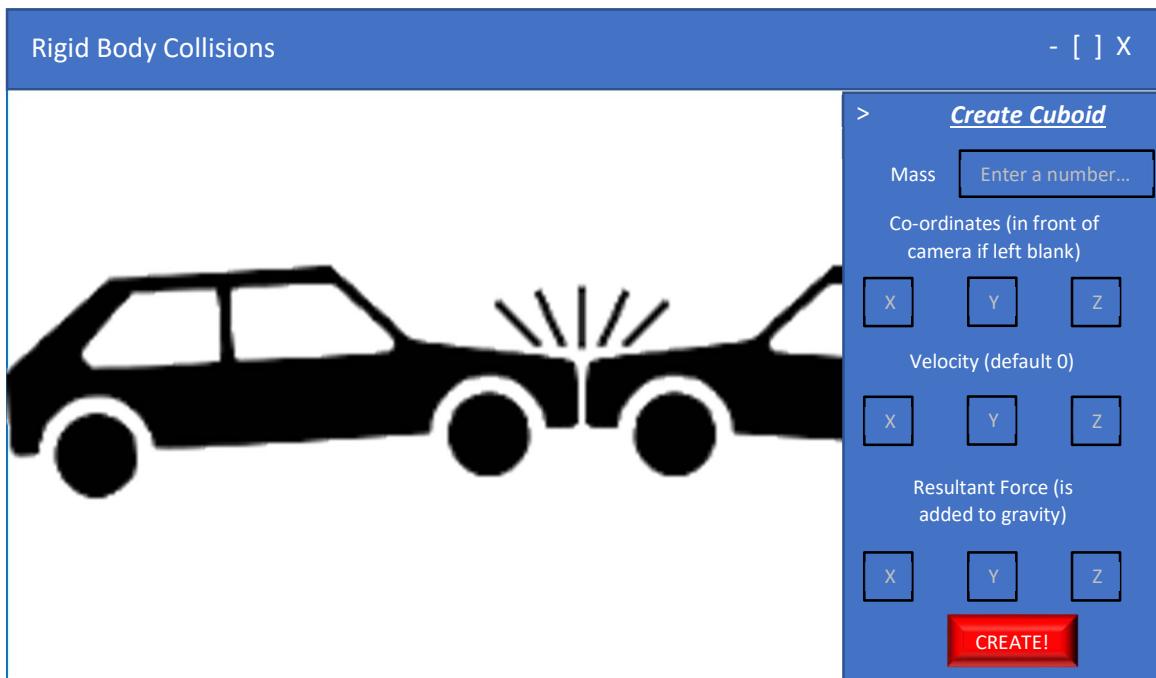
There is a settings button (gear) on the menu screen, but I will also make the settings accessible from within each of the simulations. When doing this, I will leave the other form open in the background so that the simulation isn't reset each time the user wishes to update a setting.

### Accessibility

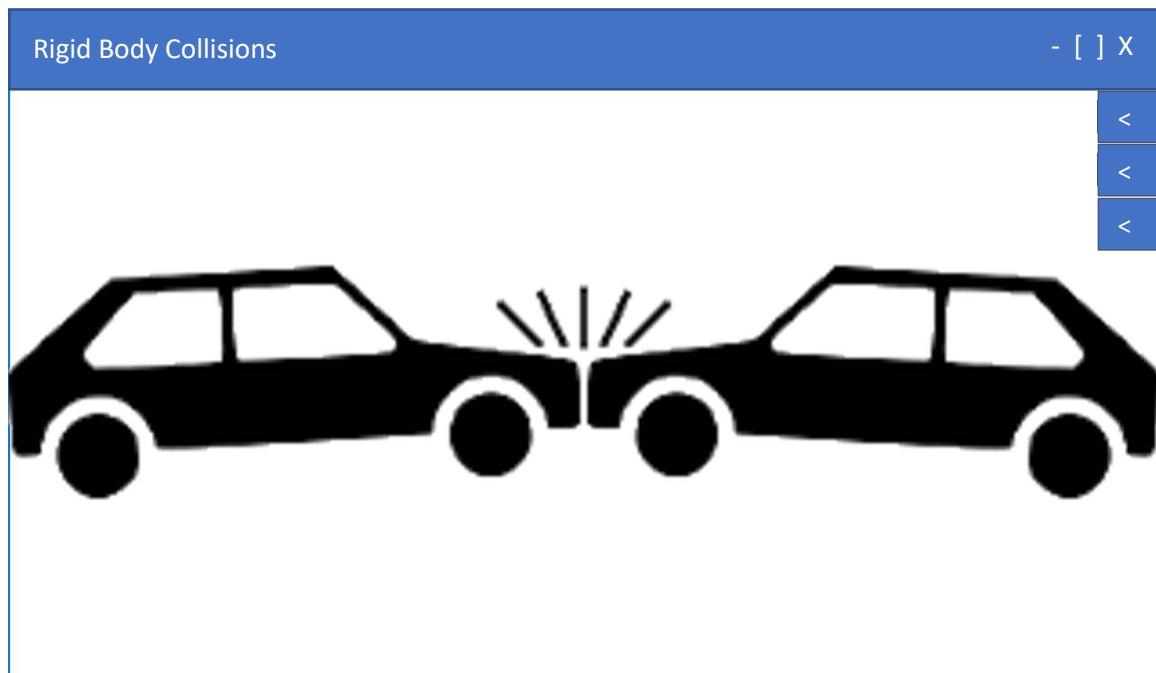
I will also introduce a setting which will allow colourblind users to alter the colours used, e.g., changing the background colour or the colour of objects. I could add this to the settings page, but I am thinking of adding it to the simulations themselves as this would allow the user to see the changes in real time.

I now need to design the finer details of the simulations. The good news is that most of the work has already been done, I simply need to add a few key features and design a layout in

terms of buttons and options which the user can make use of, just as I did for the menu and settings pages.



Above is an example of a menu for the rigid bodies simulation. I also want to add colour to this list, taken as an RGB value, and size, taken as length, width, and height, but I didn't have room here. There are other settings I wish to implement and I have come to the conclusion that I will need multiple menus if I wish to fit everything in. The “>” in the top left is a button which will minimize this menu, allowing the user to open others from a sidebar.

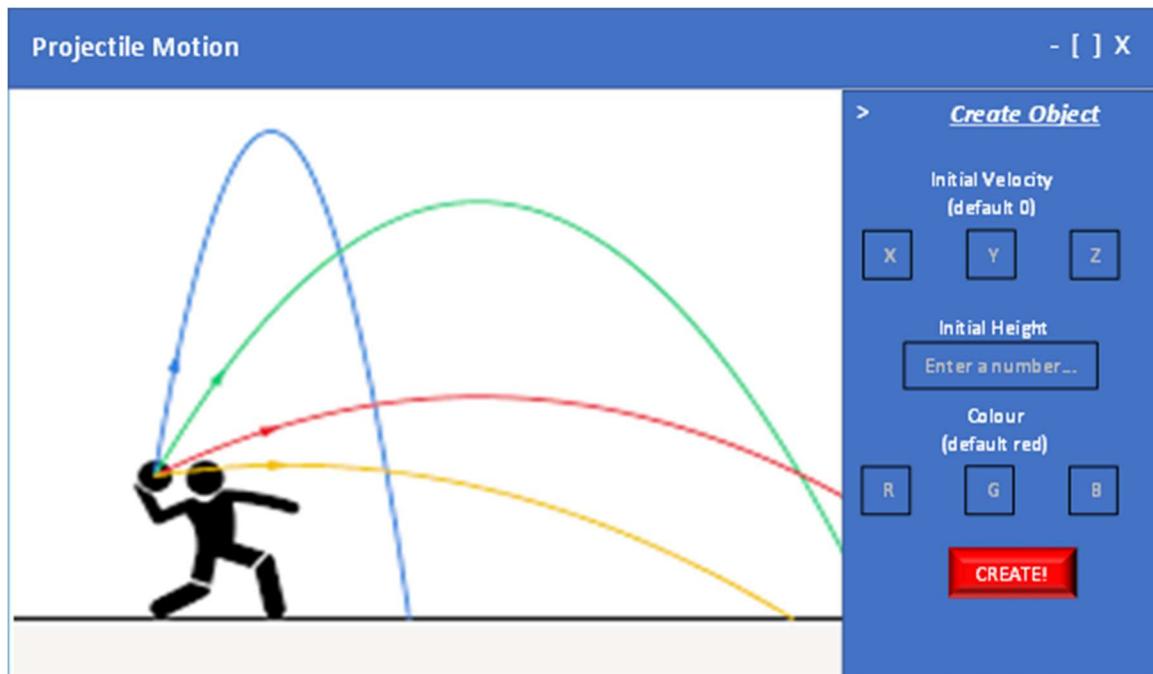


The second menu will have options related to the environment itself rather than the creation of new objects, allowing the user to alter things like the strength of gravity and the background colour.

The third tab will have options to alter things about a specific object (as selected from a list), namely the properties listed in the “Create Cuboid” tab.

The menu for the orbit simulation will be much the same as this one.

The projectile motion simulation will focus more on kinematics (motion) than on collisions. This simulation will have collisions disabled and the menu will be greatly simplified as the only necessary information for projectile motion is the starting height and the initial velocity (assuming no air resistance).



The projectile motion menu is much simpler as most of the other information is not relevant. The mass makes no difference as the only acceleration is due to gravity, which is constant. The size does not matter because the object could be modelled as a particle anyway. I decided not to include starting coordinates (except for height) because allowing users to create object with certain starting co-ordinates would not be very useful since the objects cannot interact with each other in any way. The colour option is new here but this will be on the menu for the other simulations too, it just wouldn't fit in the diagram before. The second and third menus will be the same as outlined for the first simulation.

Simulation 3 “orbit” will have the same menus as “Rigid Body Collision” except it will not have the option for a resultant force, as the only forces acting will be between object due to their gravity.

### Differences between each simulation

The “rigid body collisions” simulation will basically be everything I have created so far, but I will have a toggleable ground on which object may be able to land.

The “projectile motion” simulation will remove object collisions, the ground will be permanent, and there will be fewer user options as outlined previously.

The “orbit” simulation will have no ground, objects will be able to collide, and most importantly, instead of having gravity always acting down, there will instead be a force pulling objects of large masses together, this will also allow objects to orbit each other, much like the planets in our solar system, hence the name of this simulation.

The ground will be a plane at  $y = 0$ , the camera will not be able to go beneath the plane, meaning it should always be rendered before anything else. Checking for collisions with it will be easy since I need only check if each point has a  $y$ -coordinate less than zero, if not, there is a collision. Rendering this plane can be made much easier, by simply making it a cuboid but with a very small height and a different colour.

I want the ground to be anchored in place so that it doesn’t move when things collide with it. Essentially, I want it to have infinite mass. This is fine because the only places where mass shows up in the collision response is when dividing by it. This means that all divisions by mass should be replaced by zero.

For the “orbit” simulation, I will determine the force acting between them using Newton’s Law of Gravitation:

$$F = \frac{Gm_1m_2}{r^2}$$

Where  $F$  is the force between them,  $m_1$  and  $m_2$  are the objects’ masses,  $r$  is the distance between their centres of masses and  $G$  is the gravitational constant,  $6.67 \times 10^{-11}$ . Because this is such a small number, the gravitational forces between objects is only noticeable for very large masses.

### Note on Post-Development Testing

During post development, I will ask my stakeholders to test the program to see if its usability is good and what improvements can be made to improve the program’s usability.

## Test Plan

Test table:

Test No.	Input Data	Expected Output
18	Settings button from menu screen clicked	Settings dialogue window opens
19	Rigid Body Collisions selected	Rigid Body Collisions simulation opens
20	Projectile Motion selected	Projectile Motion simulation opens
21	Orbit selected	Orbit simulation opens
22	In simulation 1 (RBC) menu, enter 0,0,0 to "background colour" box and click "Make Changes!"	Background colour changes to black
23	Rigid Body Collisions: Set mass to 1, CoR to 1, Position to 0,0,0, velocity to 0,0,0, dimensions to 1,1,1, colour to 255,0,0, resultant force to 0,0,0 and click "Create Object!"	Red cube side length 1, created at the origin. Accelerates down due to gravity.
24	Same as above but in Projectile Motion, with height of 10, initial speed and angle as 0	Same as above, until the object hits the ground when it will bounce back up to its initial height
25	Same as above but in Orbit with radius of 0.5	Object is created at the origin but object does <b>not</b> accelerate, it remains stationary.
26	RBC: Click "Delete Selected Object!" when an object is selected.	Selected object should disappear.
27	Same as above but in Projectile Motion	Same as above
28	Same as above but in Orbit	Same as above

29	Orbit: Create an object with mass 5920000000000kg, at the origin. Create a second object of any mass, at (0,0,-10) with velocity (6.283,0,0).	The second object should orbit the first at a distance of 10 metres, once every 10 seconds.
30	Orbit: Create a set of 10 smaller, white objects (mass 1kg) to orbit a single larger blue object (mass 5920000000000kg) at a distance of 10. Create a second larger blue object, same mass as the other, a distance of 20 from the first. Set background colour to (0,0,0) (black) so the blue objects can be seen.	Smaller objects change path due to the second blue object. The blue objects should move together until they meet, where they will stop moving.

These tests are to ensure that menus and other usability features are functioning correctly, as well as the three separate simulations.

## Key Classes, Subroutines and Variables

Classes, functions/procedures and variables for prototype 2 (after refinement).

Classes designed for this prototype:

Class Name	Attributes	Description
Ground	double height, bool toggled, Vector colour	A plane with which objects can collide and on which they can collide. The attribute "toggled" can be used to enable/disable it.

I also used encapsulation because this was the only way I was able to access an instance of the Ground class from outside the Global class:

Class Name	Methods
Ground	get(), set()

groundCollisionDetection	Mesh mesh	void	Same as collisionDetection, but for collisions with the ground.
groundCollisionResponse	Entity obj, Vector point	void	Performs collision response with the ground.
regExInterp	string str	Vector	interprets a regular expression (string of a particular form) of the form (x, y, z) and extracts the numbers from it. e.g., (1,2,3) returns 1, 2 and 3.
renderGround	Ground ground, Camera camera	void	renders the ground when it is toggled on.
Open_Settings_Click	object sender, EventArgs e	void	Opens the settings window in a dialogue box when the settings button is clicked.
Exit_Click	object sender, EventArgs e	void	Closes the current window and opens the main menu.

Create_Click	object sender, EventArgs e	void	Creates an object as specified by a user's inputs. Called when the “Create Object!” button is clicked.
MakeChanges_Click	object sender, EventArgs e	void	Changes settings such as the background colour and strength of gravity according to the users inputs. Is called when the “Make Changes!” button is clicked.
Ground_Enabled_CheckedChanged	object sender, EventArgs e	void	Checkbox which toggles the ground.
Alter_Click	object sender, EventArgs e	void	Alters a selected object which currently exists, by applying the users inputted settings.
Delete_Click	object sender, EventArgs e	void	Deletes the selected object.

As before, the methods of the Global class are highlighted in green.

Variable Name	Data Type	Description
bgColour	Color	Background colour is set according to this value.
main_menu	Menu	Instance of the “Menu” form. It is a main menu from which one of 3 simulations, or settings, can be accessed.
paused	bool	Determines whether the simulation is running (false) or the simulation options are being displayed (true)
gravConst	double	gravitational constant, is used to determine how attracted objects with mass are towards each other. Used in simulation 3.
ground	Ground	Instance of the ground class. Is the ground.

The concludes the design of the final prototype.

# Prototype 3 Development

## Main Menu & Settings

I first made a few small adjustments. The updatePosition procedure has been renamed to displaceObject and now displaces the object by a specified amount (much like the rotateObject procedure) this means that now I can move objects easily within the code. This also means that I can create a new object at the origin and then just move it to the desired location.

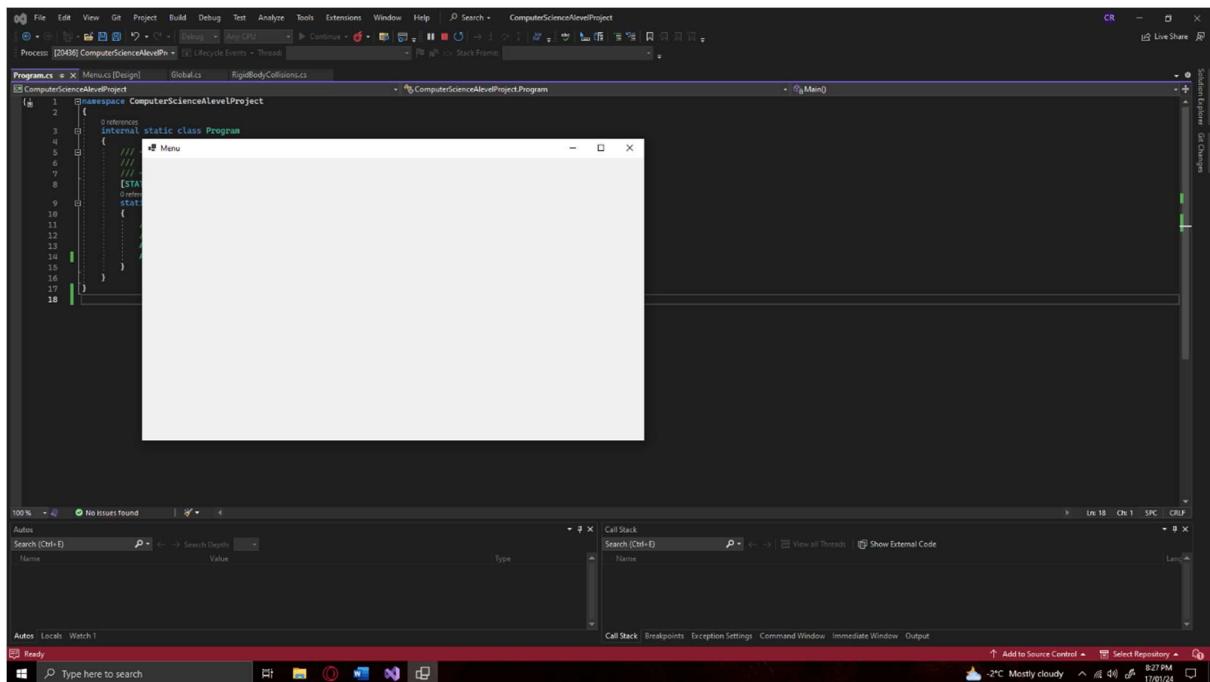
```
public static void displaceObject(Entity entity, Vector displacement)
{
    entity.rigidBody.centreOfMass = vectorAddVector(entity.rigidBody.centreOfMass, displacement);
    for (int i = 0; i < entity.mesh.vertices.Length; i++) //Loops through every vertex of the
    {
        entity.mesh.vertices[i] = vectorAddVector(entity.mesh.vertices[i], displacement);
    }
}
```

The first major thing I will do is the renaming and creation of new forms. I first renamed “Form1” to “RigidBodyCollisions”. I then created a new form called “Menu”. When the program is run, the old form “RigidBodyCollisions” is run, but I instead want the new “Menu” form to open instead. To change this, inside of Program.cs (prewritten code) I can change the form which opens when the application is run, to whichever I wish.

From Program.cs:

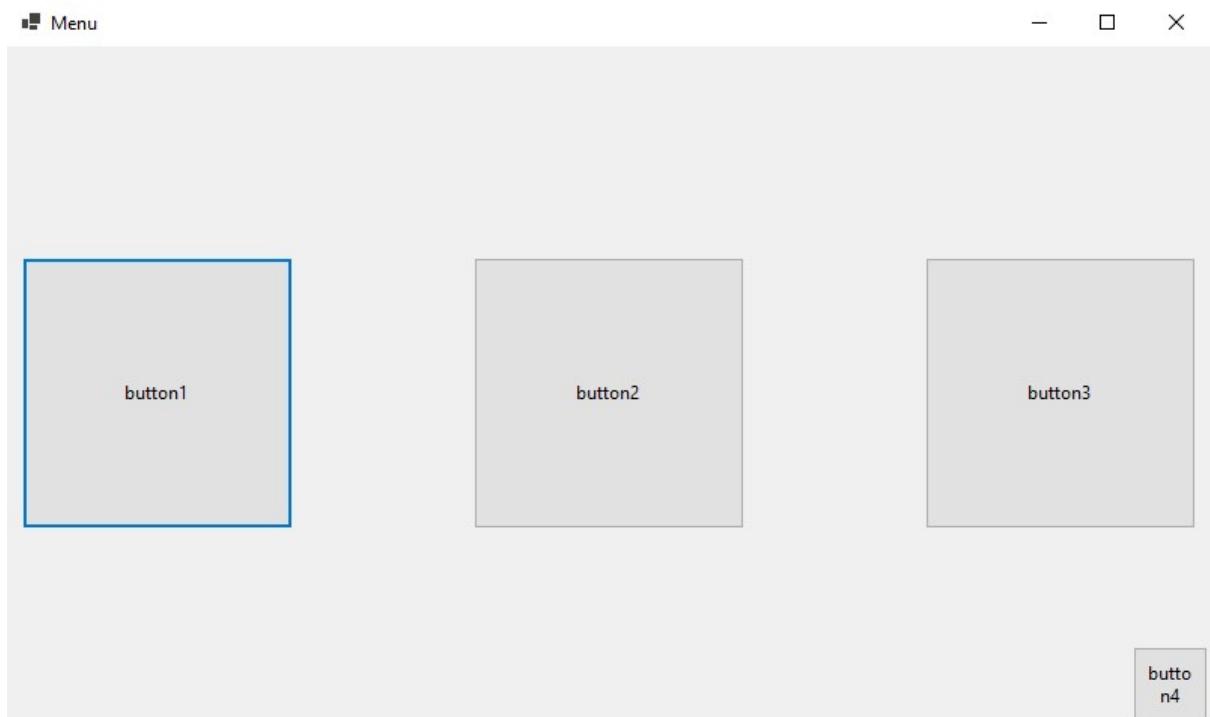
```
1  namespace ComputerScienceAlevelProject
2  {
3      0 references
4      internal static class Program
5      {
6          /// <summary>
7          /// The main entry point for the application.
8          /// </summary>
9          [STAThread]
10         0 references
11         static void Main()
12         {
13             // To customize application configuration such as set high DPI settings or default font,
14             // see https://aka.ms/applicationconfiguration.
15             ApplicationConfiguration.Initialize();
16             Application.Run(new Menu());
17         }
18     }
```

When I run this, the new menu form (which is currently blank) now appears instead.

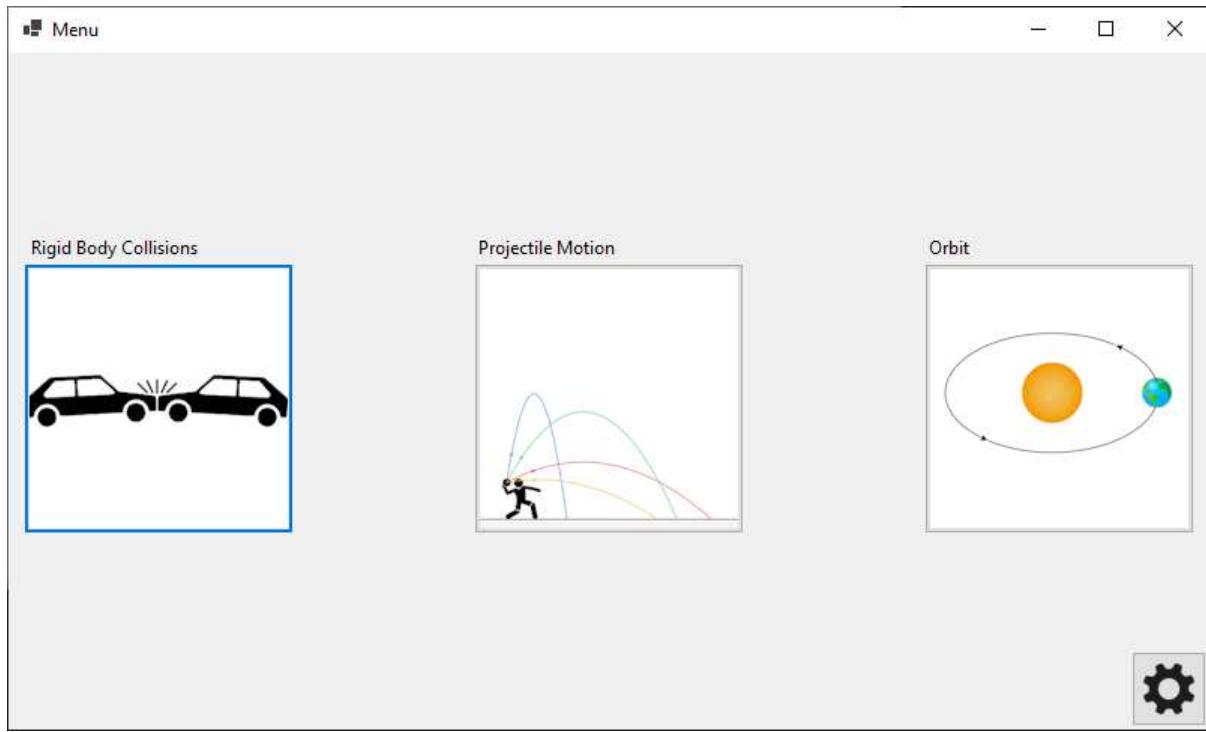


I can use the winforms “Designer” to easily add buttons to this menu.

After using the Designer, I now have three main buttons, in addition to a small settings button, although they of course don’t do anything yet.

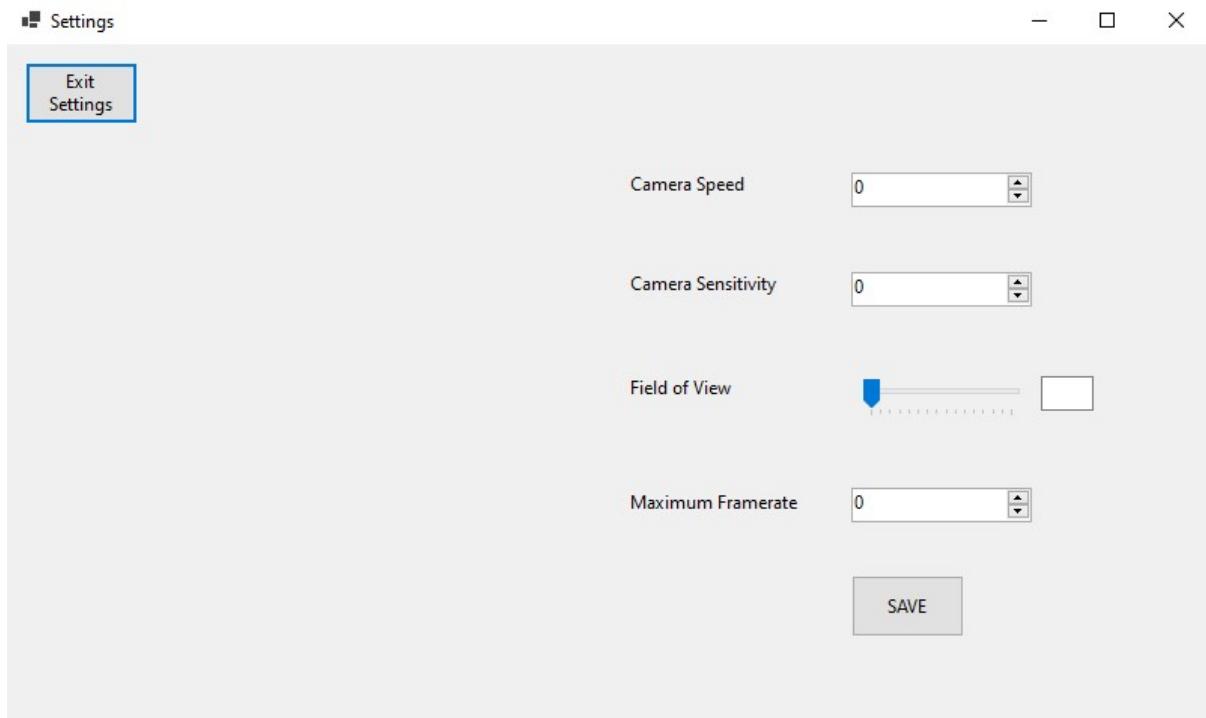


I renamed these buttons to “Play\_Sim1”, “Play\_Sim2”, “Play\_Sim3” & “Open\_Settings” respectively. I also removed the text on the buttons and added images. Finally I added labels above the buttons to say what they were for. After these changes, the final menu looks like this:



Currently, the buttons do nothing when clicked. I will first create a settings menu, which will open when the gear icon is clicked.

I have briefly altered the code so that the program launches to this new settings page. This is what it currently looks like:

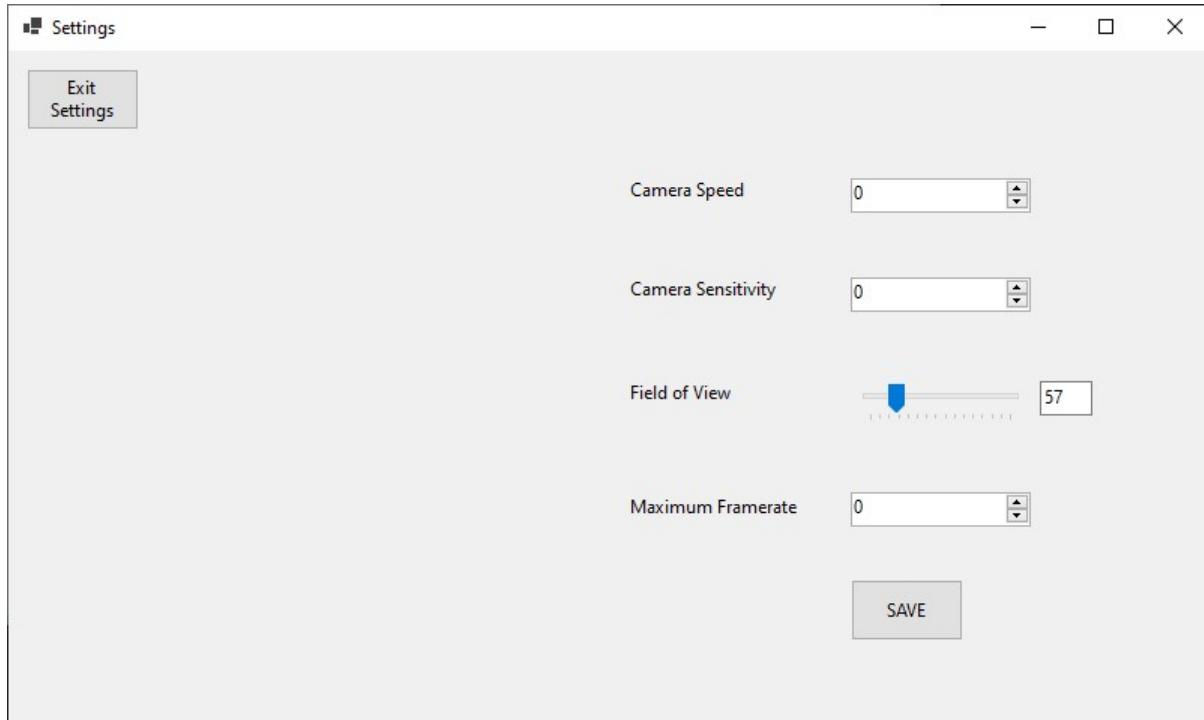


I want the textbox next to the field of view slider to display the value held in by the slider. Gave the slider (technically called a trackbar) the following code, which causes it to update the textbox with whatever value is currently held within itself.

```

11  namespace ComputerScienceAlevelProject
12  {
13      public partial class Settings : Form
14      {
15          public Settings()
16          {
17              InitializeComponent();
18          }
19
20          private void Field_of_View_Scroll(object sender, EventArgs e)
21          {
22              textBox1.Text = Field_of_View.Value.ToString();
23          }
24      }
25  }

```



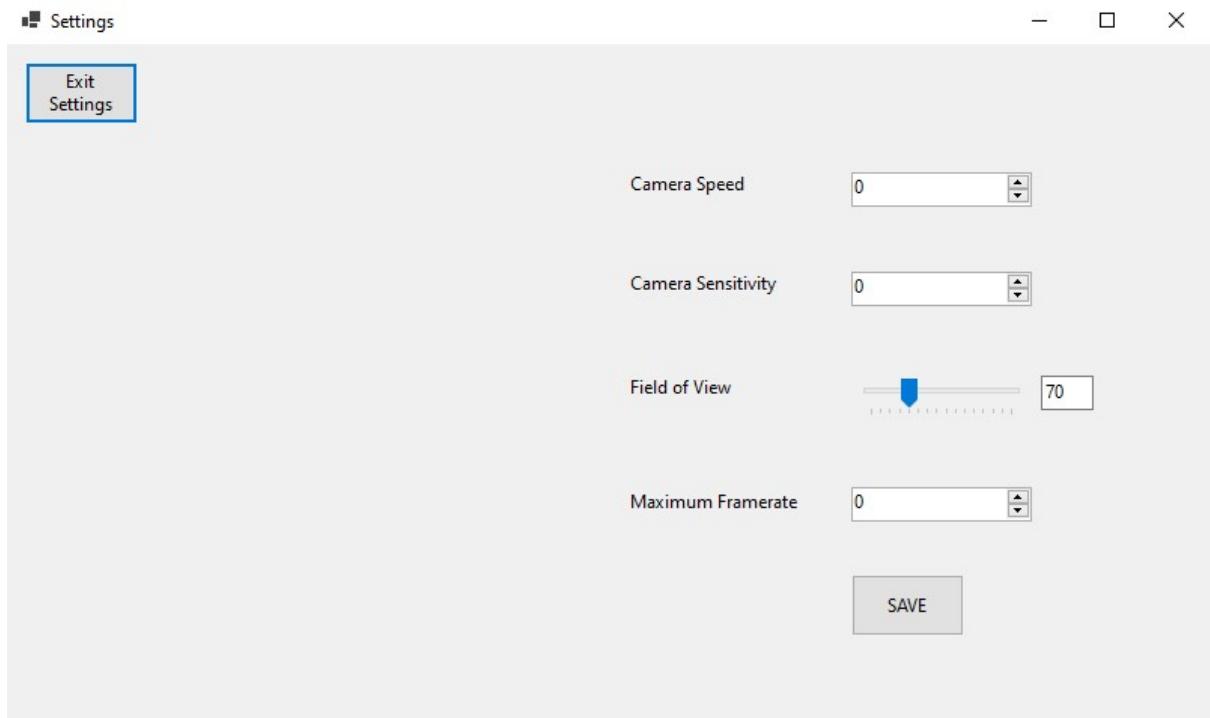
The textbox now displays the value held in the trackbar as required. The value is only displayed though, when the value in the trackbar is updated. I will make it so that the value in the trackbar and the text in the textbox is set to whatever the current field of view is, whenever the settings menu is opened.

```

private void Settings_Load(object sender, EventArgs e)
{
    //Set field values to their actual program values.
    Field_of_View.Value = Convert.ToInt32(Global.Camera.fov * 180 / Math.PI);
    textBox1.Text = Convert.ToInt32(Global.Camera.fov * 180 / Math.PI).ToString();
}

```

The trackbar and textbox now show the current field of view (converted to degrees) when the window is first opened.



I will also implement the same feature for the other fields here. First, I had to create variable to store the values for camera speed and camera sensitivity, these are now being used by the simulation to control the camera.

Code from the “RigidBodyCollisions” form:

```
void gameLoop()
{
    Global.cameraControl(Global.camSpeed * Global.frameTime, Convert.ToDouble(Global.camSensitivity) / 50 * Math.PI * Global.frameTime, Global.Camera);
    renderAll(Global.Entities, Global.Camera);
    physics();
}
```

With the default values being 25 for speed and 50 for sensitivity (these are in arbitrary units).

This is the code to set these values when the settings form is opened:

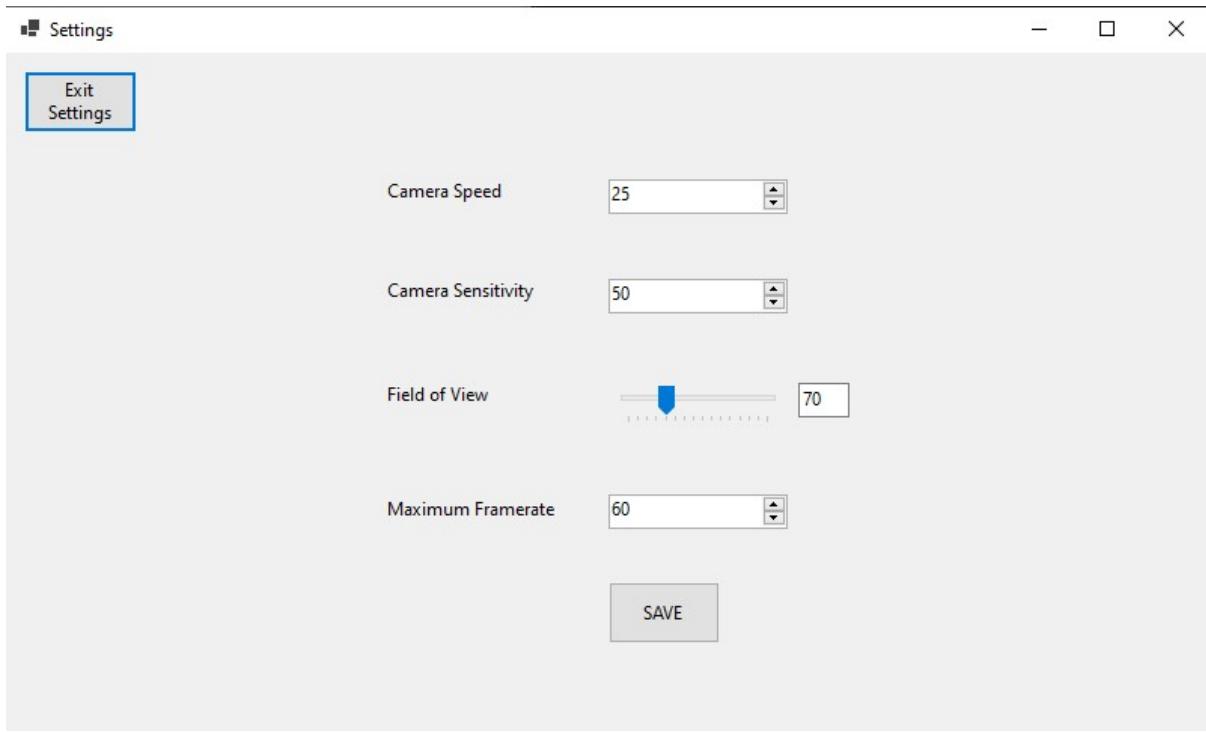
```
private void Settings_Load(object sender, EventArgs e)
{
    //Set field values to their actual program values.
    Camera_Speed.Value = Global.camSpeed;

    Camera_Sensitivity.Value = Global.camSensitivity;

    Field_of_View.Value = Convert.ToInt32(Global.Camera.fov * 180 / Math.PI);
    textBox1.Text = Convert.ToInt32(Global.Camera.fov * 180 / Math.PI).ToString();

    Maximum_Framerate.Value = Global.fps;
}
```

The default values appear as expected:

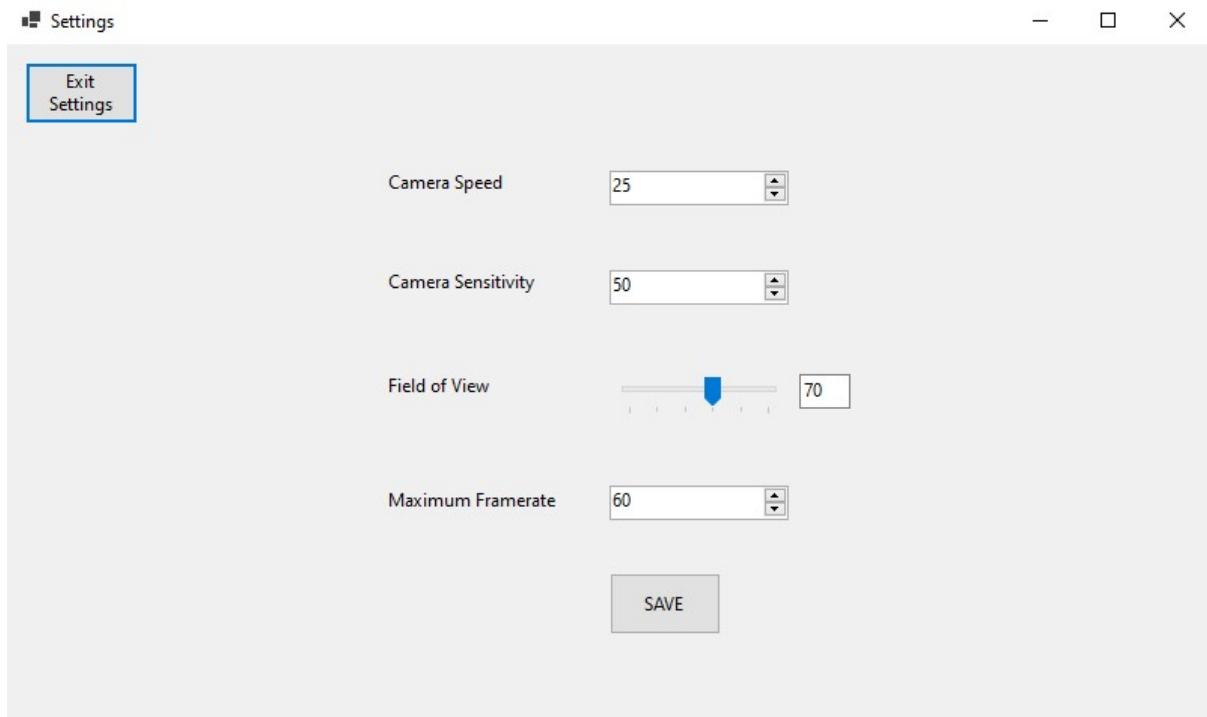


I also moved everything to be more central, as I think this looks nicer.

### *Input Validation*

To ensure that no invalid inputs are used here, I have used “NumericUpDown” field three times, which only allows integers to be entered. To ensure that these integers are of appropriate ranges, I have carefully set the minimum and maximum values. The speed can be anything from 10 to 100, to ensure that the users camera does not move painfully slowly nor too fast to be useful. The camera sensitivity can also be anywhere between 10 and 100 for the same reason. The framerate can be set anywhere from 10 to 165, as nobody is likely to have a monitor of > 165 Hz refresh rate. The field of view can be set anywhere from 40 to 90, as values outside this range tend to distort the image greatly, and 180 causes a crash. I still find that 70 is the sweet spot. The above screenshot was taken before this change hence why the blue pointer doesn’t seem to match this scale.

New screenshot:



To complete this settings page, I must add the functionality to the “SAVE” and “Exit Settings” buttons. The SAVE button, should set the values in the Global class, to those listed on the settings page.

```
private void Save_Button_Click(object sender, EventArgs e)
{
    Global.camSpeed = Convert.ToInt16(Camera_Speed.Value);
    Global.camSensitivity = Convert.ToInt16(Camera_Sensitivity.Value);
    Global.Camera.fov = Field_of_View.Value;
    Global.fps = Convert.ToInt16(Maximum_Framerate.Value);
}
```

This code is run when the save button is clicked and should update the values in the “Global” class.

To verify if this has worked, I have added some Debug.WriteLine procedures to print the values in the Global class when the program is loaded and then whenever I click save. The following was printed upon loading the settings form:

```
25 50 1.2217304763960306 60
```

25, 50 and 60 are all correct, but the third number should be 70, not 1.221... This is because I didn't convert from radians to degrees (when printing) or from degrees to radians (when setting the values).

I made the appropriate change to the save button's code, so that it now converts the value to radians first.

```
private void Save_Button_Click(object sender, EventArgs e)
{
    Global.camSpeed = Convert.ToInt16(Camera_Speed.Value);
    Global.camSensitivity = Convert.ToInt16(Camera_Sensitivity.Value);
    Global.Camera.fov = Field_of_View.Value * Math.PI / 180;
    Global.fps = Convert.ToInt16(Maximum_Framerate.Value);
}
```

I also changed the Debug line to convert to degrees before printing. The Debug lines look like this:

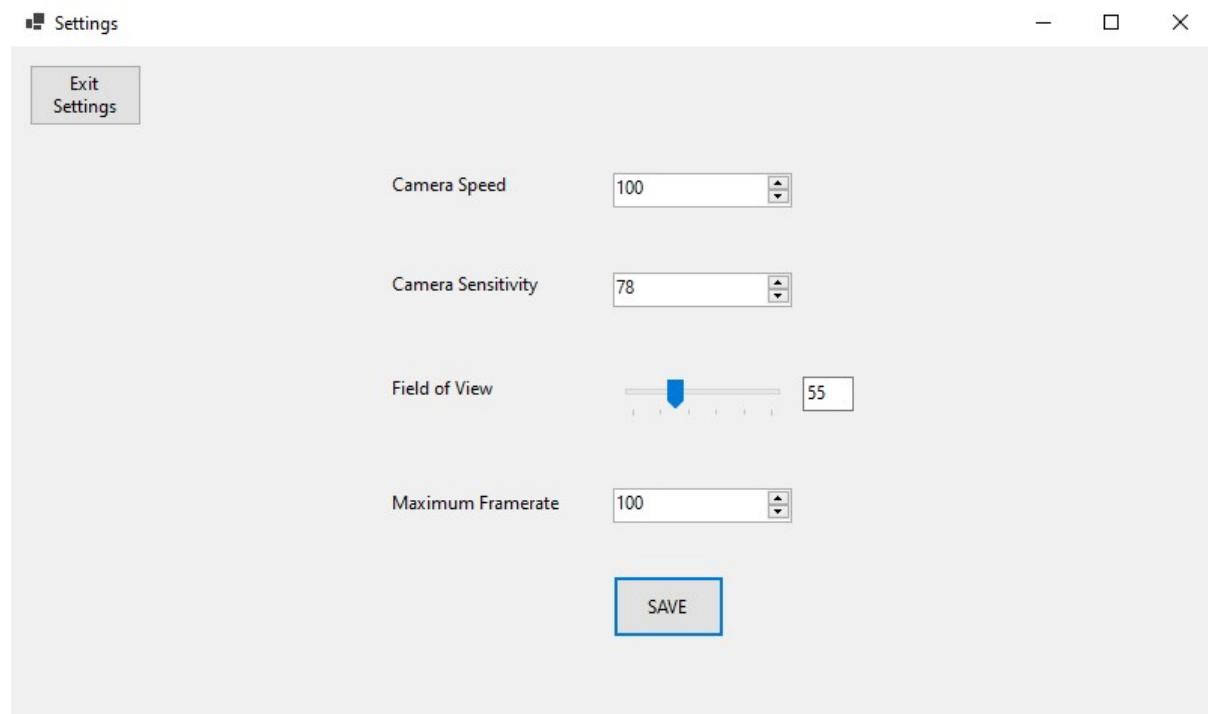
```
Debug.WriteLine("\n" + "Form Loaded:" + "\n" + Global.camSpeed + " " +
Global.camSensitivity + " " + Global.Camera.fov * 180 / Math.PI + " " + Global.fps
+ "\n");

Debug.WriteLine("\n" + "Save Clicked:" + "\n" + Global.camSpeed + " " +
Global.camSensitivity + " " + Global.Camera.fov * 180 / Math.PI + " " + Global.fps
+ "\n");
```

I will run the code, change the options, press save, and close the application.

I expect the output to be “Form Loaded:” followed by the default settings, then “Save Clicked:” followed by the values which I changed the settings to.

I took a screenshot of the random values I choose.



The following was printed to the debug screen:

```
Form Loaded:
25 50 70 60
```

```
Save Clicked:  
100 78 55 100
```

Finally, to complete the settings form, I need to add functionality to the “Exit Settings” button. I want the settings to appear as a dialogue form, meaning it is like a child to the form from which it was opened, so the form which it was opened from was never closed. This means that the previous form is already open, and so I don’t need to reopen it. The “Exit Settings” button, needs only to close the settings window.

The code for this is very simple.

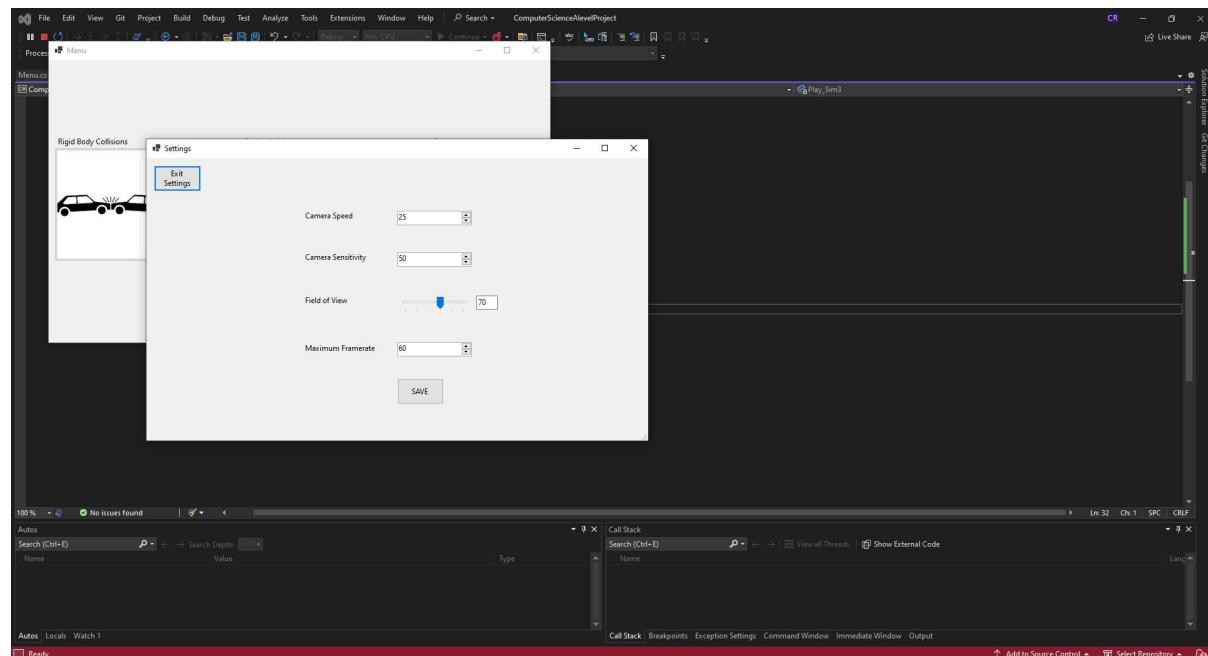
```
private void Exit_Settings_Click(object sender, EventArgs e)  
{  
    this.Close();  
}
```

Where “this” refers to the window itself.

The last thing I need to do for now in terms of the settings form, is have a way to open it. I want the gear icon on the main menu screen to open this window as a dialogue box. This involves two steps. First creating an instance of the “Settings” form which I called “settings”. Next, using settings.ShowDialog(); to show the form as a dialogue box.

```
private void Open_Settings_Click(object sender, EventArgs e)  
{  
    Settings settings = new Settings();  
    settings.ShowDialog();  
}
```

The result of this code is shown in here:



and [Video29](#) shows the save functionality working correctly, as well as the “Exit Settings” button. When the save button is used, the values are still there when the form is next opened. If the save button is not used, then they are not updated. One more minor change, I changed the small textbox to display the value held in the trackbar to be read only to make it clear that changing the value in the textbox, does not set the field of view. Now that it read only, the user cannot modify its contents.

The next thing I need to do is add the features which make each simulation different, as well as adding their own settings which can be tweaked as previously outlined. I will then allow the buttons on the main menu to open these simulations. At this point, the development stage of this project will be finished.

## Implementing The Ground Plane

The next thing, and likely the final difficult thing I will need to do, is implement the ground to be used in the projectile motion simulation, and to be toggleable in the rigid body collisions simulation. During design, I said that I would simply create and render a mesh which is large but with a very small height. I have decided that there may be a better way. I will create a new class, called “ground”. This class will have a few attributes:

- colour, this will need an R, G & B value, hence it will be vector.
- height, this will determine the y level at which the plane will be created and will be a double.
- toggled, this will determine whether the plane is enabled or not and will be a bool.

An instance of this class will be created in the Global class and can be referenced throughout the code. Any code which I will write in this section will not be present on the “orbit” simulation as this one does not need to reference this object. The projectile motion simulation will always have the ground there, and so its “toggled” attribute will be set to true when the form is opened, and the user will have no options to turn it off from within the form. When the rigid body collisions simulation is opened, the “toggled” attribute will be set to false by default but the user can change this from within the form.

Ground Class:

```

Ground.cs  X
ComputerScienceAlevelProject  ComputerScienceAlevelProject.Ground  toggled
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ComputerScienceAlevelProject
8  {
9      internal class Ground
10     {
11         public double height;
12         public bool toggled;
13         public Vector colour;
14
15         public Ground(double height, bool toggled, Vector colour)
16         {
17             this.height = height;
18             this.toggled = toggled;
19             this.colour = colour;
20         }
21     }
22 }
23

```

75 % No issues found | Ln: 23 Ch: 1 OVR SPC CRLF

Creating an instance of this class inside Global to be accessed throughout the project:

```

public static Ground ground = new Ground(0, false, new Vector(0, 255, 0));
0 references
public static Ground Ground
{
    get { return ground; }
    set { ground = value; }
}

```

To render the ground, I will use the render face procedure which I have already created. The coordinates of the inputted vertices will be relative to the camera's position, (so the user cannot move away from it, the ground will always be beneath the user). Physics with the ground will work regardless of whether it is rendered in that particular location or not.# Because the camera will never (nor will objects) be able to go beneath the ground, it will be rendered first. I added this line of code to the renderAll function, just before putting the meshes in order of closeness.

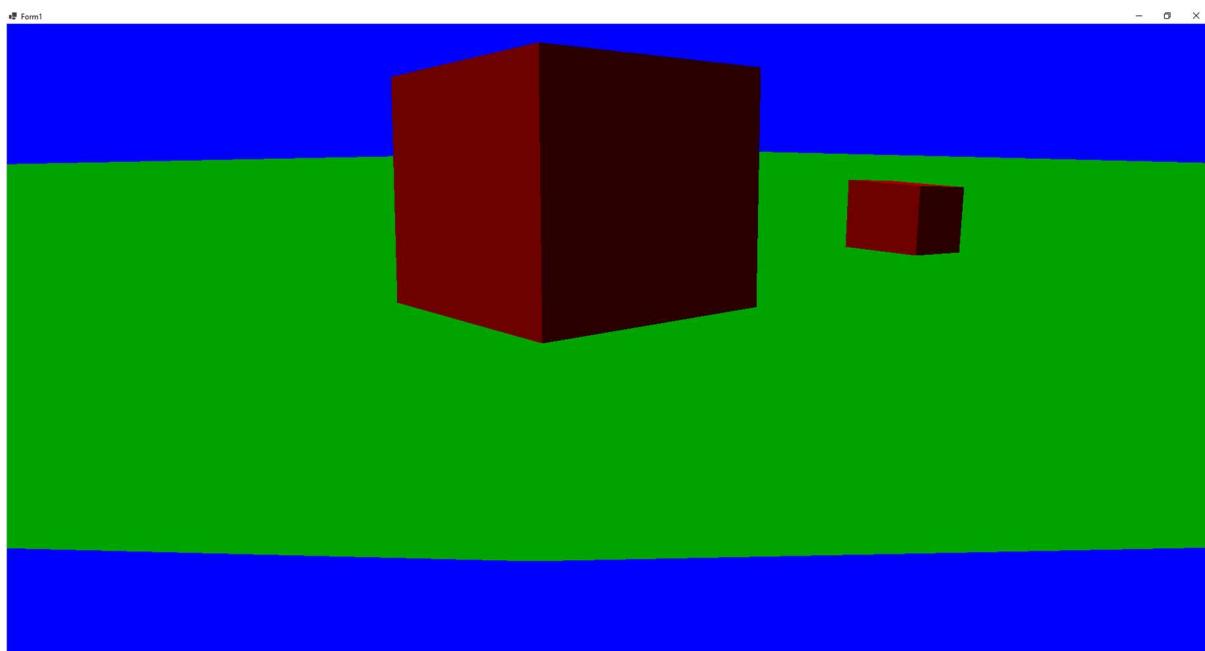
```
void renderAll(List<Entity> entities, Camera camera)
{
    double groundHeight = Global.Ground.height;
    Vector[] vertices = { new Vector(camera.position.x + 1000, gr
    Face ground = new Face(vertices, new Vector(0, -1, 0));
    renderFace(ground, Global.Ground.colour);
```

The line which it cut off was too long to include in the screenshot, it just defined the list of coordinates for the corners relative to the camera's position. You may notice that renderFace procedure no longer takes a mesh as a parameter. This is because that parameter used to be used to determine the normal to the face, but since the normal is now simply an attribute of the face, this parameter is no longer necessary so it was removed.

```
//Rendering individual face:
void renderFace(Face face, Vector colour)
{
    //Colour determination
    Vector perpVect = face.normal;
    double vectMag = Global.vectMag(perpVect);
    double shade = Global.getBrightnessLevel(perpVect, vectMag);
    Brush brush = new SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade), Convert.ToInt16(colour.y * shade), C
```

The renderFace procedure also now takes a vector as a parameter, which stores three values, representing red green and blue. This allows the colour of a face to be determined here. As shown in the above screenshot, this vector is being used to determine the colour with which the face is drawn. To accommodate this change, I added a new attribute colour, to the mesh class, which is used to be inputted to this procedure. I then added vectors (255, 0, 0) (for red) to the initialisation of the two sample cubes. I expect to see the two red cubes with a green plane rendered beneath them.

After running the code, I see this strange result:



This is almost correct, but the plane is not rendering close to the camera (at the bottom of the image you can see the blue background where you should see the green plane). I thought this may be because the camera is clipping through the plane (because I haven't prevented this from happening yet) so I moved the plane down, by setting its height to 10 (remember, a higher y value means it is lower down). After doing this, the result hasn't changed. I think the calc y coord function from the beginning may be broken, which would explain why certain values for the field of view were not working earlier and why certain visual glitches occur when very close to either cube.

Whilst searching for the answer to this, I found that I made type long ago when defining vertTanFov. I wrote:

```
double vertTanFov = Math.Tan(vertFov) / 2;
```

Where it should have said:

```
double vertTanFov = Math.Tan(vertFov / 2);
```

After making this change (which I suppose didn't make much difference for 70°), other fields of view now look normal, so I will extend the range of values which the field of view trackbar can take. The range of values is now from 20 to 170.

This fix (although necessary) did not actually fix the problem with the plane. To investigate this problem further, I want to fix another problem, that is that the plane renders even when it is behind the camera. To fix this, I could use the check that I was using previously for other faces, but because the plane is so large, this would not work. I will instead simply check if the angle the camera makes with the horizontal is beyond a certain angle, specifically, half of the vertical field of view of the camera, because if the camera is angled up more than this, then the plane will not be visible. This solution will only work because the plane is always horizontal.

To find the angle, I will use  $\tan \alpha = \frac{\text{camera.direction.y}}{\sqrt{(\text{camera.direction.x})^2 + (\text{camera.direction.z})^2}}$ .

I know that  $-90 < \alpha < 90$  so if  $\alpha < \frac{\text{vertFov}}{2}$  then  $\tan \alpha < \tan\left(\frac{\text{vertFov}}{2}\right)$  meaning

$\frac{\text{camera.direction.y}}{\sqrt{(\text{camera.direction.x})^2 + (\text{camera.direction.z})^2}} < \tan\left(\frac{\text{vertFov}}{2}\right)$ . This is the condition which if true, means the ground should be rendered.

```
//RENDER ALL:
void renderAll(List<Entity> entities, Camera camera)
{
    double groundHeight = Global.Ground.height;
    Vector[] vertices = { new Vector(camera.position.x + 1000, groundHeight, camera.position.z + 1000), new Vector(camera.position.x - 1000, groundHeight, camera.position.z + 1000), new Vector(camera.position.x + 1000, groundHeight, camera.position.z - 1000), new Vector(camera.position.x - 1000, groundHeight, camera.position.z - 1000) };
    Face ground = new Face(vertices, new Vector(0, -1, 0));
    if (camera.direction.y/Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2)) < Math.Tan(vertFov / 2))
    {
        renderFace(ground, Global.Ground.colour);
    }
}
```

With the new check in place, the plane is rendered when the camera is looking too far up, but not when it is looking down. This is the opposite of what I wanted, so I will just change the sign from  $<$  to  $>$ . This happened because the y value is negative (hence the angle is negative) when looking up, because down is the positive y direction. After changing the sign, it works much better much still is not perfect. When the camera's z component is negative, the plane renders the wrong way round, this is because of some from earlier which flips the sign of the y component to prevent some other issue, which I had no explanation for. I need to implement a similar fix here, which looks like this:

```

if (camera.direction.z < 0)
{
    if (camera.direction.y / Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2)) < Math.Tan(vertFov))
    {
        renderFace(ground, Global.Ground.colour);
    }
}
else
{
    if (camera.direction.y / Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2)) > Math.Tan(vertFov))
    {
        renderFace(ground, Global.Ground.colour);
    }
}

```

This did fix the minor issue I was talking about.

I am experiencing so many issues related to rendering this plane. I think it may be a good idea to go back and reconsider how I render this plane altogether. This means that much of the last two pages will be scrapped (but that does not mean it was wasted, as I did fix some older problems and also learning what doesn't work is part of the process).

I will still use the ground class as it has all the relevant information about the ground. I will also keep the changes made with regards to how meshes can now have their own assigned colour etc.,

I am first considering the fact that this plane is infinitely large, and its horizon would appear as a perfectly horizontal line. All I need to do is work out the  $y$ -coordinate (as in the height on the screen) of this line.

I will begin by defining some variables (don't worry this is the last time you will see equations).

$H$  = height of horizon on the screen

$F$  = vertical field of view of the camera

$E$  = height of the camera above the ground

$\theta$  = angle made by the camera against the horizontal

$H$  is what I want to find and is measured using the same normalised on screen coordinates used in prototype 1 (a value between  $-1$  and  $1$ , with  $0$  being the centre of the screen,  $1$  being the bottom and  $-1$ , the top).

To find the other variables defined here in terms of things we know in the program:

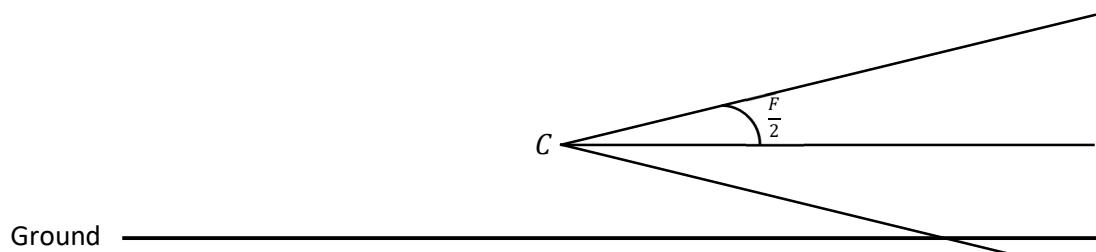
$$F = \text{vertFov}$$

$$E = \text{Global.ground.height} - \text{camera.position.y}$$

$$\theta = \tan^{-1} \left( \frac{\text{camera.direction.y}}{\sqrt{(\text{camera.direction.x})^2 + (\text{camera.direction.z})^2}} \right)$$

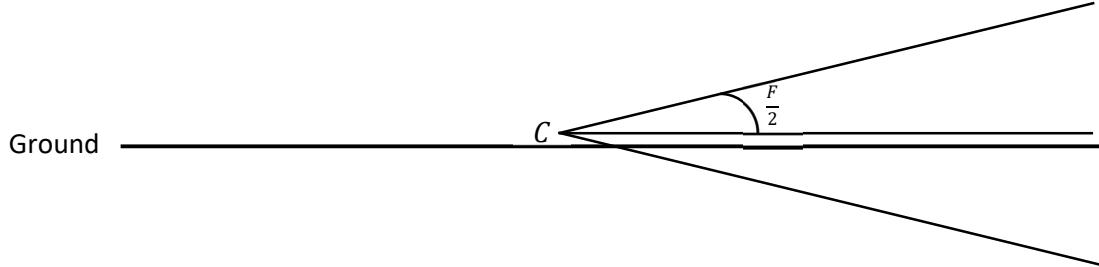
I first did the following working on paper and am transcribing it here.

I first considered the case where  $\theta = 0$ :



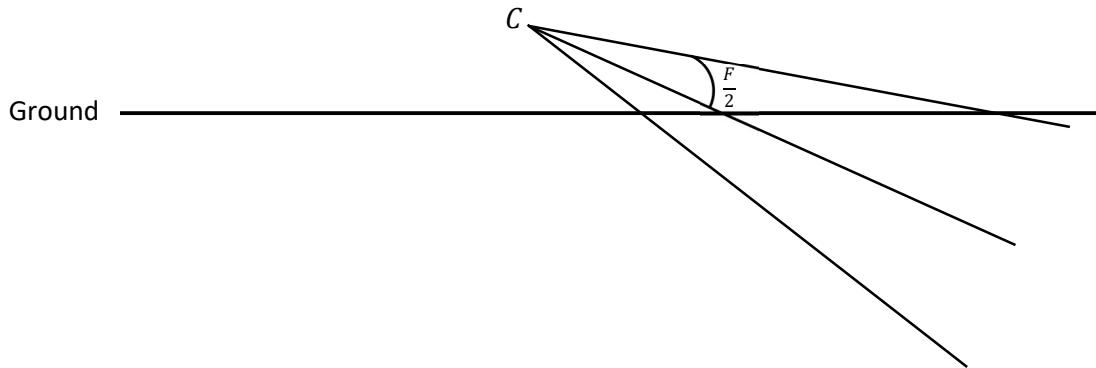
$C$  is the camera.

My first observation was that you can zoom out because the ground stretches infinitely. Also zooming out does not affect angles. After zooming out you would see something like this:

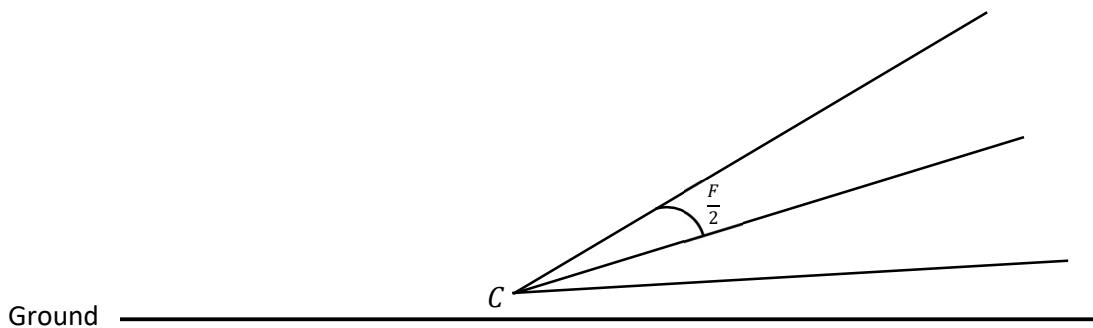


If you keep zooming out, you can make the camera appear to be as close to the ground as you like whilst preserving the proportion of the screen taken up by the ground. This all tells me that the value  $H$  which I am trying to find is independent of  $E$ , the height of the camera above the ground. For this reason, I will *Let  $E = 0$*  as this will make my work easier. Given these observations, it becomes clear that if you zoom out sufficiently, the proportion of the screen taken up by the camera is  $\frac{1}{2}$ , meaning the horizon is at the centre of the screen. In other words,  $H = 0$  in this case.

My next observation was that, if the camera is tilted far enough down that the top field of view line tilts below the horizontal, then the top field of view line will at some point intersect the plane, meaning the entire screen will be covered by the plane:



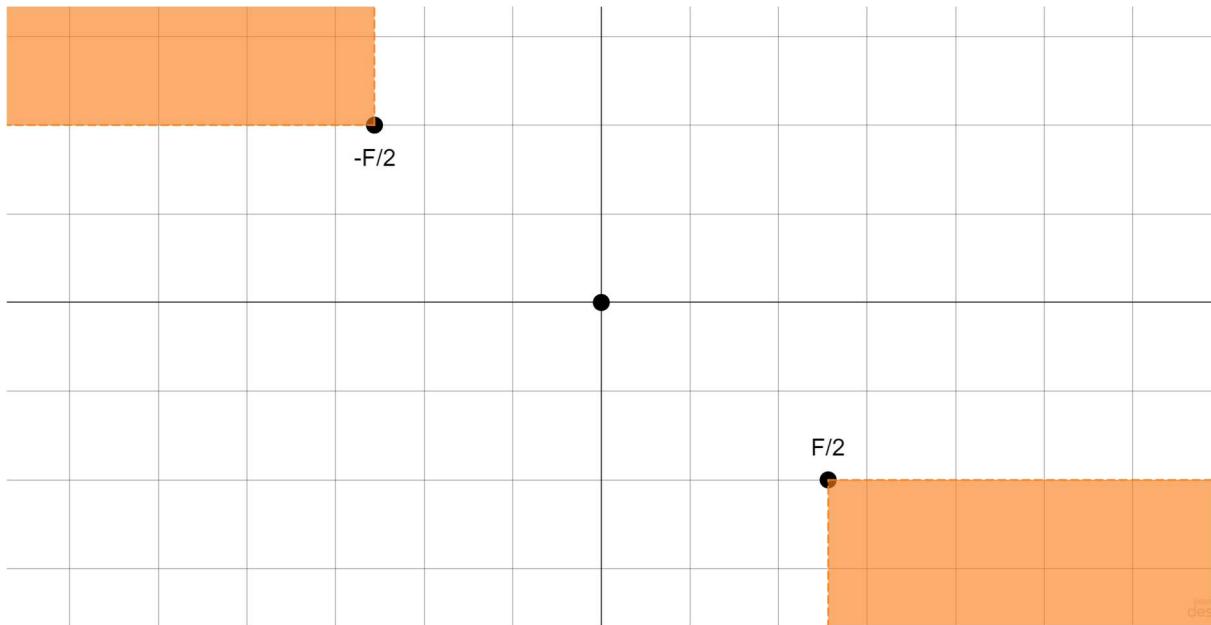
For this to happen, the camera must be below the horizontal by an angle,  $\theta$  greater than  $\frac{F}{2}$ . For the entire screen to contain the plane, the horizon line must be above the screen, meaning  $H \leq -1$ . To summarise this diagram, when  $\theta > \frac{F}{2}$ ,  $H < -1$ . We can make a similar argument with the reverse diagram to argue the converse.



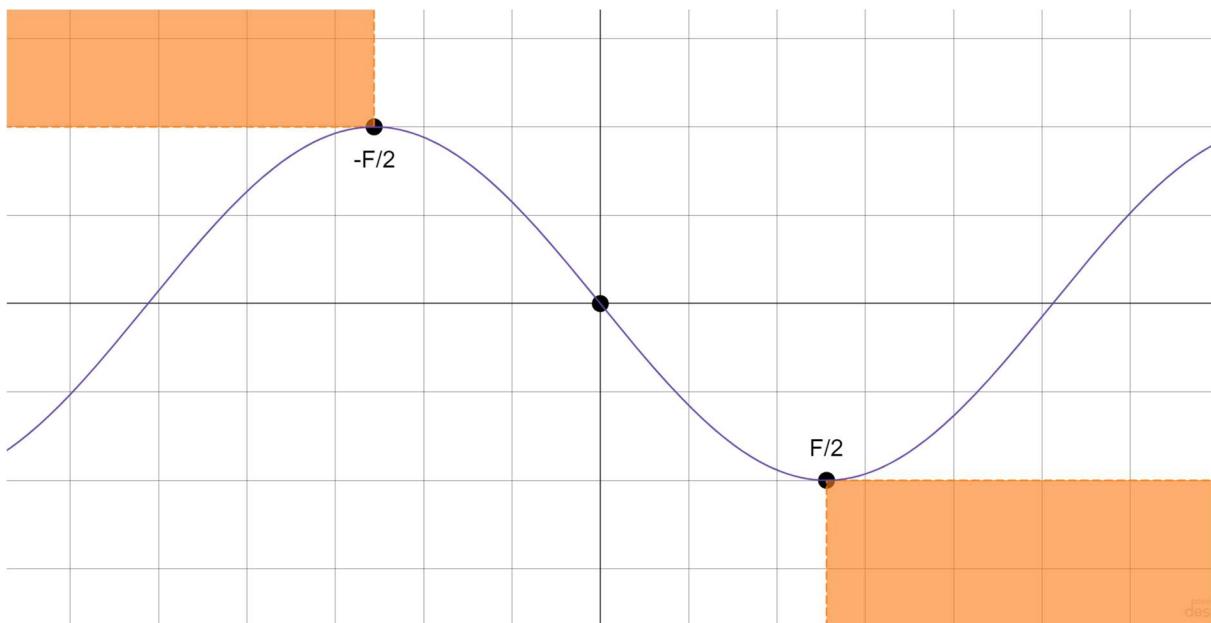
When  $\theta < -\frac{F}{2}$ ,  $H > 1$  which means when the camera is angled above the horizontal with an angle greater than  $\frac{F}{2}$  (it's negative in the equation because up is the negative direction) then the horizon is below the screen, so the ground will not be visible on the screen.

My next thought was to *Let  $H = f(\theta)$* . I already know that  $f(0) = 0$  and that  $f(\theta) > 1$  for  $\theta < -\frac{F}{2}$ , and that  $f(\theta) < -1$  for  $\theta > \frac{F}{2}$ . It is also apparent that  $f\left(\frac{F}{2}\right) = -1$  and  $f\left(-\frac{F}{2}\right) = 1$  by zooming out or setting  $E = 0$ .

This plot shows what information I currently know about the graph of  $H = f(\theta)$ :

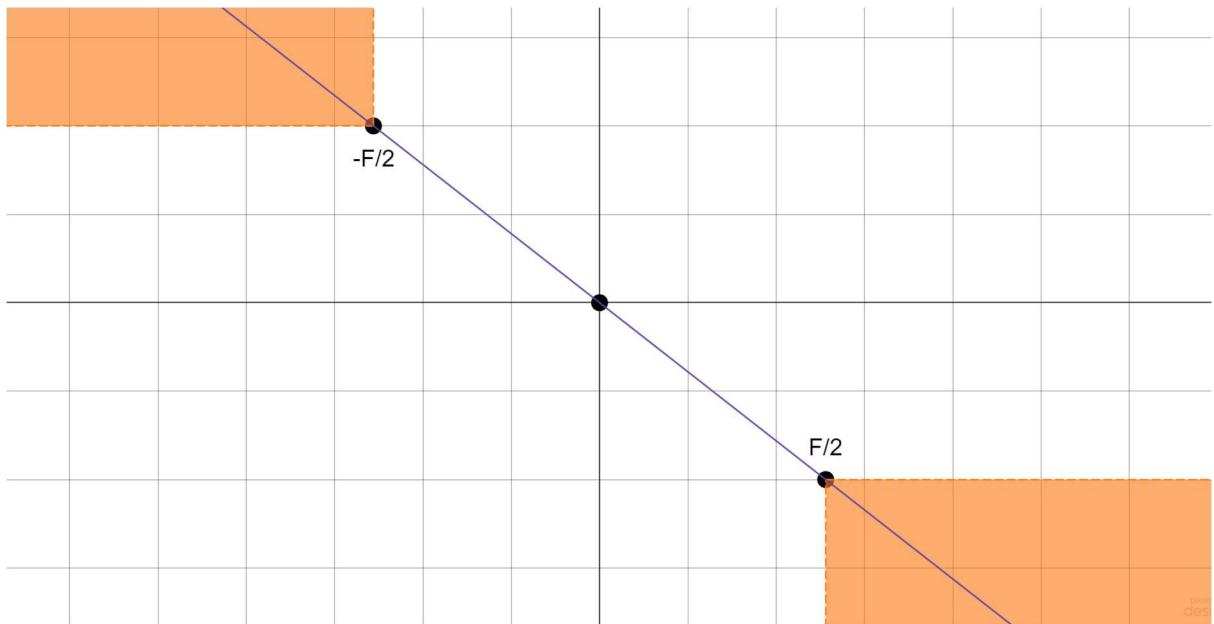


The curve should pass through each labelled point. My initial thought was  $H = -\sin\left(\frac{\theta\pi}{F}\right)$  as this goes through all three points:



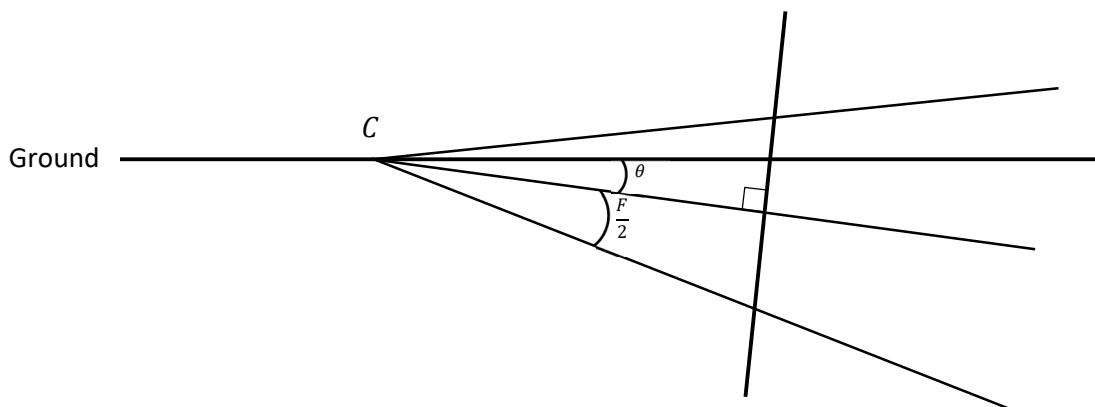
This doesn't work however, because this curve does not go inside the shaded orange region.

I next thought of a straight line going through all three points. Specifically  $H = -\frac{2\theta}{F}$ :

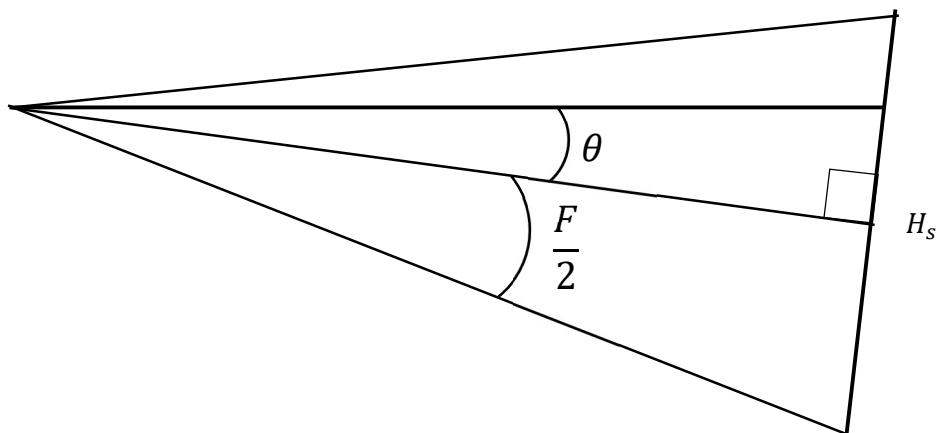


This satisfies all the conditions about the function that we already have figured out, but there are an infinite number of functions which do so. In order to be certain, I had to geometrically find an answer (in the end it wasn't this one).

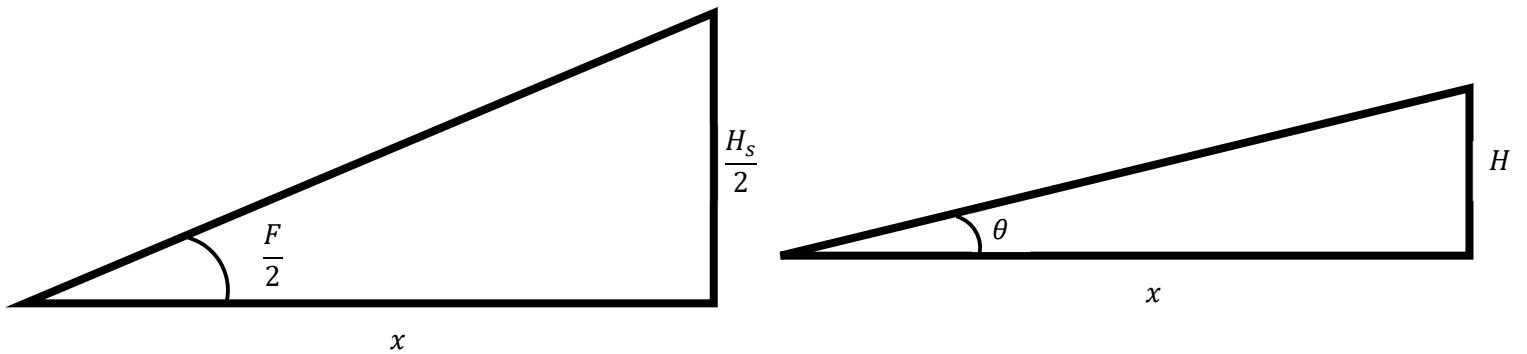
Next I considered the following diagram, letting  $E = 0$ .



The new line is perpendicular to the camera's direction vector and represents the screen. From this diagram, I isolated the following figure:



$H_s$  here represents the height of the screen (in pixels). I then separately considered two right triangles, within this diagram:



$H$  on the right hand triangle is the final value, I want to get. From the left triangle,

$$\tan\left(\frac{F}{2}\right) = \frac{H_s}{2x} \Rightarrow x = \frac{H_s}{2 \tan\left(\frac{F}{2}\right)}$$

From the right triangle,

$$\tan \theta = \frac{H}{x} \Rightarrow H = x \tan \theta = \frac{H_s \tan \theta}{2 \tan\left(\frac{F}{2}\right)}$$

I still need to normalise this value by multiplying by  $\frac{2}{H_s}$  which gives me:

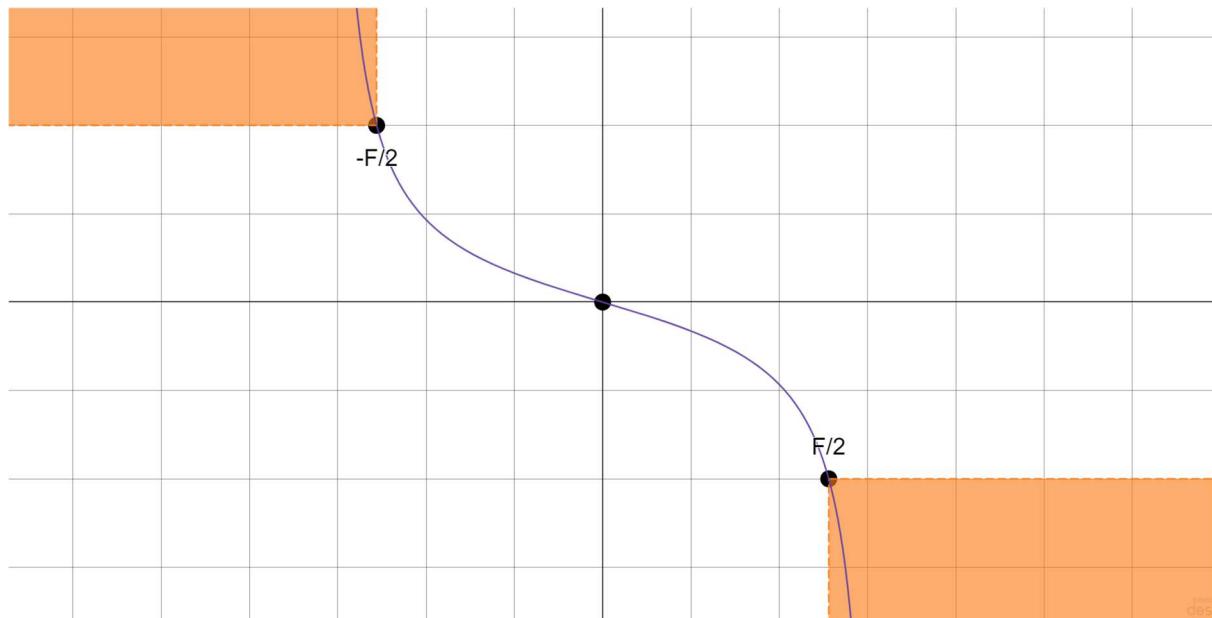
$$H = \frac{\tan \theta}{\tan\left(\frac{F}{2}\right)}$$

Technically this is the magnitude of  $H$  (because I was considering side lengths only), when setting  $\theta = \frac{F}{2}$  or  $\theta = -\frac{F}{2}$  it becomes apparent that the sign needs to be reversed.

Finally, I get the result:

$$H = \frac{-\tan \theta}{\tan\left(\frac{F}{2}\right)}$$

Plotting this on the graph from before (for  $-\frac{\pi}{2} < \theta < \frac{\pi}{2}$ ) to ensure that this solution satisfies the criteria (passes through the points and the orange regions):

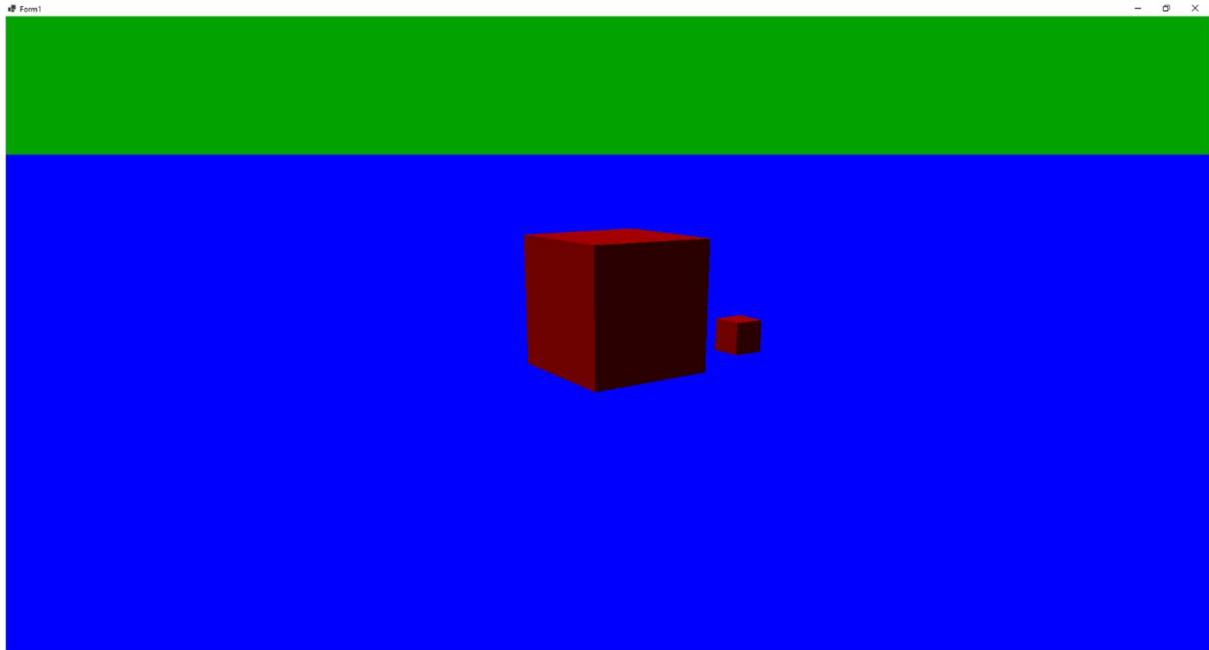


I see that it indeed does, so I can be confident that this is correct.

I wrote the following code which uses the above formula to find the height of the horizon on the screen. This code then denormalises this value, by using the function made back in prototype 1. The appropriate points are then determined, as well as the colour, the shape is then rendered.

```
void renderGround(Ground ground, Camera camera)
{
    double tanTheta = camera.direction.y / Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2));
    double H = -tanTheta / vertTanFov; //Calculates H based on the derived formula
    H = Global.denormaliseY(H, formHeight); //denormalises this value
    double shade = Global.getBrightnessLevel(new Vector(0, -1, 0), 1);
    Vector colour = ground.colour; //colour of the ground
    Brush brush = new SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade), Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade)));
    Point point1 = new Point(0, Convert.ToInt16(H)); //brush and points are created so that the shape can be rendered
    Point point2 = new Point(formWidth, Convert.ToInt16(H));
    Point point3 = new Point(formWidth, 0);
    Point point4 = new Point(0, 0);
    Point[] points = { point1, point2, point3, point4 };
    g.FillPolygon(brush, points);
}
```

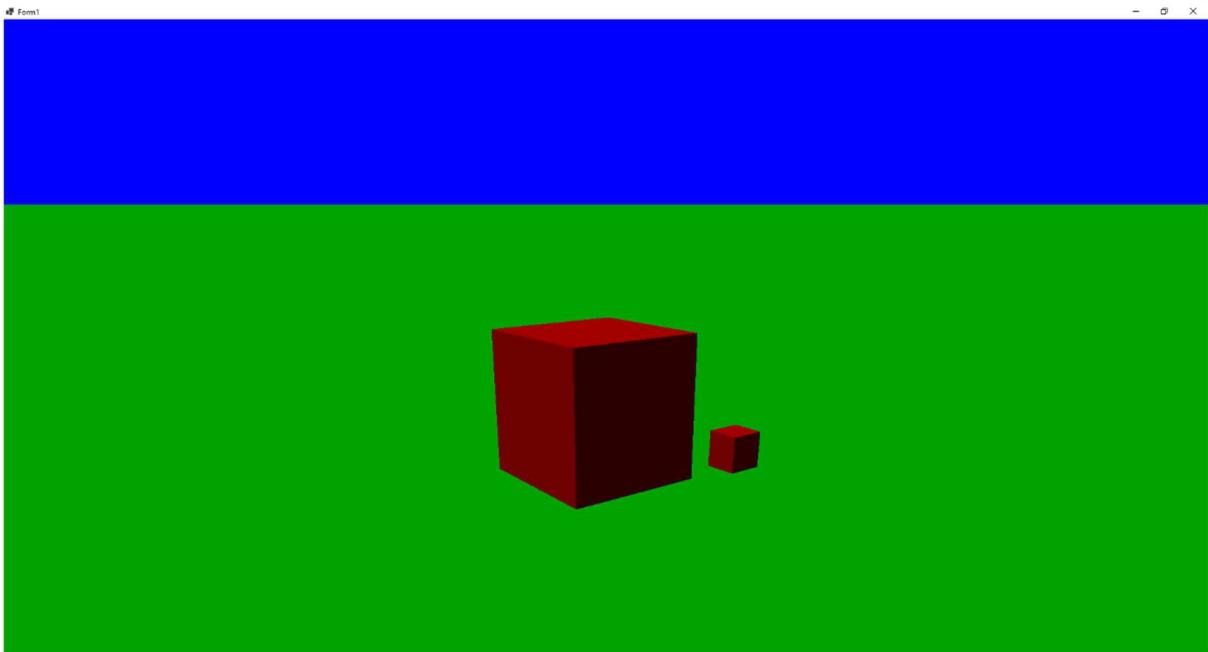
The next step was calling this “renderGround” procedure. This is done in the “renderAll” procedure using the line `renderGround(Global.Ground, camera);`; before anything else is rendered so that it always appears at the back. After these implementations, this is what is seen when the program is run:



The horizon moves as expected when the camera moves, the problem is that the ground is rendering above the horizon line instead of below it. This is because when I defined the points, (e.g., point1, point2, point3, point4), I used 0 to reference the bottom of the screen, but 0 is actually the top, hence why the plane renders at the top. I should have used `formHeight`, instead. These four lines have now been changed to read:

```
Point point1 = new Point(0, Convert.ToInt16(H));
Point point2 = new Point(formWidth, Convert.ToInt16(H));
Point point3 = new Point(formWidth, formHeight);
Point point4 = new Point(0, formHeight);
```

The plane now renders (mostly) normally:



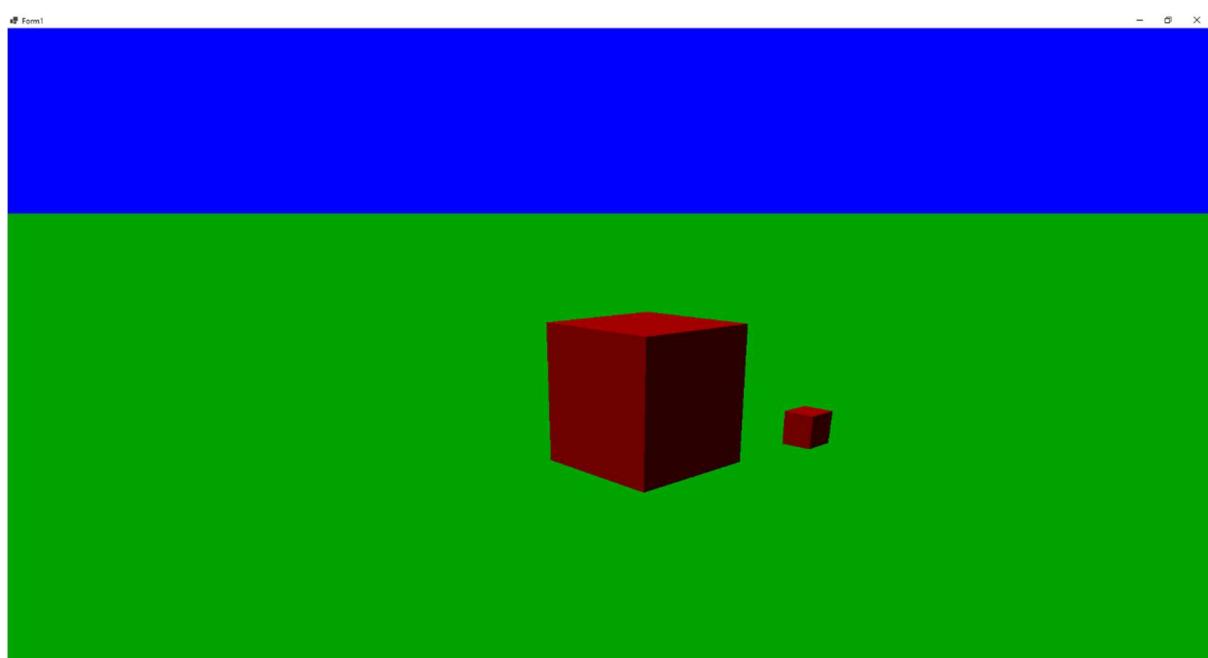
The only problem is the same one which I faced during prototype 1, in which the sign of the z component of the camera's direction, flips everything upside down. In this case that means the height of the horizon changes sign each time the z component changes sign. Fixing this is trivial, I simply added an if statement which flips the sign of  $H$  depending on the sign of the z component. In code, I replaced the line:

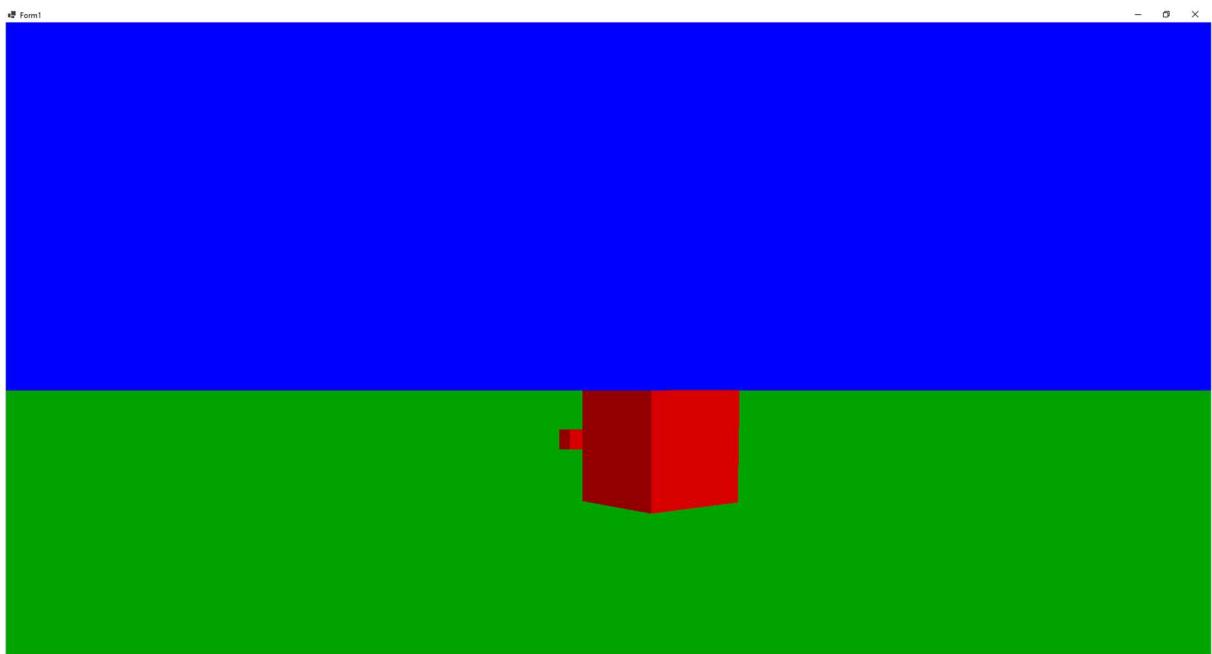
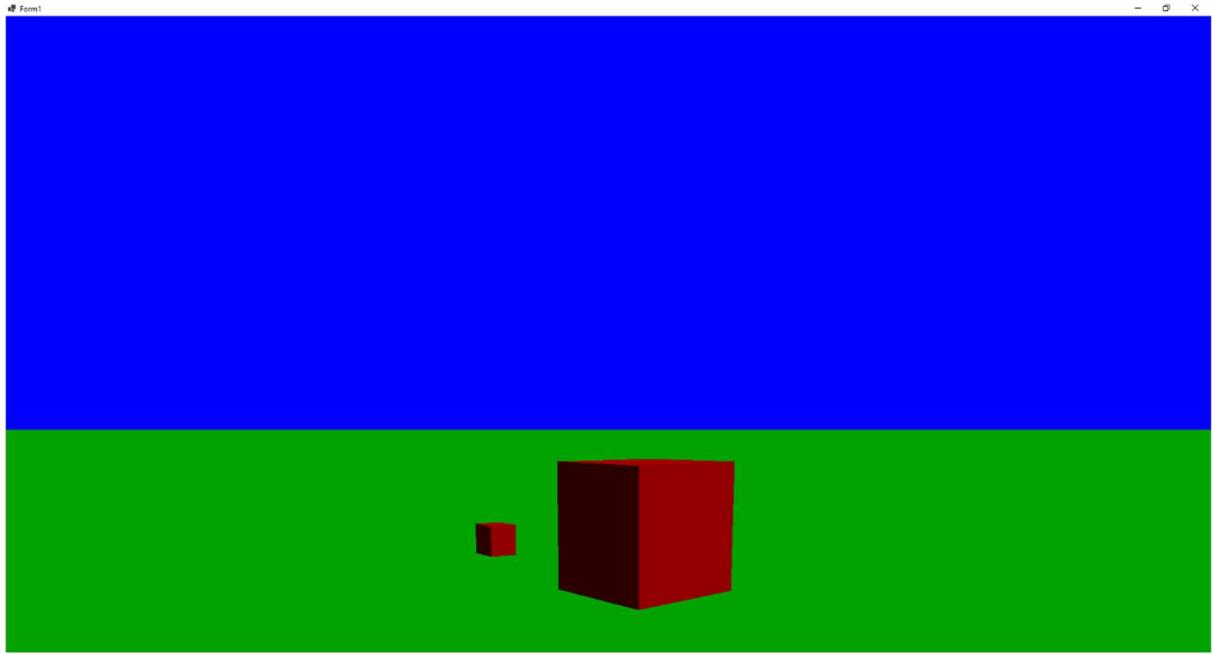
`H = -tanTheta / vertTanFov;`

with:

```
double H;  
if (camera.direction.z >= 0)  
{  
    H = -tanTheta / vertTanFov;  
}  
else  
{  
    H = tanTheta / vertTanFov;  
}
```

The plane now renders correctly in all cases as shown in the next few screenshots and [Video30](#).





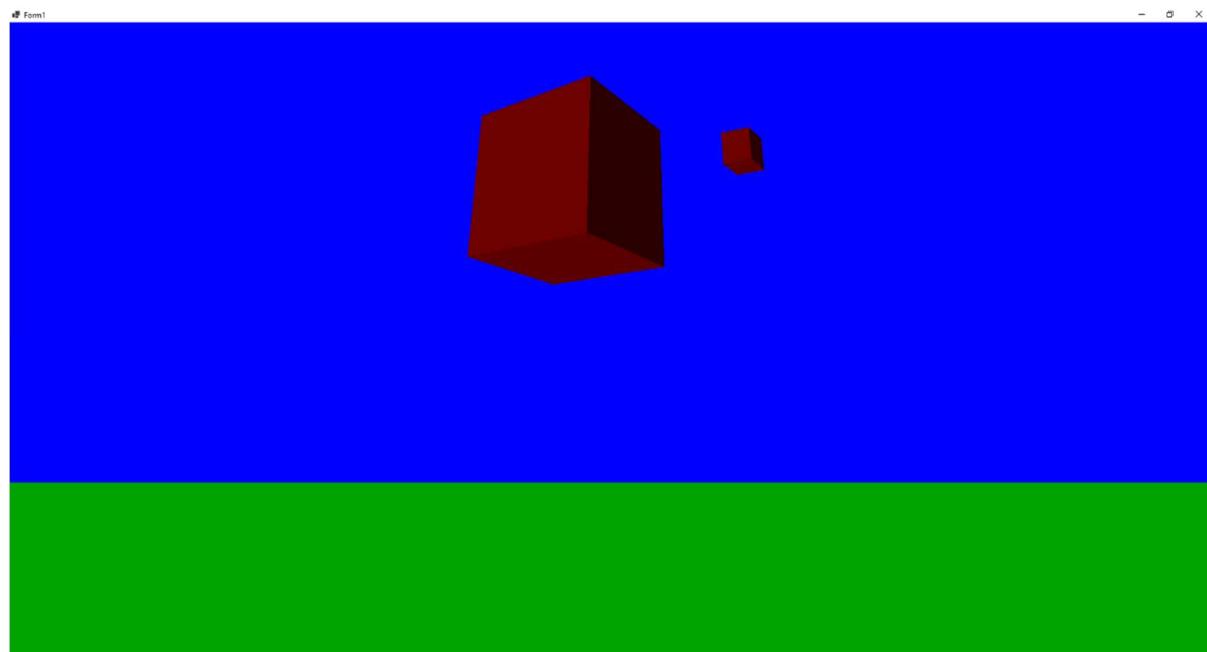
One thing which became apparent about this plane is that (as previously mentioned) the camera's height (or position at all) makes no difference as to how it appears. This can make it very confusing to know where the plane actually is. To address this, I will implement the next feature which is simply to prevent the camera from going beneath the plane when it is enabled. I will also ensure that it is only rendered when it is enabled by adding a simple if statement.

```
if (Global.Ground.toggled)
{
    renderGround(Global.Ground, camera);
}
```

To prevent the camera going below the ground, I will add the same if statement inside the game loop, I will then push the camera back up if necessary.

```
void gameLoop()
{
    if (Global.Ground.toggled)
    {
        if (Global.Camera.position.y > Global.Ground.height)
        {
            Global.Camera.position.y = Global.Ground.height;
        }
    }
}
```

This now works as intended:



Though the fact that this actually works is better demonstrated by [Video31](#). I also seem to have inadvertently added "sneaking" (pressing shift causes the camera to move down slightly, it moves back up when you release shift) but I like it so I am going to leave it in. When setting the "toggled" attribute of the ground instance to False, it does not render and the user's downwards movement is not impeded.

The final thing I need to implement is collisions with the ground. For this I will create a new global function to detect collisions and a global procedure to execute the response.

The collision detection function will check if each vertex of the mesh is intersecting the plane (by checking its y-coordinate). Any which are will be added a list. At the end the list will be returned.

```
public static List<Vector> groundCollisionDetection(Mesh mesh)
{
    List<Vector> points = new List<Vector>();
    for(int i = 0; i < mesh.vertices.Length; i++)
    {
        if (mesh.vertices[i].y > ground.height)
        {
            points.Add(mesh.vertices[i]);
        }
    }
    return points;
}
```

If this list is empty, no response will occur, if it is not then the average of the points in the list will be found. The new collision response procedure will be called with this point.

```
//Collision detection and response with the ground
List<Vector> points = Global.groundCollisionDetection(Global.Entities[i].mesh);
if(points.Count != 0)
{
    Vector position = Global.averagePoint(points.ToArray());
    Global.groundCollisionResponse(Global.Entities[i], position);
}
```

This code was place inside the Physics() procedure after collision flags and before updating the velocities etc of rigid bodies.

The ground response procedure will be the same as the old one, but all references to division by m<sub>2</sub> (the mass of the plane in this place) will be replaced by zero (as the mass of the plane is infinite). Additionally, any other references to properties of the second object will be regarded as zero. The coefficient of restitution used will simply be that of the only object being referenced. These changes make this algorithm much shorter than the other one.

```
public static void groundCollisionResponse(Entity obj, Vector point)
{
    Vector n = new Vector(0, -1, 0);
    double e = obj.rigidBody.CoR;
    double m = obj.rigidBody.mass;
    Vector u = obj.rigidBody.velocity;
    Vector omegaInitial = obj.rigidBody.angularVelocity;
    Vector rp = vectBetweenPoints(obj.rigidBody.centreOfMass, point);
    Vector up = vectorAddVector(u, crossProduct(omegaInitial, rp));
    double part1 = -((e + 1) * dotProduct(up, n));
    Vector part2 = vectorTimesScalar(n, 1 / m);
    Vector v = crossProduct(rp, n);
    Vector tensor1 = obj.rigidBody.inertiaInv;
    Vector part3 = crossProduct(new Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z), rp);
    Vector j = vectorTimesScalar(n, Math.Abs(part1 / dotProduct(vectorAddVector(part2, part3), n)));
    Vector v1 = vectorAddVector(u, vectorTimesScalar(j, 1 / m));
    v = crossProduct(rp, j);
    Vector omegaFinal = vectorAddVector(omegaInitial, new Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z));

    obj.rigidBody.velocity = v1;
    obj.rigidBody.angularVelocity = omegaFinal;
}
```

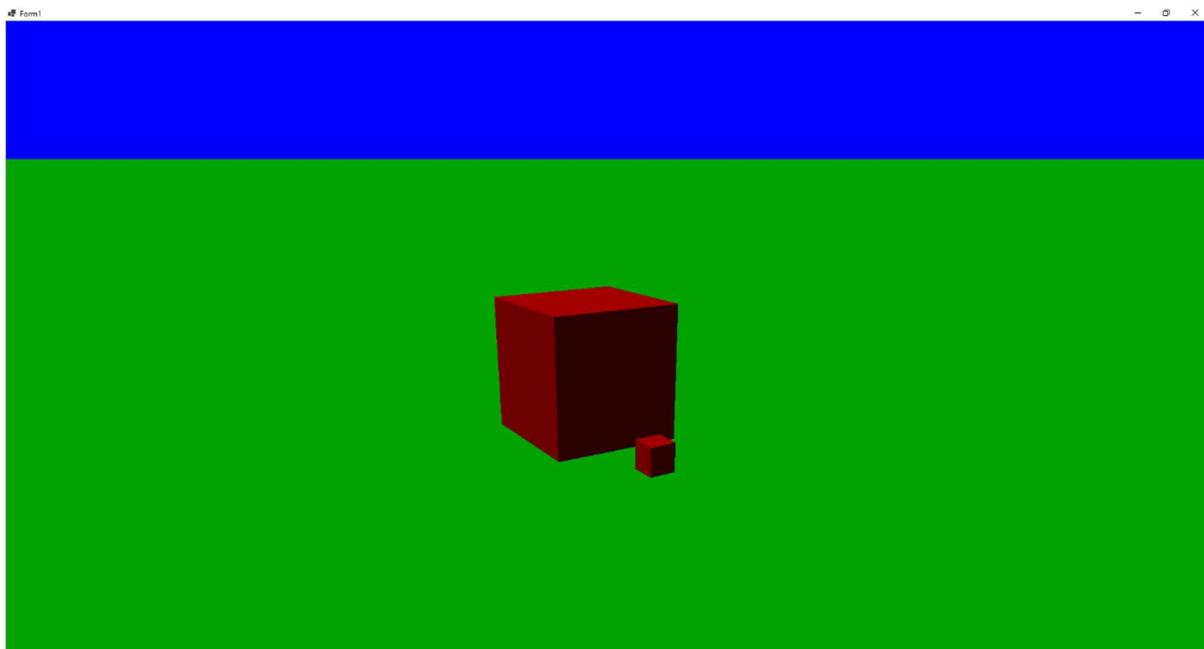
With this code, objects now behave as expected when colliding with the ground (as in they rebound away from it). They do however continue to move through the ground at a constant speed after they should have come to rest.

I created a boolean value, “groundCollided” which is initially set to false. If the object is found to have collided with the ground then the value is set to true. If the value is true then the vertical component of the object’s force is subtracted from the reaction force to get a new reaction force which is instead used. This means that the object’s vertical acting forces are cancelled out by the ground’s normal reaction force. These changes are shown in this code snippet:

```
//Collision detection and response with the ground
List<Vector> points = Global.groundCollisionDetection(Global.Entities[i].mesh);
bool groundCollided = false;
if(points.Count != 0)
{
    Vector position = Global.averagePoint(points.ToArray());
    Global.groundCollisionResponse(Global.Entities[i], position);
    groundCollided = true;
}

//Find with which object and normal (if any) this object collided with each frame and apply normal reaction
Vector reaction = new Vector(0, 0, 0);
for (int j = 0; j < check.Count; j++)
{
    if (Global.Entities[i] == check[j].Item2)
    {
        reaction = Global.getNormalReactions(check[j].Item2, check[j].Item3, Global.normalOfClosestFace(check[j].Item3));
    }
    if (Global.Entities[i] == check[j].Item3)
    {
        reaction = Global.getNormalReactions(check[j].Item2, check[j].Item3, Global.normalOfClosestFace(check[j].Item2));
    }
}
if(groundCollided)
{
    reaction = Global.vectorAddVector(reaction, new Vector(0, -Global.Entities[i].rigidBody.force.y, 0));
}
```

The cropped-out part is not new. This screenshot shows the cubes resting on the plane after falling due to gravity:



The ground is now complete. I will next create menus for the simulations and separate the form into three different ones for each simulation.

## Simulation Menus

I started by duplicating and renaming the RigidBodyCollisions form twice and renamed the duplicates. I now have forms for all three simulations (though they are currently identical). I now added functionality to the buttons on the menu screen, to close the menu and open the new window. Copying and pasting the entire form lead to some errors, so instead I created two new forms and just copied and pasted the code instead.

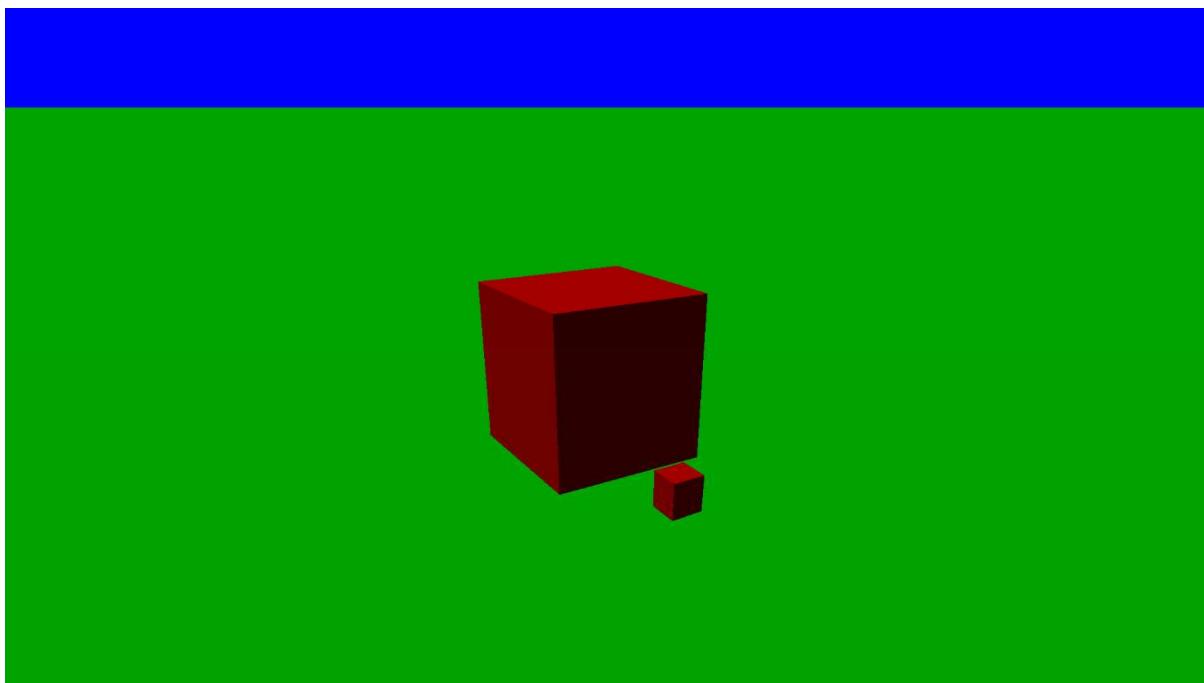
This is the code to make the menu buttons open the corresponding simulation form in full screen with no border, and then hide the menu form:

```
private void Play_Sim1_Click(object sender, EventArgs e)
{
    RigidBodyCollisions sim = new RigidBodyCollisions();
    sim.WindowState = FormWindowState.Maximized;
    sim.FormBorderStyle = FormBorderStyle.None;
    sim.Show();
    this.Hide();
}

1 reference
private void Play_Sim2_Click(object sender, EventArgs e)
{
    ProjectileMotion sim = new ProjectileMotion();
    sim.WindowState = FormWindowState.Maximized;
    sim.FormBorderStyle = FormBorderStyle.None;
    sim.Show();
    this.Hide();
}

1 reference
private void Play_Sim3_Click(object sender, EventArgs e)
{
    Orbit sim = new Orbit();
    sim.WindowState = FormWindowState.Maximized;
    sim.FormBorderStyle = FormBorderStyle.None;
    sim.Show();
    this.Hide();
}
```

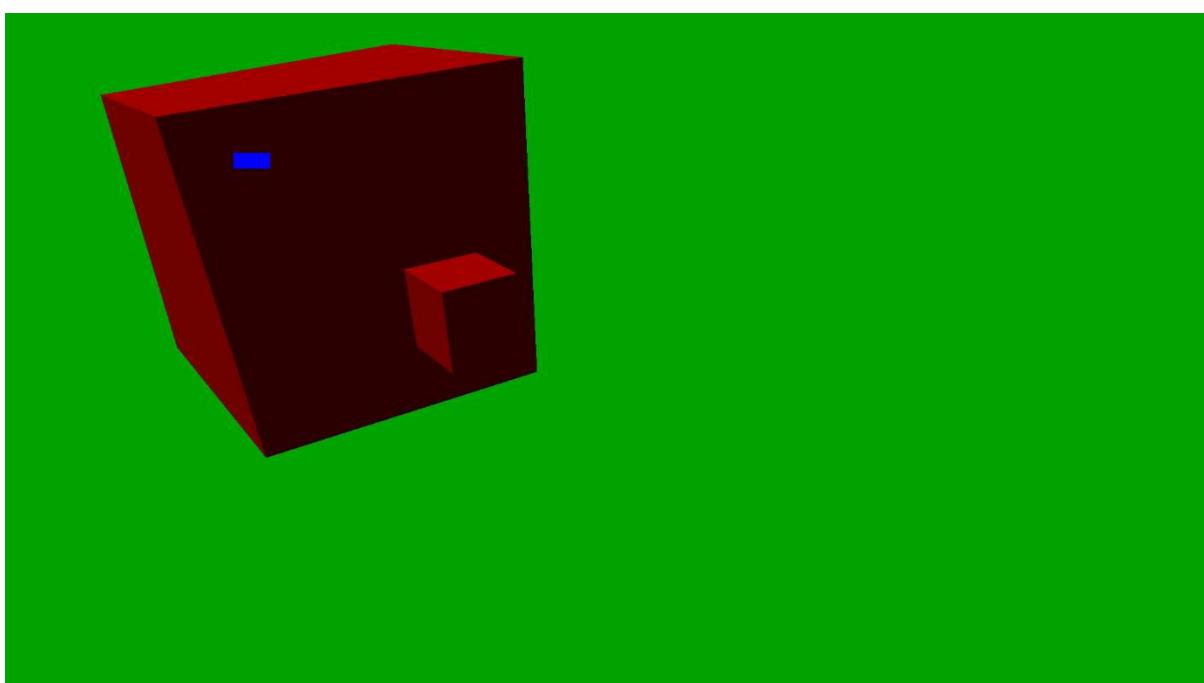
This screenshot shows the “projectile motion” simulation open in fullscreen. Notice the lack of border.



This simulation currently looks identical to what we already had but this will be changed when I make the three simulation forms different.

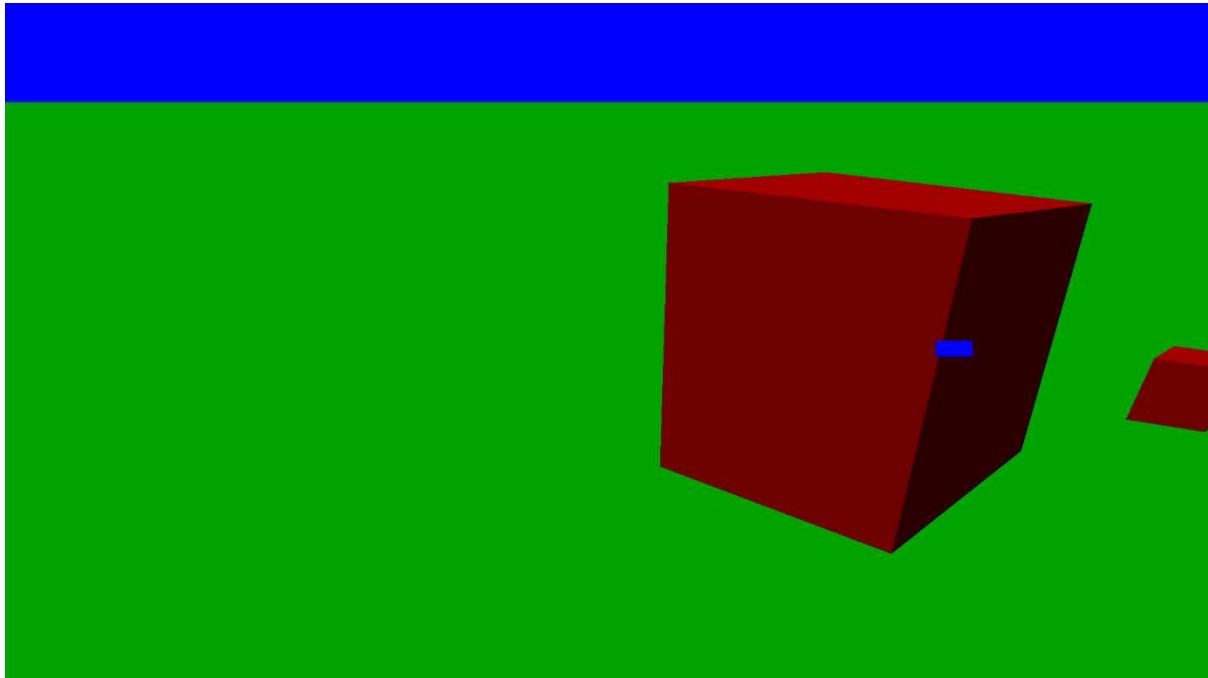
[Video32](#) shows this form being opened from the menu screen. The same happens with all three. I have set the accelerations of each object to zero for the next section as we are no longer testing physics.

The next thing I want to see before trying to create menus is if user input (namely camera movement) still works when there are UI elements (such as buttons and labels) on the screen. I also want to see if these UI elements will render in front of or behind whatever has been painted (I want them to render in front so that they are always visible). I first created a single label in the centre of the form to see what would happen. This was the result:

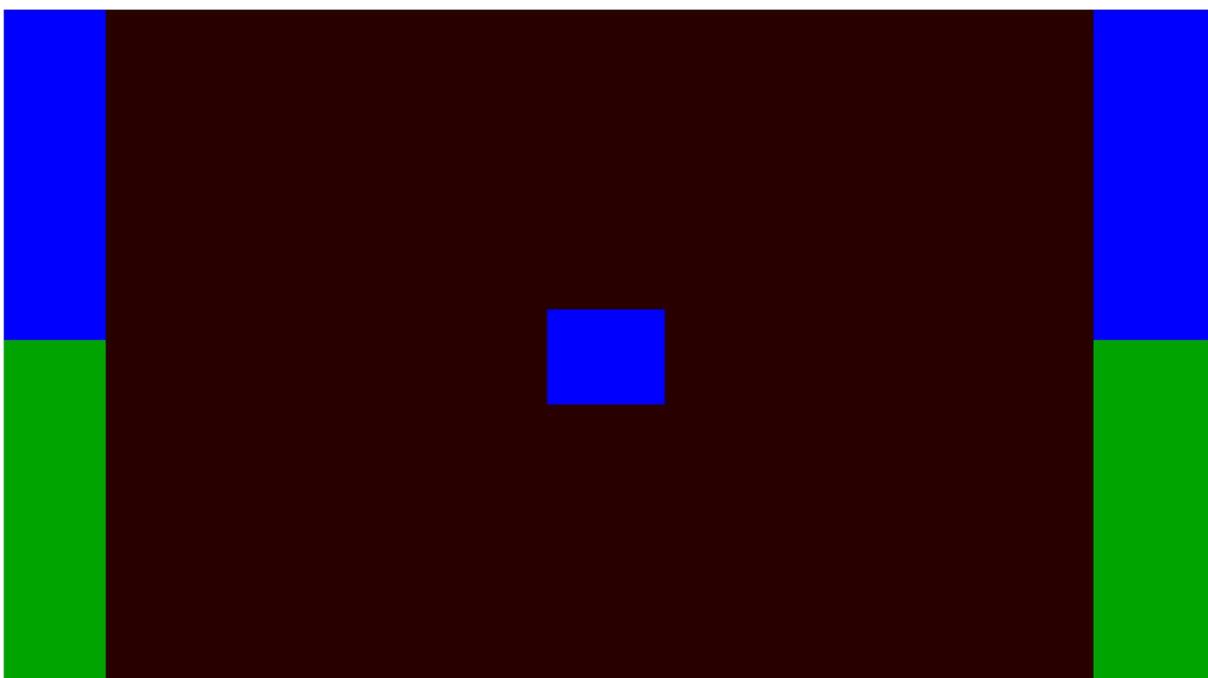


I was still able to control the camera despite the label which is good. I placed the label in the centre of the form in the designer, but it appears here to the top left. This is because the designer uses absolute coordinates relative to the top left corner, to fix this, I could set it to align to the centre (by setting the “anchor” property to “None”), but since the final menu I am aiming to create will be on the right side of the form, I will instead set it to be relative to the right side of the form by setting “Anchor” to “Right”.

It now appears anchored to the right-hand side as expected:



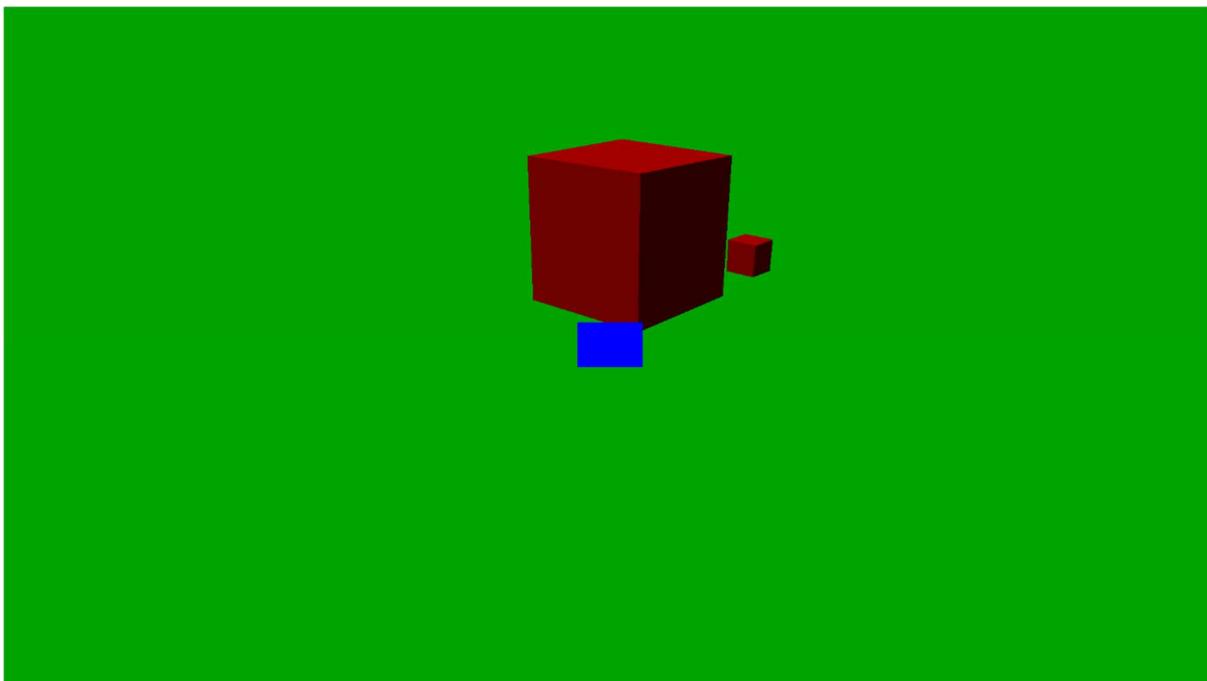
The other obvious problem is that this label is not displaying any text (it should display the string “label1”), instead it appears as a blue rectangle. This is because the background colour of the label is set to blue. I will later be able to change this to whatever colour I choose to make the background colour of the menu. The issue of not displaying the text it is supposed to hold will be dealt with later, for now I will see if buttons work properly. This is the result of creating a large button in the centre of the screen:



The result seems the same as before, except I now cannot move the camera. I also changed the background colour of the button to white, but the result was the same. This means that the reason why the button (and the label before) appears as a blue rectangle is not because that is the background colour of the UI element, but because that is the background element of the form though I still don't know why this happens. I do know how to address the issue of keyboard inputs not being registered.

The reason why this happens is because the button is looking for keyboard inputs which means that the rest of the program is unable to. To resolve this, I will disable the button, and have it enabled on when the mouse is hovering over it.

After setting its "Enabled" property to false:



the keyboard inputs work as expected, but it is still blue.

After a few hours with no luck, I have had an idea, since it seems unlikely that I will be able to get the buttons and painting to the screen to work simultaneously, I will use a keyboard shortcut, (esc) to take the user to a menu from which they can access all of the previously discussed properties. I do not plan on making this menu a different window. What I will do is, when the user presses escape, a property "paused" will be toggled between true and false. Each time a game loop is going to be run, a check on this value is done. If it is set to false, the program will run as it currently does. If not, the program's operations will be paused and instead, the screen will be cleared and the background colour changed, with a menu with all of the buttons etc., I require being drawn. This menu will also have access to the settings form and an option to exit the simulation and relocate back to the main menu. I created "paused" as a global variable which is set to true upon loading the form. When the escape key is pressed, the value is toggled by this code:

```
if (e.KeyCode == Keys.Escape)
{
    Global.paused = !Global.paused;
```

This code causes the code to operate as it currently does only when paused is set to false:

```
if (Global.paused == false)
{
    Stopwatch timer = new Stopwatch();
    timer.Start();
    gameLoop();
    timer.Stop();
    Global.frameTime = timer.Elapsed.TotalSeconds;

    if (Global.frameTime < 1.0 / Global.fps)
    {
        Stopwatch timer2 = new Stopwatch();
        timer2.Start();

        while (timer2.Elapsed.TotalSeconds < 1.0 / Global.fps - Global.frameTime)
        {
        }

        timer2.Stop();
        Global.frameTime = 1.0 / Global.fps;
    }
}

Invalidate();
```

With this, pressing the escape key causes the screen to go blank, pressing it again causes it to resume as it was as shown in [Video33](#). This is the expected result. There is no need for me to wipe the screen if the property is true because if it is true, then nothing is drawn in the first place. For the menu to show, I must create each of the buttons, labels, etc., disable them all and make them invisible if “paused” is false, and set them to enabled and visible when “paused is true”. After making these changes, the above code now looks like this with new parts highlighted in blue:

```
if (Global.paused == false)
{
    for (int i = 0; i < this.Controls.Count; i++)
    {
        this.Controls[i].Visible = false;
        this.Controls[i].Enabled = false;
    }
    Stopwatch timer = new Stopwatch();
    timer.Start();
    gameLoop();
    timer.Stop();
    Global.frameTime = timer.Elapsed.TotalSeconds;

    if (Global.frameTime < 1.0 / Global.fps)
    {
        Stopwatch timer2 = new Stopwatch();
        timer2.Start();

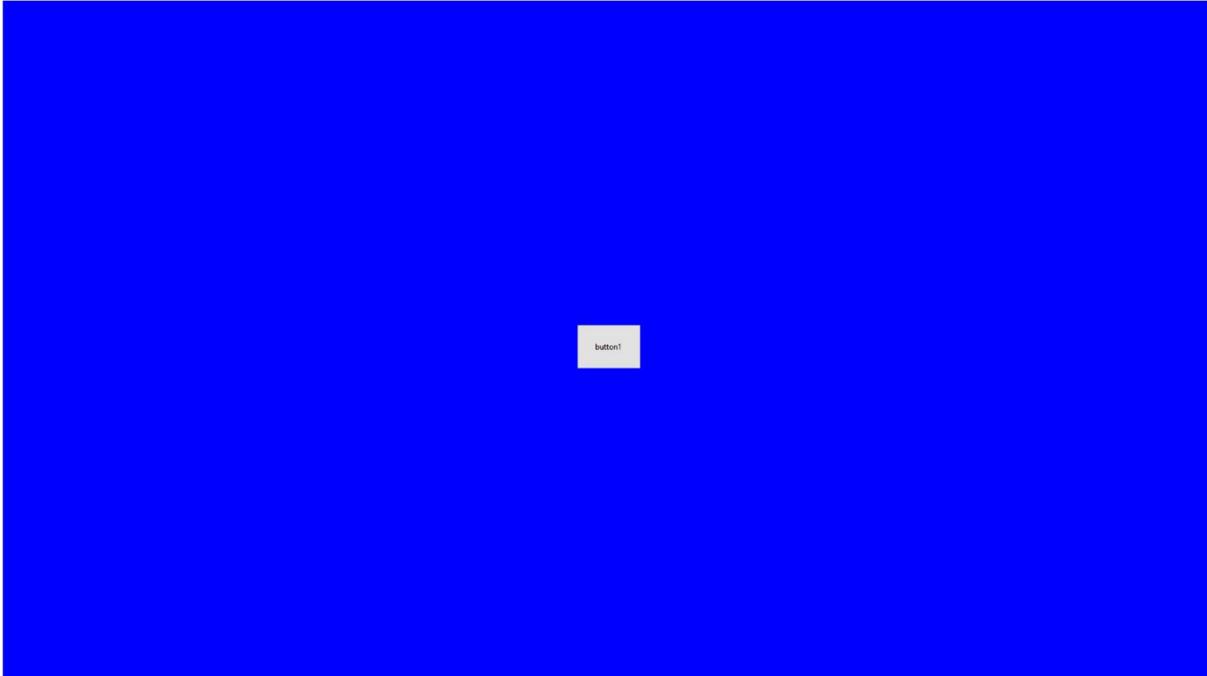
        while (timer2.Elapsed.TotalSeconds < 1.0 / Global.fps - Global.frameTime)
        {

        }

        timer2.Stop();
        Global.frameTime = 1.0 / Global.fps;
    }
}
```

```
else
{
    for (int i = 0; i < this.Controls.Count; i++)
    {
        this.Controls[i].Visible = true;
        this.Controls[i].Enabled = true;
    }
}
Invalidate();
```

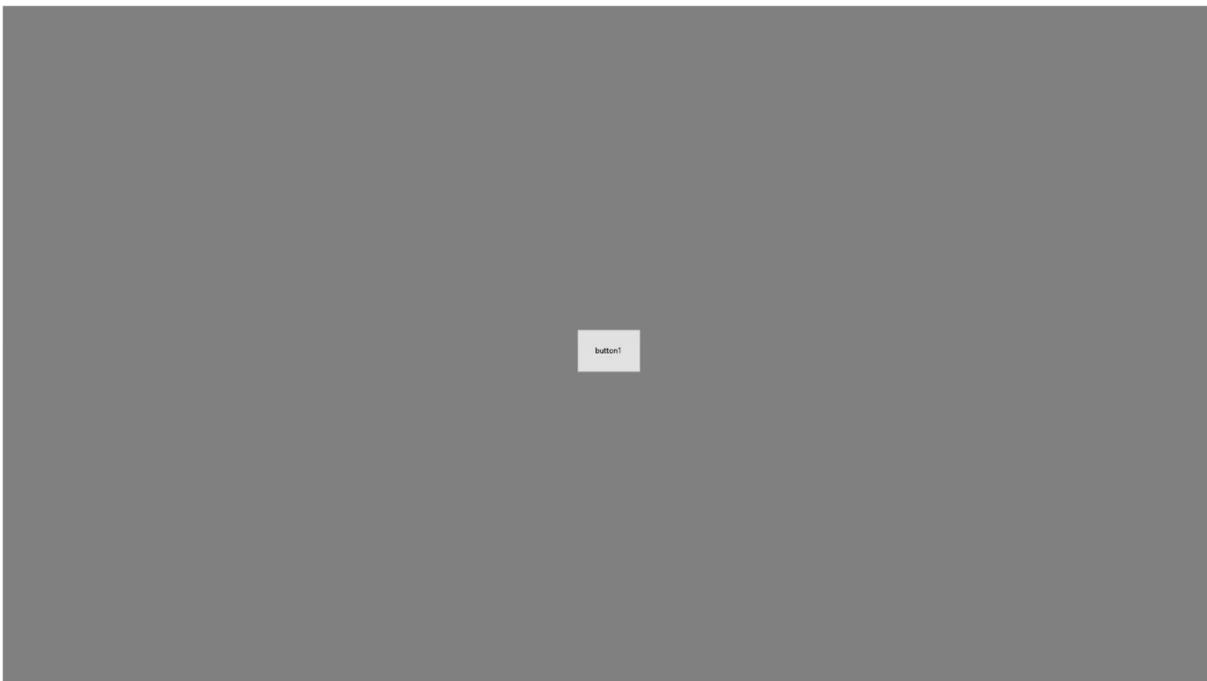
This is what is now seen, whenever the escape key is pressed.



The program resumes and the button disappears when escape is pressed again. I have added this line of code to change the background colour between blue and grey depending on the state of "paused".

```
this.BackColor = Color.Blue;
this.BackColor = Color.Gray;
```

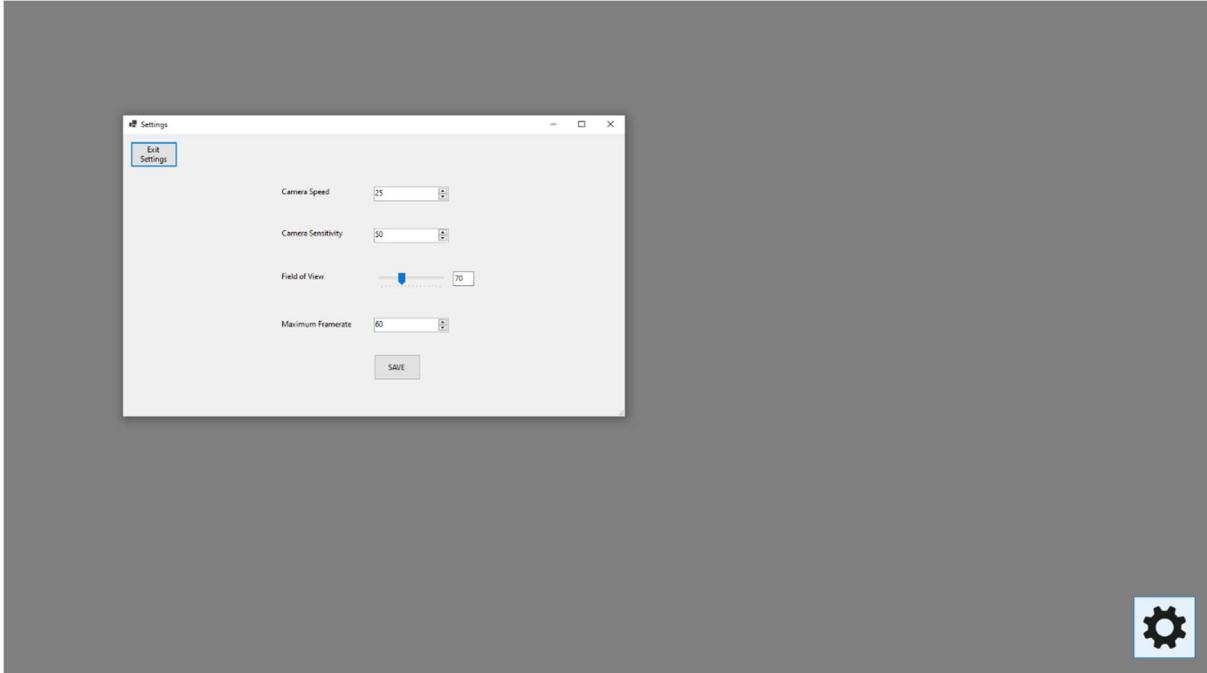
The new menu is shown here:



The background colour switches itself back as intended, when the simulation is resumed.

The full result of this new menu is shown in [Video34](#).

I made the above button into a settings button to appear in the bottom left corner. I gave it the same code as the settings button in the menu, so the user can now open the settings page from within a simulation.



I have noticed a single problem. When I interact with the button, the escape key to close the menu stops working. This is because the button is enabled and is looking for keyboard inputs. This overrides the part of my program which recognises escape key input. This is easily resolved by adding the line "`this.KeyPreview = true;`" to my form code, just after the line "`InitializeComponent();`". After this step, the menu works perfectly. The next

button I will add is to close the form and reopen the menu form. If you recall, the menu was never actually closed, only hidden meaning I need to show it.

To do this, I would need a way of referencing the hidden menu form from within the RigidBodyCollisions form. I created an instance of Menu in the Global class by writing:

```
public static Menu main_menu;
```

Next, I set this to be the instance of the menu which is created when the program loads, by writing the following code in the Menu form:

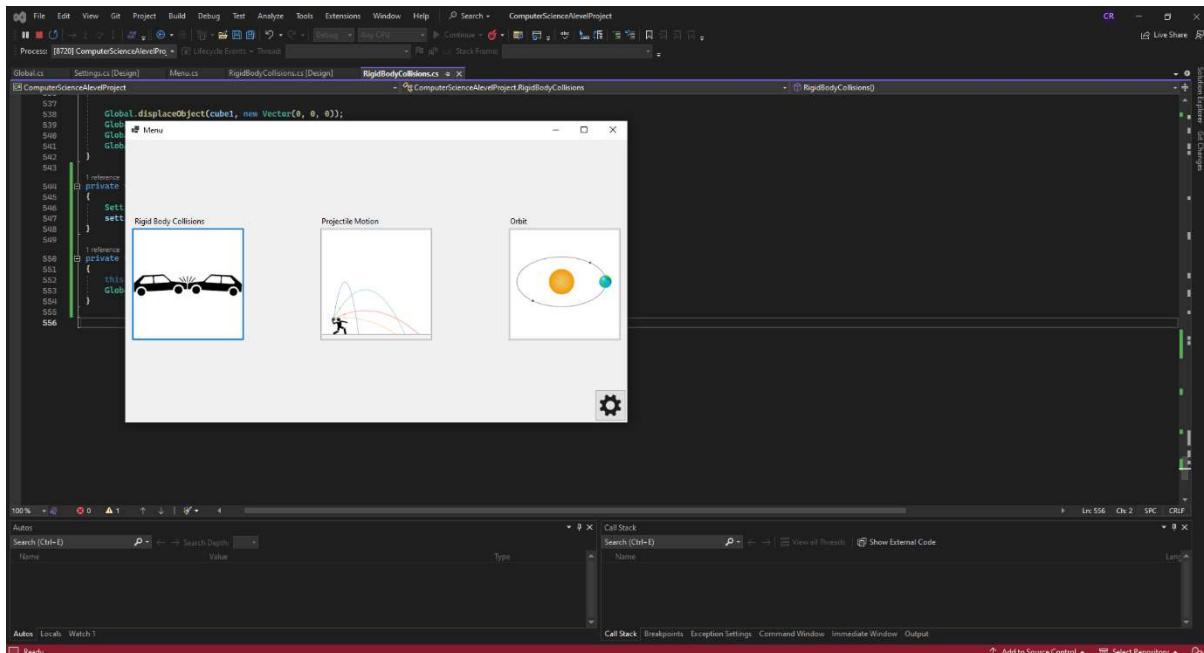
```
private void Menu_Load(object sender, EventArgs e)
{
    Global.main_menu = this;
}
```

Finally, I referenced this Global variable, in order to show the menu form after closing the current form when the new exit button is clicked:

```
private void Exit_Click(object sender, EventArgs e)
{
    this.Close();
    Global.main_menu.Show();
}
```

The result is that the window closes when this button is clicked, and the main menu is shown again.





This is more clearly shown to be working in [Video35](#).

Before adding the next set of buttons, I have removed the default cubes. I also used the line “`Global.Entities = new List<Entity>();`” to reset the list of entities each time the form is opened because this means that simulations cannot be interfered with by those which were previously running.

Instead of using the three tabs outlined in design, I will instead set out the settings in three columns, beneath headings as shown here:



The “Object Creation” heading will need controls for mass, position, velocity, dimensions, colour, resultant force (excluding gravity), and a button to create the object.

The “Alter Environment” heading should have controls for acceleration due to gravity, background colour.

The “Alter Objects” column will have a drop-down list allowing the user to select an object. They will then be able to alter this objects’ properties including position, rotation, velocity, colour, resultant force, etc., To make this part easier for the end user, I will add the attribute “name” to the Entity class and will add the option to enter a name under “Object Creation”.

I have added the labels, which now look like this:

Main Menu	Object Creation	Alter Environment	Alter Objects
<p>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)"</p>	<p>Mass: <input type="text"/> kg</p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Dimensions: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p>	<p>Strength of: <input type="text"/> Gravity</p> <p>Background: <input type="text"/> Colour</p>	
	<p>Create Object!</p>		

For some reason, the simulation (which is currently paused) is visible through the textboxes. I attempt to fix this, but I think it actually makes the menu feel more inviting, like it has a window to the outside.

## More Input Validation

I will need ensure that the inputs to these textboxes are as expected before I start trying to do things with these inputs. For this I will need to use input validation. If the input is not as expected, I will display an error message in red, next to the create button using a label. I created a label with an error message, which is disabled and invisible. It only becomes visible when invalid data is entered and the create button is clicked.

Input validation code: (I will explain on the next page).

```
private void Create_Click(object sender, EventArgs e)
{
    //Validate user inputs
    bool check = true;
    string pattern = @"^(\()?\d+(\.\d+)?,( +)?\d+(\.\d+)?,( +)?\d+(\.\d+)?(\())?$$";

    string sMass = massInput.Text;
    if (!double.TryParse(sMass, out double ...))
    { check = false; }
    string sPosition = positionInput.Text;
    if (!Regex.IsMatch(sPosition, pattern))
    { check = false; }
    string sVelocity = velocityInput.Text;
    if (!Regex.IsMatch(sVelocity, pattern))
    { check = false; }
    string sDimensions = dimensionsInput.Text;
    if (!Regex.IsMatch(sDimensions, pattern))
    { check = false; }
    string sColour = colourInput.Text;
    if (!Regex.IsMatch(sColour, pattern))
    { check = false; }
    string sForce = forceInput.Text;
    if (!Regex.IsMatch(sForce, pattern))
    { check = false; }

    if (check == true)
    {
        //Create object
    }
    else
    {
        //Display validation label
        ValidationLabel.Visible = true;
        ValidationLabel.Enabled = true;
    }
}
```

The first input (mass) is simply checked to be a number (double specifically) by using double.TryParse. If the check fails, then it is an invalid input so check is set to false.

The rest of the inputs must be of the form (number1, number2, number3). To ensure that they are, I have used Regex (regular expression) which checks each string against a “pattern” string which I have created to ensure that they match. My pattern string was:

`@""^(\()?\d+(\.\d+)?,( )?\d+(\.\d+)?,( )?\d+(\.\d+)?(\())?$$"`

The @ is simply to let the program know that the following string is a pattern. The ^ and \$ signify the beginning and end of the pattern respectively. “\(/“ means “(“ the backslash is necessary because brackets are used elsewhere to signify other things. Putting this in brackets and following with a question mark, “(\()?” means “(“ but now it’s optional. “\d” means a digit from 0 to 9, adding a “+”: “\d+” now means at least 1 digit from 0 to 9 in a row. “.\d+” means a period, “.” followed by at least 1 digit. Placing this in brackets and adding “?”, I get “(.d\+)?” meaning the same thing, but now it is optional. The comma is just a comma. “( +)?” means any number of spaces (also optional). At the end, “(\())?” means “)” (optional). The rest is repeated what I have already said. All together, I get the above expression which means:

“( (optional) followed by some number (with or without a decimal point with numbers after it) followed by a comma followed by some spaces (optional) followed by... followed by ”) optional. All in all, these are some examples of valid inputs:

“(12, -15, 0)”

“12,32,12.8”

“0.4, 99, -2)”

and these are examples of invalid inputs:

“(four, eight, -three)” – Should be (4, 8, 3)

“(., .12, -9)” – Should be (0.4, 0.12, -9)

“(7, 18, 5.)” – Should be (7, 18, 5)

To ensure that the validation label is invisible when it should be, I had to make the part of the code which makes all controls visible when the simulation is paused, ignore this one control:

```
else
{
    this.BackColor = Color.Gray;
    for (int i = 0; i < this.Controls.Count; i++)
    {
        if (Controls[i] != ValidationLabel)
        {
            this.Controls[i].Visible = true;
            this.Controls[i].Enabled = true;
        }
    }
}
```

I first entered what should be valid inputs into each box and clicked the button, this was the result:

# Object Creation

tion,  
nsions,  
ant force  
d in the  
. Mass  
mber

**Mass:**  kg

**Position:**

**Velocity:**

**Dimensions:**

**Colour:**

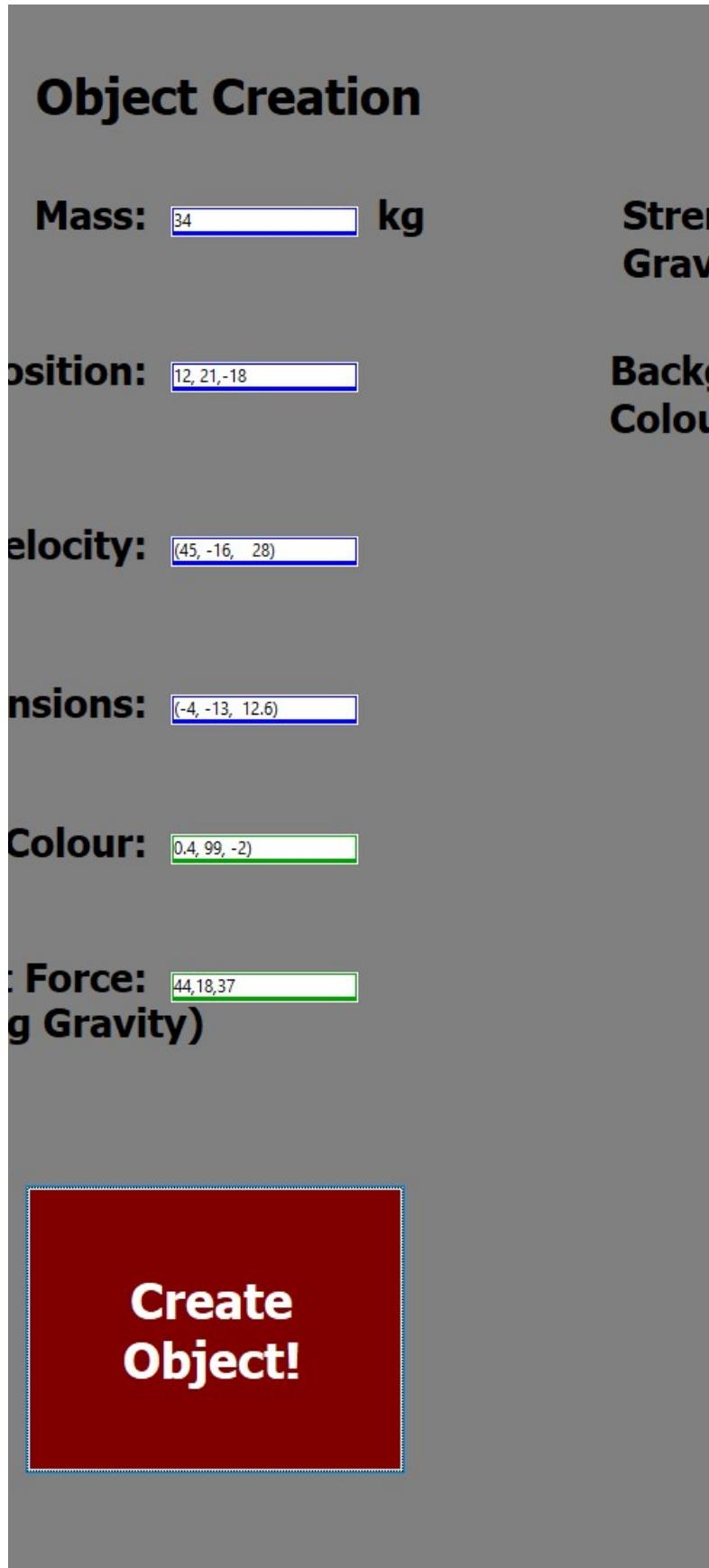
**Resultant Force:  
(Excluding Gravity)**

**Create Object!**

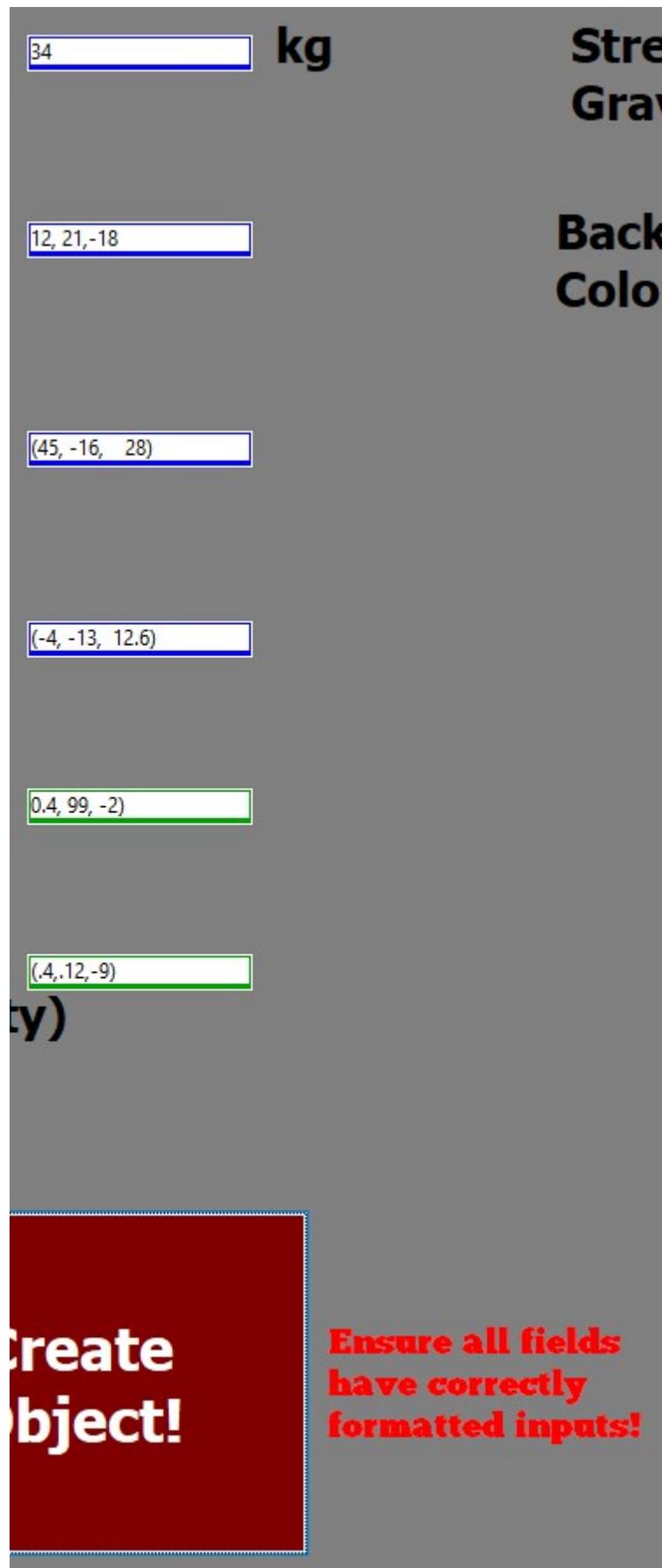
Ensure all fields have correctly formatted inputs!

The error message appeared, despite what I thought were valid inputs. The reason why this happened is because I was using negative numbers and my pattern does not check for these. To fix this, I will add optional negative signs into the string “(-)?”.

`@"^(\\()?-\\d+(.\\d+)?,(\\+)?(-)?\\d+(.\\d+)?,(\\+)?(-)?\\d+(.\\d+)?(\\()?)?"`  
After this change, negative numbers are accepted as valid inputs.



Next I replaced the last string with one which should be invalid “(.4, .12, -9)” (third example from before).



And this input was rejected as expected.

### User Created Objects

I will now finally allow users to create their own objects. The first step is to define the vertices of the cube. I will do this centred at the origin, I will then translate the entire object to its correct position at the end using Global.displaceObject().

Before I can start using the user inputs (which were strings) I first need to extract from these strings the relevant data. I can do this by using more of the Regular Expressions Library. The pattern of each of my inputs (except mass which is trivial) is as follows:

(open bracket) + (negative sign [optional]) + (digits) + (decimal point and more digits [optional]) + (comma and spaces [spaces optional]) + (negative sign [optional]) + (digits) + (decimal point and more digits [optional]) + (comma and spaces [spaces optional]) + (negative sign [optional]) + (digits) + (decimal point and more digits [optional]) + (close bracket [optional]).

Each of these is considered as a RegEx Group which can be handled separately. Groups[0] is the entire string, Group[1] is the opening bracket (empty string if one was not used), Group[2] is the negative sign etc., According to this numbering scheme, groups 2,3 and 4 contain the first number, 6, 7 and 8 contain the second, and 10, 11 and 12, the third. The following code then, should assign the length width and height as entered by the user:

```
Match match = Regex.Match(sDimensions, pattern);
double depth = Convert.ToDouble(match.Groups[2].Value + match.Groups[3].Value +
match.Groups[4].Value);
double height = Convert.ToDouble(match.Groups[6].Value + match.Groups[7].Value +
match.Groups[8].Value);
double width = Convert.ToDouble(match.Groups[10].Value + match.Groups[11].Value +
match.Groups[12].Value);
```

I used Debug.WriteLine(depth+" , "+height+" , "+width) to print out the new width, height and depth. I opened the form and entered the string “(-18.5, 12.23, -32)” into the “Dimensions” box and clicked the create button, the expected result was:

“-18.5, 12.23, -32”

I instead got an error message claiming that my string was in the incorrect format (which it wasn't as it was already verified by my input validation). I next used the line Debug.WriteLine(match.Groups.Count) to find out how many groups my string was split into, it was 11. I then used a for loop to print the contents of each group:

```
for (int i = 0; i < 11; i++)
{
    Debug.WriteLine(match.Groups[i].Value);
}
```

The output was:

```
(-18.5, 12.23, -32)
(
-
.5

.23
-
)
```

The problem is that the digits before the decimal point are not being recognised as groups so they are being skipped over. To resolve this, I placed brackets around these parts in the pattern, which now looks like this:

```
@"^(\\()?-\\)?(\\d+)(\\.\\d+)?,(\\ +)?(-)?(\\d+)(\\.\\d+)?,(\\ +)?(-)?(\\d+)(\\.\\d+)?(\\))?$"
```

The result from:

```
Debug.WriteLine(depth+, "+height+, "+width);
```

is now: “-18.5, 12.23, -32” as expected.

Instead of storing width, height and depth as three separate variables, I will store them instead inside of a vector. Since I will be doing this to each input box, I decided it would make sense to have a function do this for me. I created a function called regExInterp in the global class to interpret these regular expressions for me and return the corresponding vector,

```
public static Vector regExInterp(string str)
{
    string pattern = @"^(\\()?-\\)?(\\d+)(\\.\\d+)?,(\\ +)?(-)?(\\d+)(\\.\\d+)?,(\\ +)?(-)?(\\d+)(\\.\\d+)?(\\))?$";
    Match match = Regex.Match(str, pattern);
    double v1 = Convert.ToDouble(match.Groups[2].Value + match.Groups[3].Value + match.Groups[4].Value);
    double v2 = Convert.ToDouble(match.Groups[6].Value + match.Groups[7].Value + match.Groups[8].Value);
    double v3 = Convert.ToDouble(match.Groups[10].Value + match.Groups[11].Value + match.Groups[12].Value);
    return (new Vector(v1, v2, v3));
}
```

which is working as expected.

I will now use this new function to create vectors for each of the user inputs:

```
//if (check == true)
if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel.Visible = false;
    ValidationLabel.Enabled = false;

    //Assign object creation variables
    Vector position = Global.regExInterp(sPosition);
    Vector velocity = Global.regExInterp(sVelocity);
    Vector dimensions = Global.regExInterp(sDimensions);
    Vector colour = Global.regExInterp(sColour);
    Vector force = Global.regExInterp(sForce);
}
```

Mass is already stored inside the double variable “mass” because of the line:

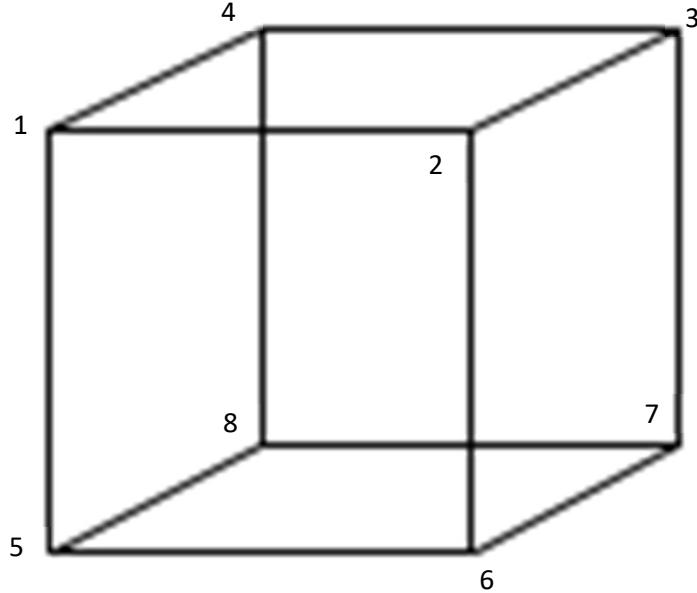
```
if (!double.TryParse(sMass, out double mass))
```

After writing this, I have realised that a negative mass can be entered. To prevent this I have added an if statement to set check=false if mass <= 0. I have used a nested if statement here because this check should (and can) only be done if we already know if the mass entered is actually a number.

```
if (double.TryParse(sMass, out double mass))
{
    if (mass <= 0) { check = false; }
    //entered mass is negative which is not allowed
}
else { check = false; } //entered mass is not a number
```

I am now at last ready to write the code which will actually create the users’ objects.

As a reminder, this diagram from long ago, shows the order in which the vertices must be created:



though here I relabelled the diagram to start from 1 instead of 0 for my convenience.

I must next create a list of these vertices. I then create a mesh from these vertices which will allow the shape to be rendered, creating this mesh requires the list of vertices which I just created, the dimensions vector which I created before, and the colour vector which I also created before.

I then create the tensor by using the following:

$$tensor = \frac{12}{mass} \begin{pmatrix} \frac{1}{height^2 + width^2} \\ \frac{1}{depth^2 + width^2} \\ \frac{1}{depth^2 + height^2} \end{pmatrix}$$

I create the rigid body using all of the values I have so far.

Finally create the object from the mesh and rigidbody, displace it to its correct location and add it to the list of entities.

Before I show you the code I just wrote for the above, I realised that I did not include the coefficient of restitution among my user inputs, so I will do this momentarily. First I will test what I have done so far:

```

//Remove validation label (in case it is there)
ValidationLabel.Visible = false;
ValidationLabel.Enabled = false;

//Assign object creation variables
Vector position = Global.regexInterp(sPosition);
Vector velocity = Global.regexInterp(sVelocity);
Vector dimensions = Global.regexInterp(sDimensions);
Vector colour = Global.regexInterp(sColour);
Vector force = Global.regexInterp(sForce);

//Create Object
//Create vertices
double depth = dimensions.x / 2;
double height = dimensions.y / 2;
double width = dimensions.z / 2;
//I have divided these by 2 because I need the distance
//from centre to face, not face to face.
Vector v1 = new Vector(-depth, -height, -width);
Vector v2 = new Vector(-depth, -height, width);
Vector v3 = new Vector(depth, -height, width);
Vector v4 = new Vector(depth, -height, -width);
Vector v5 = new Vector(-depth, height, -width);
Vector v6 = new Vector(-depth, height, width);
Vector v7 = new Vector(depth, height, width);
Vector v8 = new Vector(depth, height, -width);
Vector[] vertices = { v1, v2, v3, v4, v5, v6, v7, v8 };

//Create mesh:
Mesh mesh = new Mesh(vertices, dimensions, colour);

//Create Tensor and RigidBody
Vector v0 = new Vector(0, 0, 0); //zero vector
Vector tensor = Global.vectorTimesScalar(new Vector(1 / (height * height + width * width), 0, 0));
Vector force = Global.vectorAddVector(force, new Vector(0, 9.81 * mass, 0));
RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, v0, force, 1, tensor);

//Create object
Entity obj = new Entity(mesh, rb);
Global.displaceObject(obj, position);
Global.Entities.Add(obj);

```

To test this I will attempt to create a cube with side length 1 at the origin (I will disable the ground for this test), which will be red (RGB(255, 0, 0)) with mass 1, 0 initial velocity, and no forces acting (besides gravity).

I will also create a cuboid of side lengths 2, 2 and 3. It will be yellow (RGB(255, 255, 0)) with a mass of 10, it will have an initial position of (12, 0, 0) which is in front of the first cube. Its initial velocity

will be  $(-5, 2.5, 0)$  (towards the first cube but upwards). It will have no forces acting on it (excluding gravity).

I will create a third object, which will be green ( $\text{RGB}(0, 255, 0)$ ), a cube of side length 2, a mass of 1, a resultant force of  $(0, -12, 0)$  (this upward force will allow it to overcome gravity and slowly accelerate upwards). Its initial velocity will be  $(0, 3, 0)$  (3 down). Its initial position will be  $(-6, 3, 7)$ .

I will create these three objects simultaneously (since the simulation is paused when in the menu).

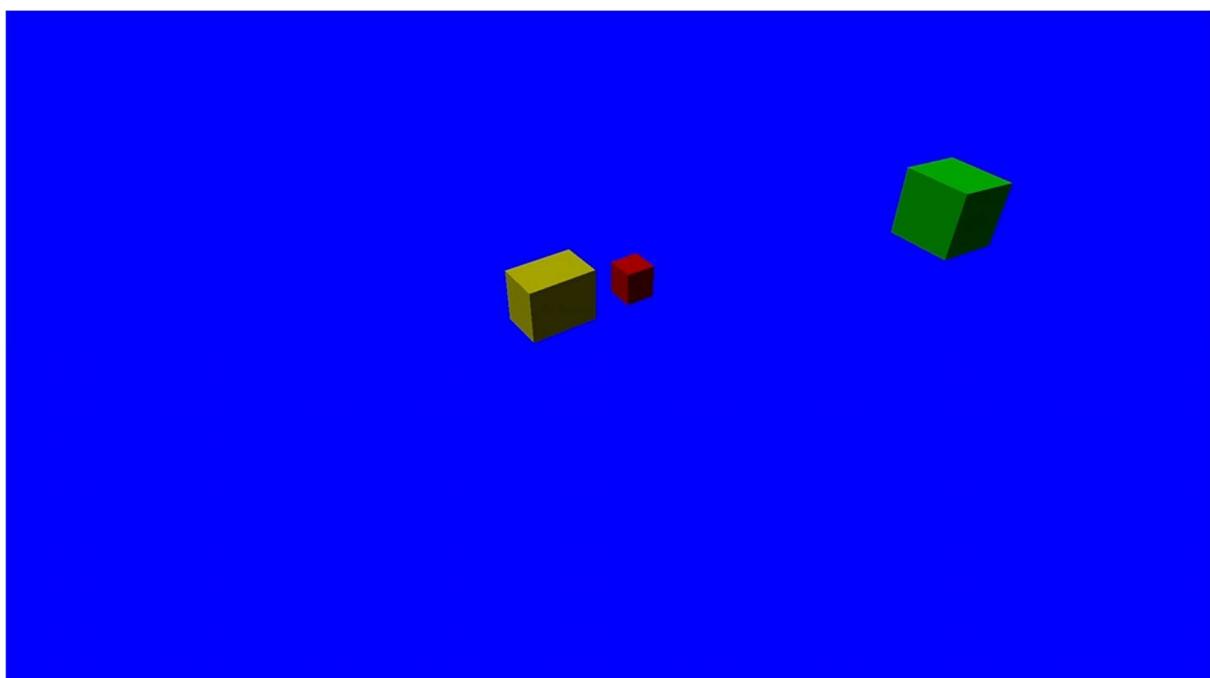
When I ran this test everything seemed to run perfectly, until the objects started to collide with the ground and bounce away from it (the ground which I supposedly disabled). I discovered that the reason for this was that when I was dealing with collisions of objects with the ground, I never checked whether the ground was enabled or not first. I added an if statement and then some separate code to update rigidBody attributes in the case where the ground is disabled.

```
//Collision detection and response with the ground
if (Global.Ground.toggled)
{
    .
    .
    .

else
{
    Global.updateAcceleration(Global.Entities[i]);
    Global.updateVelocity(Global.Entities[i]);
    Global.displaceObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity,
Global.frameTime));

    Global.updateAngularVelocity(Global.Entities[i]);
    Global.rotateObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity,
Global.frameTime));
}
```

With this new code, objects do not collide with the ground when it is not there. [Video36](#) shows the Create object button now working as intended.



This screenshot shows objects having been created, the video of course does a better job of showing their movement and also shows their creation via the menu.

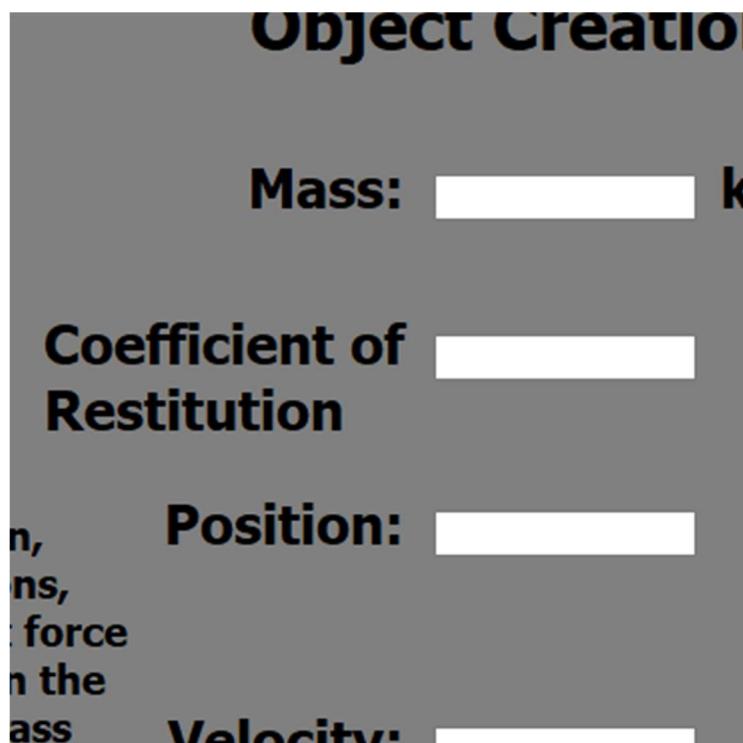
I have been playing around with new this system for a while and I have noticed that often (every third collision between distinct objects) I get crashes and errors when objects collide, always from this line of code here:

```
if ((Global.collisionFlags[k].Item2 == check[j].Item2 &&
Global.collisionFlags[k].Item3 == check[j].Item3)
    || (Global.collisionFlags[k].Item2 == check[k].Item3 &&
Global.collisionFlags[k].Item3 == check[j].Item2))
```

This is code from the collision flags section. Specifically it is an “index out of range” error. As it turns out, there was a simple typo, `check[k]` should have been `check[j]`. Now many collisions can occur error free!

These successive collisions are shown in [Video37](#).

I will now, as promised, add a field for the user to enter coefficient of restitution.



The control looks like this:

The corresponding code:

```

//Validate user inputs
bool check = true;
string pattern = @"^((\()?(-)?(\d+)(.\d+)?,( +)?(-)?(.
string sMass = massInput.Text;
string sCor = corInput.Text;
if (double.TryParse(sMass, out double mass))
{
    if (mass <= 0) { check = false; }
    //entered mass is negative which is not allowed
}
else { check = false; } //entered CoR is not a number

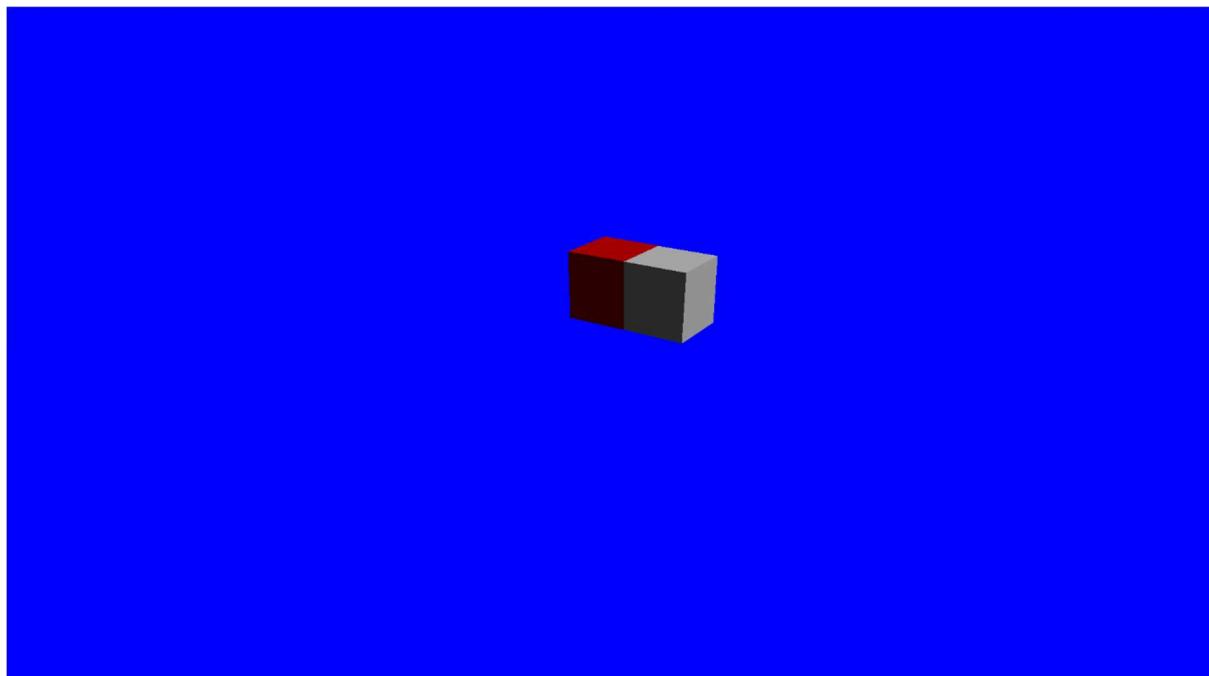
if (double.TryParse(sCor, out double cor))
{
    if (cor < 0 || cor > 1) { check = false; }
    //checks that entered CoR is between 0 and 1
}
else { check = false; }

```

The line where the RigidBody is created was changed to:

```
RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, v0, force, cor, tensor);
```

To test this, when two objects of coefficient of restitution 0 collide, they should coalesce (move together without separating). This is precisely what actually happens.



I next need to add functionality to the environment controls so that the user will be able to alter the acceleration due to gravity and the background colour.

# **Alter Environment**

**Strength of:**   
**Gravity**

**Background:**   
**Colour**

**Make  
Changes!**

I added this make changes button which I will now give code to:

```
private void MakeChanges_Click(object sender, EventArgs e)
{
    bool check = true;
    string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d-
    string sGrav = gravityInput.Text;
    string sColour = bgColourInput.Text;
    double grav;

    if (sGrav == "") //if sGrav is empty it should pass the
    {
        grav = 9.81;
    }
    else
    {
        if (!double.TryParse(sGrav, out grav))
        {
            check = false;
        }
    }

    //make sure colour is of the form (x,y,z) or similar
    if (!Regex.IsMatch(sColour, pattern))
    { check = false; }

    //if (check == true)
    if (check == true)
    {
        //Remove validation label (in case it is there)
        ValidationLabel2.Visible = false;
        ValidationLabel2.Enabled = false;

        Vector colour = Global.regExInterp(sColour);

        //apply background colour
        Global.bgColour = Color.FromArgb(Convert.ToInt16(co
           

        //apply gravity
        Global.gravity = grav;
    }
    else
    {
        //Display validation label
        ValidationLabel2.Visible = true;
        ValidationLabel2.Enabled = true;
    }
}
```

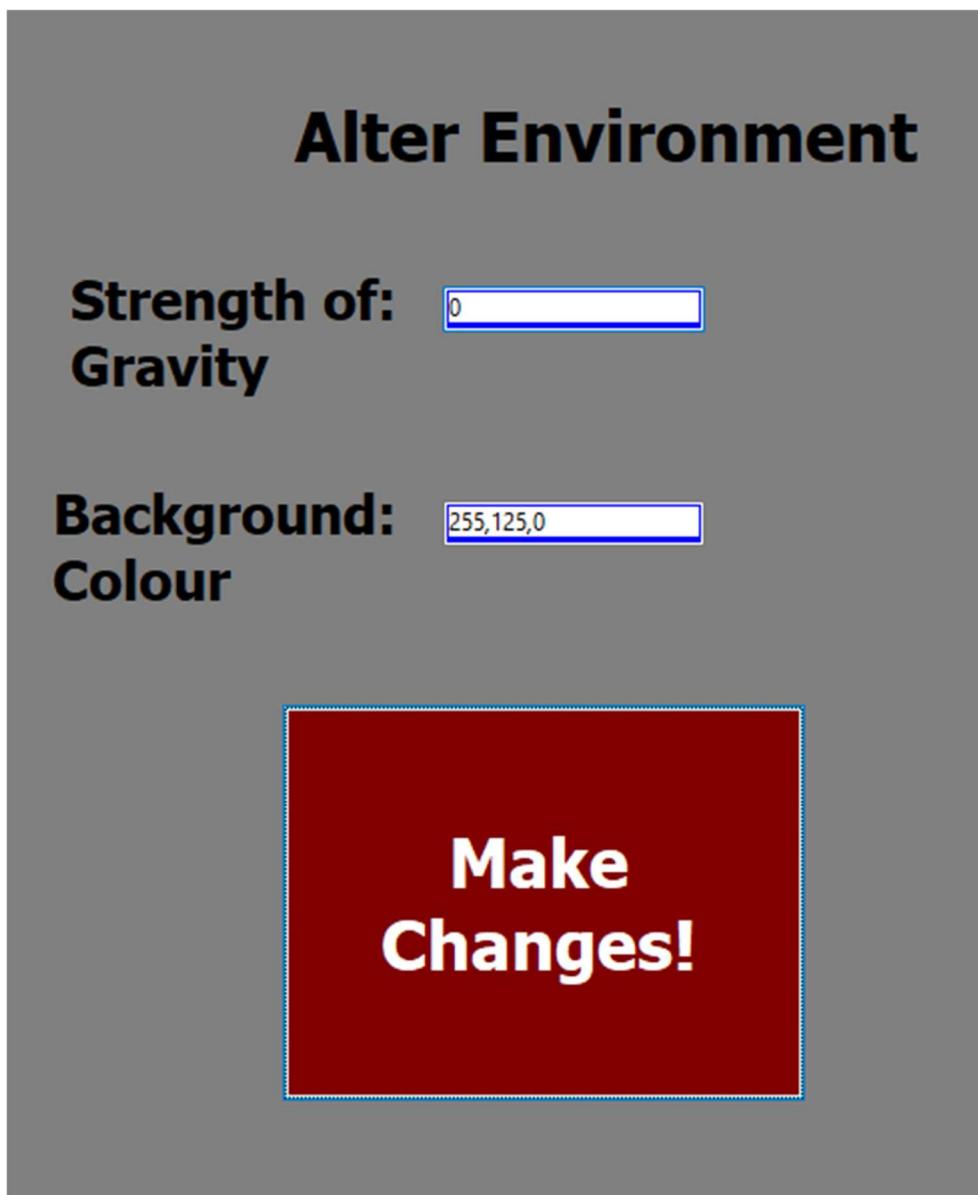
This code uses the same pattern as before to check that the colour is entered in an appropriate form. If the gravity field is left blank, it defaults to 9.81. I created a new global variable called “gravity” to which I here assign the user’s input value to. This is then used inside the updateVelocity procedure to take it into account before applying change in velocity.

I also made a new Global variable for background colour. The line  
“Global.bgColour = Color.FromArgb(Convert.ToInt16(colour.x),  
Convert.ToInt16(colour.y), Convert.ToInt16(colour.z));”

Sets this variable to the user’s inputted variable.

I then replaced the line which sets the background colour as “color.Blue” to instead set it as “Global.bgColor”.

I created a second validation label called ValidationLabel2 which is identical to the other aside from its position and its name.





Before I continue, I wish to add some more input validation, e.g., to ensure that each value for the “colour” vector is between 0 and 255 (inclusive) as these are the only acceptable RGB values. Also to make sure that dimensions are always strictly  $> 0$  (if negative values are entered it makes no difference but the program crashes when 0 is entered). This was relatively straightforward:

```
string sDimensions = dimensionsInput.Text;
if (!Regex.IsMatch(sDimensions, pattern))
{ check = false; }
else
{
    dimensions = Global.regExInterp(sDimensions);
    if (!(dimensions.x > 0 && dimensions.y > 0 && dimensions.z > 0))
        { check = false; }
}
string sColour = colourInput.Text;
if (!Regex.IsMatch(sColour, pattern))
{ check = false; }
else
{
    colour = Global.regExInterp(sColour);
    if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y && colour.y <= 255 &&
0 <= colour.z && colour.z <= 255))
        { check = false; }
}
```

The validation box now shows up for these kinds of invalid inputs.

Invalid Colour input (126 is too big).

**Dimensions:**

**Colour:**

**Resultant Force:**   
(including Gravity)

**Create Object!**

**Ensure all fields have correctly formatted inputs!**

Invalid colour input (negative numbers aren't allowed):

**Dimensions:**

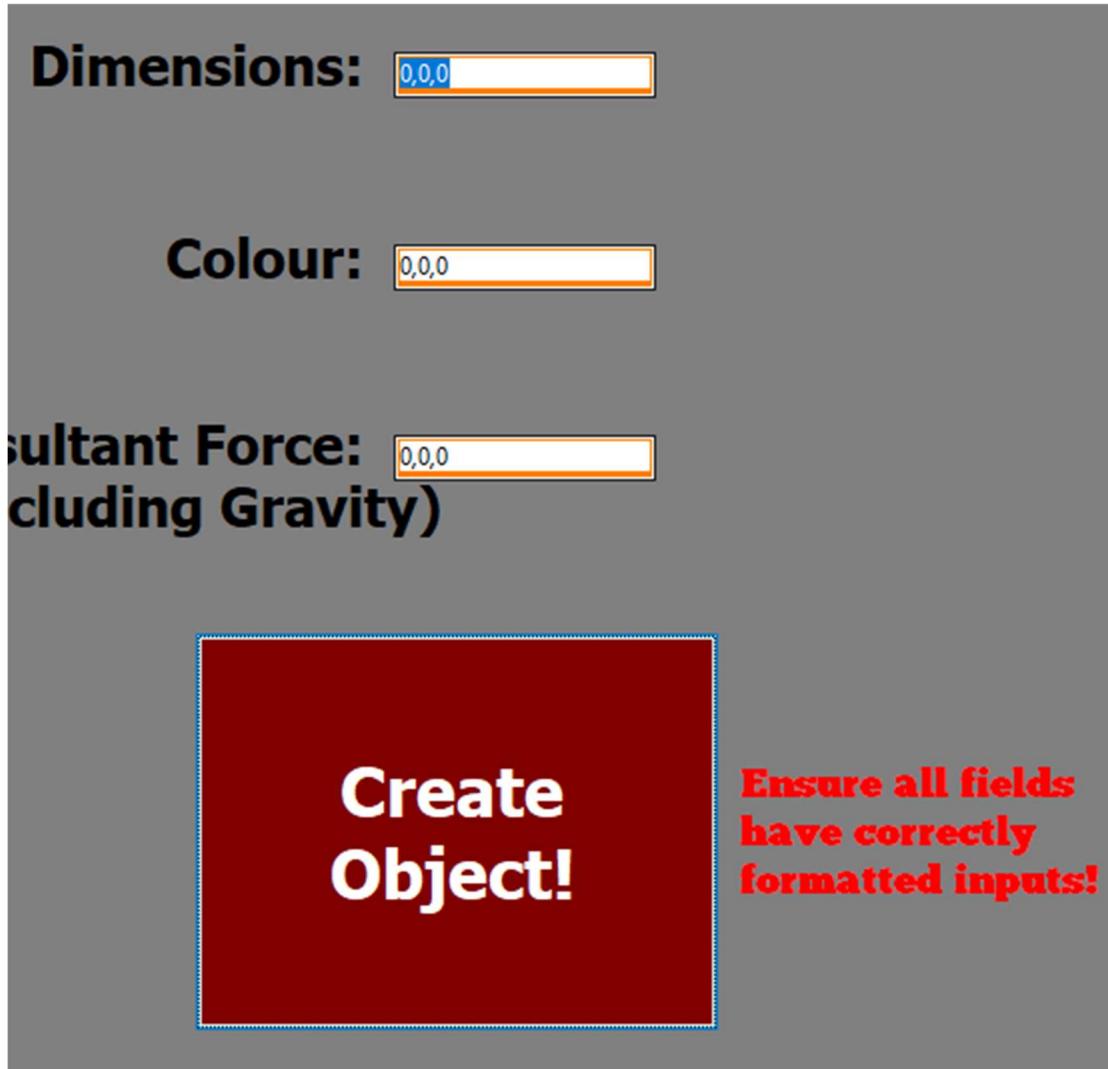
**Colour:**

**Resultant Force:**   
**(including Gravity)**

**Create  
Object!**

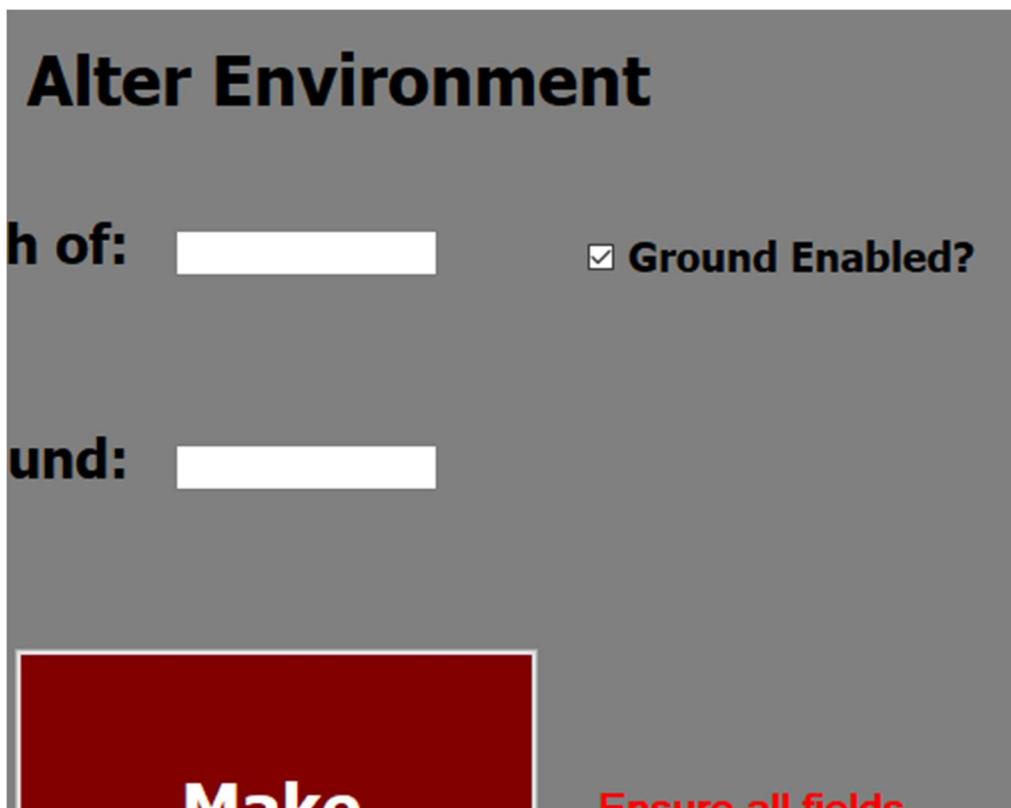
**Ensure all fields  
have correctly  
formatted inputs!**

Invalid dimensions input (0 is not allowed):



Back to the alter environment column for a moment, I now whish to add a tick box to enable/disable the ground, I don't want the use to need to click the button for this one to take effect, so I added the following code to the event which occurs whenever the checkbox is toggled instead of, instead of adding it to the Make Changes button click event.

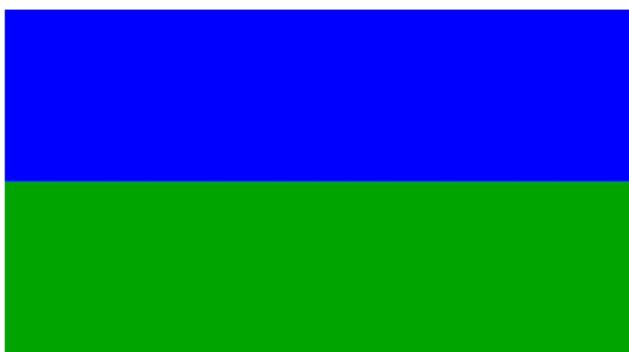
The checkbox (enabled by default, I also set the ground to be enabled when the form loads):



Its code:

```
private void GroundEnabled_CheckedChanged(object sender, EventArgs e)
{
    if (GroundEnabled.Checked)
    {
        Global.Ground.toggled = true;
    }
    else
    {
        Global.Ground.toggled = false;
    }
}
```

When ticked, the ground is enabled, when unticked, it is not:



After playing around with the plane enabled, I noticed that a lot of the time, objects would bounce away to a higher position than where they started, this should not be possible. After reviewing my code, I believe that the reason for this occurring is that I have not implemented collision flags for collisions with the ground as I did for collisions with other objects.

I first gave the `rigidBody` class a new attribute, “`groundFlag`”, which would act as a collision flag between the object and the ground. This attribute is, of course, a bool. I then check these collision flags before applying any collision response with the ground, updating the flags where necessary.

```
List<Vector> points = Global.groundCollisionDetection(Global.Entities[i].mesh);
bool groundCollided = false;

if (Global.Entities[i].rigidBody.groundFlag == false)
{
    if (points.Count != 0)
    {
        //The flag is false and there was an intersection:
        //Do collision response and set flag to true.
        Vector position = Global.averagePoint(points.ToArray());
        Global.groundCollisionResponse(Global.Entities[i], position);
        groundCollided = true;
        Global.Entities[i].rigidBody.groundFlag = true;
    }
    else
    {
        //The flag was false and there was no intersection:
        //Do nothing
    }
}
else
{
    if (points.Count != 0)
    {
        //The flag is true and there was an intersection:
        groundCollided = true;
    }
    else
    {
        //The flag was true but there was no intersection:
        //Set flag to false:
        Global.Entities[i].rigidBody.groundFlag = false;
    }
}
```

Moving on to the third column of the menu screen:

I first realised that in order to select an object, each object would need to have some way of identifying it. I decided to give each object a name. I added “name” to the Entity class, as well as a field which takes this as an optional input. If no name is entered it will simply be named “Cube0”, “Cube1”, “Cube2” etc., based on how many entities that currently exist.



```
//Create name:  
string sName = nameInput.Text;  
string name = "";  
if (sName == "")  
{  
    name = "Cube" + Global.Entities.Count;  
}  
else  
{  
    name = sName;  
}
```

(The above code is just before object creation in the “Create\_Click” event).

Inside the Entity class I created an override method which sets the name of the Entity as the one entered by the user, this means that name will be used in the drop down list (technically a DomainUpDown).

```
namespace ComputerScienceAlevelProject
{
    99+ references
    internal class Entity
    {
        public Mesh mesh;
        public RigidBody rigidBody;
        public string name;
        5 references
        public Entity(Mesh mesh, RigidBody rigidBody, string name)
        {
            this.mesh = mesh;
            this.rigidBody = rigidBody;
            this.name = name;
        }
        0 references
        public override string ToString()
        {
            return this.name;
        }
    }
}
```

Now when an object is created with a specified name, that name appears in the list, if there is no specified name, it is simply named as "Cube0", "Cube1", "Cube2" etc.,



# Alter Objects

Epic Name

**Alter!**

I will next add code for the “Alter” button which will allow object properties to be changed. Before I could do this though, I had to add fields for all of the data which the user can enter. These are shown on the next page and each one is optional.

# Alter Objects

Enabled?

Select Object



**Mass:**

**Coefficient of Restitution**

**Position:**

**Velocity:**

**Angular Velocity:**

**Colour:**

**Resultant Force:  
(Excluding Gravity)**

**Alter!**

This is the code for the “Alter” button:

```
private void Alter_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem == null)
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
    else
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;

        //Validate user inputs
        bool check = true;
        string pattern = @"^(\()?( -)?(\d+)( \d+)?,(\ +)?(-)?(\d+)( \d+)?,(\ +)?(-)?(\d+)( \d+)?(\))?$";
        string sMass = massInput2.Text;
        string sCor = corInput2.Text;
        Vector colour = new Vector(0, 0, 0);

        double mass;
        if (double.TryParse(sMass, out mass))
        {
            if (mass <= 0) { check = false; }
            //entered mass is negative which is not allowed
        }
        else if (sMass != "")
        { check = false; } //entered mass is not a number and is non-empty

        double cor;
        if (double.TryParse(sCor, out cor))
        {
            if (cor < 0 || cor > 1) { check = false; }
            //checks that entered CoR is between 0 and 1
        }
        else if (sCor != "")
        { check = false; }

        //make sure the rest are of the form (x,y,z) or similar
        string sPosition = positionInput2.Text;
        if (!Regex.IsMatch(sPosition, pattern) && sPosition != "")
        { check = false; }

        string sVelocity = velocityInput2.Text;
        if (!Regex.IsMatch(sVelocity, pattern) && sVelocity != "")
        { check = false; }

        string sAngularVelocity = angularVelocityInput2.Text;
        if (!Regex.IsMatch(sAngularVelocity, pattern) && sAngularVelocity != "")
        { check = false; }

        string sColour = colourInput2.Text;
        if (!Regex.IsMatch(sColour, pattern) && sColour != "")
        { check = false; }
        else
        {
            if (sColour != "")
            {
                colour = Global.regExInterp(sColour);
            }
            if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y && colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
            { check = false; }
        }
    }
}
```

```
string sForce = forceInput2.Text;
if (!Regex.IsMatch(sForce, pattern) && sForce != "")
{ check = false; }

//if (check == true)
if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel3.Visible = false;
    ValidationLabel3.Enabled = false;

    //Get object from SelectedItems (the DomainUpDown Control)

    Entity obj = SelectObject.SelectedItem as Entity;

    if (sMass != "")
    {
        obj.rigidBody.mass = mass;
    }

    if (sCor != "")
    {
        obj.rigidBody.CoR = cor;
    }

    if (sColour != "")
    {
        obj.mesh.colour = colour;
    }

    if (sPosition != "")
    {
        Vector position = Global.regExInterp(sPosition);
        //To set position, I must displace it to the origin
        //and then displace it to the desired location
        Global.displaceObject(obj, Global.vectorTimesScalar(obj.rigidBody.centreOfMass, -1));
        Global.displaceObject(obj, position);
    }

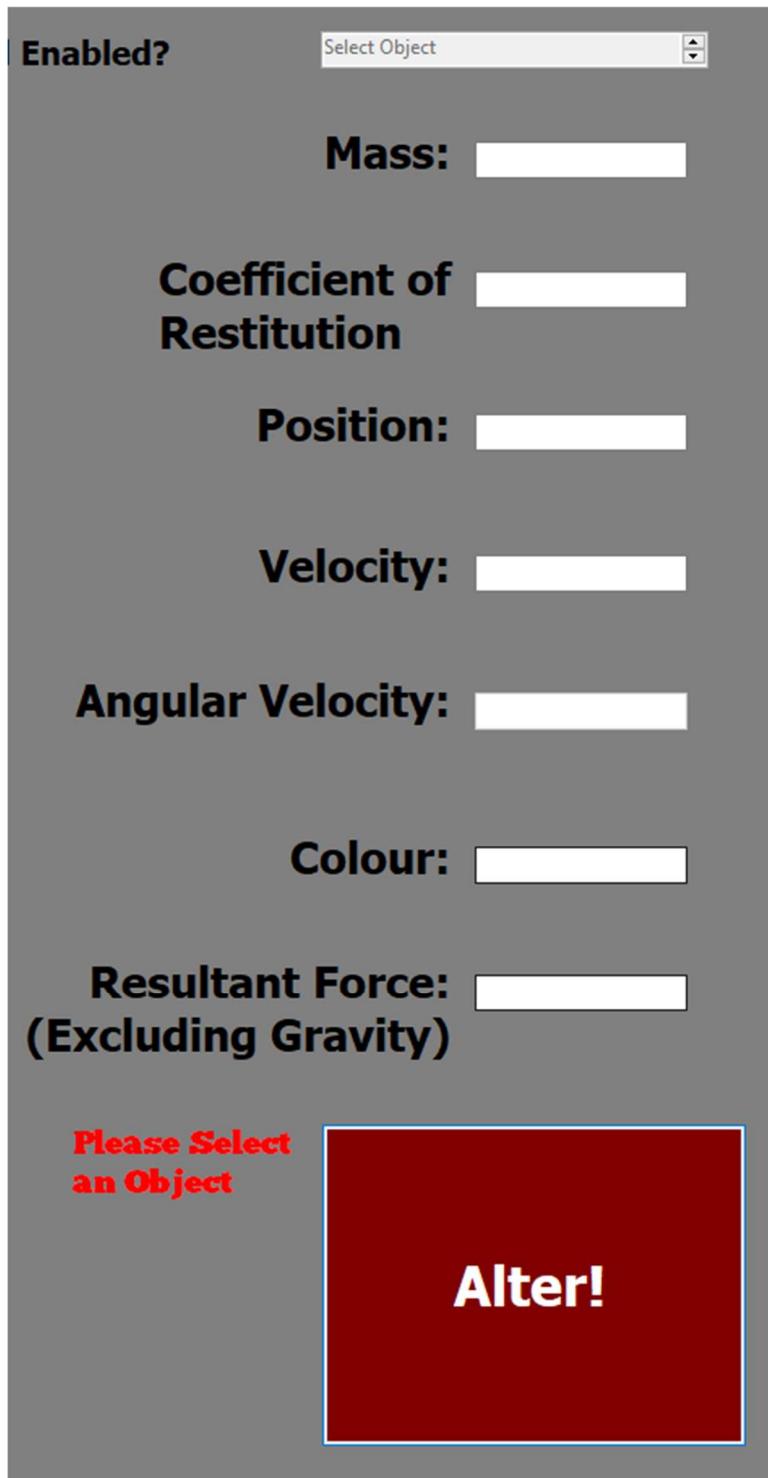
    if (sVelocity != "")
    {
        Vector velocity = Global.regExInterp(sVelocity);
        obj.rigidBody.velocity = velocity;
    }

    if (sForce != "")
    {
        Vector force = Global.regExInterp(sForce);
        obj.rigidBody.force = force;
    }

    if (sAngularVelocity != "")
    {
        Vector angularVelocity = Global.regExInterp(sAngularVelocity);
        obj.rigidBody.angularVelocity = angularVelocity;
    }
}

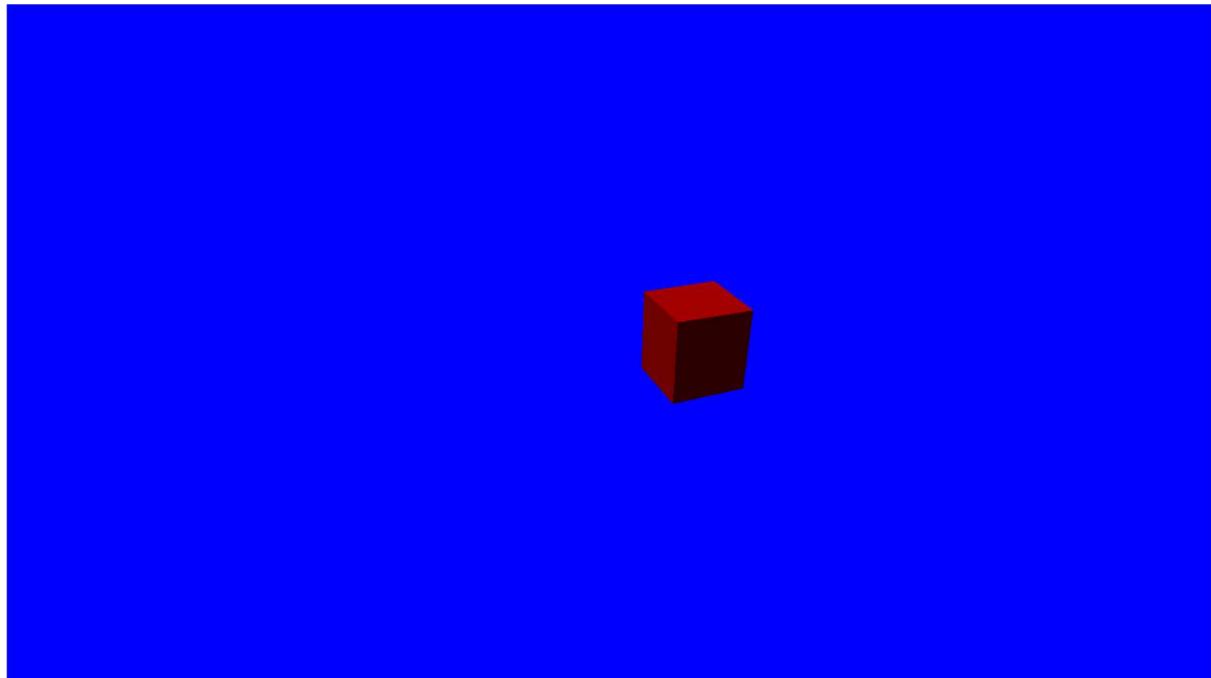
else
{
    //Display validation label
    ValidationLabel3.Visible = true;
    ValidationLabel3.Enabled = true;
}
```

Much of this code is copied from the create button but there are some key changes. Firstly, I removed dimensions, and added angular velocity. Because I wanted the button to change nothing for fields which are left blank, I added if statements at the end to only apply changes when the field in question does not contain an empty string. When setting the position of an object, I moved it to the origin by displacing it by the negative of its own position, I then displace it by the specified amount to get it to the correct location. I added an if statement at the beginning to check whether the DomainUpDown had a value selected or was null. This is because if it was null then the program would throw an error because I would be trying to change properties of an unspecified object. I added a fourth validation label for this purpose, it only appears if no object is selected and it tells the user to “please select an object”.

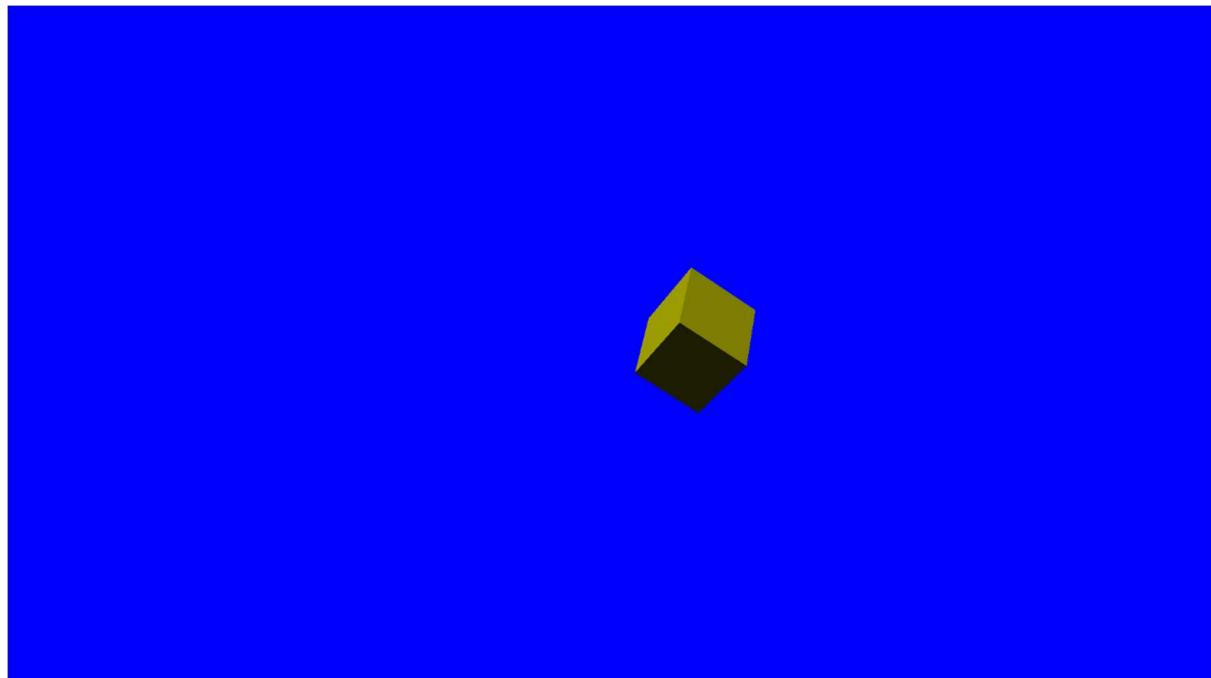


To demonstrate this working, I have created a red cube, and then set its colour to yellow (RGB (255, 255, 0))and angular velocity to (1, 1, 1) so it begins rotating:

Before:



After:



Inputted values to cause this change:

# Alter Objects

**Enabled?**

**Mass:**

**Coefficient of Restitution**

**Position:**

**Velocity:**

**Angular Velocity:**

**Colour:**

**Resultant Force:  
(Excluding Gravity)**

**Alter!**

The last feature I will add to this menu is a button to remove the selected object. This should be quite easy to implement.

The button looks like this:

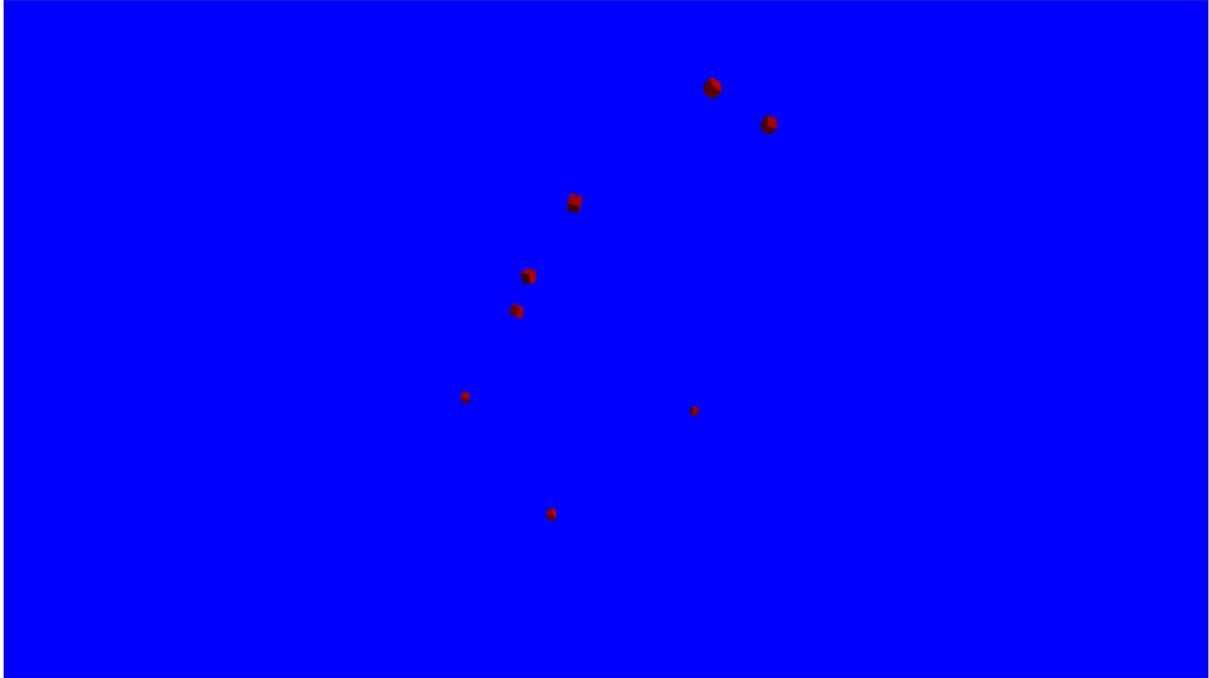


I will now write some code which will remove the selected object from the list of entities (Global.Entities).

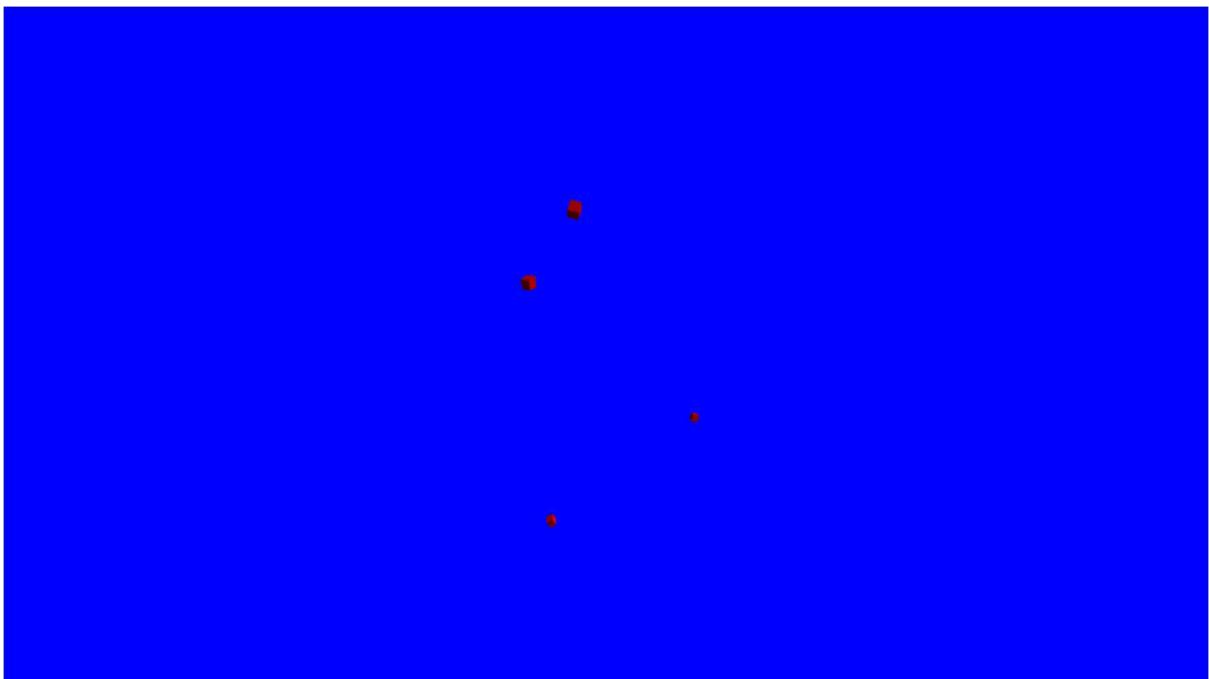
```
private void Delete_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem != null)
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;
        Entity obj = SelectObject.SelectedItem as Entity;
        Global.Entities.Remove(obj);
        SelectObject.Items.Remove(obj);
    }
    else
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
}
```

The above code first checks that there is actually a selected object, if not the label to tell the user to select one is displayed, otherwise it is hidden. If an object is selected, it is removed from the list of entities and is then removed from the DomainUpDown Control also. Objects can now be removed by using the button.

Before:



After:



I made multiple objects in one place so that they would move far apart in a seemingly random way. I then set each of their velocities and angular velocities to zero so that they wouldn't move between screenshots. I then deleted 4 of these objects, leaving four remaining as shown above.

I think it would be more convenient to have each of the fields reset to being blank each time the menu is opened. The code for this is simple, I just clear each field when the escape key is pressed.

```
if (e.KeyCode == Keys.Escape)
{
    Global.paused = !Global.paused;
    massInput2.Clear();
    corInput2.Clear();
    positionInput2.Clear();
    velocityInput2.Clear();
    angularVelocityInput2.Clear();
    colourInput2.Clear();
    forceInput2.Clear();
}
```

This works as expected:

Before Reopening Menu:

Alter Objects	
<b>Enabled?</b>	<input type="button" value="Select Object"/>
<b>Mass:</b>	<input type="text" value="21"/>
<b>Coefficient of Restitution</b>	<input type="text" value="124"/>
<b>Position:</b>	<input type="text" value="12"/>
<b>Velocity:</b>	<input type="text" value="12"/>
<b>Angular Velocity:</b>	<input type="text" value="325"/>
<b>Colour:</b>	<input type="text" value="12,43,234"/>
<b>Resultant Force: (Excluding Gravity)</b>	<input type="text" value="0,0,0"/>
<input type="button" value="Alter!"/>	

After Reopening Menu:

The menu is now completed. This means it is time for me to move on to my final task to meet each of the success criteria. I will copy this menu to the other two forms and alter them appropriately to meet the previously outlined requirements for these other simulations. (I also need to add Newton's law for Gravitation to the "Orbit" simulation).

## Refining/Differentiating Each Simulation

A while ago I made three copies of the simulation form. One for the rigid body collisions simulation (the one I've been working on this whole time). I now wish to copy everything I have done since then (i.e., the menu) into these other forms. After doing this all three forms are now identical. I want to make some changes to the second two. First, I remove the checkbox for the ground and change the code in the load event to make the ground enabled by default (since the user can no longer disable it this means the ground will always be present in the projectile motion simulation).

Menu with no checkbox:

Main Menu

Object Creation

Alter Environment

Alter Objects

Strength of: Gravity

Background: Colour

Make Changes!

Delete Selected Object!

Alter!

Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number

Mass: kg

Coefficient of Restitution

Position:

Velocity:

Dimensions:

Colour:

Resultant Force: (Excluding Gravity)

Name: (optional)

Create Object!

Select Object

Mass:

Coefficient of Restitution

Position:

Velocity:

Angular Velocity:

Colour:

Resultant Force: (Excluding Gravity)

Global gravity = 9.81;

Code to enable ground when the form is loaded:

```
private void ProjectileMotion_Load(object sender, EventArgs e)
{
    Global.Ground.toggled = true;
    Global.paused = false;
    Global.Entities = new List<Entity>();
    Global.collisionFlags = new List<Tuple<bool, Entity, Entity>>();
    Global.Camera = new Camera(new Vector(-1, 0, 0), new Vector(1, 0.01, 0.01), 7 * Math.PI / 18);
    Global.gravity = 9.81;
}
```

I previously stated that this simulation would contain no collisions. This is not completely true as I still want objects to have collisions with the ground but not with each other. This means that mass and coefficient of restitution are still important.

Size of objects doesn't matter because this will not affect how they move, so each object will just be a unit cube.

I decided not to include starting coordinates (except for height) because allowing users to create object with certain starting co-ordinates would not be very useful since the objects cannot interact with each other in any way (but height can be controlled because they can interact with the ground).

Colour will still be a field, as well as resultant force and force due to gravity and background colour. I will replace velocity with two fields, an initial speed, and the initial angle at which the object was launched (in degrees). This is because projectile motion is usually discussed using initial speed and angle instead of velocity as a vector.

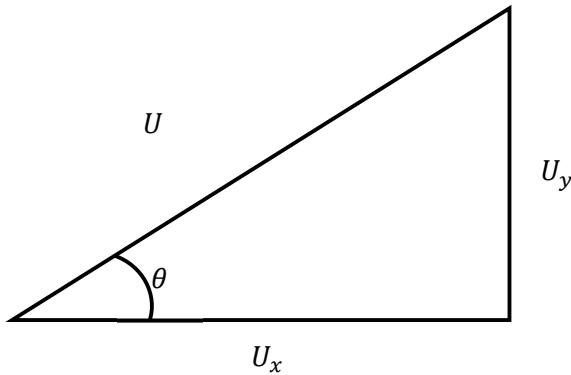
The third column will be the third column will only have the same variables previously outlined as well as resultant force and the deletion button.

This is what the menu for the projectile motion simulation now looks like:

Main Menu	Alter Environment	Alter Objects
Mass: <input type="text"/> kg Coefficient of Restitution: <input type="text"/> <small>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number</small> Starting: Height: <input type="text"/> Initial: Speed: <input type="text"/> Initial: Angle: <input type="text"/> Colour: <input type="text"/>  Resultant Force: <input type="text"/> (Excluding Gravity)  Name: <input type="text"/> (optional)	Strength of: Gravity: <input type="text"/> Background: Colour: <input type="text"/> <div style="background-color: red; color: white; padding: 10px; text-align: center;"> <b>Make Changes!</b> </div>	Select Object: <input type="text"/> Mass: <input type="text"/> Coefficient of Restitution: <input type="text"/> Colour: <input type="text"/> Resultant Force: (Excluding Gravity): <input type="text"/> <div style="background-color: red; color: white; padding: 10px; text-align: center;"> <b>Alter!</b> </div> <div style="background-color: green; color: white; padding: 5px; margin-top: 10px;"> <b>Delete Selected Object!</b> </div>
		

Most of the code can just be copied from the previous section. A key change which must be made is in converting the initial speed and initial angle into a vector, angle being in degrees.

Using trigonometry:



Where  $U$  is the initial speed and  $\theta$  is the initial angle, the horizontal and vertical components  $U_x$  and  $U_y$  respectively are given by:  $U_x = U \cos \theta$  &  $U_y = U \sin \theta$ . The angle must first be converted from degrees to radians by writing `angle = angle * Math.PI / 180;`

```
private void Create_Click(object sender, EventArgs e)
{
    //Validate user inputs
    bool check = true;
    string pattern = @"^((\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?(\))?$";
    string sMass = massInput.Text;
    string sCor = corInput.Text;
    string sHeight = heightInput.Text;
    string sSpeed = speedInput.Text;
    string sAngle = angleInput.Text;
    Vector colour = new Vector(0, 0, 0);

    //Scalars
    if (double.TryParse(sMass, out double mass))
    {
        if (mass <= 0) { check = false; }
        //entered mass is negative which is not allowed
    }
    else { check = false; } //entered mass is not a number

    if (double.TryParse(sCor, out double cor))
    {
        if (cor < 0 || cor > 1) { check = false; }
        //checks that entered CoR is between 0 and 1
    }
    else { check = false; }

    if (double.TryParse(sHeight, out double height))
    {
        if (height <= 0) { check = false; }
        //entered height is negative which is not allowed
    }
    else { check = false; } //entered height is not a number

    if (double.TryParse(sSpeed, out double speed))
    {
        if (speed < 0) { check = false; }
    }
    else { check = false; }

    if (double.TryParse(sAngle, out double angle))
    {
        if (!( -90 <= angle && angle <= 90)) { check = false; }
    }
    else { check = false; }

    //Vectors
    string sColour = colourInput.Text;
    if (!Regex.IsMatch(sColour, pattern))
    { check = false; }
    else
    {
        colour = Global.regExInterp(sColour);
        if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y && colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
        { check = false; }
    }
    string sForce = forceInput.Text;
    if (!Regex.IsMatch(sForce, pattern))
    { check = false; }
}
```

```

if (check == true)
{
    ValidationLabel.Visible = false;
    ValidationLabel.Enabled = false;

    //Assign object creation variables
    Vector position = new Vector(0, 10 - height, 0);
    angle = angle * Math.PI / 180;
    double Ux = speed * Math.Cos(angle);
    double Uy = -speed * Math.Sin(angle);
    Vector velocity = new Vector(0, Uy, Ux);
    Vector force = Global.regExInterp(sForce);

    //Create Object
    Vector v1 = new Vector(-0.5, -0.5, -0.5);
    Vector v2 = new Vector(-0.5, -0.5, 0.5);
    Vector v3 = new Vector(0.5, -0.5, 0.5);
    Vector v4 = new Vector(0.5, -0.5, -0.5);
    Vector v5 = new Vector(-0.5, 0.5, -0.5);
    Vector v6 = new Vector(-0.5, 0.5, 0.5);
    Vector v7 = new Vector(0.5, 0.5, 0.5);
    Vector v8 = new Vector(0.5, 0.5, -0.5);
    Vector[] vertices = { v1, v2, v3, v4, v5, v6, v7, v8 };

    //Create mesh:
    Mesh mesh = new Mesh(vertices, new Vector(1,1,1), colour);

    //Create Tensor and RigidBody
    Vector v0 = new Vector(0, 0, 0); //zero vector
    Vector tensor = new Vector(6/mass, 6/mass, 6/mass);
    RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, force, cor, tensor, false);

    //Create name:
    string sName = nameInput.Text;
    string name = "";
    if (sName == "")
    {
        name = "Projectile" + Global.Entities.Count;
    }
    else
    {
        name = sName;
    }

    //Create object
    Entity obj = new Entity(mesh, rb, name);
    Global.displaceObject(obj, position);
    Global.Entities.Add(obj);
    SelectObject.Items.Add(obj);
}

else
{
    //Display validation label
    ValidationLabel.Visible = true;
    ValidationLabel.Enabled = true;
}
}

```

The code for all of this is shown above, most of it is very similar to what we had before, except speed, angle and height are all scalars (instead of the velocity and position vectors before). These vectors are constructed from the other entered information.

This is shown to be working in [Video38](#) and here:

## Object Creation

Mass:  kg

Coefficient of Restitution:

d Starting:   
l be Height

x, y, a Initial:   
Initial Speed

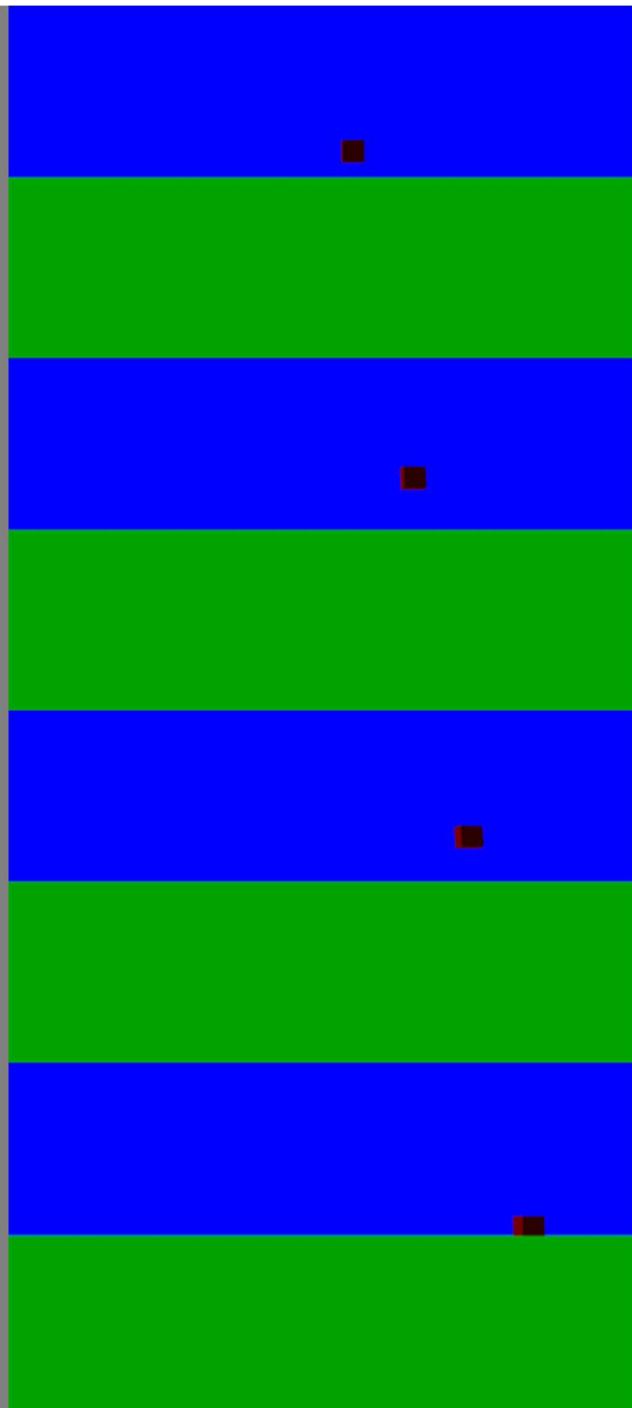
Initial:   
Angle

Colour:

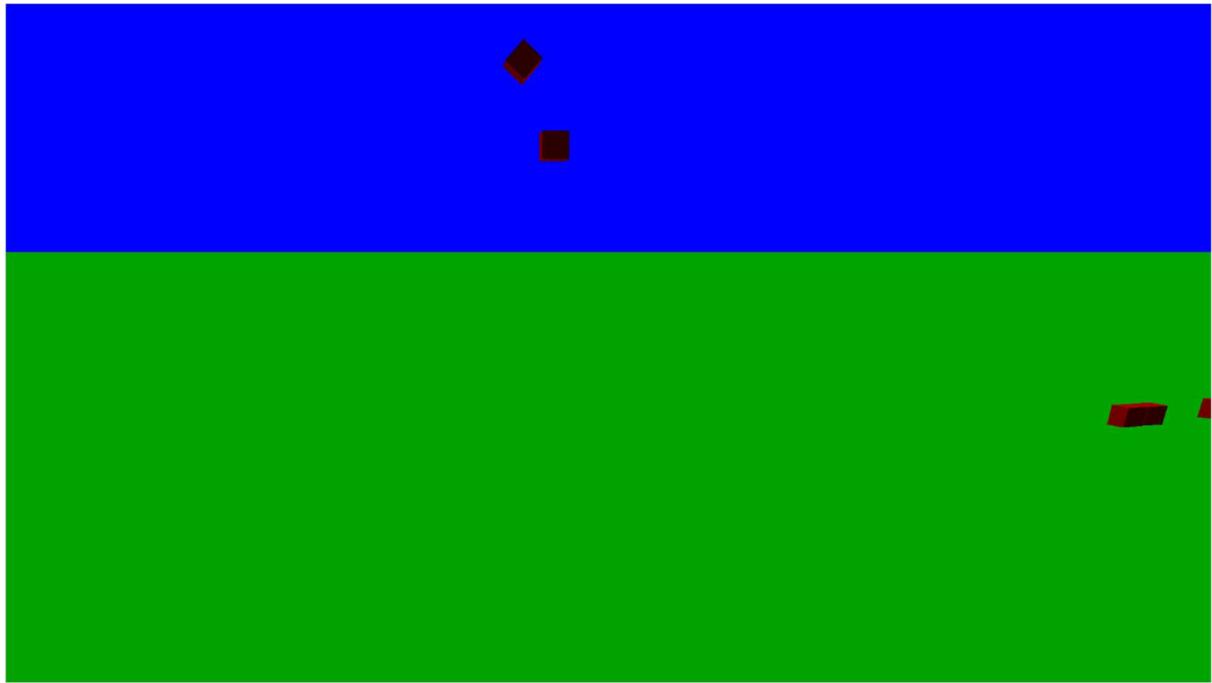
sultant Force:   
(including Gravity)

Name:   
(optional)

Create  
Object!



If too many objects are created near each other, they collide (which they shouldn't). Also when there are many objects (even if not close together) there is still a drop in performance due to the collision detection.



To fix this, I will simply remove the collision code from this simulation (leaving in the code for collision with the plane).

```
//Physics
void physics()
{
    //Collision Detection
    List<Tuple<Vector, Entity, Entity>> check = Global.checkAllCollisions();
    for (int i = 0; i < Global.Entities.Count; i++)
    {
        //Add weight to resultant force.
        Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0, Global.Entities[i].rigidBody.mass * Global.gravity, 0));

        //Don't need to check if the ground is enabled because it always is.

        //Collision detection and response with the ground
        List<Vector> points = Global.groundCollisionDetection(Global.Entities[i].mesh);
        bool groundCollided = false;

        if (Global.Entities[i].rigidBody.groundFlag == false)
        {
            if (points.Count != 0)
            {
                //The flag is false and there was an intersection:
                //Do collision response and set flag to true.
                Vector position = Global.averagePoint(points.ToArray());
                Global.groundCollisionResponse(Global.Entities[i], position);
                groundCollided = true;
                Global.Entities[i].rigidBody.groundFlag = true;
            }
        }
        else
        {
            //The flag was false and there was no intersection:
            //Do nothing
        }
    }
    else
    {
        if (points.Count != 0)
        {
            //The flag is true and there was an intersection:
            groundCollided = true;
        }
    }
}
```

:

```

else
{
    //The flag was true but there was no intersection:
    //Set flag to false:
    Global.Entities[i].rigidBody.groundFlag = false;
}

//Find with which object and normal (if any) this object collided with each frame and apply normal reaction.
//This is done regardless of collision flags.
Vector reaction = new Vector(0, 0, 0);
if (groundCollided)
{
    reaction = Global.vectorAddVector(reaction, new Vector(0, -Global.Entities[i].rigidBody.force.y, 0));
}

Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, reaction);
Global.updateAcceleration(Global.Entities[i]);
Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, Global.vectorTimesScalar(reaction, -1));

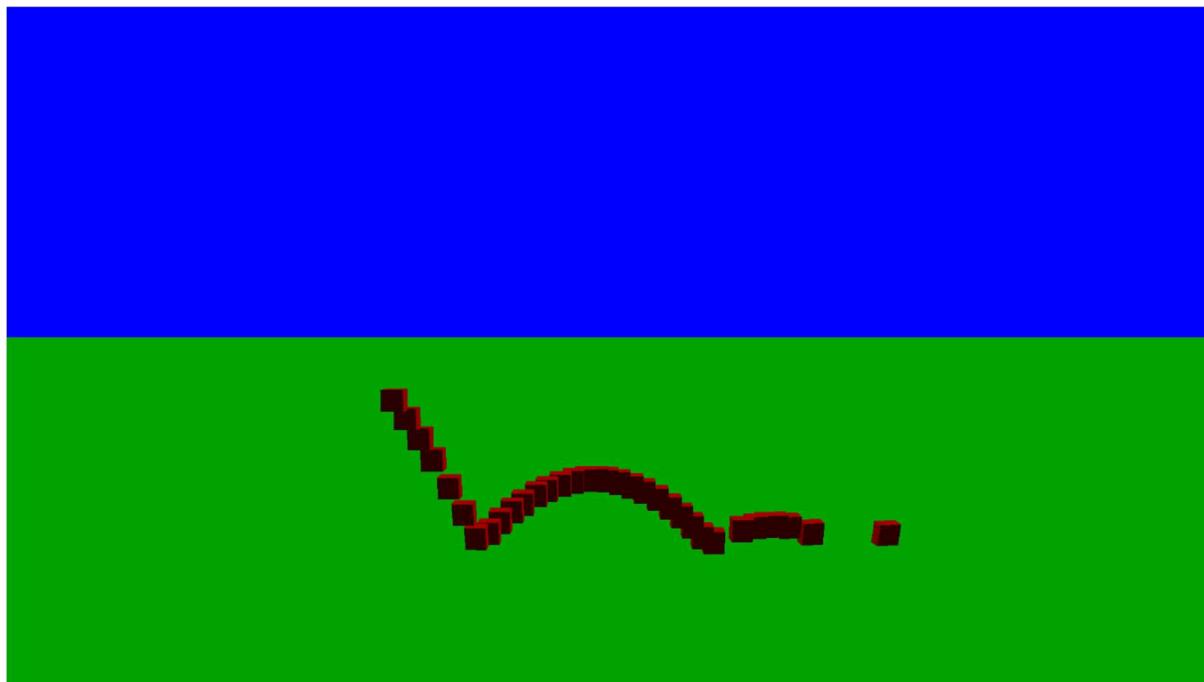
Global.updateVelocity(Global.Entities[i]);
Global.displaceObject(Global.Entities[i], Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity, Global.frameTime));

Global.updateAngularVelocity(Global.Entities[i]);
Global.rotateObject(Global.Entities[i], Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity, Global.frameTime));

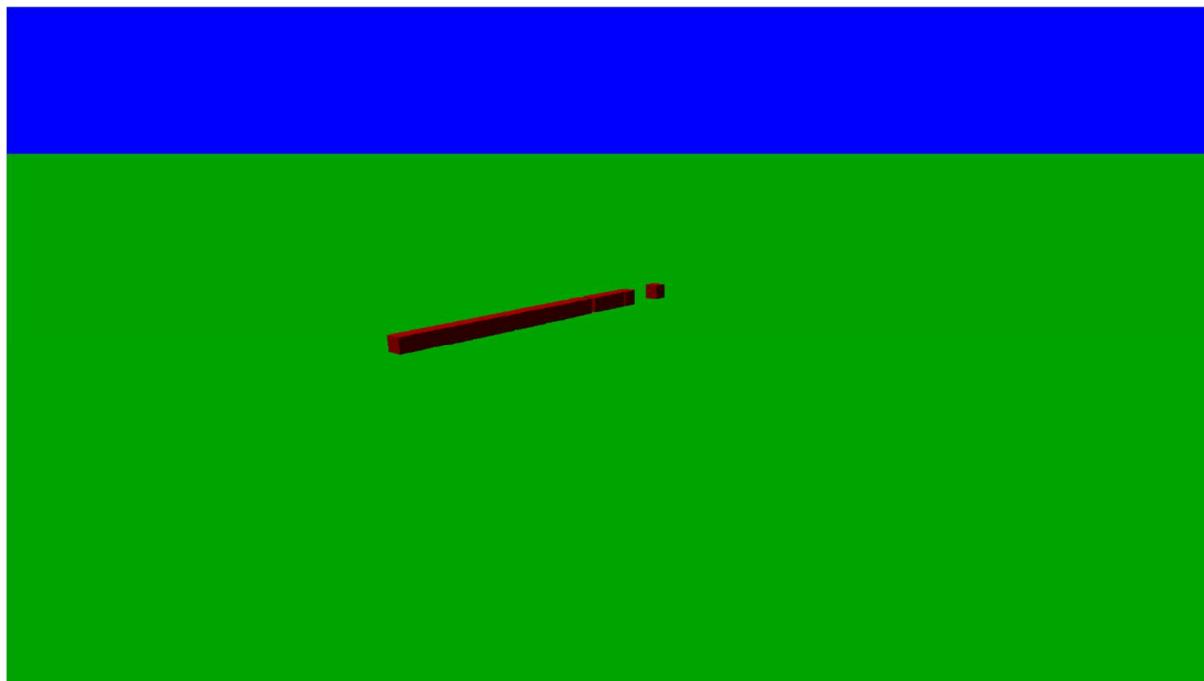
//Take away gravity
Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0, -Global.Entities[i].rigidBody.mass * Global.gravity, 0));
}

```

Objects no longer collide with each other, but they do collide with the ground.



Normal reaction with the ground is also still working:

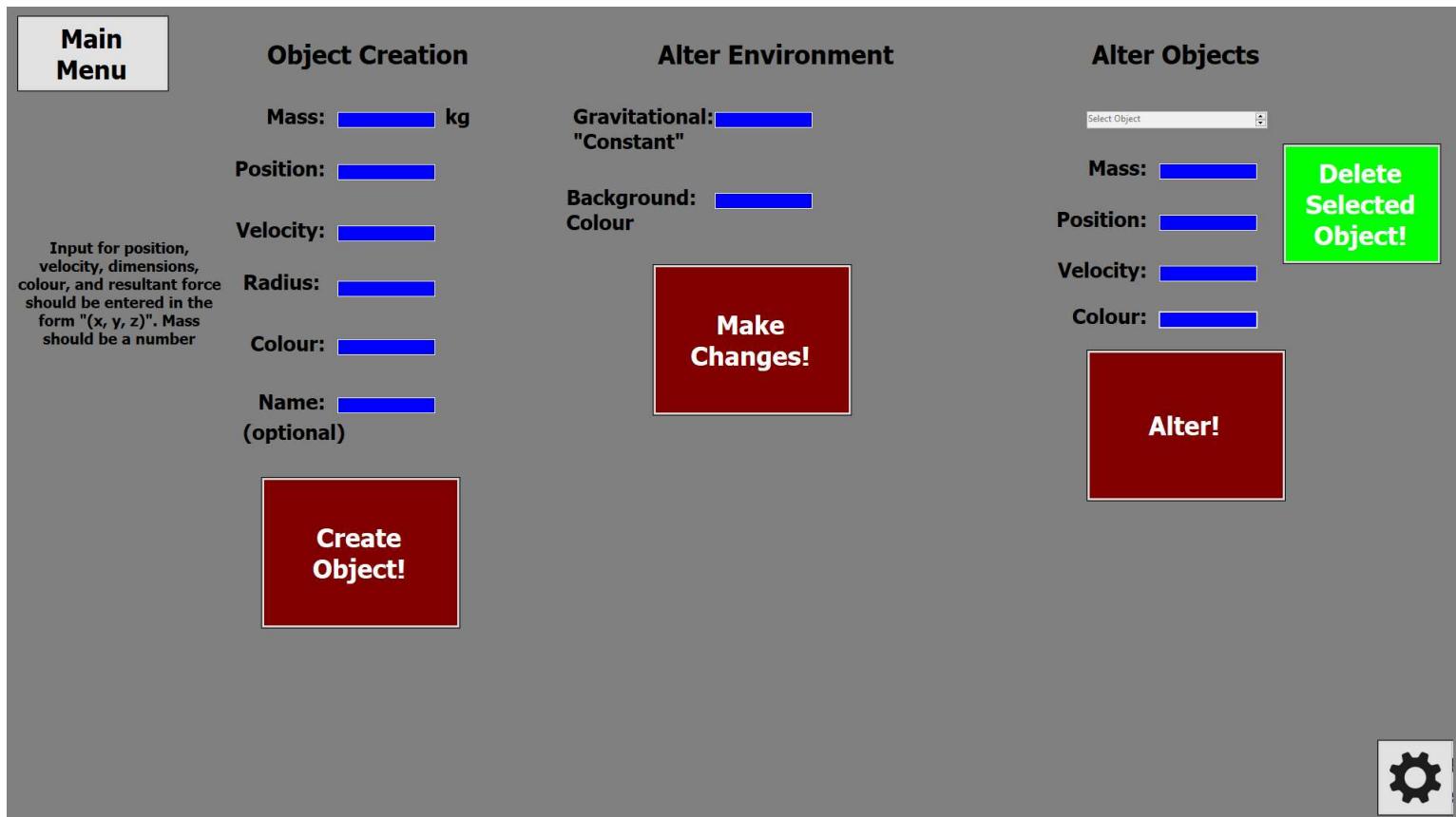


The projectile motion simulation is now complete.

The final thing I must do is finalise the orbit simulation. At this point, each of the success criteria will have been completed.

For this simulation, only cubes can be created, so instead of dimensions as a vector, there will be only “radius” which will half of the cube’s side length. There will be no collisions at all in this simulation. The biggest difference will of course be that there will no be a gravitational force between objects, which if set to well chosen values can cause one object to orbit another, or indeed to objects to orbit each other. I may change the units used in this simulation, e.g., distance and mass so that there will be noticeable effects without needing to enter very large values.

The first step, as before, is to copy the UI elements from simulation 1 into the new simulation. Then remove and rename as necessary.



I now need to add functionality to these textboxes. The only new textboxes are “radius” which replaces “dimensions” and “Gravitational “Constant””. I put “Constant” in quotation marks because I am allowing the user to change it, but in reality, it is simply always the same.

Changing the dimensions textbox to radius, was as simple as changing the line of code which validates the user input to, instead of ensuring that it is of the form (x,y,z) to instead ensure that it is a number > 0 and then changing the line of code which assigns the dimensions

vector to the entered values (by using Global.regExInterp()) to a line which simply sets it to be a vector, all of which's components are simply twice the radius.

```

private void Create_Click(object sender, EventArgs e)
{
    //Validate user inputs
    bool check = true;
    string pattern = @"^(\()?( -)?(\d+)(.\d+)?,( +)?(-)?(\d+)";
    string sMass = massInput.Text;
    string sRadius = radiusInput.Text;
    Vector colour = new Vector(0, 0, 0);
    if (double.TryParse(sMass, out double mass))
    {
        if (mass <= 0) { check = false; }
        //entered mass is negative which is not allowed
    }
    else { check = false; } //entered mass is not a number

    if (double.TryParse(sRadius, out double radius))
    {
        if (radius <= 0) { check = false; }
        //entered radius is negative which is not allowed
    }
    else { check = false; } //entered radius is not a number

    //Assign object creation variables
    Vector position = Global.regExInterp(sPosition);
    Vector velocity = Global.regExInterp(sVelocity);
    Vector dimensions = new Vector(2 * radius, 2 * radius, 2 * radius);
}

```

I will also remove the collision code for this form, although in this case I can also remove the code for collisions wtith the ground, as well as everything else to do with the ground.

```

//Physics-
void physics()
{
    //Collision Detection
    List<Tuple<Vector, Entity, Entity>> check = Global.checkAllCollisions();
    for (int i = 0; i < Global.Entities.Count; i++)
    {
        //Add weight to resultant force.
        Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0, Global.Entities[i].rigidBody.mass * Global.gravity, 0));

        Global.updateAcceleration(Global.Entities[i]);
        Global.updateVelocity(Global.Entities[i]);
        Global.displaceObject(Global.Entities[i], Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity, Global.frameTime));

        Global.updateAngularVelocity(Global.Entities[i]);
        Global.rotateObject(Global.Entities[i], Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity, Global.frameTime));

        //Take away weight
        Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0, -Global.Entities[i].rigidBody.mass * Global.gravity, 0));
    }
}

```

(Ignore the crossed out lines)

The code which adds the weight (force due to gravity) and then removes it, is still there. I wish to replace this with some code which will calculate the force due to gravity which the objects apply upon each other. For this I have first created a new Global variable, gravConst, which is set by default to a value of  $6.67 \times 10^{-11}$ .

```
public static double gravConst = 6.67*Math.Pow(10, -11); //Gravitational Constant = 6.67x10^-11
```

The make changes button, now sets this variable, with the above being the default if left blank.

```
private void MakeChanges_Click(object sender, EventArgs e)
{
    bool check = true;
    string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?";
    string sGrav = gravityInput.Text;
    string sColour = bgColourInput.Text;
    double grav;
    Vector colour = new Vector(0, 0, 0);

    if (sGrav == "") //if sGrav is empty it should pass the test immediately
    {
        grav = 6.67*Math.Pow(10, -11);
    }
    else if (!double.TryParse(sGrav, out grav))
    {
        check = false;
    }

    //make sure colour is of the correct form or is an empty string
    else if (sColour != "" && !Regex.IsMatch(sColour, pattern))
    { check = false; }

    //if (check == true)
    if (check == true)
    {
        //Remove validation label (in case it is there)
        ValidationLabel2.Visible = false;
        ValidationLabel2.Enabled = false;

        if (sColour != "")
        {
            colour = Global.regExInterp(sColour);
            //apply background colour
            Global.bgColour = Color.FromArgb(Convert.ToInt16(colour.x), Convert.ToInt16(colour.y), Convert.ToInt16(colour.z));
        }

        //apply gravity
        Global.gravConst = grav;
    }
    else
    {
        //Display validation label
        ValidationLabel2.Visible = true;
        ValidationLabel2.Enabled = true;
    }
}
```

The final thing I must do is replace the lines of code which add and remove gravity before and after updating acceleration, velocity and position.

```
//Physics
void physics()
{
    for (int i = 0; i < Global.Entities.Count; i++)
    {
        //calculate the force of gravity from all other objects
        Vector gravity = new Vector(0, 0, 0);
        for (int j = 0; j < Global.Entities.Count; j++)
        {
            if (i != j) //Checking every *other* object
            {
                Vector vectBetweenObjects = Global.vectBetweenPoints(Global.Entities[i].rigidBody.centreOfMass, Global.Entities[j].rigidBody.centreOfMass);
                double r = Global.vectMag(vectBetweenObjects);
                Vector newGrav = Global.vectorTimesScalar(vectBetweenObjects, Global.gravConst * Global.Entities[i].rigidBody.mass * Global.Entities[j].rigidBody.mass / Math.Pow(r, 3));
                gravity = Global.vectorAddVector(gravity, newGrav);
            }
        }
        //Add this force:
        Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, gravity);

        Global.updateAcceleration(Global.Entities[i]);
        Global.updateVelocity(Global.Entities[i]);
        Global.displaceObject(Global.Entities[i], Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity, Global.frameTime));

        Global.updateAngularVelocity(Global.Entities[i]);
        Global.rotateObject(Global.Entities[i], Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity, Global.frameTime));

        //Take away gravitational force
        Global.Entities[i].rigidBody.force = Global.vectorAddVector(Global.Entities[i].rigidBody.force, Global.vectorTimesScalar(gravity, -1));
    }
}
```

I will quickly walk through the new code:

```
Vector gravity = new Vector(0, 0, 0);
for (int j = 0; j < Global.Entities.Count; j++)
//This for loop checks through each object
{
    if (i != j) //The if statement ensures that we don't apply an objects own
    gravity to itself (though this would end up being zero anyway, the if statement
    saves time by preventing these unnecessary calculations.
    {
        Vector vectBetweenObjects =
Global.vectBetweenPoints(Global.Entities[i].rigidBody.centreOfMass,
Global.Entities[j].rigidBody.centreOfMass); //Find the vector going from our
object to the other.
        double r = Global.vectMag(vectBetweenObjects); //Distance between the
objects
        Vector newGrav = Global.vectorTimesScalar(vectBetweenObjects,
Global.gravConst * Global.Entities[i].rigidBody.mass *
Global.Entities[j].rigidBody.mass / Math.Pow(r, 3)); //Gravitational force
contributed by the jth object.
        gravity = Global.vectorAddVector(gravity, newGrav); //This contribution
is added to the total
    }
}
```

The force is then removed again at the end.

The line which calculates “newGrav” uses Newton’s law of Gravitation:

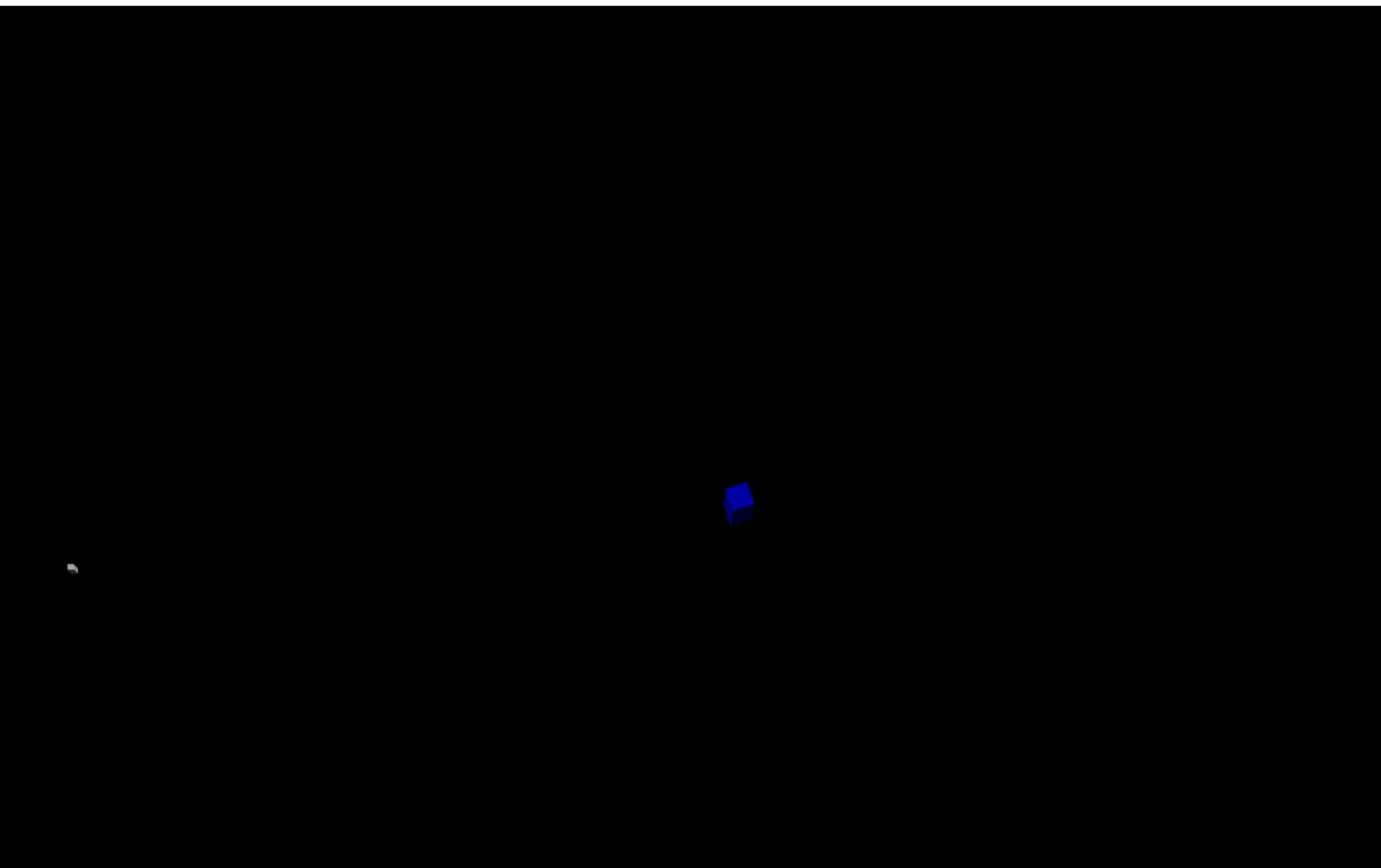
$$F = \frac{Gm_1m_2}{r^2}$$

The reason why I have used an  $r^3$  instead of  $r^2$  is because I am using the vector going from the centre’s of masses of the objects, which I need to divide by  $r$  in order to get a unit vector. Dividing  $\frac{Gm_1m_2}{r^2}$  by  $r$  gives  $\frac{Gm_1m_2}{r^3}$  hence the  $r^3$ .

To test this, I have created two massive objects, one to represent the moon, and one the earth.

The Earth is blue with a radius of 6371000m and a mass of 59720000000000000000000kg. This will be created at the origin. The moon has a radius of 1737400m, a mass of 7300000000000000000000kg, with an initial position of 384400000m from the origin, (average distance from moon to earth). Finally its velocity will be 1022000ms<sup>-1</sup>.

After creating these objects, the moon starts moving as expected:



The only problem is that the movement is so slow that it is difficult to tell if the motion is circular or not.

I will need to test this implementation another way.

Using the formulae for circular motion and gravitation, we get

$$F = \frac{mv^2}{r} = \frac{GMm}{r^2} \Rightarrow v = \sqrt{\frac{GM}{r}}$$

I can use this formula to determine how fast an object must move around another of mass  $M$ , in order to orbit it with at a distance of  $r$ .

If I created one object of mass 1000000 (1 million) kg with a radius of 1. I will create another at a distance of 10 (its mass doesn't matter so I will let it be 1). This second object would need to have a velocity of

$$\sqrt{\frac{GM}{r}} = \sqrt{\frac{6.67 \times 10^{-11} \times 10^6}{10}} = 0.00258 \text{ ms}^{-1}$$

in order to orbit the first object. The issue is that, since this velocity is very small compared to the path it will need to travel it will take a very long time to do so. Specifically:

$$t = \frac{2\pi r}{v} = \frac{20\pi}{0.00258} = 24329 \text{ seconds} = 6 \text{ hours, 45 minutes and 29 seconds}$$

I don't want to wait around for that long. I can use another formula here to decide the time period for myself, lets say  $T$  seconds.

$$v = \frac{2\pi r}{T}$$

The other equation still needs to be satisfied, so I can solve them each simultaneously to work out the appropriate radius and velocity.

$$\frac{2\pi r}{T} = \sqrt{\frac{GM}{r}} \Rightarrow \frac{4\pi^2 r^2}{T^2} = \frac{GM}{r} \Rightarrow 4\pi^2 r^3 = GMT^2$$

This result is called kepler's third law.

For a radius of 10m, if I wanted a time period of 10 s, the mass would need to be

$$M = \frac{4\pi r^3}{GT^2} = \frac{4\pi^2 \times 10^3}{6.67 \times 10^{-11} \times 10^2} = 5.92 \times 10^{12} \text{ kg}$$

The orbital velocity would then need to be

$$v = \frac{2\pi r}{T} = \frac{2\pi \times 10}{10} = 6.283 \text{ ms}^{-1}$$

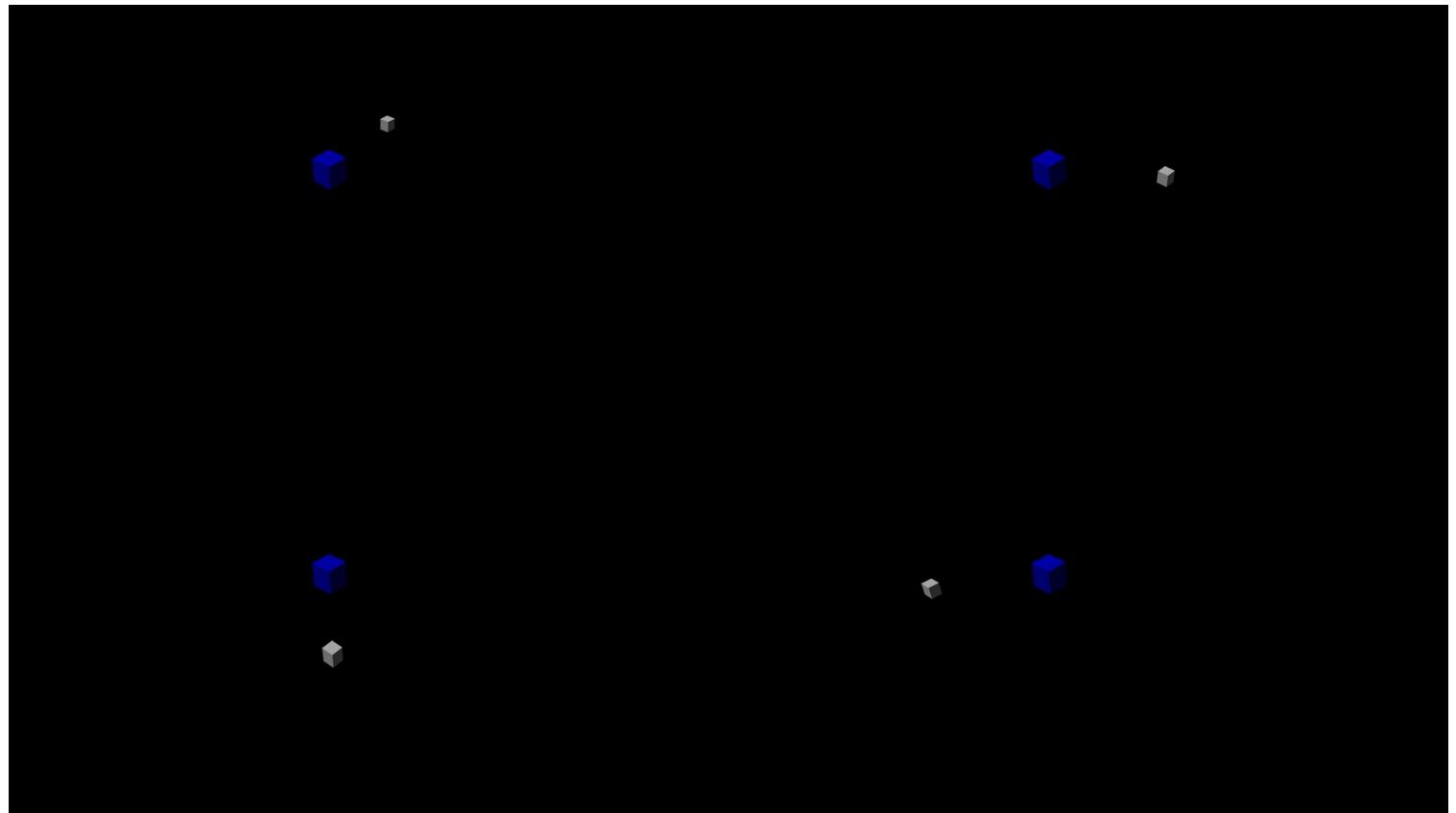
So, finally, to test this simulation, I will create an object of mass 592000000000kg at the origin. This object will be blue. I will create a second (white) object at a distance of 10 from the origin, with a velocity of  $6.283 \text{ ms}^{-1}$ . The second object should (hopefully) orbit the first in about 10 seconds.

This is shown to be working here:

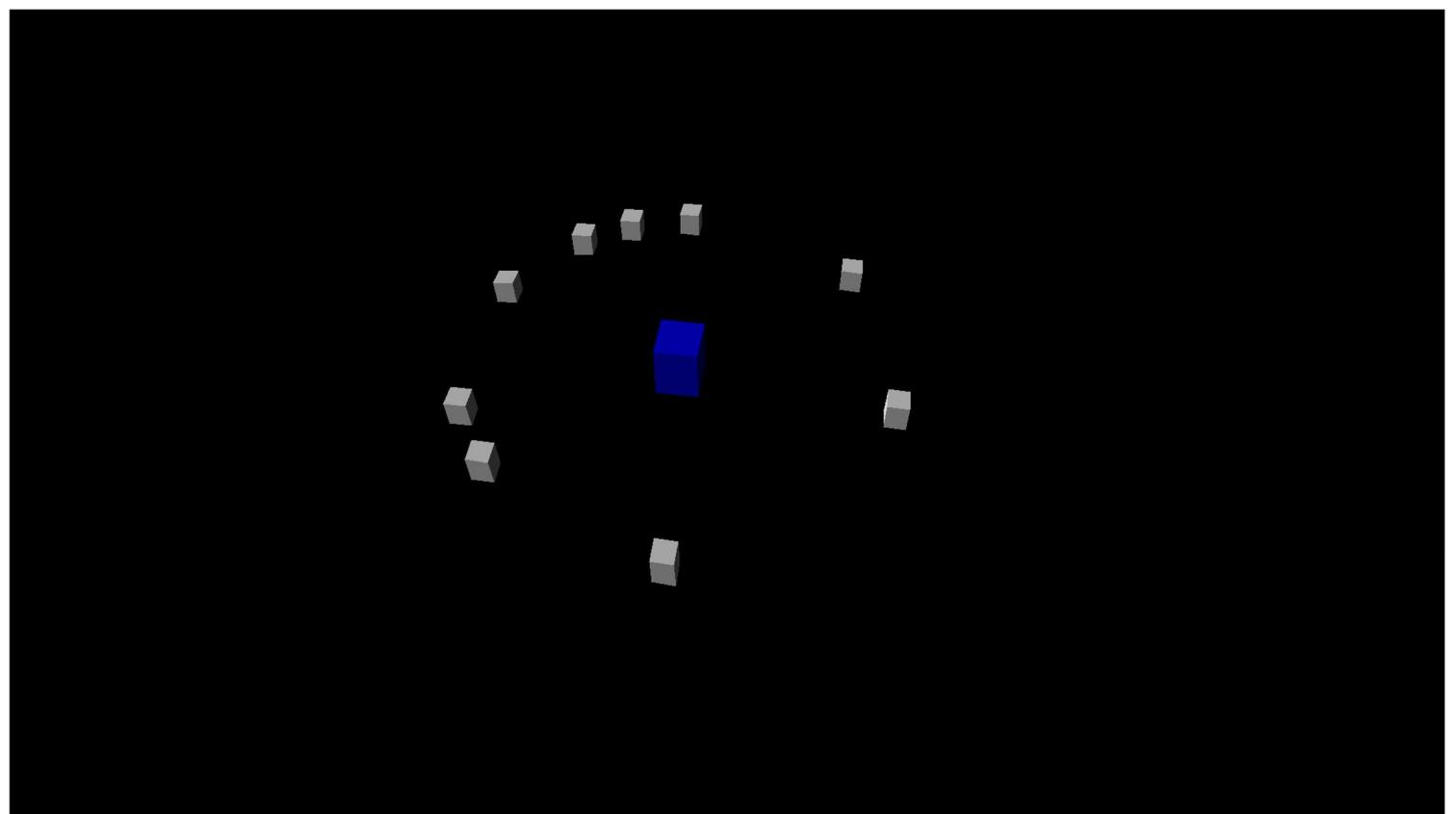
Object creation:

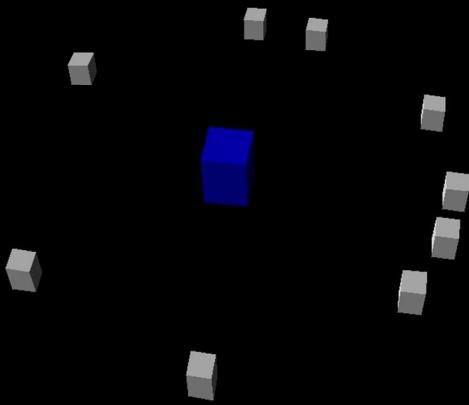
<b>Object Creation</b>	<b>Object Creation</b>
<b>Mass:</b> <input type="text" value="5920000000000"/> kg	<b>Mass:</b> <input type="text" value="1"/> kg
<b>Position:</b> <input type="text" value="0,0,0"/>	<b>Position:</b> <input type="text" value="0,0,-10"/>
<b>Velocity:</b> <input type="text" value="0,0,0"/>	<b>Velocity:</b> <input type="text" value="6.283,0,0"/>
<b>Radius:</b> <input type="text" value="1"/>	<b>Radius:</b> <input type="text" value="0.5"/>
<b>Colour:</b> <input type="text" value="0,0,255"/>	<b>Colour:</b> <input type="text" value="255,255,255"/>
<b>Name:</b> <input type="text" value="Obj1"/> <b>(optional)</b>	<b>Name:</b> <input type="text" value="Obj2"/> <b>(optional)</b>
<b>Create Object!</b>	<b>Create Object!</b>

One orbiting the other:



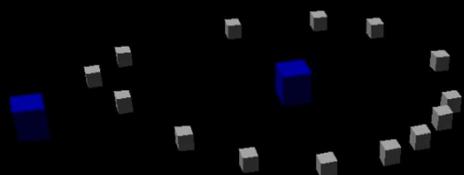
I can create multiple objects which continue to orbit as shown here:

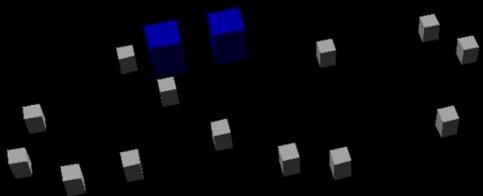
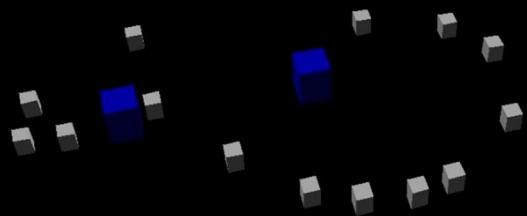


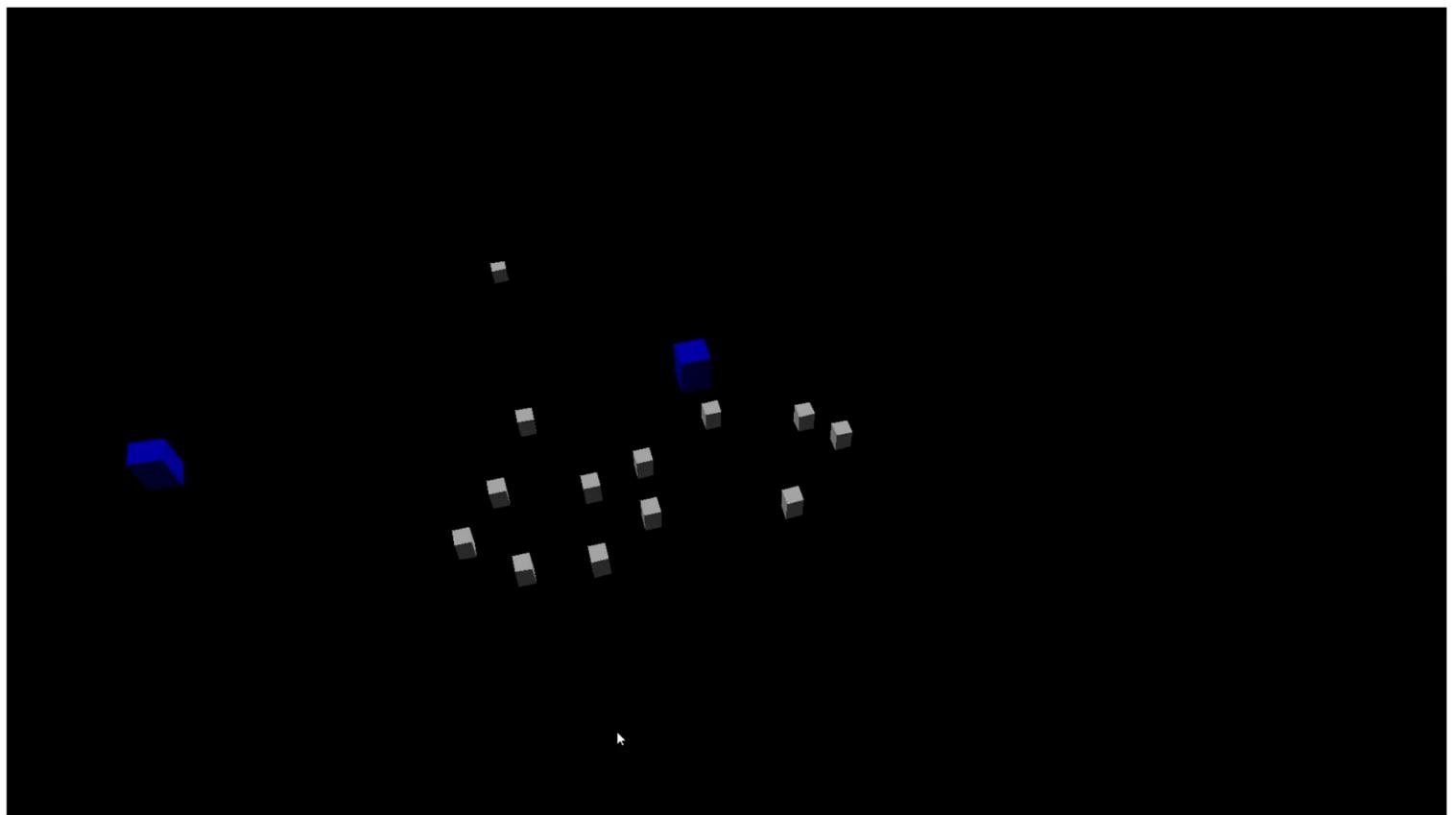


If I create another blue cube, with the same mass as the current one, the smaller objects should change their trajectory.

This is exactly what we see. We also see that the two massive objects, move towards each other due to their gravitational pull on each other:

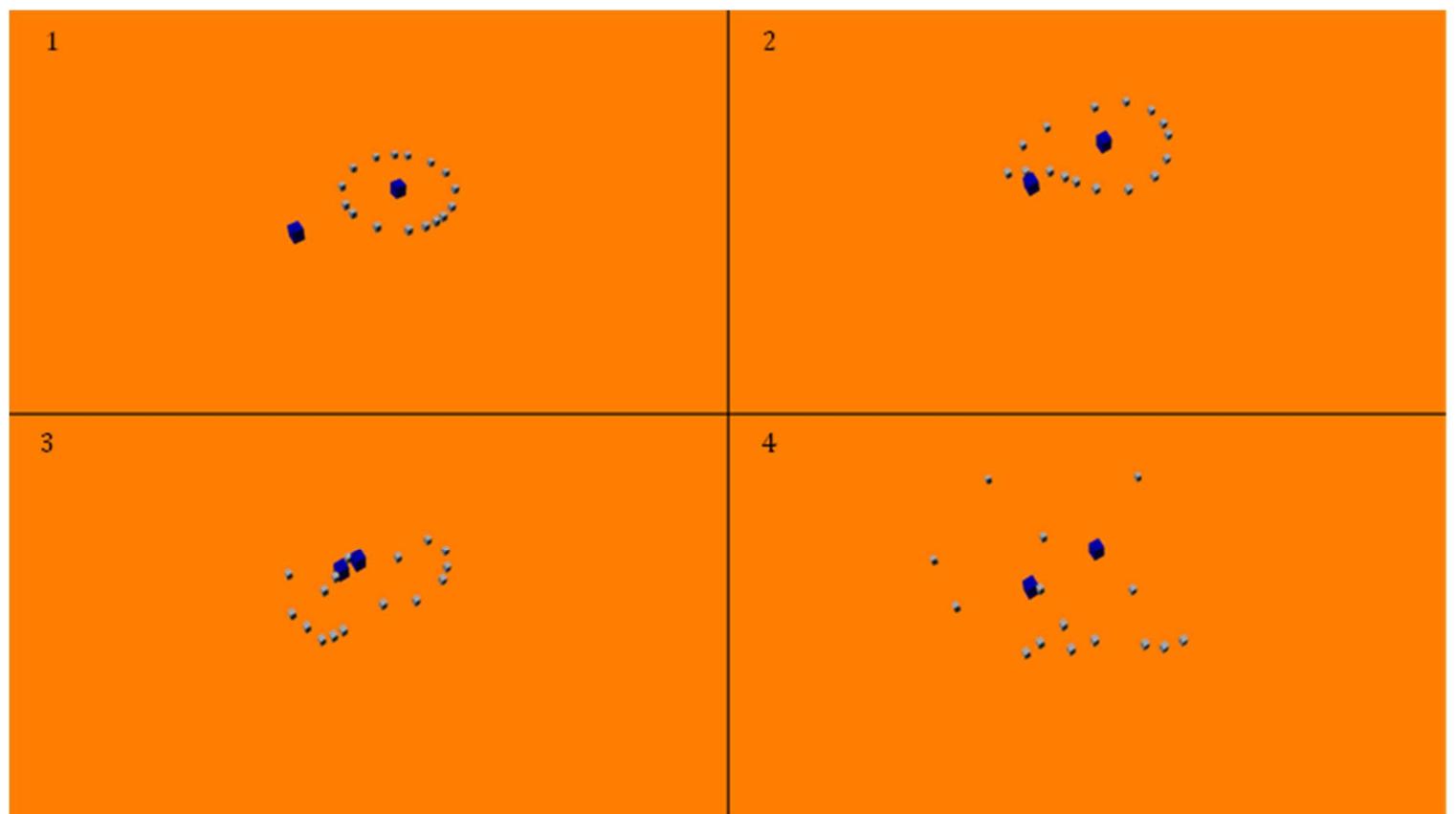






Because I removed collisions between objects, the two massive objects pass through each other. This decreases the distance between them to near zero, giving them a great gravitational force, which pushes them apart. To fix this, I will disable the gravitational force if the objects are touching (that is if the distance between their centres is less than the sum of their radii). After implementing this, the large masses no longer exert a huge force upon each other when they get to close, they instead just pass through.

I'll give you an orange background to make things more interesting:



### Testing Prototype 3

This table shows mainly tests performed at the end of development of this prototype, although there are a few failed tests which reference previous sections of this work at which certain features did not work. It is explained how these problems were solved. Each successful test has been performed after development has completed.

Test No.	Input Data	Expected Output	Actual Output	Success?
18	Settings button from menu screen clicked	Settings dialogue window opens	Settings dialogue window opens	Yes
19	Rigid Body Collisions selected	Rigid Body Collisions simulation opens	Rigid Body Collisions simulation opens	Yes
20	Projectile Motion selected	Projectile Motion simulation opens	Projectile Motion simulation opens	Yes
21	Orbit selected	Orbit simulation opens	Orbit simulation opens	Yes
22	In simulation 1 (RBC) menu, enter 0,0,0 to “background colour” box and click “Make Changes!”	Background colour changes to black	Bg colour changes to black	Yes
23	Rigid Body Collisions: Set mass to 1, CoR to 1, Position to 0,0,0, velocity to 0,0,0, dimensions to 1,1,1, colour to 255,0,0, resultant force to 0,0,0 and click “Create Object!”	Red cube side length 1, created at the origin. Accelerates down due to gravity.	A red cube of side length 1 is created at the origin which accelerates down due to gravity.	Yes
24	Same as above but in Projectile Motion, with height of 10, initial speed and angle as 0	Same as above, until the object hits the ground when it will bounce back up to its initial height	Object accelerates downward until it hits the ground, it then rebounds up to its starting height	Yes
25	Same as above but in Orbit with radius of 0.5	Object is created at the origin but object does <b>not</b> accelerate, it remains stationary.	Object is created at the origin and accelerates downwards.	No
26	RBC: Click “Delete Selected Object!”	Selected object should disappear.	Selected object disappears.	Yes

	when an object is selected.			
27	Same as above but in Projectile Motion	Same as above	Same as above	Yes
28	Same as above but in Orbit	Same as above	Same as above	Yes
29	Orbit: Create an object with mass 592000000000kg, at the origin. Create a second object of any mass, at (0,0,-10) with velocity (6.283,0,0).	The second object should orbit the first at a distance of 10 metres, once every 10 seconds.	The second object orbits the first object at a distance of 10 metres, once every 10 seconds.	Yes
30	Orbit: Create a set of 10 smaller, white objects (mass 1kg) to orbit a single larger blue object (mass 592000000000kg) at a distance of 10. Create a second larger blue object, same mass as the other, a distance of 20 from the first. Set background colour to (0,0,0) (black) so the blue objects can be seen.	Smaller objects change path due to the second blue object. The blue objects should move together until they meet, where they will stop moving.	Smaller objects change path due to the second blue object. The blue objects move together until they meet, where the gravitational force between becomes very large, pushing them apart very quickly.	No

Addressing the two failed tests:

Test 25 failed, because the object accelerated downwards, despite being in the “orbit” simulation, where gravity should be disabled. This happened because the code which runs when the simulation

```
//calculate the force of gravity from all other objects
Vector gravity = new Vector(0, 0, 0);
for (int j = 0; j < Global.Entities.Count; j++)
{
    if (i != j) //Checking every *other* object
    {
        Vector vectBetweenObjects = Global.vectBetweenPoints(Global.Entities[i].rigidBody.centreOfMass, Global.Entities[j].rigidBody.centreOfMass);
        double r = Global.vectMag(vectBetweenObjects);
        if (r > Global.Entities[i].mesh.dimensions.x / 2 + Global.Entities[i].mesh.dimensions.x / 2)
        {
            Vector newGrav = Global.vectorTimesScalar(vectBetweenObjects, Global.gravConst * Global.Entities[i].mass);
            gravity = Global.vectorAddVector(gravity, newGrav);
        }
    }
}
```

begins reads:

The gravity is set to 9.81, it should be set to 0. This is easily corrected:

Test 25 now succeeds.

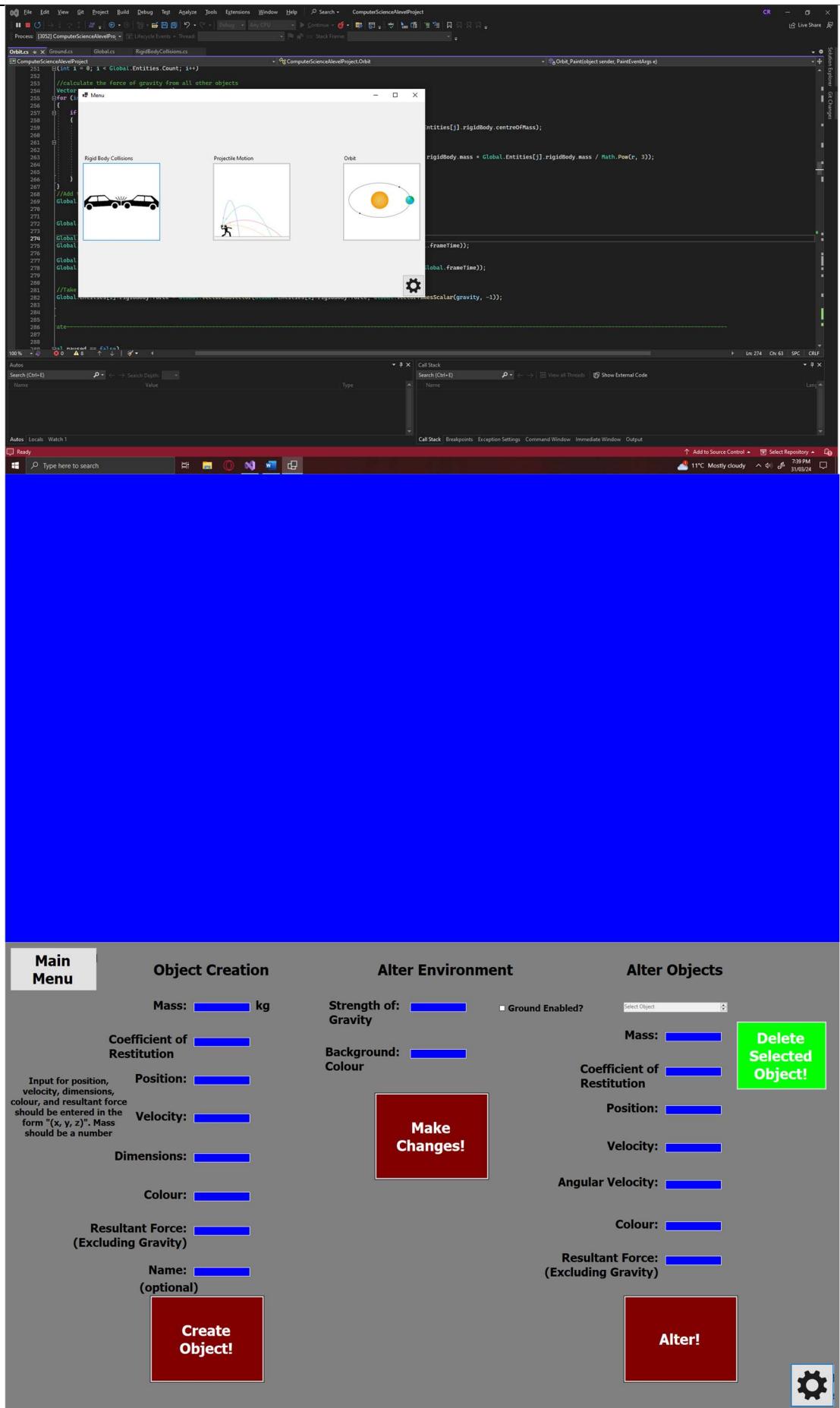
```
private void Orbit_Load(object sender, EventArgs e)
{
    Global.Ground.toggled = false;
    Global.paused = false;
    Global.Entities = new List<Entity>();
    Global.collisionFlags = new List<Tuple<bool, Entity, Entity>>();
    Global.Camera = new Camera(new Vector(-1, 0, 0), new Vector(1, 0.01, 0.01), 7 * Math.PI / 18);
    Global.gravity = 9.81;
}
```

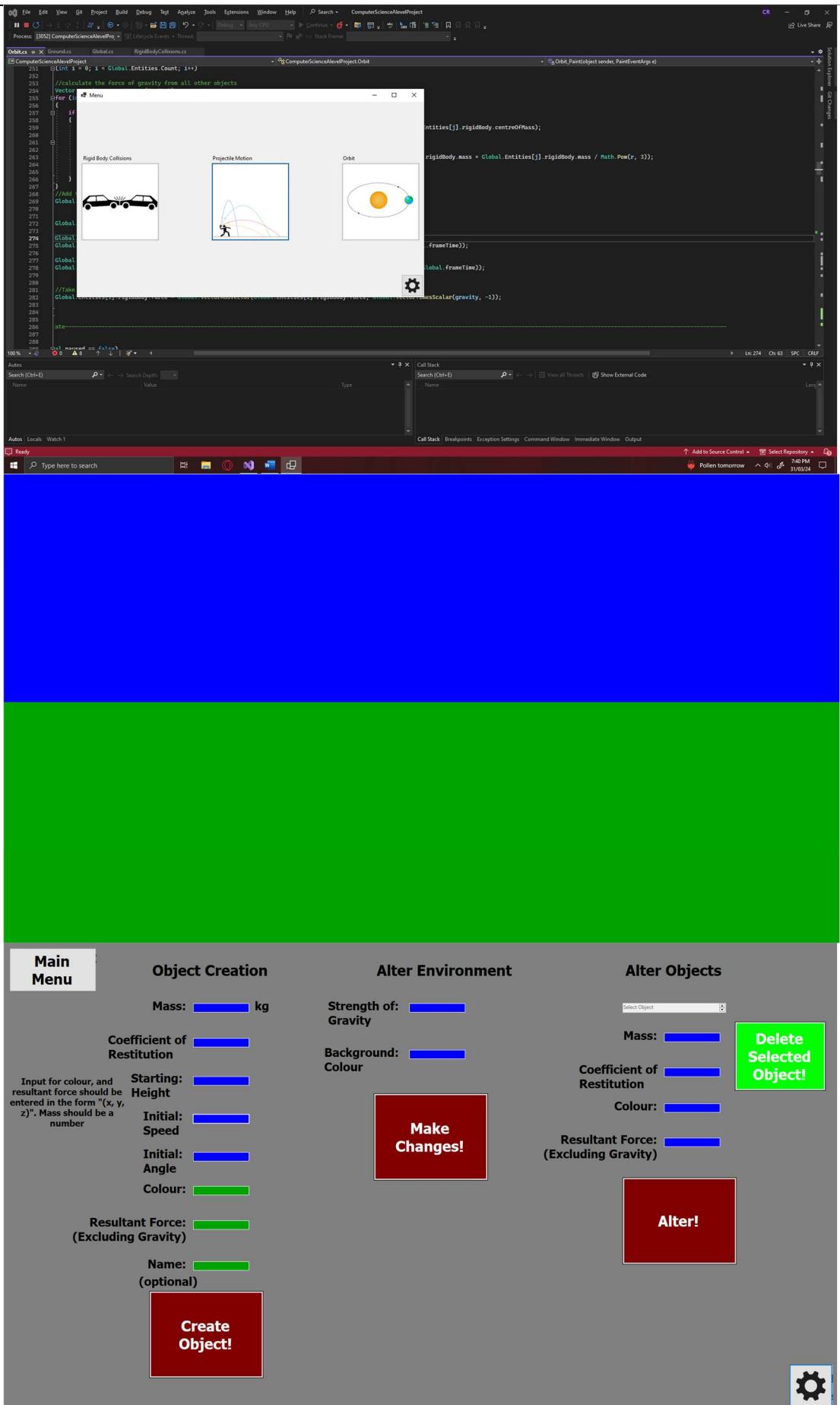
```
private void Orbit_Load(object sender, EventArgs e)
{
    Global.Ground.toggled = false;
    Global.paused = false;
    Global.Entities = new List<Entity>();
    Global.collisionFlags = new List<Tuple<bool, Entity, Entity>>();
    Global.Camera = new Camera(new Vector(-1, 0, 0), new Vector(1, 0.01, 0.01), 7 * Math.PI / 18);
    Global.gravity = 0;
}
```

Test 30 also, failed (referring to the test performed on page 260-261). This was corrected by disabling gravity between objects which are too close together.

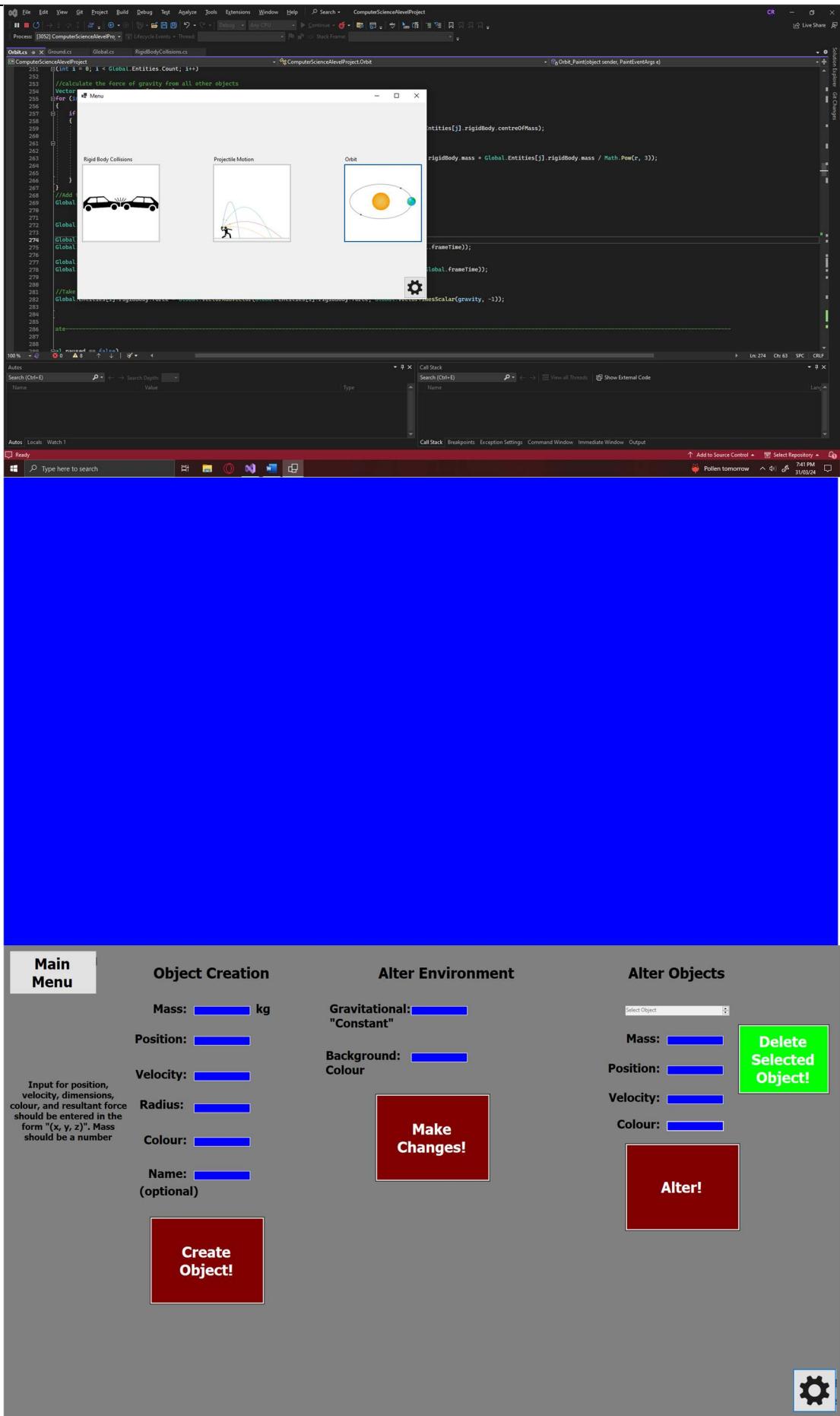
Test 30 now succeeds, (referencing the test done on page 262).

### Evidence for each test:





21



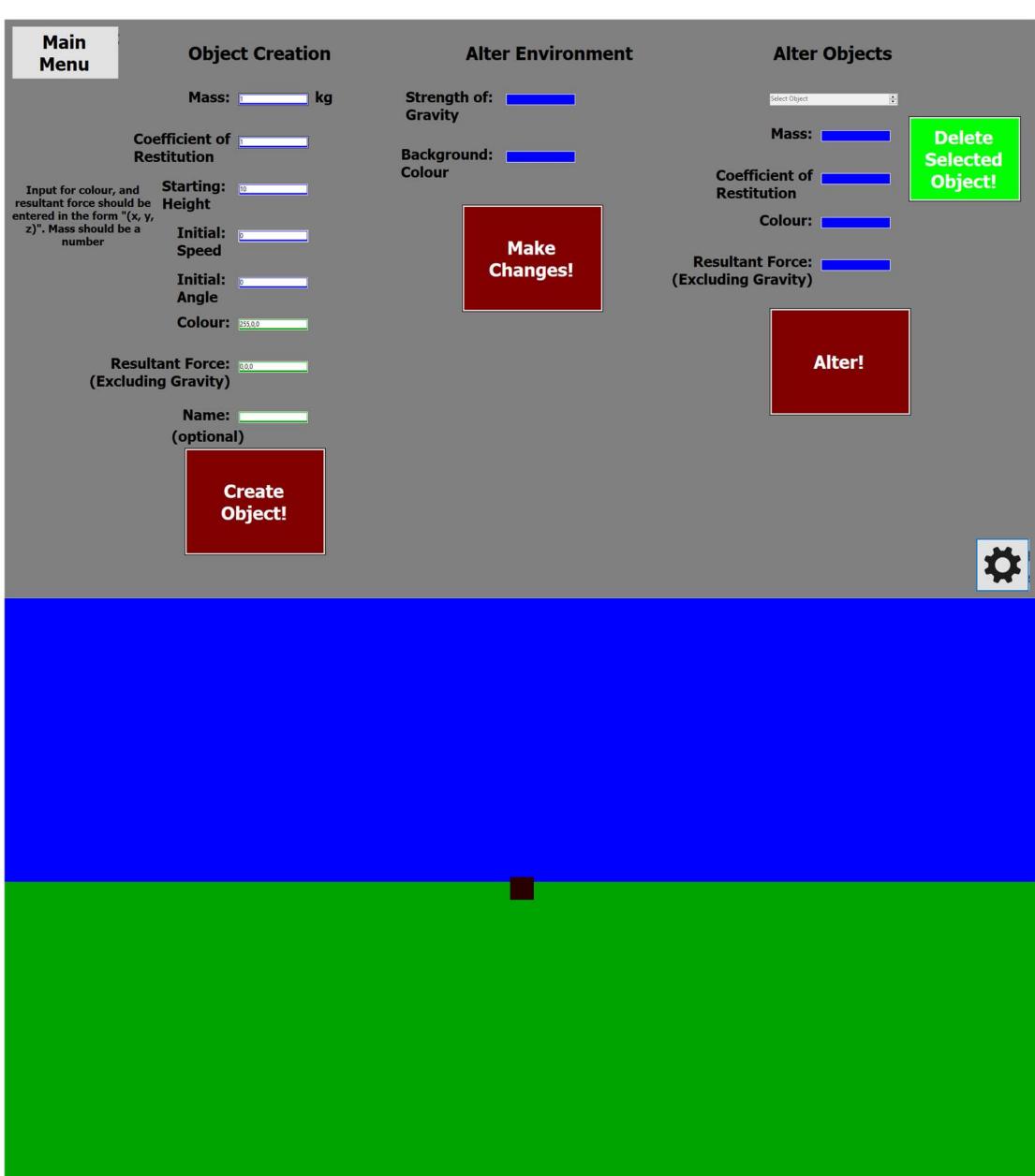
Main Menu	Object Creation	Alter Environment	Alter Objects
	<p>Mass: <input type="text"/> kg</p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number</p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Dimensions: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> <p>Name: <input type="text"/> (optional)</p>	<p>Strength of: <input type="text"/> Gravity: <input type="checkbox"/> Ground Enabled?</p> <p>Background: <input type="text"/> Colour: <input type="text"/></p>	<p>Select Object: <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Angular Velocity: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p>
	<input style="background-color: #800000; color: white; padding: 5px; width: 100px; height: 50px; font-size: 10px;" type="button" value="Create Object!"/>	<input style="background-color: #800000; color: white; padding: 5px; width: 100px; height: 50px; font-size: 10px;" type="button" value="Make Changes!"/>	<input style="background-color: #00ff00; color: black; padding: 5px; width: 100px; height: 50px; font-size: 10px;" type="button" value="Delete Selected Object!"/> <input style="background-color: #800000; color: white; padding: 5px; width: 100px; height: 50px; font-size: 10px;" type="button" value="Alter!"/>

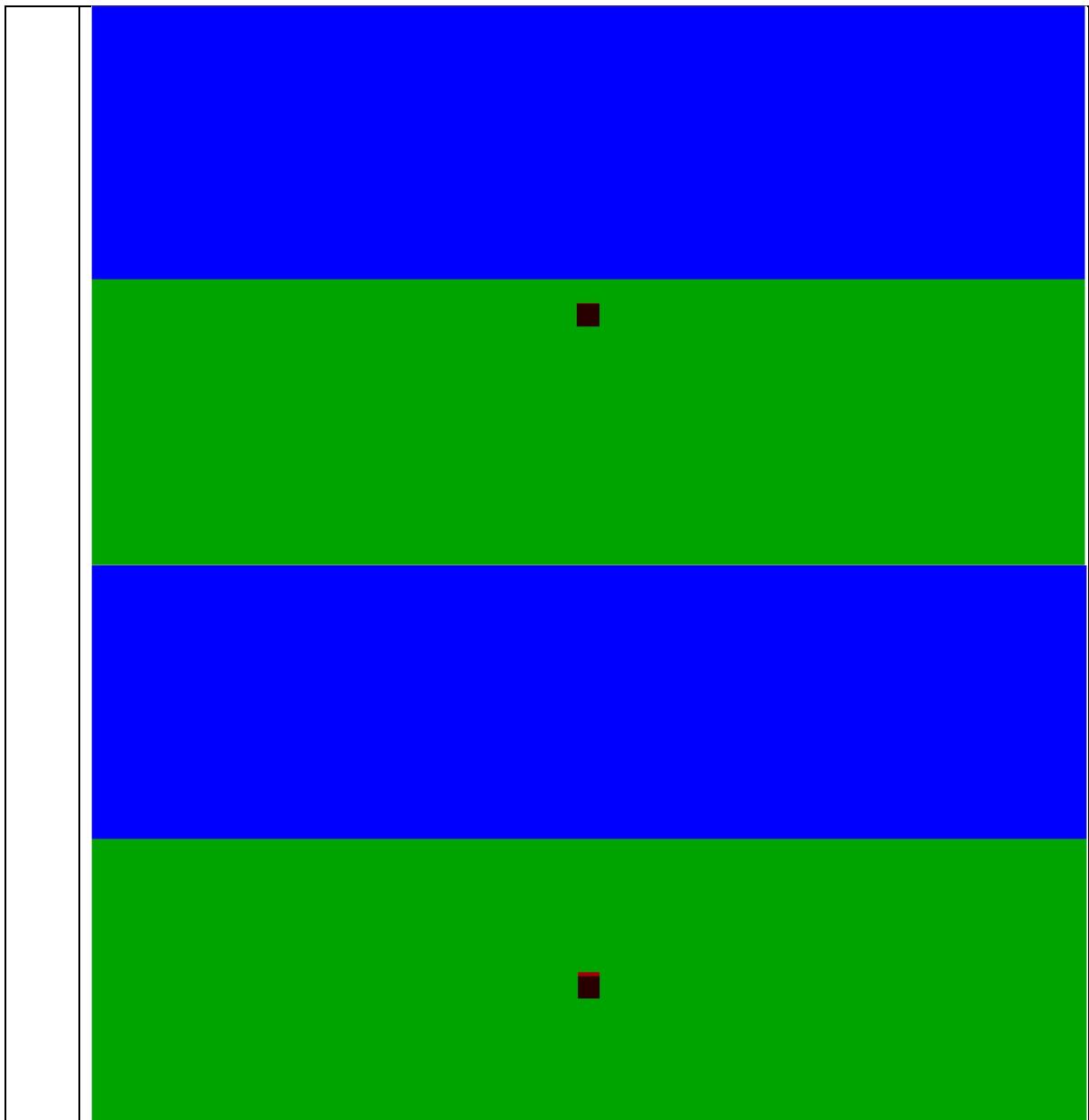
Main Menu	Object Creation	Alter Environment	Alter Objects
	<p><b>Object Creation</b></p> <p>Mass: <input type="text"/> kg</p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number</p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Dimensions: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> <p>Name: <input type="text"/> (optional)</p> <p><b>Create Object!</b></p>	<p><b>Alter Environment</b></p> <p>Strength of Gravity: <input type="text"/></p> <p>Background Colour: <input type="text"/></p> <p><b>Make Changes!</b></p>	<p><b>Alter Objects</b></p> <p>Ground Enabled? <input type="checkbox"/></p> <p>Select Object: <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Angular Velocity: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> <p><b>Delete Selected Object!</b></p> <p><b>Alter!</b></p> 

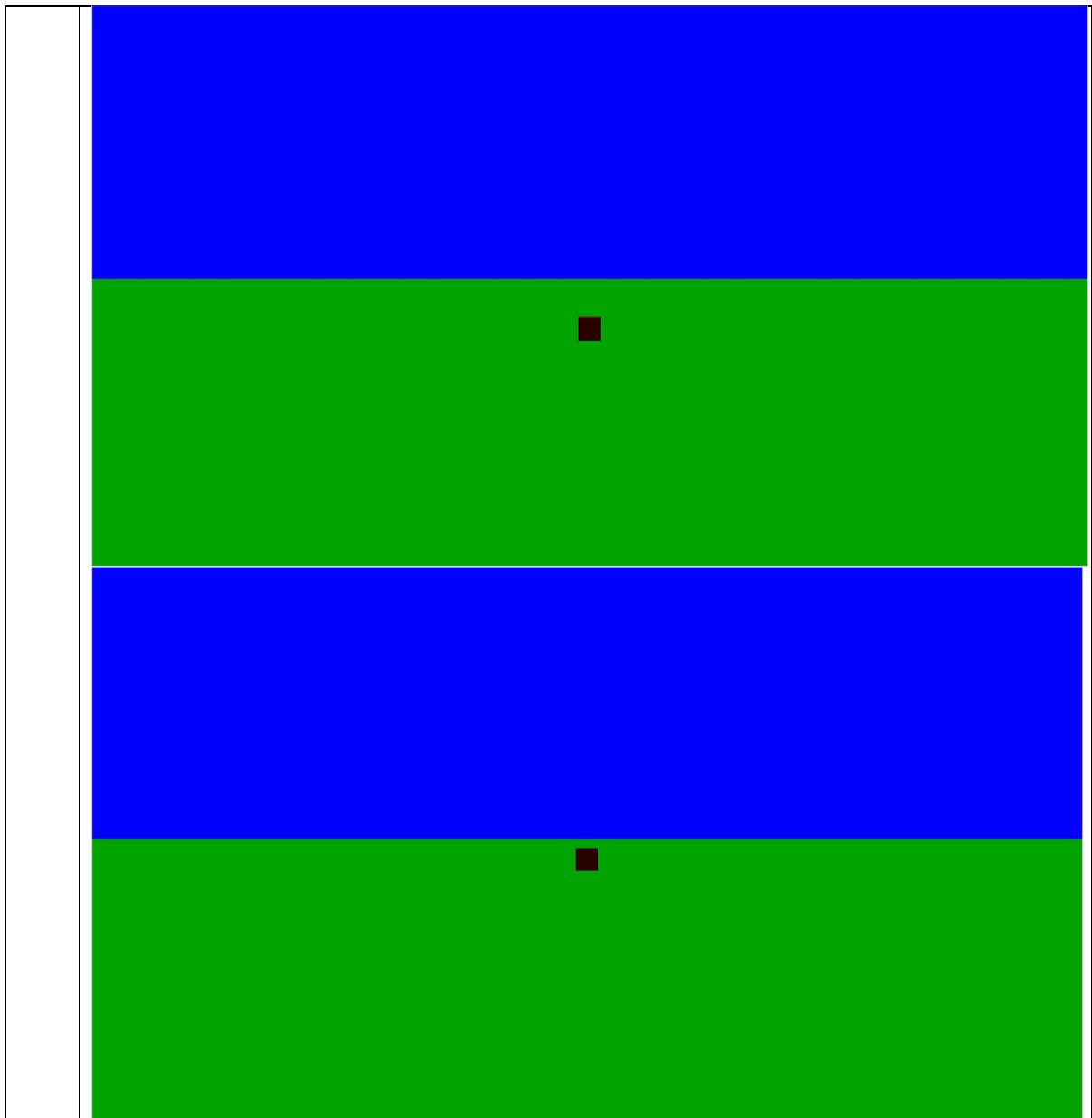
23

The image shows a software interface for creating and altering objects in a physics simulation environment. The interface is divided into several sections:

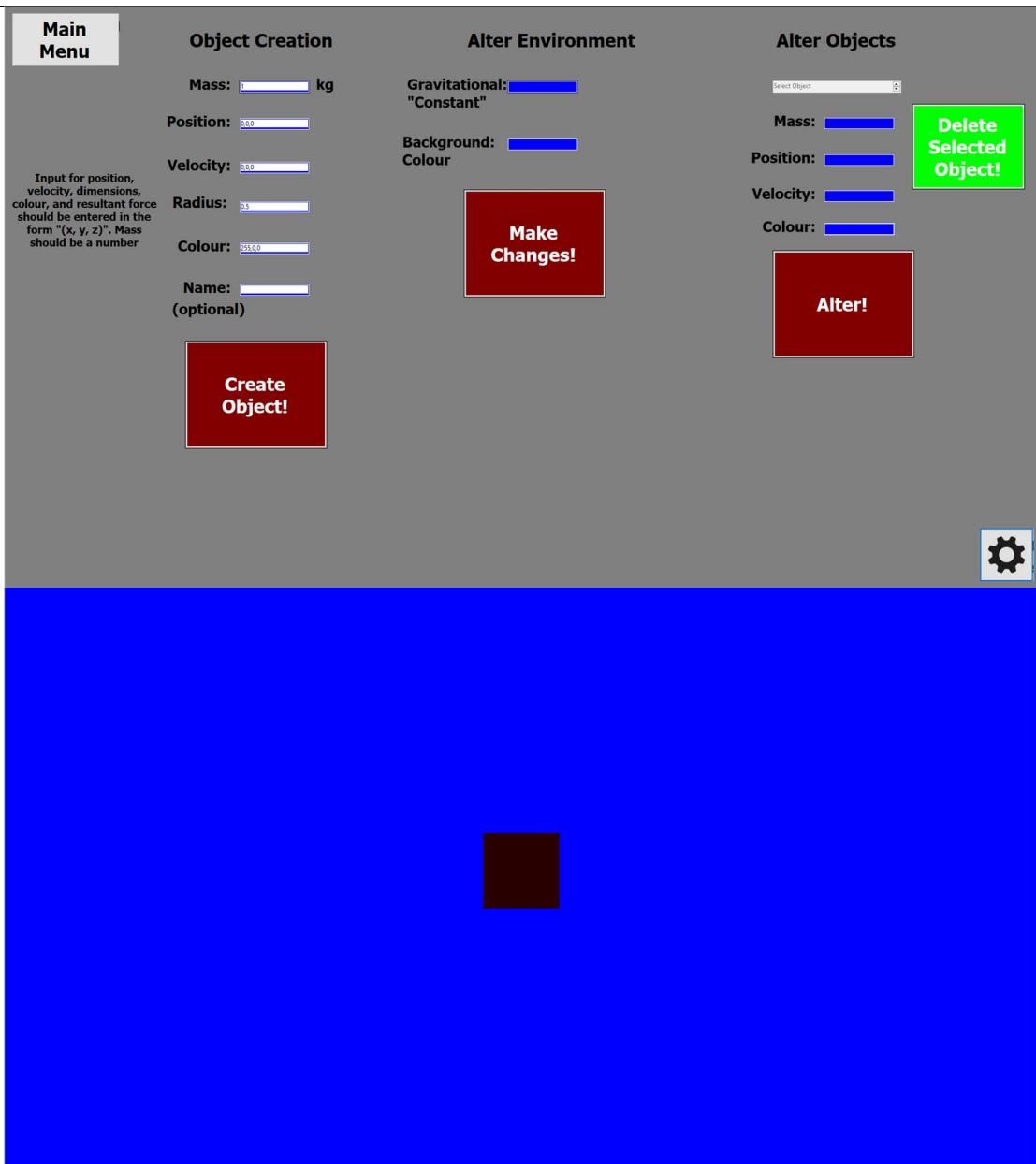
- Main Menu**: Located at the top left.
- Object Creation**: A section on the left containing fields for Mass, Coefficient of Restitution, Position, Velocity, Dimensions, Colour, and Resultant Force (Excluding Gravity). It also includes a note about input format and a **Create Object!** button.
- Alter Environment**: A section in the center containing fields for Strength of Gravity, Background Colour, and a **Make Changes!** button.
- Alter Objects**: A section on the right containing fields for Mass, Coefficient of Restitution, Position, Velocity, Angular Velocity, Colour, and Resultant Force (Excluding Gravity). It includes a **Alter!** button and a green box labeled **Delete Selected Object!**.
- Input Note**: A note at the top left specifies that input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)".
- Background**: A large blue rectangular area representing the simulation environment.





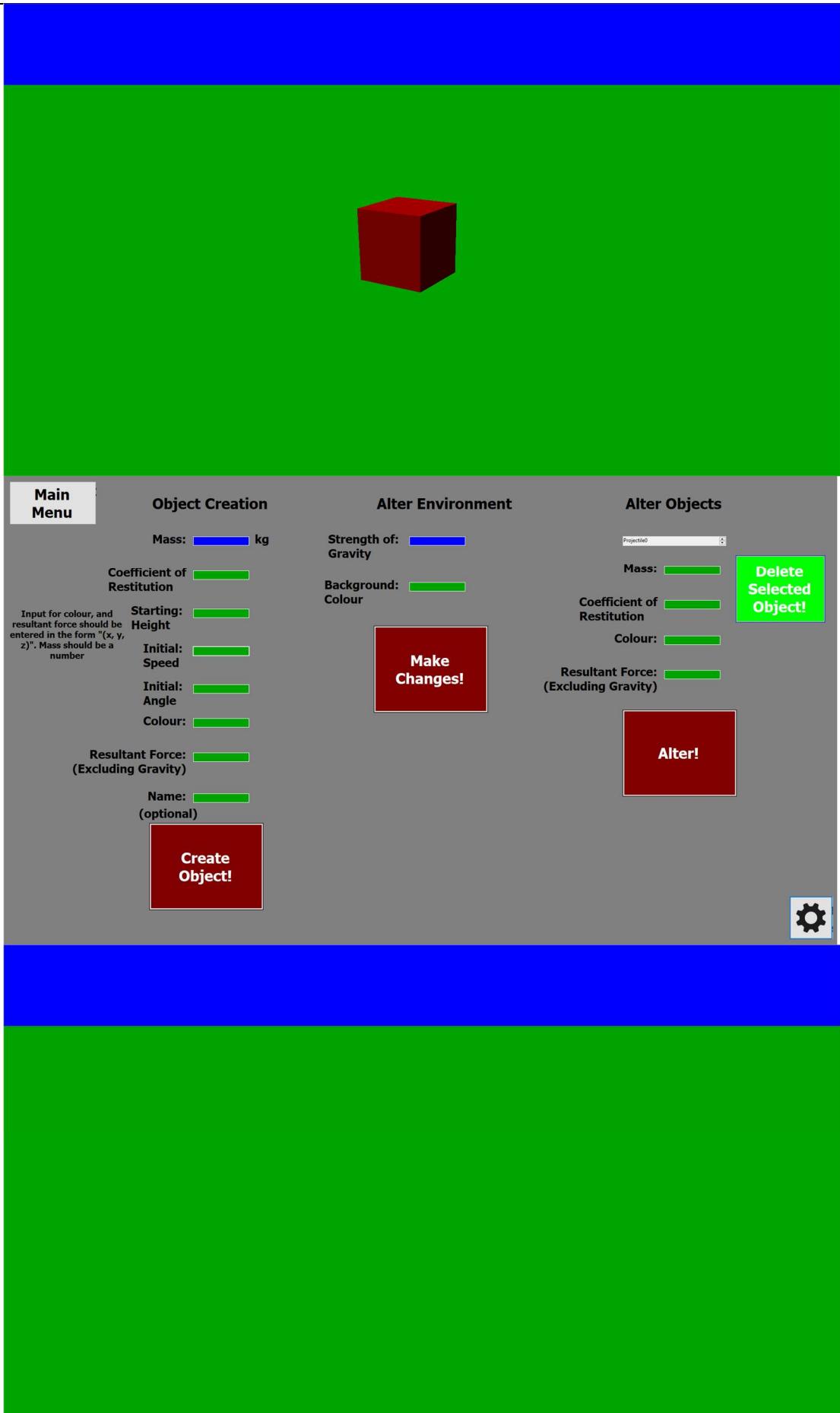


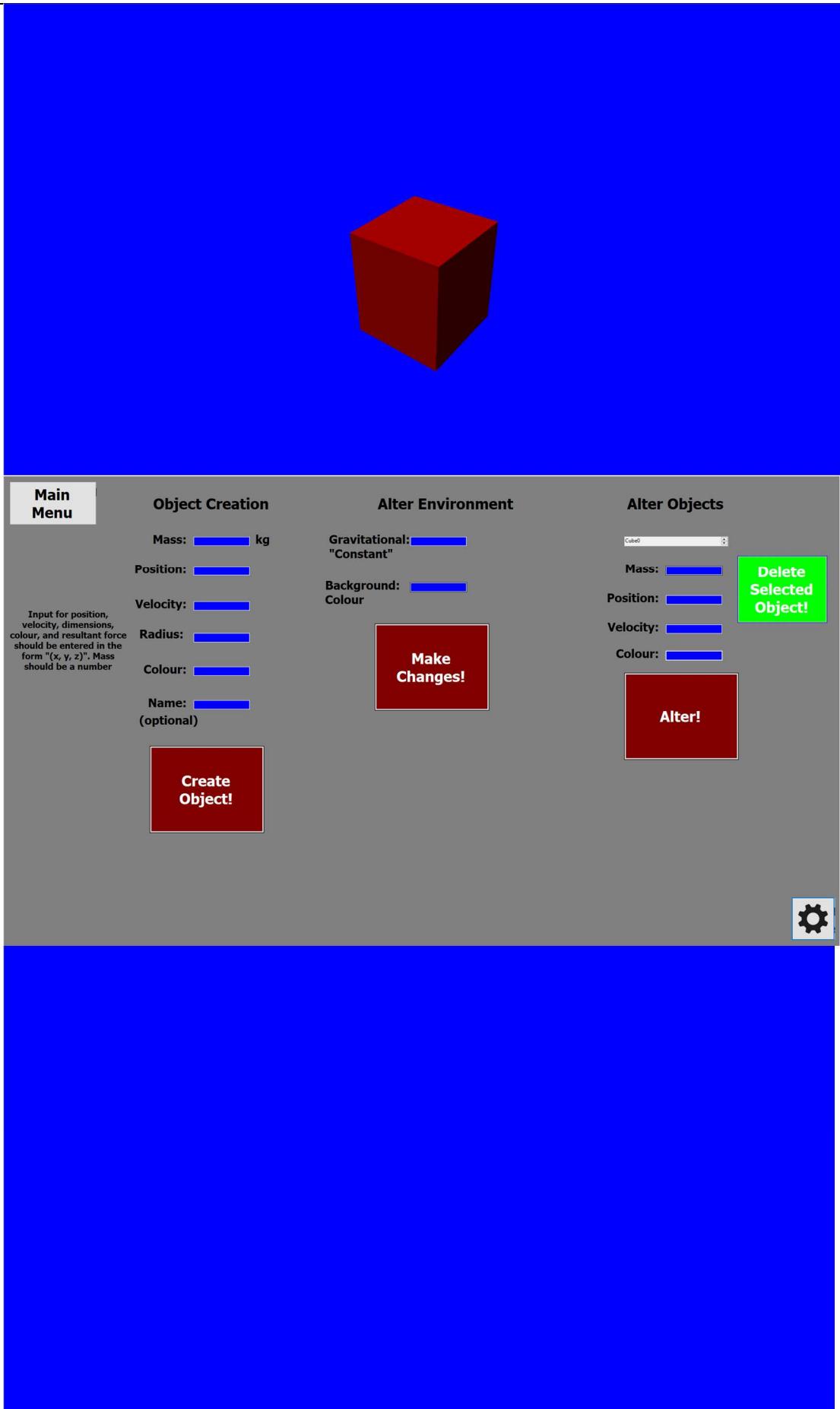
25



Main Menu	Object Creation	Alter Environment	Alter Objects
	Mass: <input type="text"/> kg Coefficient of Restitution: <input type="text"/> Position: <input type="text"/> <small>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number</small> Velocity: <input type="text"/> Dimensions: <input type="text"/> Colour: <input type="text"/> Resultant Force: <input type="text"/> (Excluding Gravity) Name: <input type="text"/> (optional)	Strength of: <input type="text"/> Gravity: <input type="checkbox"/> Ground Enabled? Background: <input type="text"/> Colour: <input type="text"/>	Cube: <input type="radio"/> Mass: <input type="text"/> Coefficient of Restitution: <input type="text"/> Position: <input type="text"/> Velocity: <input type="text"/> Angular Velocity: <input type="text"/> Colour: <input type="text"/> Resultant Force: <input type="text"/> (Excluding Gravity)
	<b>Create Object!</b>	<b>Make Changes!</b>	<b>Delete Selected Object!</b>
			<b>Alter!</b>

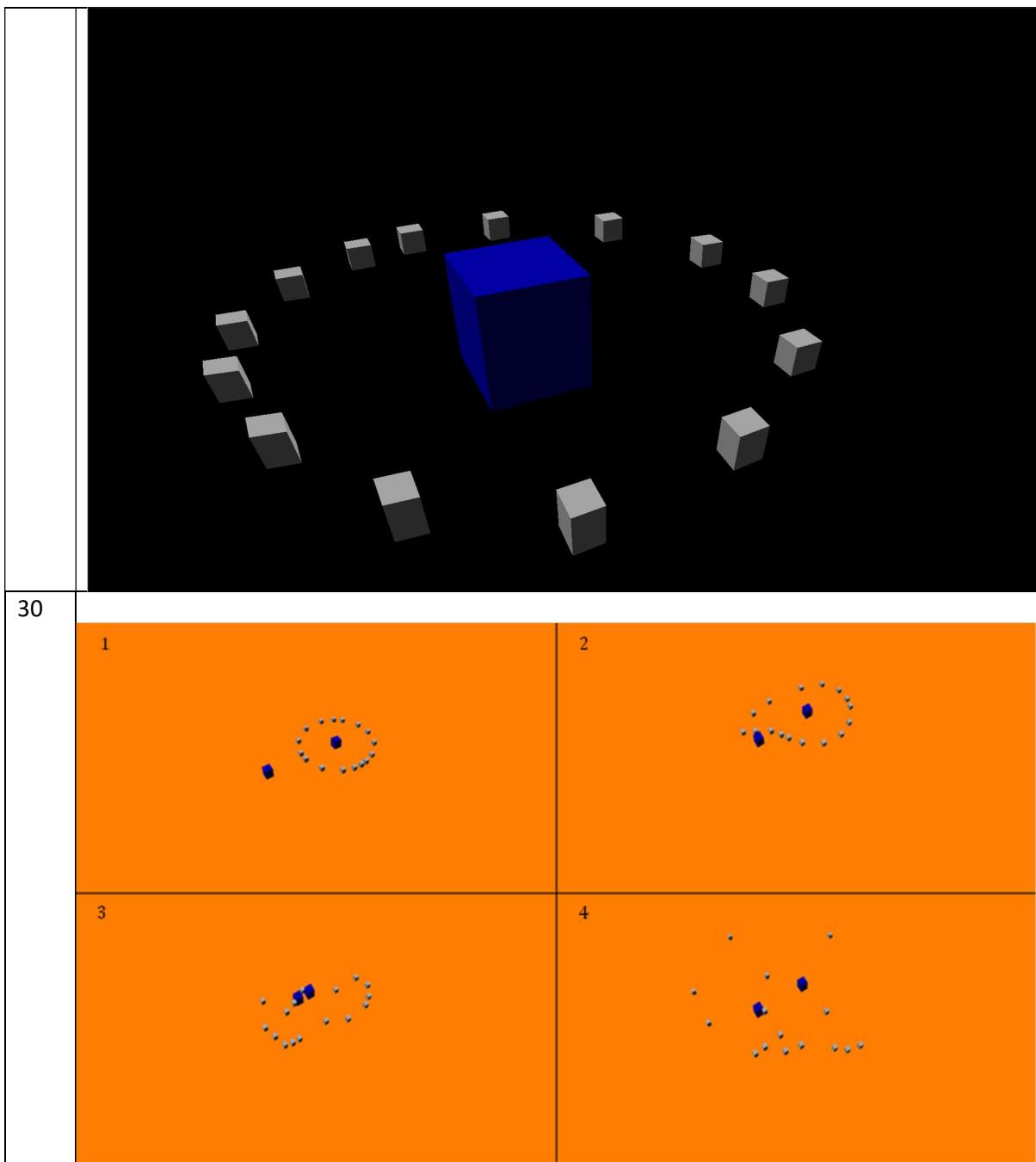






29

Main Menu	Object Creation	Alter Environment	Alter Objects
	<p><b>Object Creation</b></p> <p>Mass: <input type="text" value="55200000000"/> kg</p> <p>Position: <input type="text" value="0,0"/></p> <p>Velocity: <input type="text" value="0,0"/></p> <p>Radius: <input type="text" value="2"/></p> <p>Colour: <input type="text" value="0,0,255"/></p> <p>Name: <input type="text" value=""/></p> <p>(optional)</p> <p><b>Create Object!</b></p>	<p>Gravitational: <input type="text" value="Constant"/></p> <p>Background: <input type="text" value=""/></p> <p>Colour</p> <p><b>Make Changes!</b></p>	<p>Cube0 <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p><b>Alter!</b></p>
	<p><b>Object Creation</b></p> <p>Mass: <input type="text" value="1"/> kg</p> <p>Position: <input type="text" value="0,0,10"/></p> <p>Velocity: <input type="text" value="0,0,0"/></p> <p>Radius: <input type="text" value="0,5"/></p> <p>Colour: <input type="text" value="255,255,255"/></p> <p>Name: <input type="text" value=""/></p> <p>(optional)</p> <p><b>Create Object!</b></p>	<p>Gravitational: <input type="text" value="Constant"/></p> <p>Background: <input type="text" value=""/></p> <p>Colour</p> <p><b>Make Changes!</b></p>	<p>Cube0 <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p><b>Alter!</b></p>



These tests address Success Criteria 2, 4, 10.

## Success Criteria Review

Main window which is moveable and resizable. X

Menu screen. ✓

Lightweight and easy to navigate. (Will review during evaluation).

Multiple different options of different simulations. ✓

3D rasteriser. ✓

Shading. ✓

Working rigid body motion. ✓

Working rigid body collision. ✓

Normal reaction force. ✓

Options next to each simulation such as creating objects, applying forces to objects etc.,. ✓

Camera control. ✓

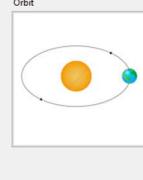
Out of the 11 success criteria, 9 have been definitely completed, and one will be judged in the next section. Notice that criteria 1 has been unchecked, this is because since adding the menus, the window is no longer resizable. This is a change which unfortunately had to be made, since with WinForms it is not very easy (as far as I am aware) to create a UI which scales with the window. The main point of this first criteria was to create a window in which things could be displayed, which we still have, it just can't be resized.

I have now completed 9/11 success criteria. The 11<sup>th</sup> will be judged in the next section.

# Evaluation

## Testing to inform evaluation

### Testing for Function

Success Criteria	Evidence
Menu Screen	   The screenshot shows a Windows application interface. At the top, there is a menu bar with "File", "Edit", "View", "Help", and "About". Below the menu is a toolbar with icons for "New", "Open", "Save", "Print", "Exit", and "Help". The main area contains three separate windows or tabs. The first tab, titled "Rigid Body Collisions", displays a small image of two cars colliding. The second tab, titled "Projectile Motion", displays a small image of a projectile's path. The third tab, titled "Orbit", displays a small image of a celestial body orbiting a larger one. Below these tabs is a gear icon. The bottom half of the screen shows the code for these functions in a C# editor. The code includes logic for checking if a collision has occurred and creating rigid bodies and tensors for projectile motion and orbits. It also includes settings for camera parameters like speed and sensitivity.

## Multiple different options of different simulations

Process: [1324] ComputerScienceAlevelProject Lifecycle Events - Thread: Stack Frame: Diagnostic

RigidBodyCollisions.cs RigidBody.cs Program.cs ComputerScienceAlevelProject ComputerScienceAlevelProject.RigidBodyCollisions Create\_Click(object sender, EventArgs e)

```

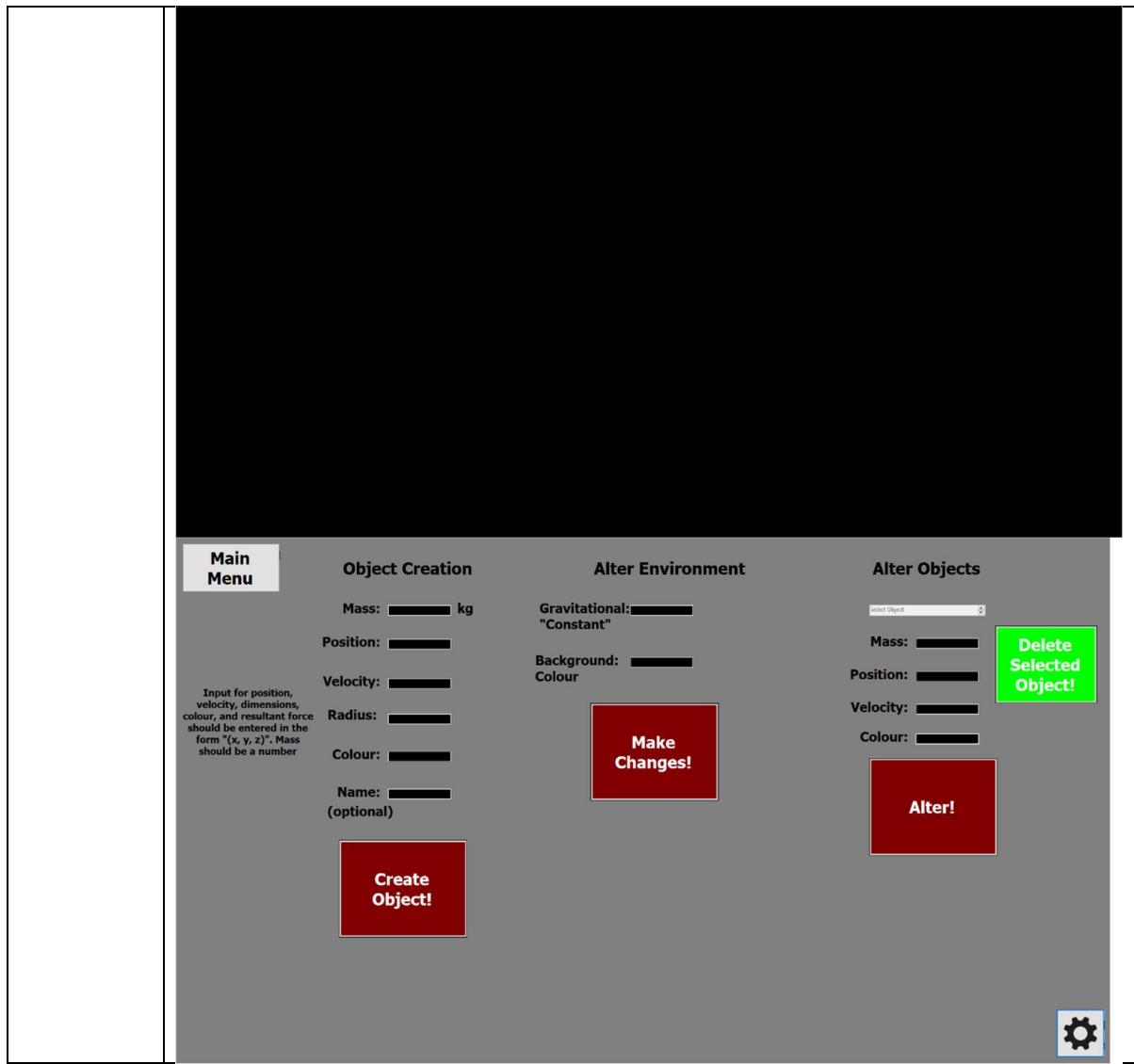
621
622
623     if (check == false)
624     {
625         if (check == true)
626         {
627             // Menu
628
629             Rigid Body Collisions
630             Projectile Motion
631             Orbit
632
633             Rigid Body Collisions
634             Projectile Motion
635             Orbit
636
637             Rigid Body Collisions
638             Projectile Motion
639             Orbit
640             Rigid Body Collisions
641             Projectile Motion
642             Orbit
643             Rigid Body Collisions
644             Projectile Motion
645             Orbit
646
647             Mesh mesh = new Mesh(vertices, dimensions, colour);
648
649             //Create Tensor and RigidBody
650             Vector v0 = new Vector(0, 0, 0); //zero vector
651             Vector tensor = Global.vectorTimesScalar(new Vector(1 / (height * height + width * width), 1 / (depth * depth + width * width)));
652             RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, force, cor, tensor, false);
653
654             //Create name:
655
656             Rigid Body Collisions
657             Projectile Motion
658             Orbit
659
660             Rigid Body Collisions
661             Projectile Motion
662             Orbit
663
664             Rigid Body Collisions
665             Projectile Motion
666             Orbit

```

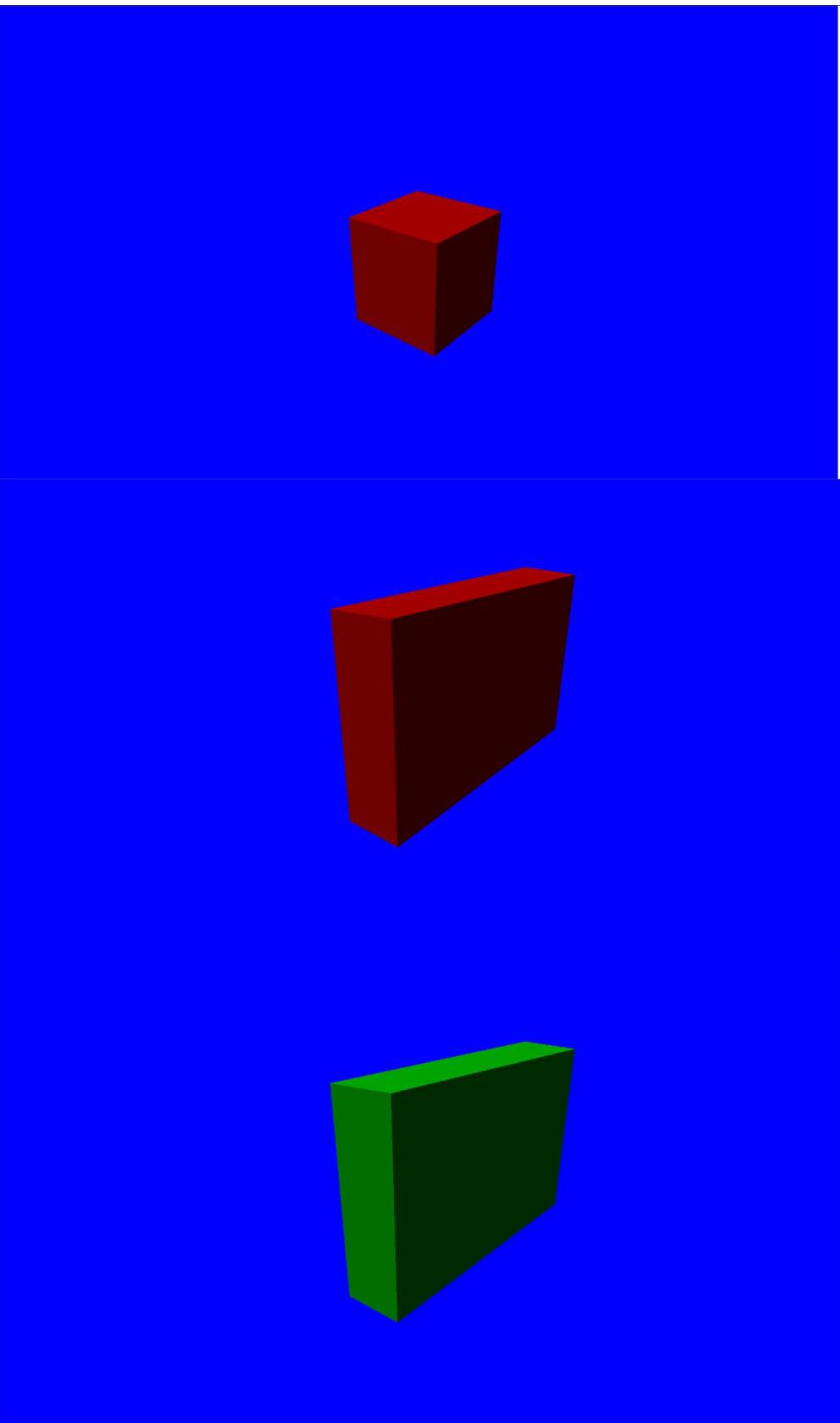
Search (Ctrl+E) Error List ... Entire Solution 0 Errors 12 Warnings 0 Messages 7

Main Menu	Object Creation	Alter Environment	Alter Objects
	Mass: <input type="text"/> kg Coefficient of Restitution: <input type="text"/> <small>Input for position, velocity, dimensions, colour and resultant force should be entered in the form "(x, y, z)". Mass should be a number</small> Position: <input type="text"/> Velocity: <input type="text"/> Dimensions: <input type="text"/> Colour: <input type="text"/> Resultant Force: <input type="text"/> <small>(Excluding Gravity)</small> Name: <input type="text"/> <small>(optional)</small>	Strength of: <input type="text"/> Gravity: <input type="checkbox"/> Ground Enabled? Background: <input type="text"/> Colour: <input type="text"/>	Select Object: <input type="text"/> Mass: <input type="text"/> Coefficient of Restitution: <input type="text"/> Position: <input type="text"/> Velocity: <input type="text"/> Angular Velocity: <input type="text"/> Colour: <input type="text"/> Resultant Force: <input type="text"/> <small>(Excluding Gravity)</small>
	<input style="background-color: red; color: white; padding: 5px; border: none;" type="button" value="Create Object!"/>	<input style="background-color: red; color: white; padding: 5px; border: none;" type="button" value="Make Changes!"/>	<input style="background-color: green; color: white; padding: 5px; border: none;" type="button" value="Delete Selected Object!"/> <input style="background-color: red; color: white; padding: 5px; border: none;" type="button" value="Alter!"/>

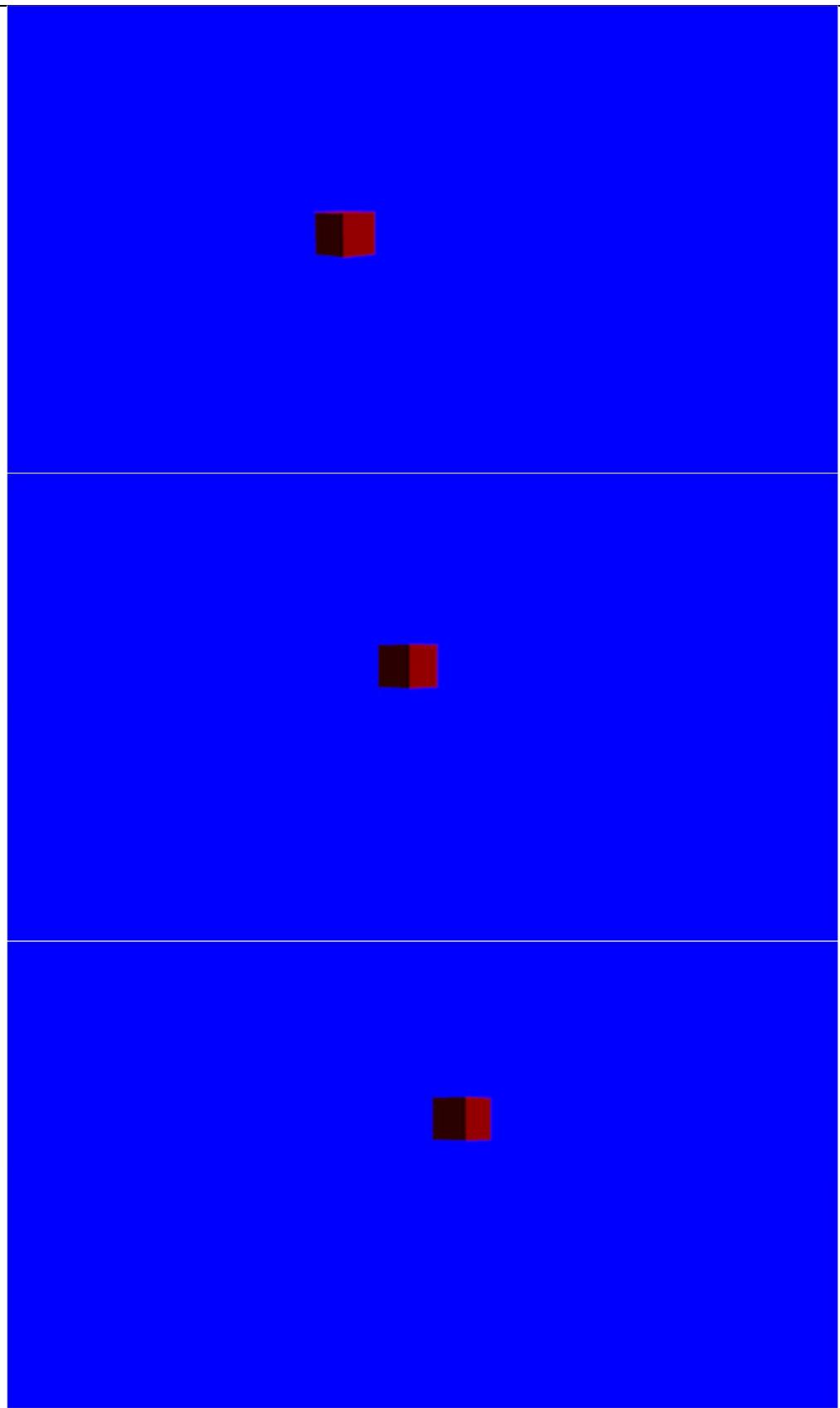
<div style="background-color: #ccc; padding: 10px;"> <div style="display: flex; justify-content: space-around;"> <div style="width: 30%;"> <p><b>Main Menu</b></p> <p>Mass: <input type="text"/> kg</p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Starting Height: <input type="text"/></p> <p>Initial Speed: <input type="text"/></p> <p>Initial Angle: <input type="text"/></p> <p>Colour: <input type="color"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> <p>Name: <input type="text"/> (optional)</p> </div> <div style="width: 30%;"> <p><b>Object Creation</b></p> </div> <div style="width: 30%;"> <p><b>Alter Environment</b></p> <p>Strength of Gravity: <input type="text"/></p> <p>Background Colour: <input type="color"/></p> </div> <div style="width: 30%;"> <p><b>Alter Objects</b></p> <p>Select Object: <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Colour: <input type="color"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> </div> </div> <div style="text-align: center; margin-top: 10px;"> <input style="background-color: red; color: white; padding: 5px; border: none;" type="button" value="Create Object!"/> </div> <div style="text-align: center; margin-top: 10px;"> <input style="background-color: red; color: white; padding: 5px; border: none;" type="button" value="Make Changes!"/> </div> <div style="text-align: center; margin-top: 10px;"> <input style="background-color: red; color: white; padding: 5px; border: none;" type="button" value="Alter!"/> </div> <div style="text-align: right; margin-top: 10px;"> </div> </div>	

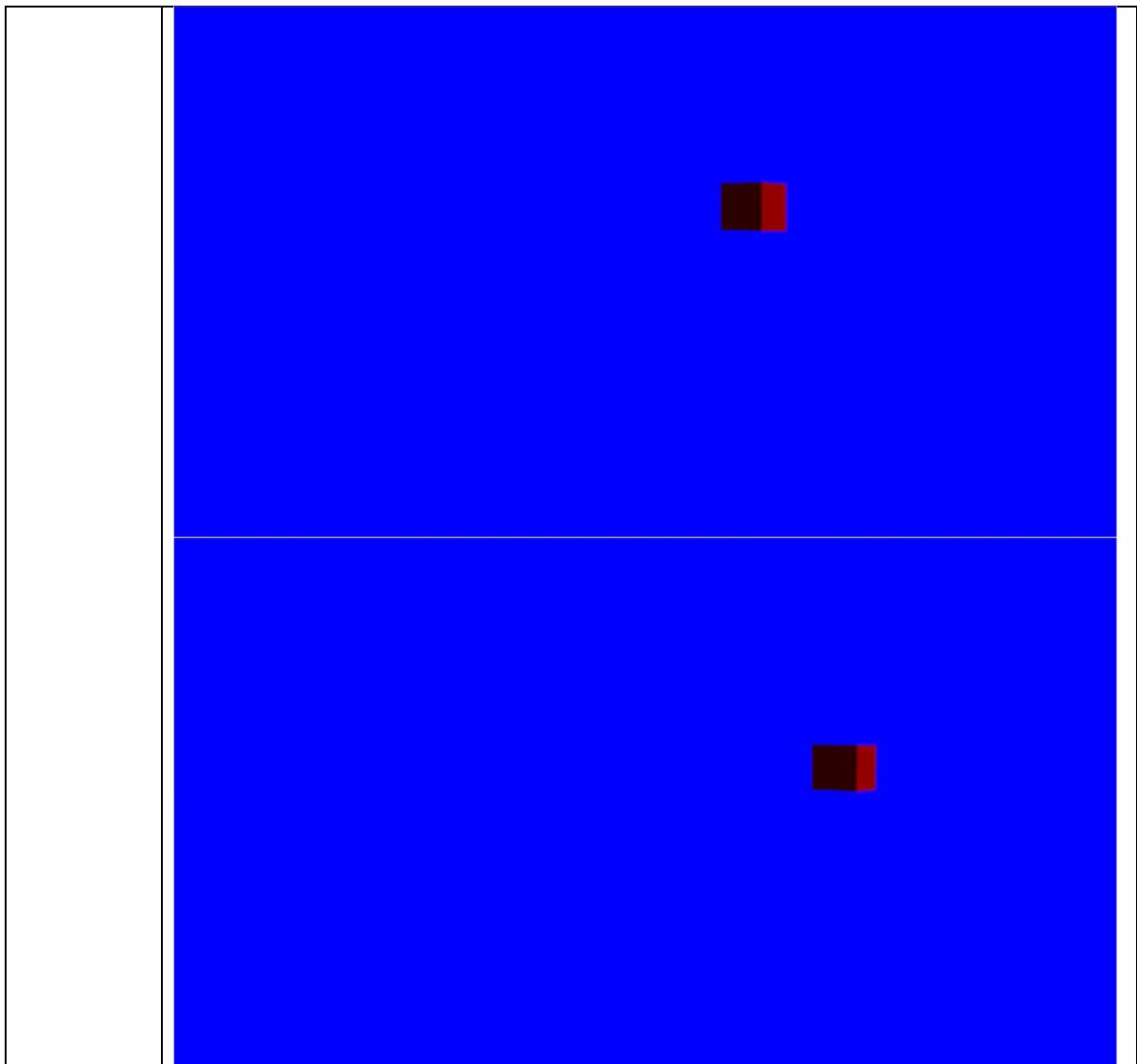


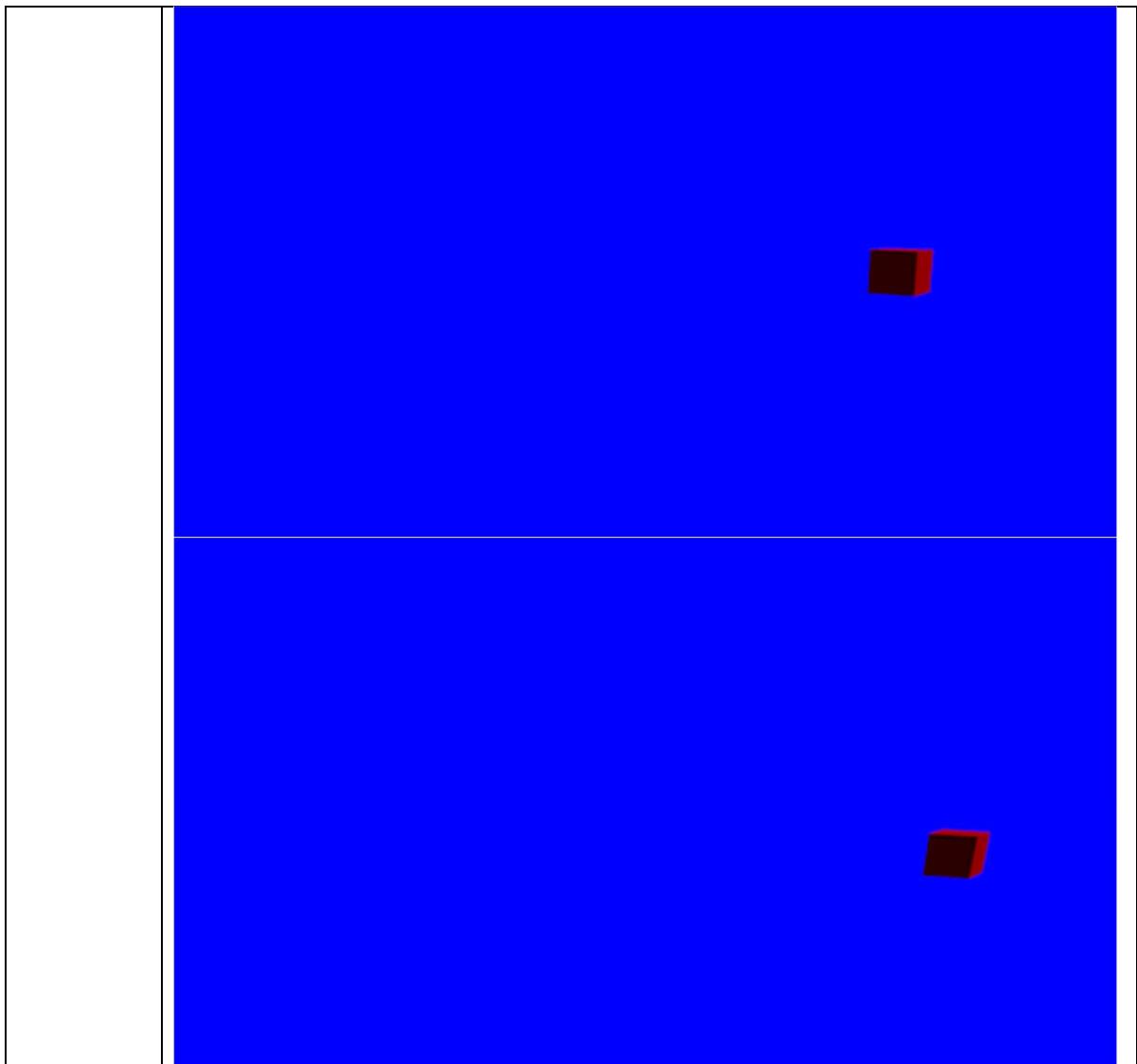
3D  
rasteriser  
& shading



Working  
rigid body  
motion

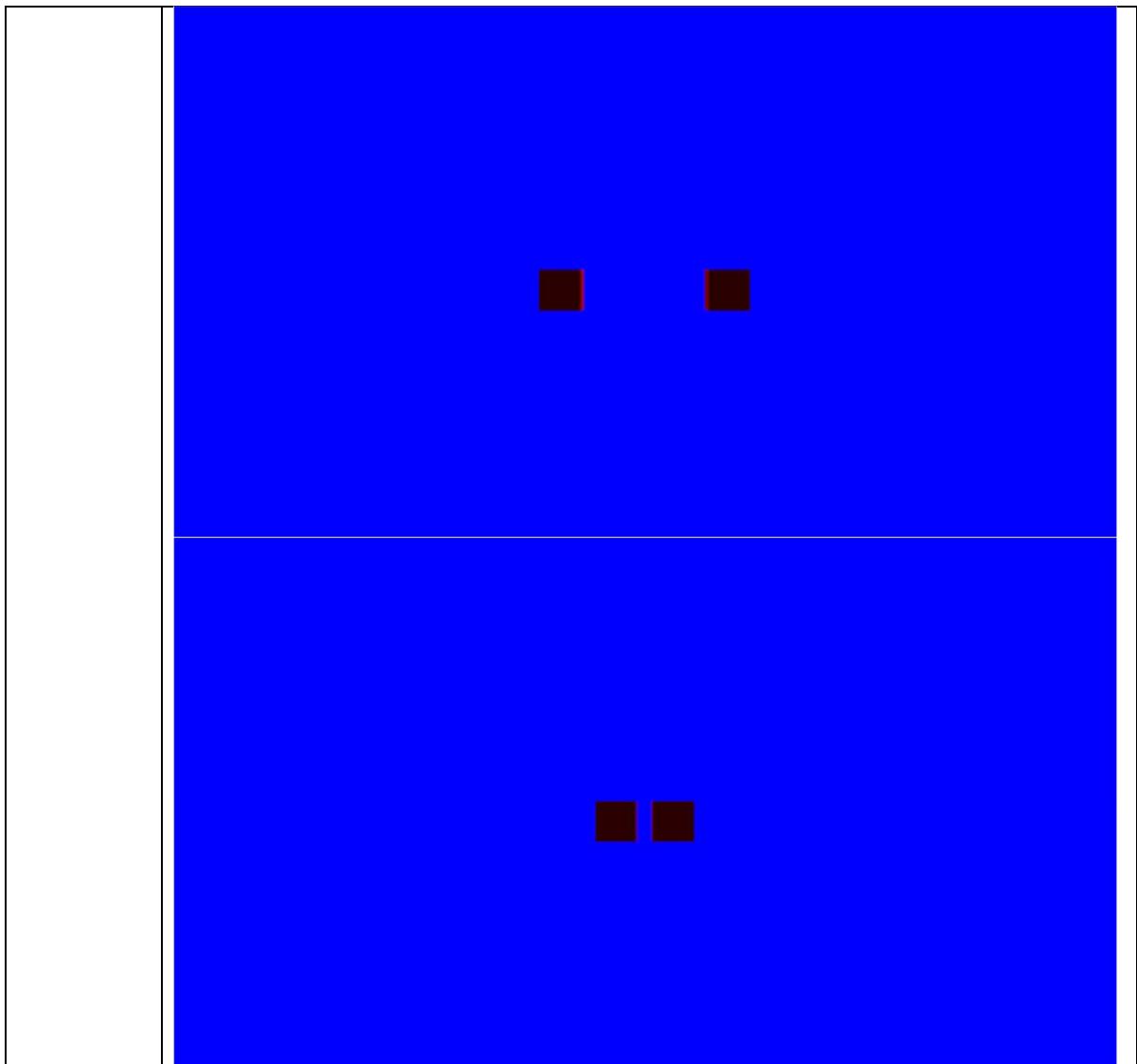


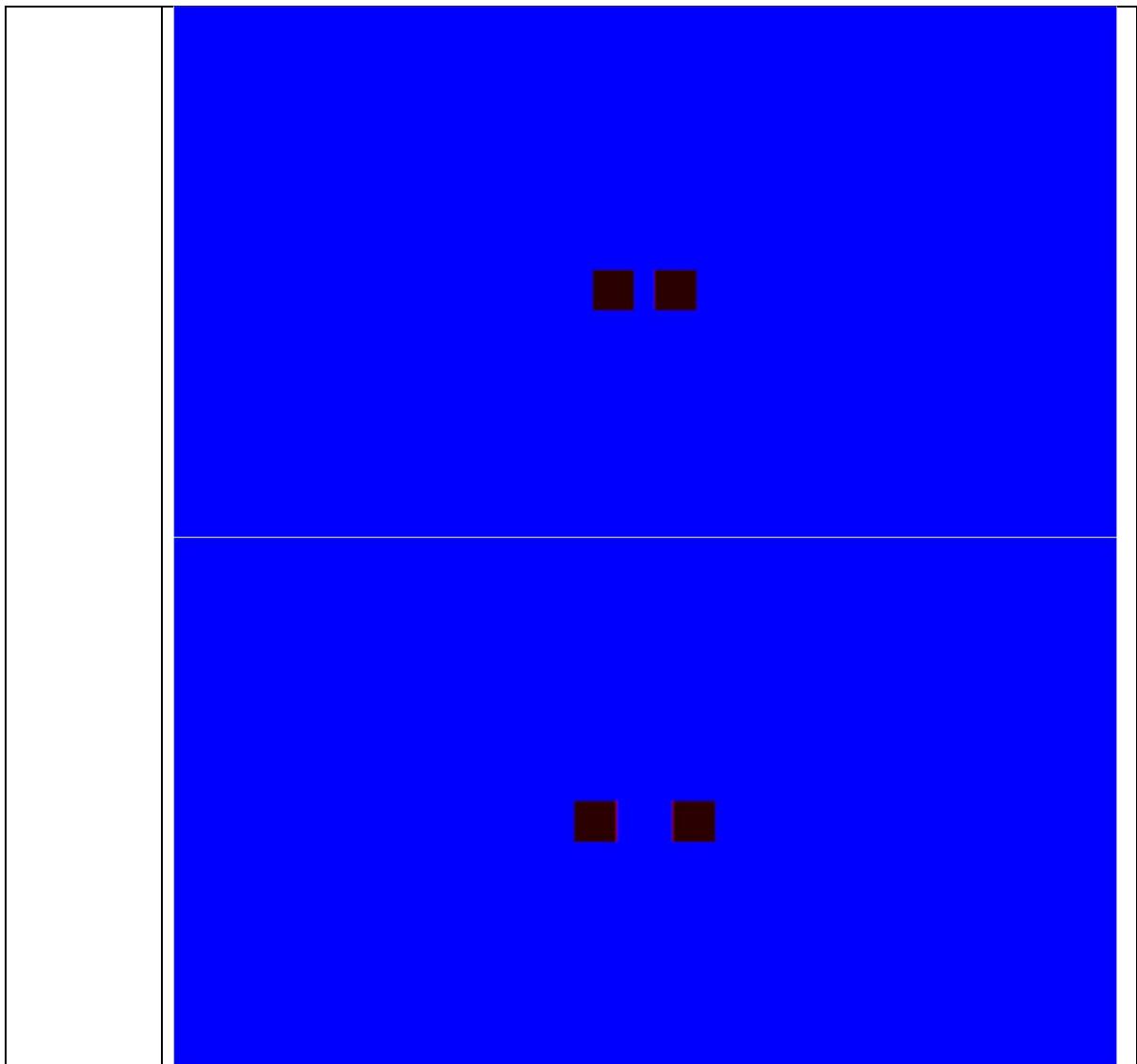


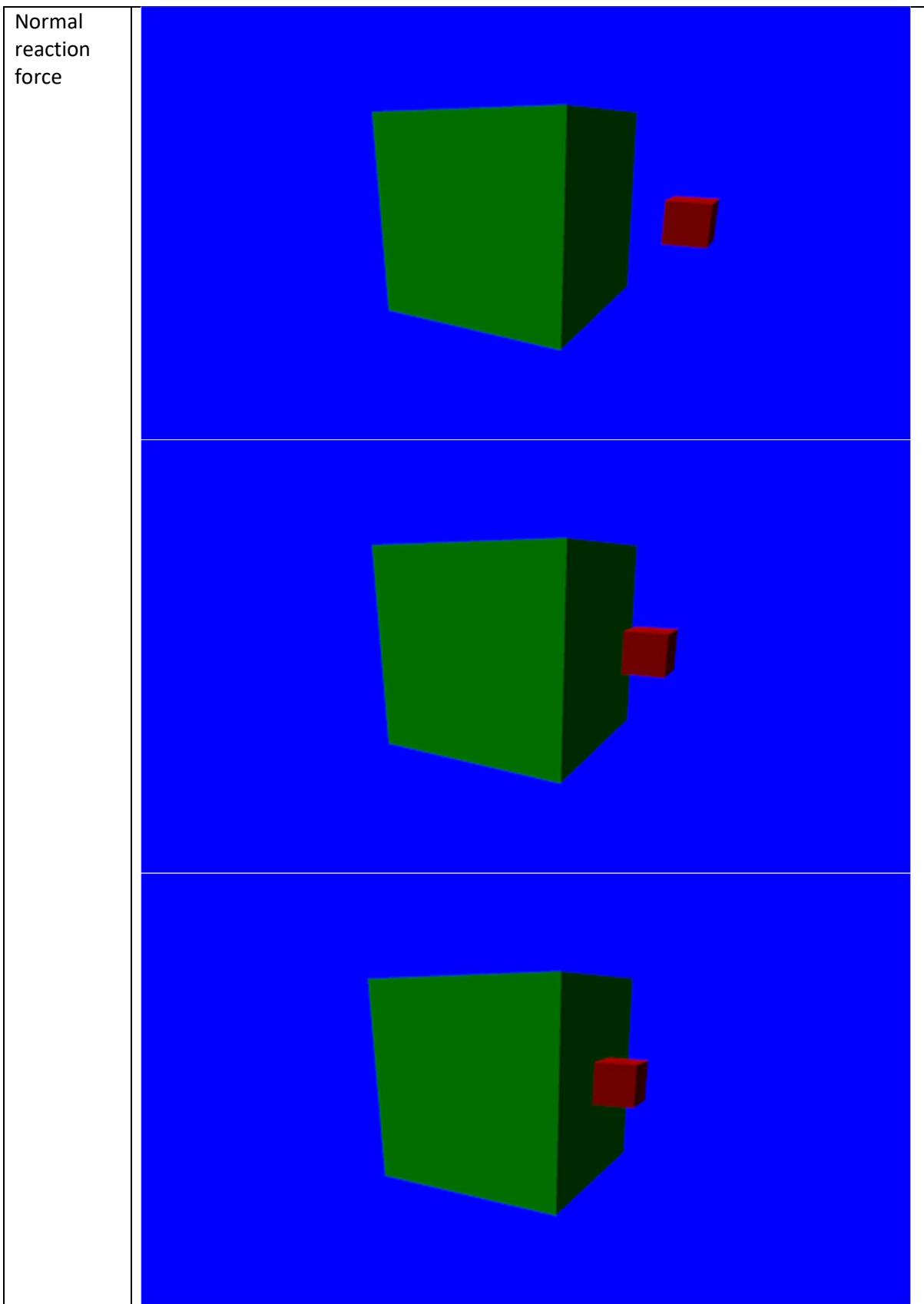


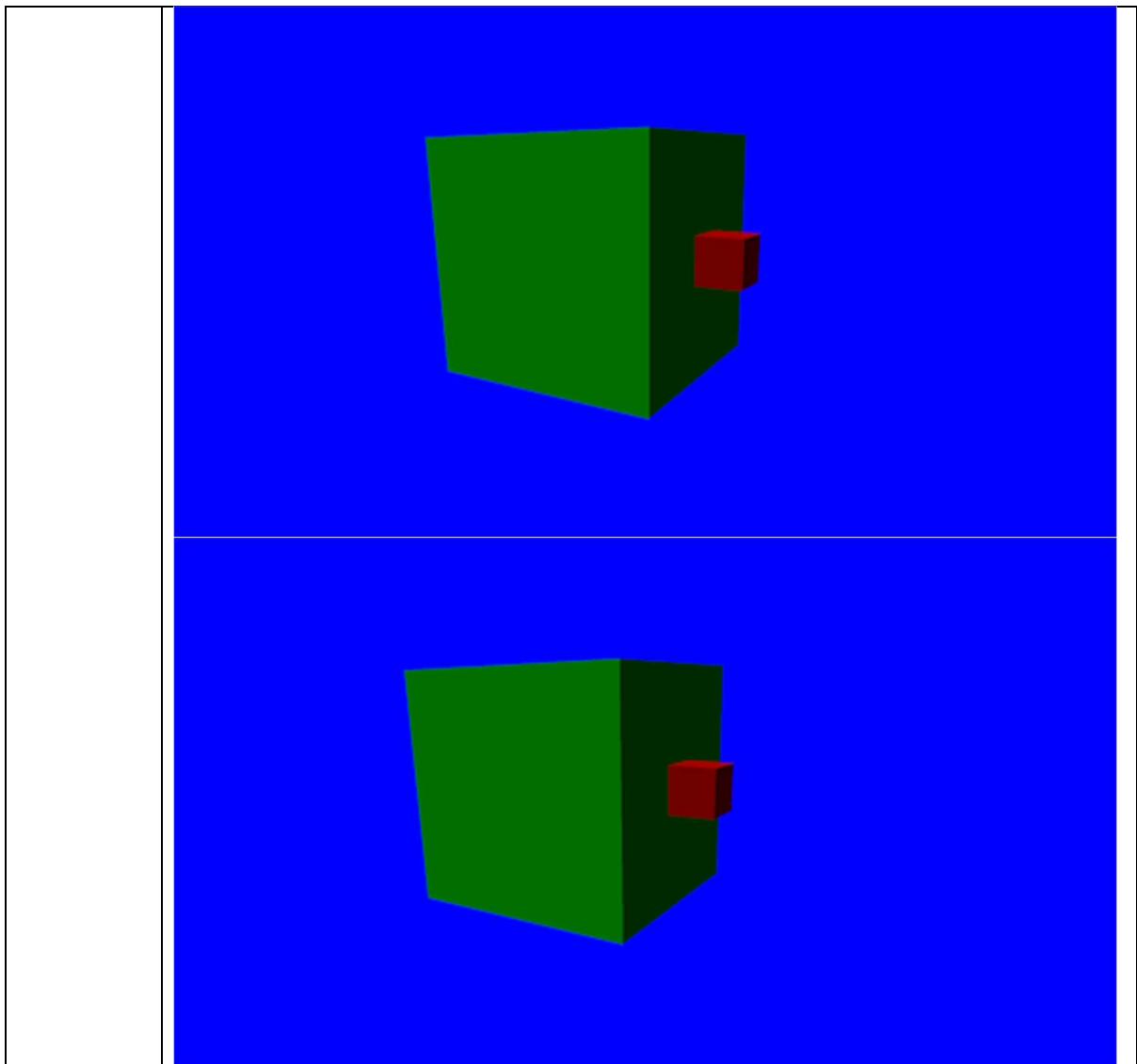
Working  
rigid body  
collisions

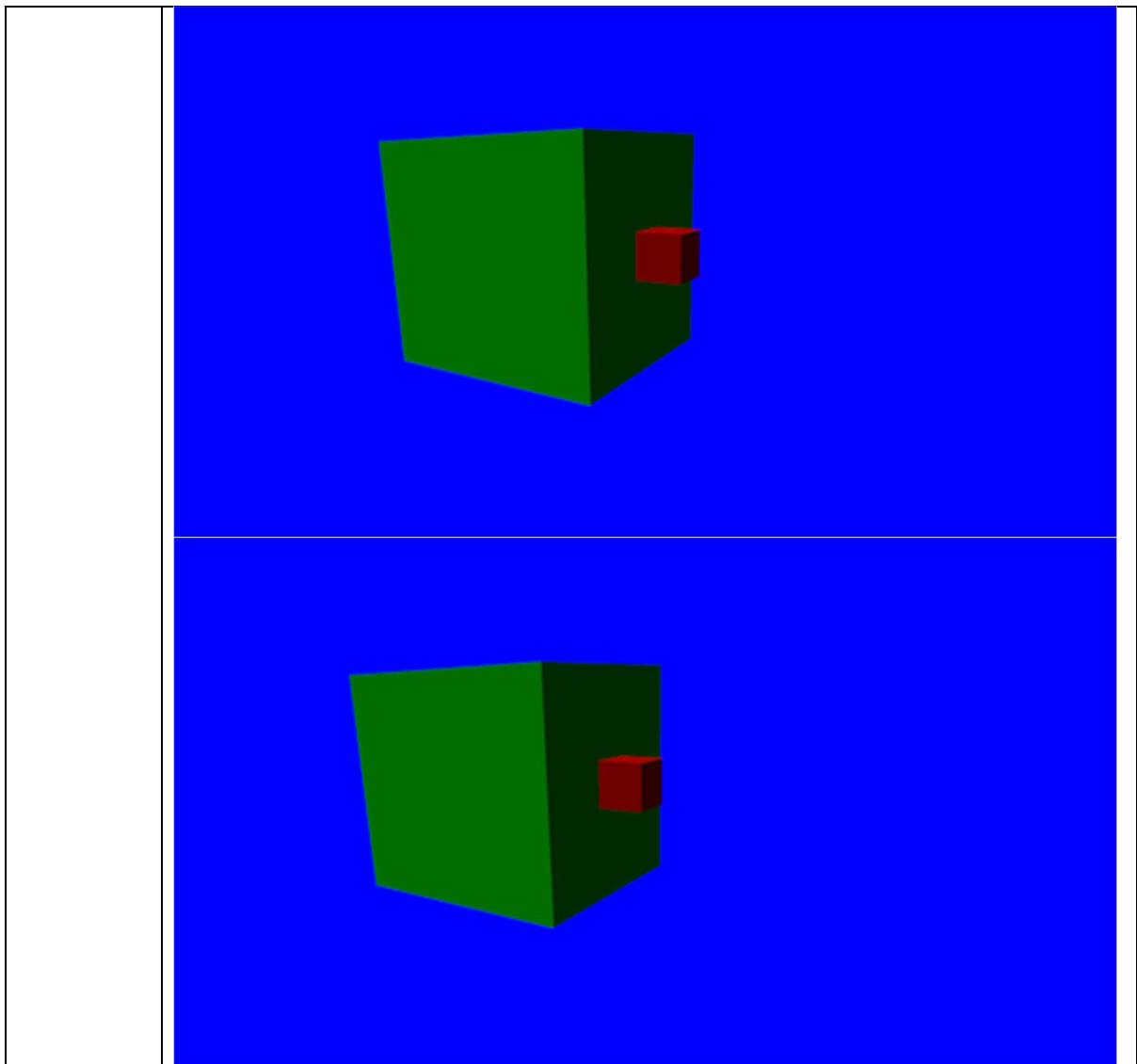








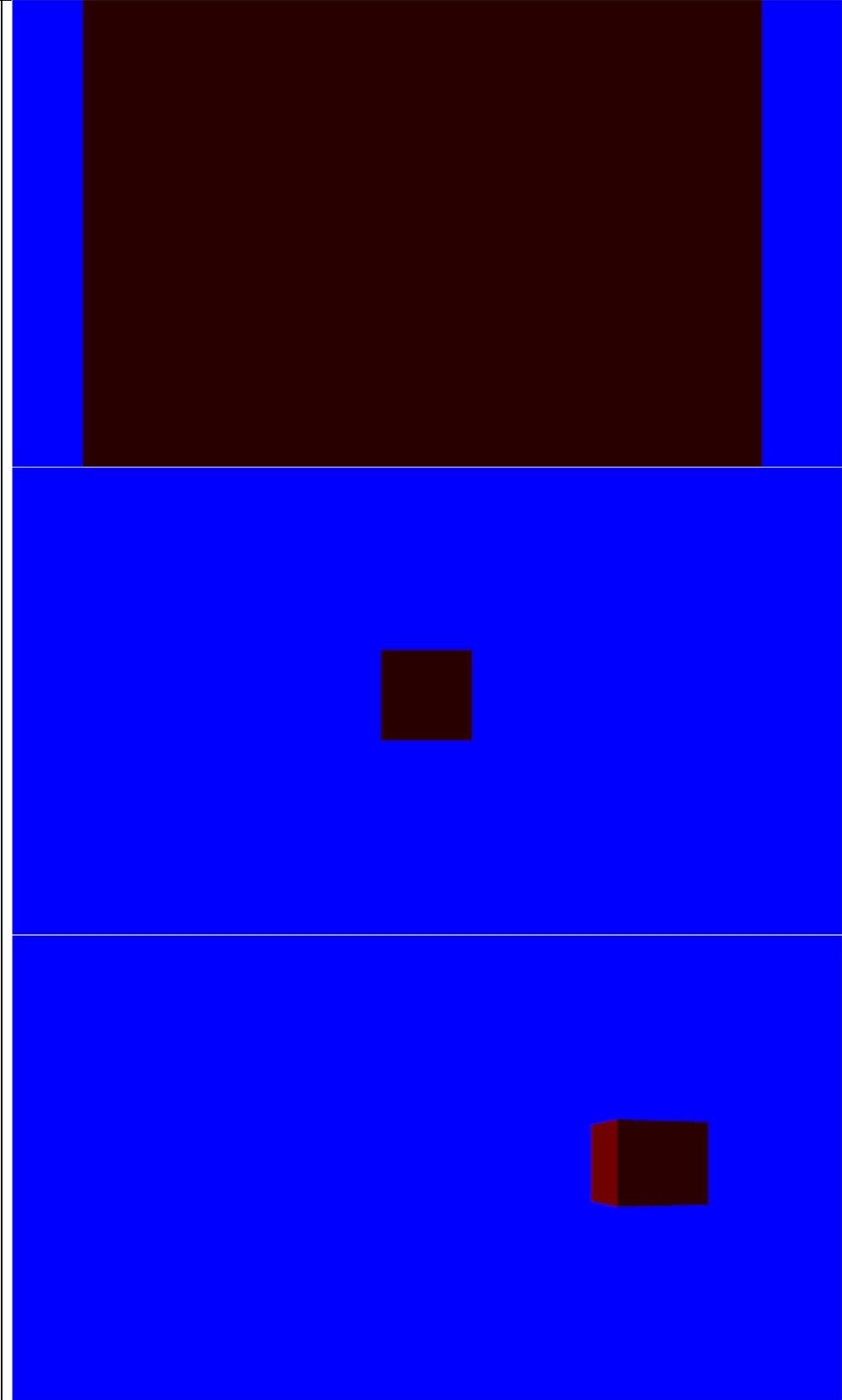


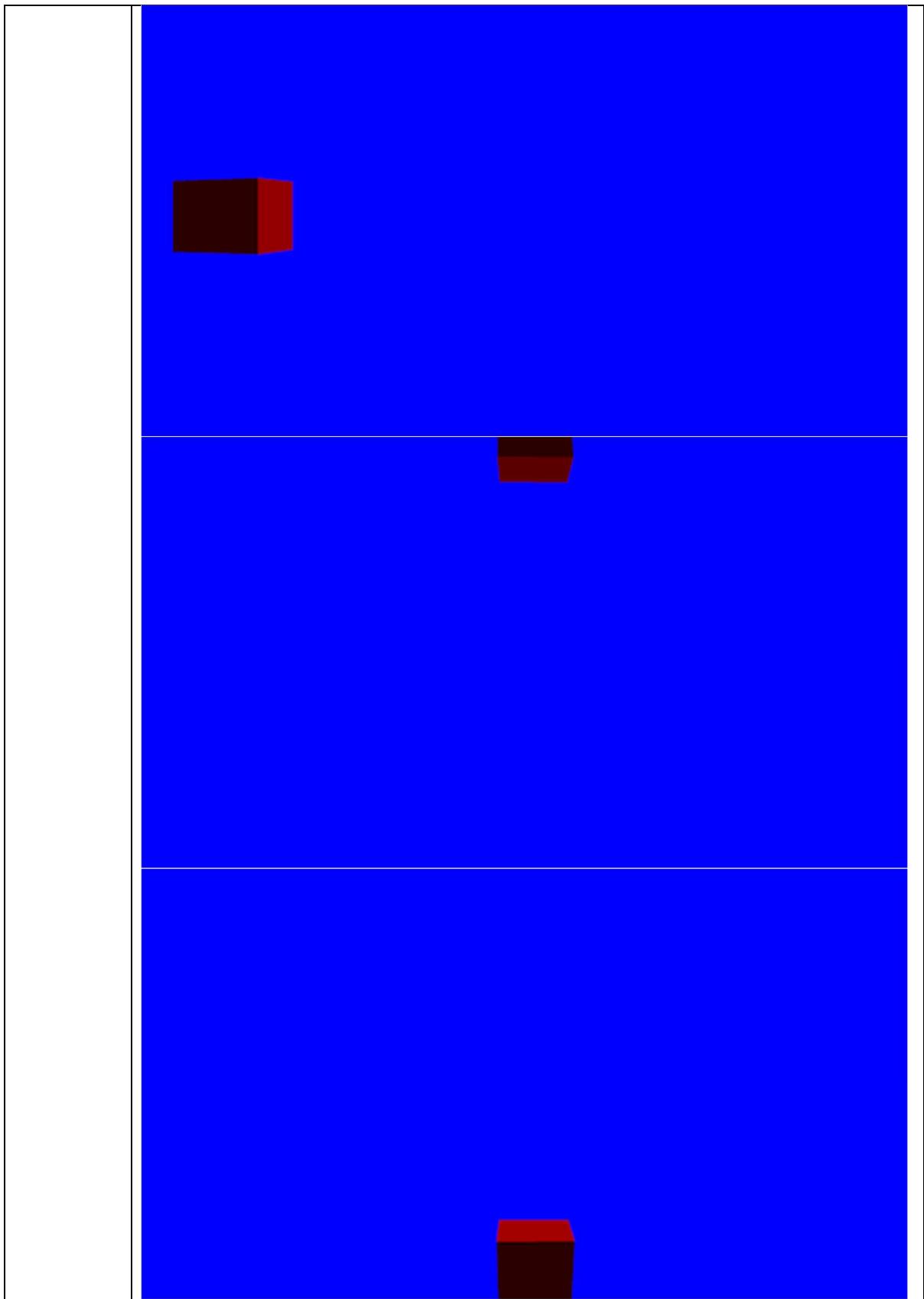


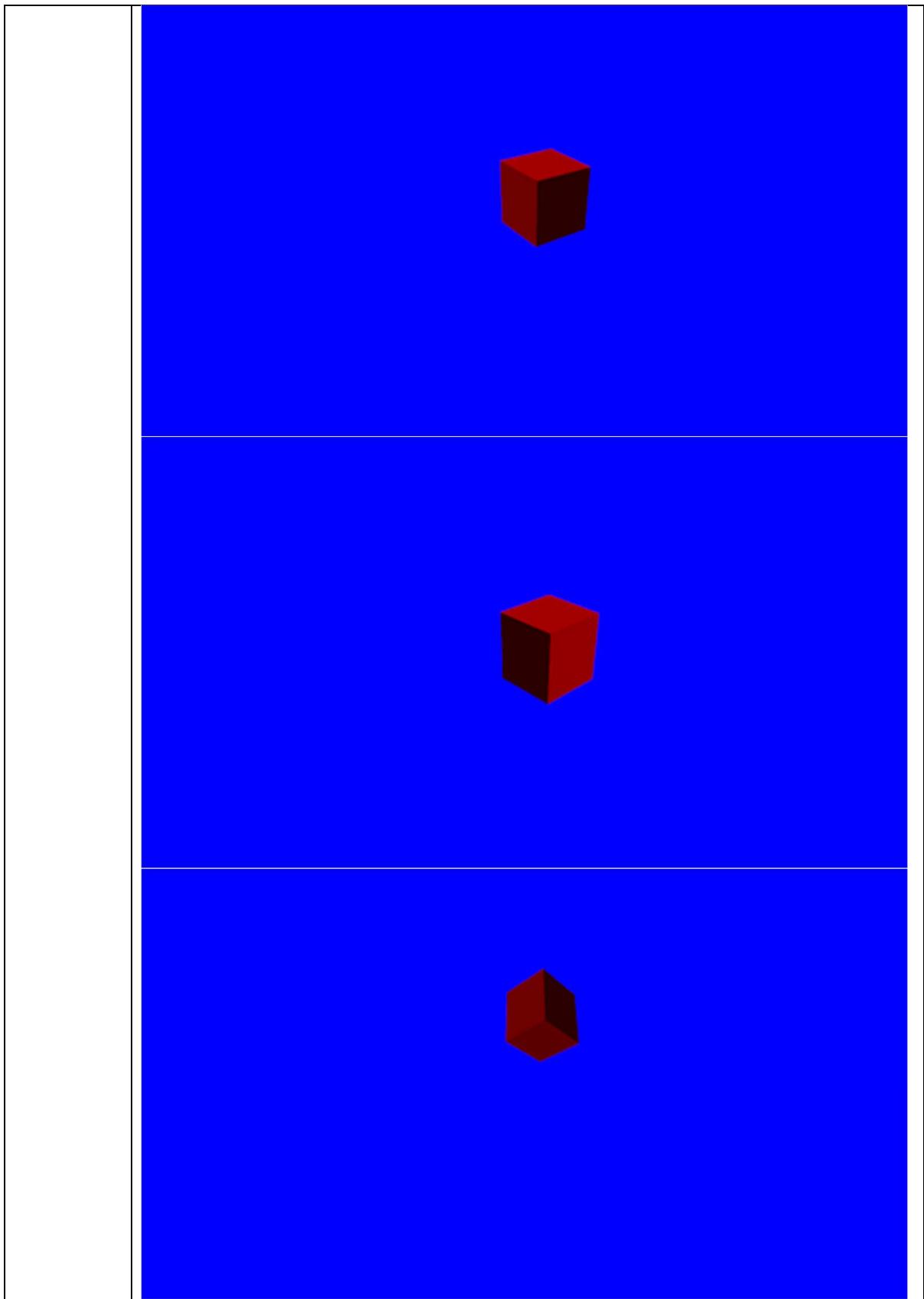
Options  
next to  
each  
simulation

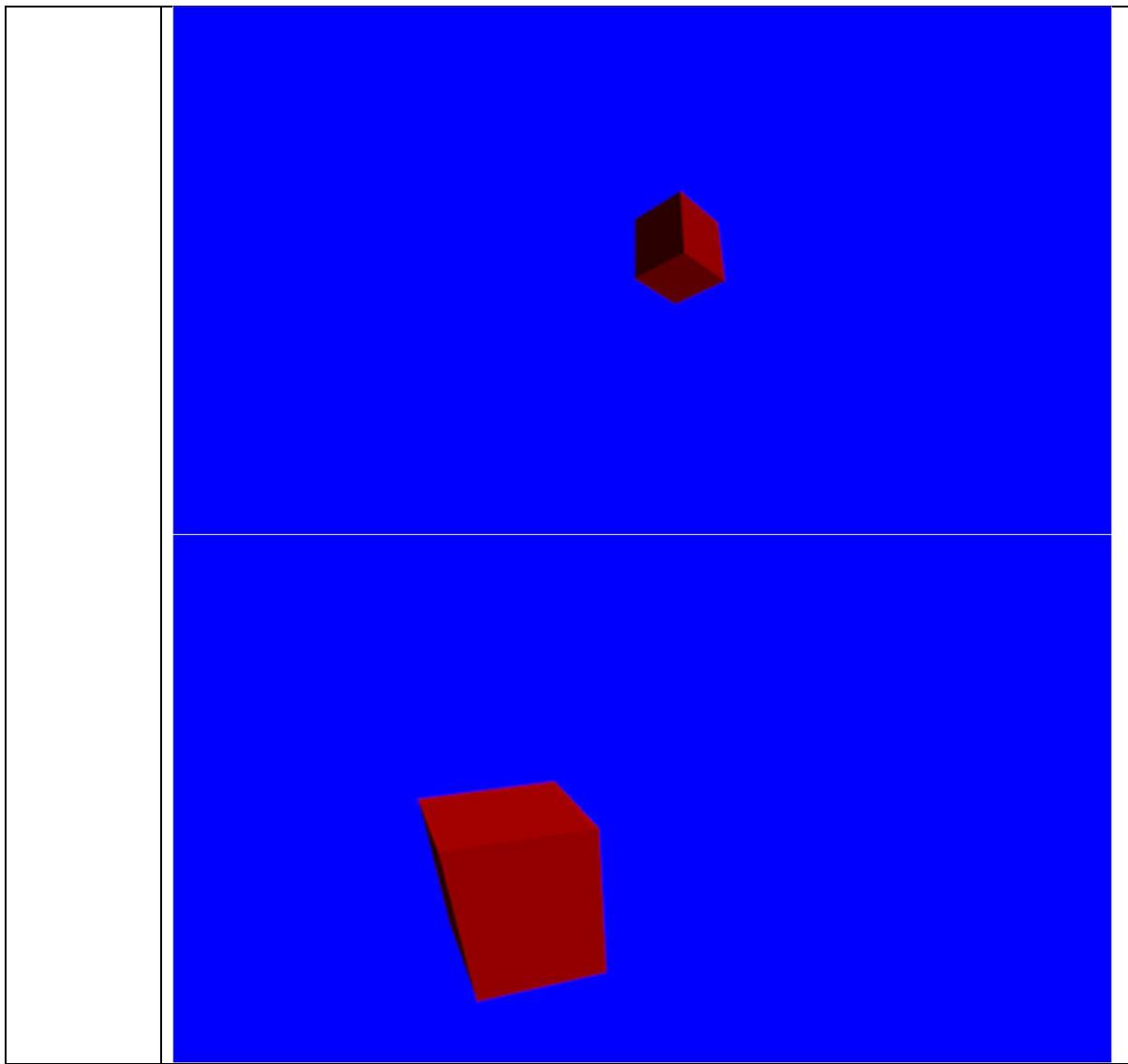
Main Menu	Object Creation	Alter Environment	Alter Objects
	<p>Mass: <input type="text"/> kg</p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Dimensions: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> <p>Name: <input type="text"/> (optional)</p>	<p>Strength of Gravity: <input type="text"/></p> <p>Background Colour: <input type="text"/></p>	<p>Ground Enabled? <input checked="" type="checkbox"/></p> <p>Select Object: <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Angular Velocity: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p>
	<b>Create Object!</b>	<b>Make Changes!</b>	<b>Delete Selected Object!</b> <b>Alter!</b>
			
Main Menu	Object Creation	Alter Environment	Alter Objects
	<p>Mass: <input type="text"/> kg</p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Starting Height: <input type="text"/></p> <p>Initial Speed: <input type="text"/></p> <p>Initial Angle: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p> <p>Name: <input type="text"/> (optional)</p>	<p>Strength of Gravity: <input type="text"/></p> <p>Background Colour: <input type="text"/></p>	<p>Select Object: <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Coefficient of Restitution: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Resultant Force: <input type="text"/> (Excluding Gravity)</p>
	<b>Create Object!</b>	<b>Make Changes!</b>	<b>Alter!</b>
			
Main Menu	Object Creation	Alter Environment	Alter Objects
	<p>Mass: <input type="text"/> kg</p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Radius: <input type="text"/></p> <p>Colour: <input type="text"/></p> <p>Name: <input type="text"/> (optional)</p>	<p>Gravitational: "Constant": <input type="text"/></p> <p>Background Colour: <input type="text"/></p>	<p>Select Object: <input type="text"/></p> <p>Mass: <input type="text"/></p> <p>Position: <input type="text"/></p> <p>Velocity: <input type="text"/></p> <p>Colour: <input type="text"/></p>
	<b>Create Object!</b>	<b>Make Changes!</b>	<b>Alter!</b>
			

Camera  
Control









The above screenshots demonstrate successful tests of the success criteria. (See test plans for each prototype for more information on each test).

I will next test my program for robustness. Will it easily break under pressure (e.g., if I have many objects in existence at the same time).

This test will entail placing 30 objects at the origin and seeing if the program is able to handle them all. I will then move a smaller object towards them to trigger a collision, what we see should be a chaotic explosion from the objects all moving apart rapidly. It is also possible that the program crashes under the stress or some unexpected result occurs. If the test is successful then the objects should all explode apart when something collides with them.

Test screenshots:

Menu where the objects are created:

Main Menu

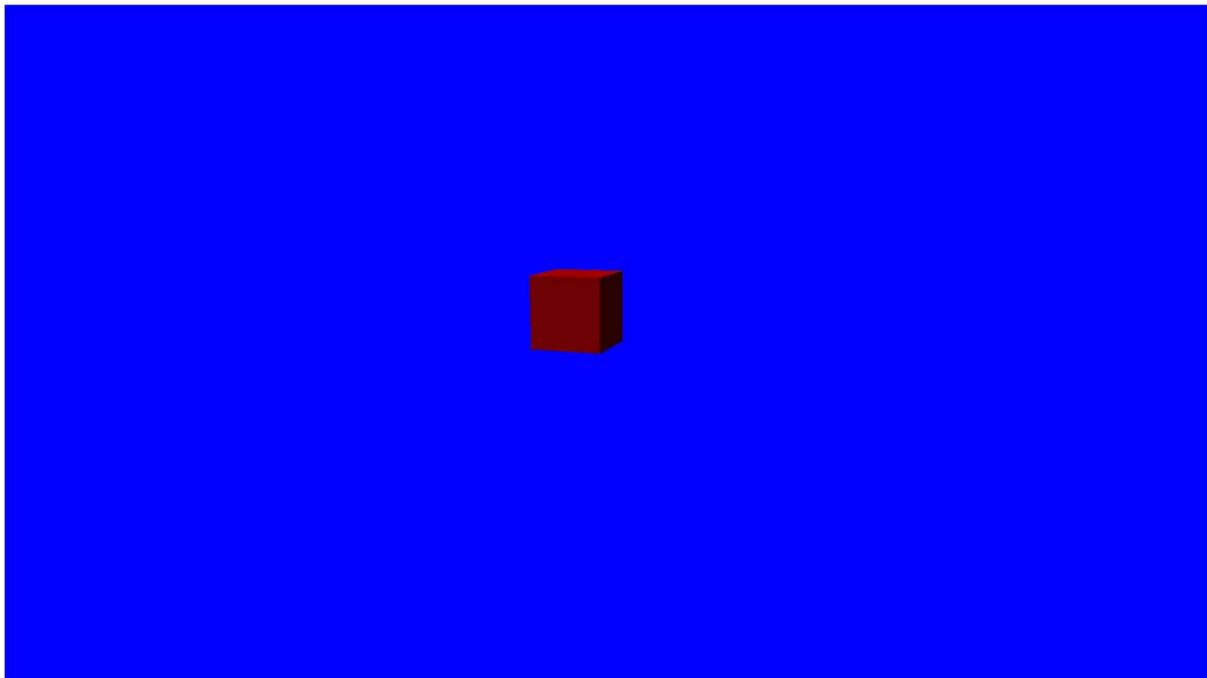
Object Creation		Alter Environment		Alter Objects	
Mass:	<input type="text" value="1"/> kg	Strength of Gravity:	<input type="text" value="0"/>	<input type="checkbox"/> Ground Enabled?	<input type="text" value="Cube0"/>
Coefficient of Restitution:	<input type="text" value="1"/>	Background Colour:	<input type="text"/>	Mass:	<input type="text"/>
Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number	Position: <input type="text" value="0.1 0.1 0.1"/>	Velocity: <input type="text" value="0 -1.0"/>	Dimensions: <input type="text" value="0.1 0.1 0.1"/>	Position: <input type="text"/>	Velocity: <input type="text"/>
Name: <input type="text" value="Bullet"/> (optional)	Colour: <input type="text" value="0,0,0"/>	Resultant Force: <input type="text" value="0,0"/> (Excluding Gravity)	Make Changes!	Angular Velocity: <input type="text"/>	Colour: <input type="text"/>
<b>Create Object!</b>		<b>Delete Selected Object!</b>			

Resultant Force:   
(Excluding Gravity)

Alter!

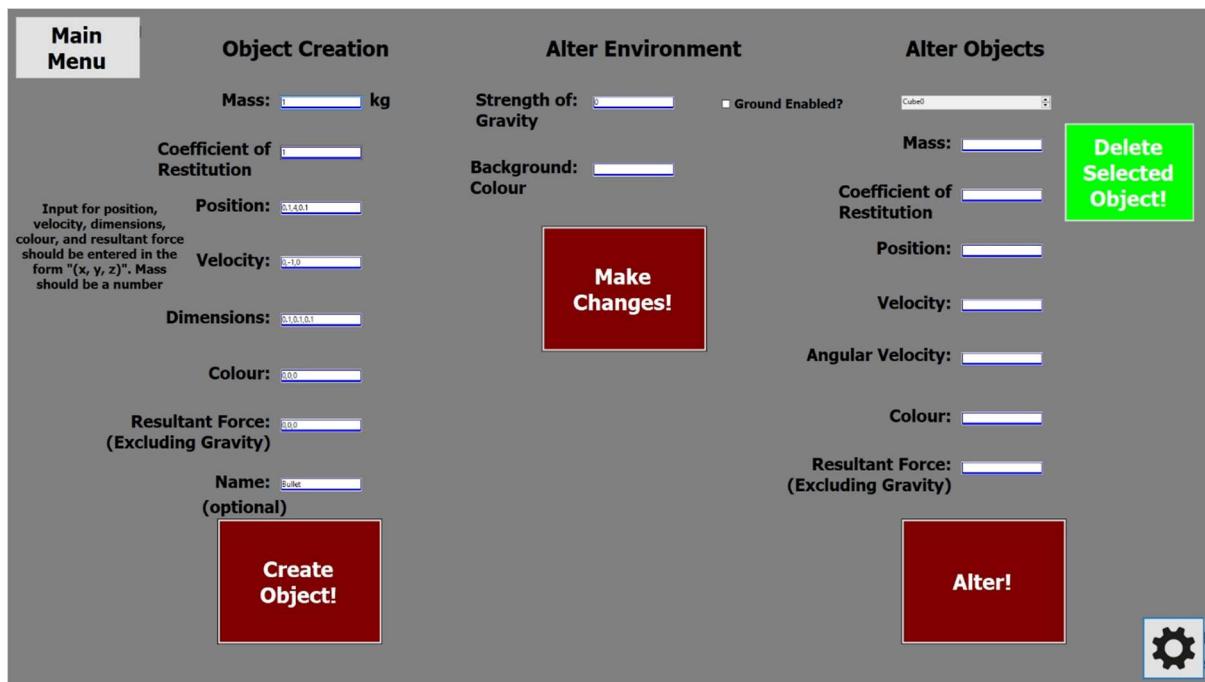
Settings icon

Objects all in one place:

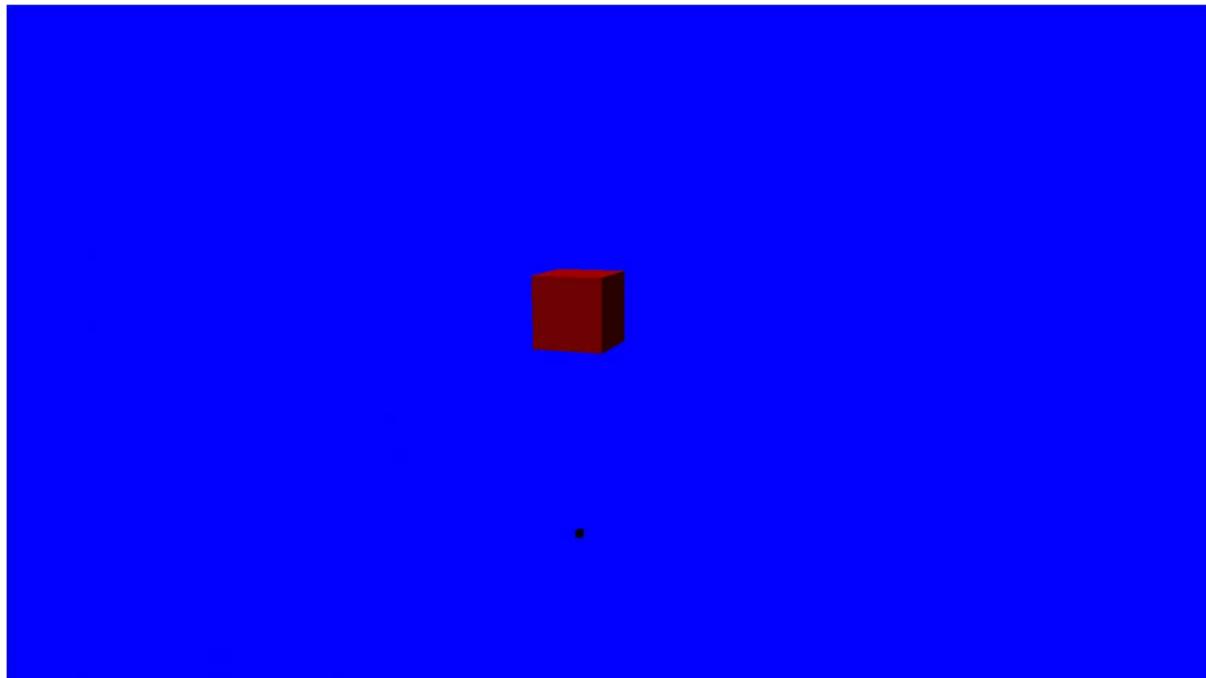


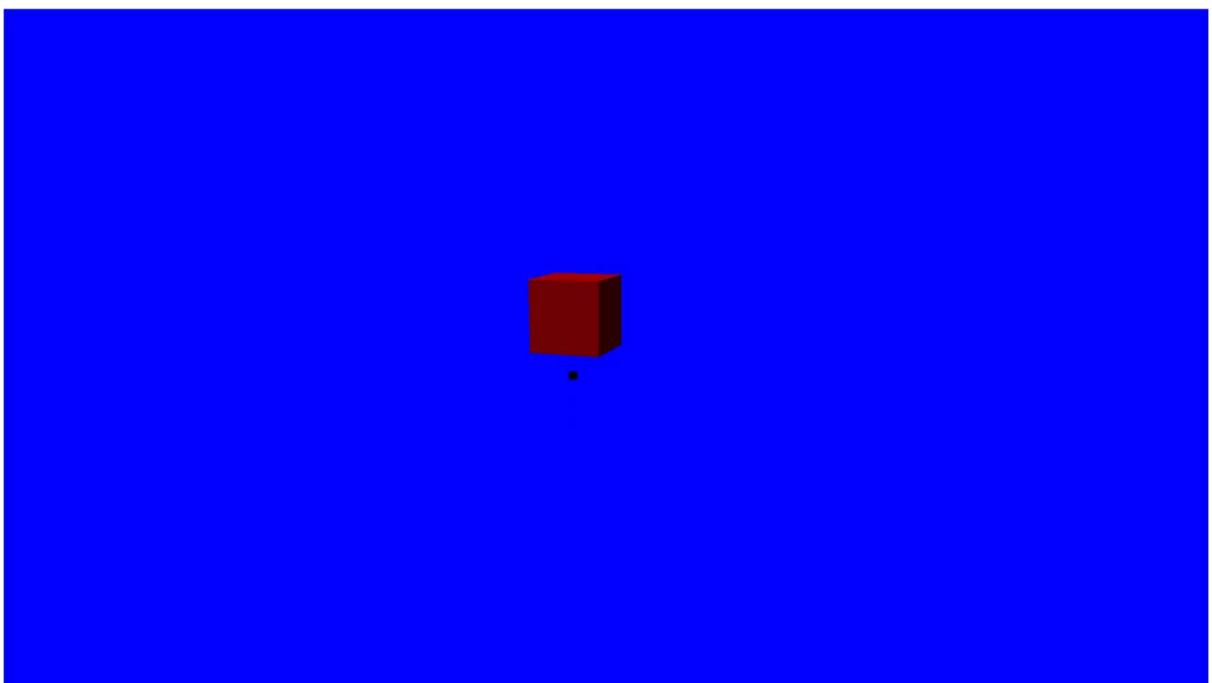
the framerate was very low here.

Menu creating the bullet (small object to move into the others):

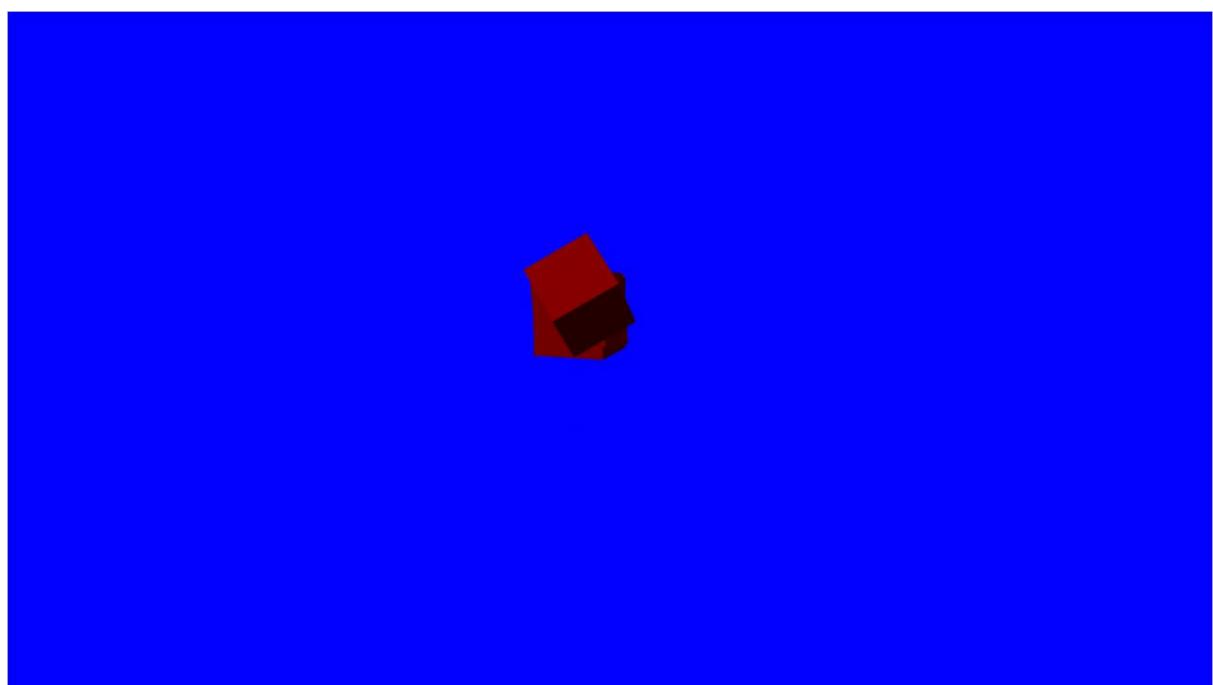


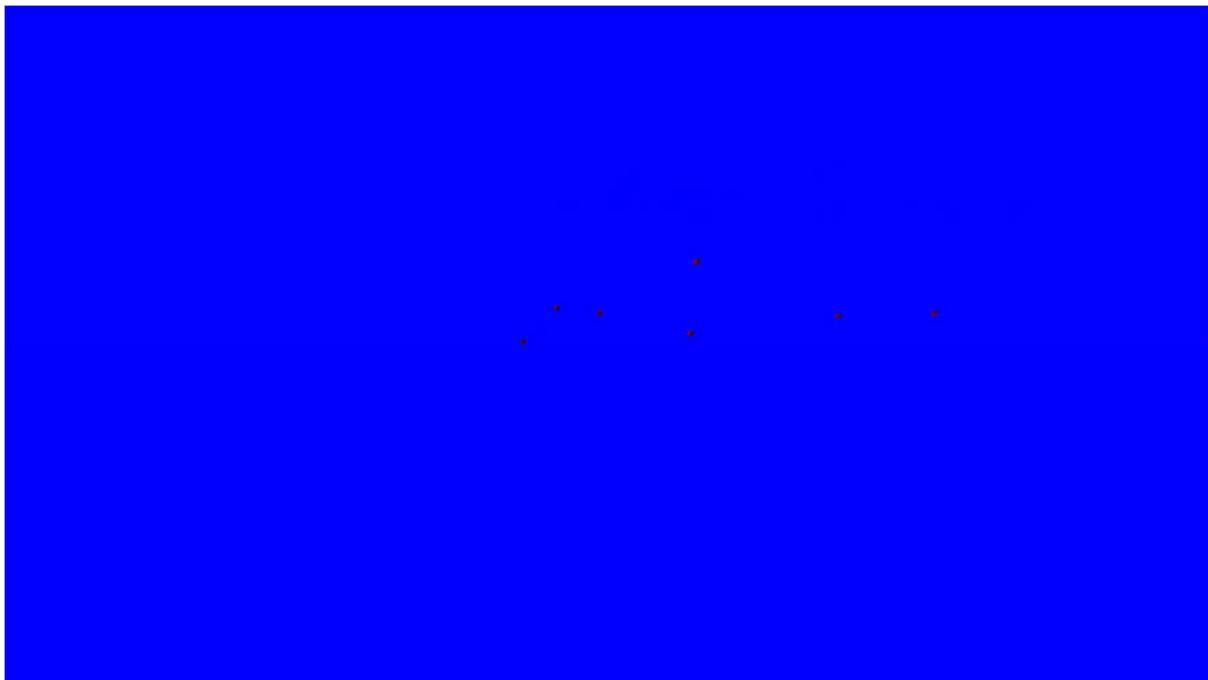
Bullet moving towards objects:





Objects exploding apart:





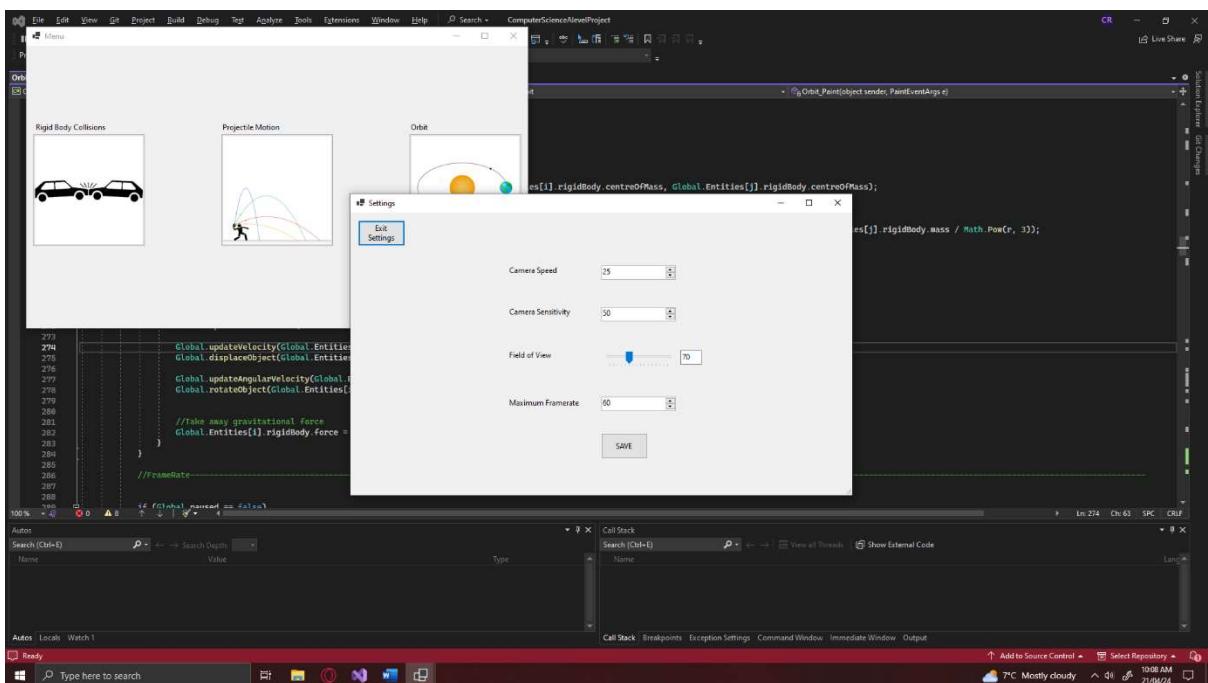
The camera was moved back for the final image.

The performance went back to normal after the objects moved apart, showing that my optimisation algorithms work as intended, the performance only suffers when the objects are close together and the more detailed collision detection is running.

## Testing for Usability

I allowed some physics students (one of my stakeholder groups) to use my program to comment on its usability. This will allow me to judge success criteria 3: “Lightweight and easy to navigate”.

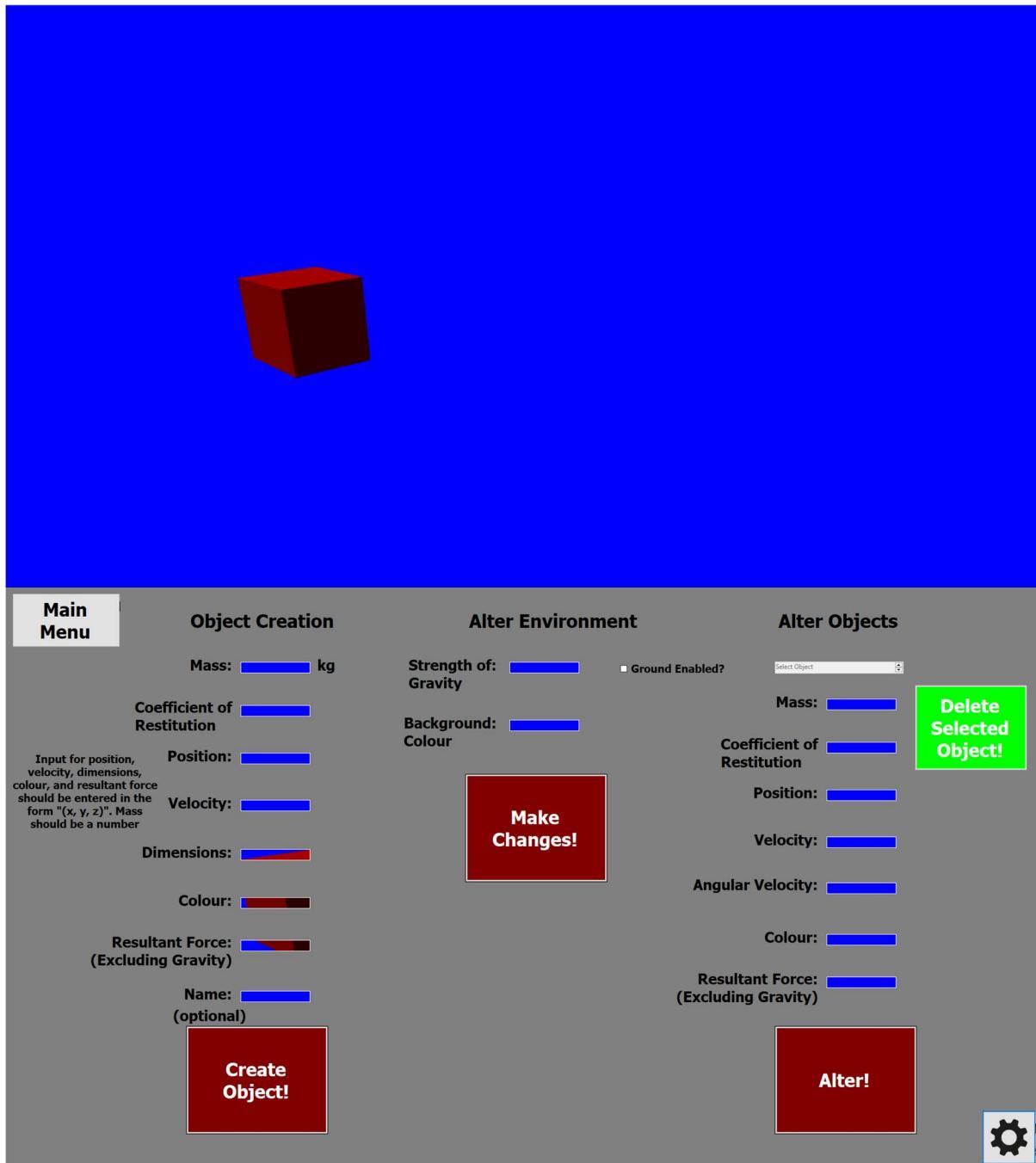
They said that the settings menu worked well.



and that it would have been better if the simulation menus were implemented in a similar way, as a dialogue box as opposed to replacing the existing window.

The menus worked and were easy to use, but it was annoying that each time the menu is opened the text boxes would all have the background image of the simulation covering the textbox until it is interacted with. (This is due to an error with WinForms, there is nothing I can do about it.)

What this looks like:



Main Menu	Object Creation	Alter Environment	Alter Objects
	Mass: <input type="text"/> kg Coefficient of Restitution: <input type="text"/> <small>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number</small> Position: <input type="text"/> Velocity: <input type="text"/> Dimensions: <input type="text"/> Colour: <input type="text"/> Resultant Force: <input type="text"/> (Excluding Gravity) Name: <input type="text"/> (optional)	Strength of Gravity: <input type="text"/> <input checked="" type="checkbox"/> Ground Enabled? Background: <input type="text"/> Colour	Select Object: <input type="button" value="..."/> Mass: <input type="text"/> Coefficient of Restitution: <input type="text"/> Position: <input type="text"/> Velocity: <input type="text"/> Angular Velocity: <input type="text"/> Colour: <input type="text"/> Resultant Force: <input type="text"/> (Excluding Gravity)
	<input style="background-color: red; color: white; padding: 10px; border: none;" type="button" value="Create Object!"/> <span style="border: 1px solid red; padding: 5px; margin-left: 20px;">Delete Selected Object!</span>		
	<input style="background-color: red; color: white; padding: 10px; border: none;" type="button" value="Make Changes!"/> <input style="background-color: red; color: white; padding: 10px; border: none;" type="button" value="Alter!"/> <span style="border: 1px solid blue; padding: 5px; margin-left: 20px;"></span>		

They also said that it was very inconvenient that for the “orbit” simulation, they would need to do calculations by hand to work out where and at which speed planets would need to be placed in order to make them fall into orbit about another body.

Main Menu	Object Creation	Alter Environment	Alter Objects
	Mass: <input type="text"/> kg Position: <input type="text"/> <small>Input for position, velocity, dimensions, colour, and resultant force should be entered in the form "(x, y, z)". Mass should be a number</small> Velocity: <input type="text"/> Radius: <input type="text"/> Colour: <input type="text"/> Name: <input type="text"/> (optional)	Gravitational: <input type="text"/> "Constant" Background: <input type="text"/> Colour	Select Object: <input type="button" value="..."/> Mass: <input type="text"/> Position: <input type="text"/> Velocity: <input type="text"/> Colour: <input type="text"/>
	<input style="background-color: red; color: white; padding: 10px; border: none;" type="button" value="Create Object!"/> <input style="background-color: red; color: white; padding: 10px; border: none;" type="button" value="Make Changes!"/> <input style="background-color: red; color: white; padding: 10px; border: none;" type="button" value="Alter!"/> <span style="border: 1px solid blue; padding: 5px; margin-left: 20px;"></span>		

I will discuss later how these comments could be addressed in further development.

## Evaluation of Solution

### Cross referencing evidence with success criteria:

As a reminder of what my 11 success criteria are:

1. Main window which is moveable and resizable.
2. Menu screen.
3. Lightweight and easy to navigate.
4. Multiple different options of different simulations.
5. 3D rasteriser.
6. Shading.
7. Working rigid body motion.
8. Working rigid body collision.
9. Normal reaction force.
10. Options next to each simulation such as creating objects, applying forces to objects etc.,.
11. Camera control.

Success criteria 1 has not been met because GUI elements do not automatically scale with window size, so I locked the window size to always be in fullscreen. This could later be addressed by finding a way to scale GUI elements with window size in the WinForms framework.

Success criteria 2 is shown by the screenshots of the menu screen as well as the settings button and simulation buttons working properly (they open the corresponding windows as shown).

Success criteria 4 is shown by all of the screenshots of menus and well labeled buttons. My test users, as previously mentioned, said the menus worked well and were easy to use, though they did have some usability suggestions which I will address later.

Success criteria 5 & 6 are demonstrated by images of a cube being shown in 3 dimensions, with different sides having slightly different shades due to how they are oriented.

Success criteria 7 is demonstrated by screenshots showing a single object moving to the side as it is accelerated downwards (it moves in a parabolic path).

Success criteria 8 is demonstrated by screenshots of 2 cubes moving towards each other, before they rebound away from each other with equal velocities (as they have the same mass).

Success criteria 9 is shown by images of a red cube resting against a green cube after moving into it. This shows the normal reaction force preventing the red object from moving through the green one.

Success criteria 10 is shown by images of the 3 menus (one for each simulation) each providing options for the user.

Success criteria 11 is shown by images of the cube from many different angles, taken by moving the camera about the cube.

### *Addressing incomplete criteria in further development*

Success criteria 1 (moveable and resizeable) has not been met because GUI elements do not automatically scale with window size, so I locked the window size to always be in fullscreen. This could later be addressed by finding a way to scale GUI elements with window size in the WinForms framework.

Success criteria 3 (lightweight and easy to navigate) could be further addressed by following by testers' suggestions. They said that menus "...were easy to use, but it was annoying that each time the menu is opened the text boxes would all have the background image of the simulation covering the textbox until it is interacted with." (This is due to an error within WinForms.) I could address this in further development by creating these menus in a dialogue box, instead of covering the whole window.

They also pointed out that to get the "orbit" simulation to work properly, you need to use very carefully chosen values which must be calculated beforehand. I could address this in further development by adding textboxes into which you write certain information, (like the radial orbit, period and velocity) to tell you what the necessary mass would be for this to occur. This would not be too difficult to implement but could make this aspect of the program much more user-friendly.

A few other suggestions for future development of this success criteria were some accessibility features such as text to speech for menus, as well as the ability to change the colour of the ground (which I initially planned to do but must have forgotten to). Adding the ability to change the ground's colour would be trivial, but a text to speech implementation would likely be quite complicated to implement, unless there are built in libraries to do it for me. This would certainly be an optional feature if it was implemented, and whether it is enabled or disabled could either be an option in settings or on the main menu.

Success criteria 5 is almost entirely met, however at very small or large scales there can be slight visual errors (e.g., long straight edges appearing curved or gaps between objects which should be touching). None of these are typically apparent when using the program, as they are quite specific edge cases. This could be addressed in further development by using a more rigorous algorithm for the rendering of 3 dimensional points. Creation of a better

algorithm would be more do-able now than at the beginning of this project, as I know much more mathematics now than I did then.

The rest of the success criteria are fully met.

### Explain how tests from before demonstrate usability

See all previous comments about success criteria 3, as this is the success criteria for usability.

### Maintenance issues and limitations

My code is overall well commented and indented (which is optional in C#) which makes it much easier to follow than it otherwise would be. One potential maintenance issue is that all three of the simulations use very similar code, meaning a lot of code is copied and pasted which could be a waste of memory resources, and is also inconvenient if I wish to make a fundamental change to all three (e.g., a new rendering algorithm). The reason I did this is because there are enough differences between the simulations that I decided it was necessary.

One way in which this could be addressed in further development is by using inheritance. I could create a single class which has the basic elements of all three simulations. Each simulation could then inherit from this one, making changes only where necessary.

A limitation of my solution is that it is only able to run on PC meaning it can not be run on Mac or Linux (or a phone or tablet).

This limitation could be addressed by making whatever technical changes would be necessary for the program to be able to run on other operating systems such as MacOS or Linux (and to replace keyboard inputs with touchscreen controls in the case of iOS, iPadOS and android). I don't think that program being unable to run on mobile devices is much of a limitation, as it would most likely be used in a setting with access to computer, but the limitation of only being able to run on Windows is certainly one which might be worth addressing.

One of the limitations mentioned in my analysis was that the program would likely suffer from performance issues due to a lack of efficiency. In the end however, it has turned out that the program actually runs very well, almost always over 165 fps. The only time when performance suffers is when there are many objects very close together, when they get further apart though, the performance improves greatly (because of my collision detection optimisation algorithm). If there was to be a version to run on phones/tablets, some more optimisation would likely need to be made (depending on the specs of the device that is).

# All project code:

Camera Class:

```
namespace ComputerScienceAlevelProject
{
    internal class Camera
    {
        public Vector position;
        public Vector direction;
        public double fov;
        public Camera(Vector position, Vector direction, double fov)
        {
            this.position = position;
            this.direction = direction;
            this.fov = fov;
        }
    }
}
```

Entity Class:

```
namespace ComputerScienceAlevelProject
{
    internal class Entity
    {
        public Mesh mesh;
        public RigidBody rigidBody;
        public string name;
        public Entity(Mesh mesh, RigidBody rigidBody, string name)
        {
            this.mesh = mesh;
            this.rigidBody = rigidBody;
            this.name = name;
        }
        public override string ToString()
        {
            return this.name;
        }
    }
}
```

Face Class:

```
namespace ComputerScienceAlevelProject
{
    internal class Face
    {
        public Vector[] vertices;
        public Vector normal;
        public Face(Vector[] vertices, Vector normal)
        {
            this.vertices = vertices;
            this.normal = normal;
        }
    }
}
```

Ground Class:

```
namespace ComputerScienceAlevelProject
{
    internal class Ground
    {
        public double height;
        public bool toggled;
        public Vector colour;

        public Ground(double height, bool toggled, Vector colour)
        {
            this.height = height;
            this.toggled = toggled;
            this.colour = colour;
        }
    }
}
```

Mesh Class:

```
namespace ComputerScienceAlevelProject
{
    internal class Mesh
    {
        public Vector[] vertices;
        public Face[] faces;
        public Vector dimensions;
        public Vector colour;
        public Mesh(Vector[] vertices, Vector dimensions, Vector colour)
        {
            this.vertices = vertices;
            this.dimensions = dimensions;
            this.colour = colour;
        }
    }
}
```

Program Class (prewritten):

```
namespace ComputerScienceAlevelProject
{
    internal static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            // To customize application configuration such as set high DPI
            settings or default font,
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize();
            Application.Run(new Menu());
        }
    }
}
```

RigidBody Class:

```
namespace ComputerScienceAlevelProject
{
    internal class RigidBody
    {
        public double mass;
        public Vector centreOfMass;
        public Vector velocity;
        public Vector acceleration;
        public Vector angularVelocity;
        public Vector angularAcceleration;
        public Vector force;
        public double CoR;
        public Vector inertiaInv;
        public bool groundFlag;
        public RigidBody(double mass, Vector centreOfMass, Vector velocity,
Vector acceleration, Vector angularVelocity, Vector angularAcceleration, Vector
force, double Cor, Vector inertiaInv, bool groundFlag)
        {
            this.mass = mass;
            this.centreOfMass = centreOfMass;
            this.velocity = velocity;
            this.acceleration = acceleration;
            this.angularVelocity = angularVelocity;
            this.angularAcceleration = angularAcceleration;
            this.force = force;
            this.CoR = Cor;
            this.inertiaInv = inertiaInv;
            this.groundFlag = groundFlag;
        }
    }
}
```

Vector Class:

```
namespace ComputerScienceAlevelProject
{
    internal class Vector
    {
        public double x;
        public double y;
        public double z;
        public Vector(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
    }
}
```

Global Class:

```
using System.Text.RegularExpressions;

namespace ComputerScienceAlevelProject
{
    static class Global
    {
        //Creating variables
        public static double gravity = 9.81;
        public static Color bgColour = Color.Blue;
        public static Menu main_menu;
        public static List<Tuple<bool, Entity, Entity>> collisionFlags = new
List<Tuple<bool, Entity, Entity>>();
        private static Vector lightVector = new Vector(-1, 0.5, -0.25);
        private static double lightMag = Math.Sqrt(Math.Pow(lightVector.x, 2) +
Math.Pow(lightVector.y, 2) + Math.Pow(lightVector.z, 2));
        public static bool paused;
        public static double gravConst = 6.67*Math.Pow(10, -11); //Gravitational
Constant = 6.67x10^-11

        private static List<Entity> entities = new List<Entity>();
        public static List<Entity> Entities
        {
            get { return entities; }
            set { entities = value; }
        }

        private static Camera camera = new Camera(new Vector(-1, 0, 0), new
Vector(1, 0.01, 0.01), 7 * Math.PI / 18);
        public static Camera Camera
        {
            get { return camera; }
            set { camera = value; }
        }

        public static Ground ground = new Ground(10, true, new Vector(0, 255,
0));
        public static Ground Ground
        {
            get { return ground; }
            set { ground = value; }
        }

        public static int camSpeed = 25;
        public static int camSensitivity = 50;
        public static double frameTime;
        public static int fps = 60;

tanfov) public static double calcXCoord(Camera camera, Vector point, double
//tanfov is tan((field of view)/2)
{
    if (camera.direction.z == 0) //special case 1
    {
        double dist = point.z - camera.position.z;
        double length = (point.x - camera.position.x) * tanfov;
        return (dist / length);
    }
    else if (camera.direction.x == 0) //special case 2
    {
        double dist = camera.position.x - point.x;
```

```

        double length = (point.z - camera.position.z) * tanfov;
        return (dist / length);
    }
    else //no special case
    {
        double p = (point.x + ((camera.direction.x * camera.position.z) /
camera.direction.z) + ((camera.direction.z * point.z) / camera.direction.x) -
camera.position.x) / ((camera.direction.x / camera.direction.z) +
(camera.direction.z / camera.direction.x));
        double q = ((camera.direction.x * p) / camera.direction.z) +
camera.position.x - ((camera.direction.x * camera.position.z) /
camera.direction.z);
        double dist = Math.Sqrt(Math.Pow(p - point.z, 2) + Math.Pow(q -
point.x, 2));
        double length = Math.Sqrt(Math.Pow(p - camera.position.z, 2) +
Math.Pow(q - camera.position.x, 2)) * tanfov;
        length = Math.Sqrt(Math.Pow(length, 2) +
Math.Pow(camera.position.y - point.y, 2));
        if (leftOrRight(point) == true)
        {
            return (dist / length);
        }
        else
        {
            return (-dist / length);
        }
    }
}
public static double calcYCoord(Camera camera, Vector point, double
tanfov) //tanfov is tan((field of view)/2)
{
    double D = (camera.direction.y) /
Math.Sqrt((Math.Pow(camera.direction.x, 4) / Math.Pow(camera.direction.z, 2)) + 2 *
Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2));
    double a;
    double b;
    double c;
    double d;
    double x;
    double y;
    double z;
    double dist;
    double length;
    double adjacent;
    //Plane is horizontal
    if (camera.direction.y == 0)
    {
        dist = point.y - camera.position.y;
        adjacent = Math.Sqrt(Math.Pow((point.x - camera.position.x), 2) +
Math.Pow(point.z - camera.position.z, 2));
        length = adjacent * tanfov;
        return dist / length;
    }
    //dy/dz=0
    else if (camera.direction.z == 0)
    {
        //Equation of plane is: y -
        (camera.direction.y/camera.direction.x)*x = camera.position.y -
        (camera.direction.y/camera.direction.x)*camera.position.x
        //In the form ax+by+cz=d:
        a = -camera.direction.y / camera.direction.x;
        b = 1;
        c = 0;
    }
}
```

```

        d = camera.position.y - (camera.direction.y / camera.direction.x)
* camera.position.x;
        //since c = 0 in this case, I find dist and length in here.
        Vector dummyPos = new Vector(camera.position.y, 0,
camera.position.x);
        Vector dummyVector = new Vector(camera.direction.y, 0,
camera.direction.x);
        Camera dummyCamera = new Camera(dummyPos, dummyVector, tanfov);
        //Replaced x by y and replaced z by x so that the calcXCoord
function can be reused with the parameters switched around.
        return calcXCoord(dummyCamera, point, tanfov);
    }

    //No special case:
    else
    {
        Vector v1 = new Vector(1, D * (camera.direction.x /
camera.direction.z), 0);
        Vector v2 = new Vector(0, D, 1);
        Vector perpVect = crossProduct(v1, v2);
        a = perpVect.x;
        b = perpVect.y;
        c = perpVect.z;
        d = a * camera.position.x + b * camera.position.y + c *
camera.position.z;

        dist = Math.Abs(a * point.x + b * point.y + c * point.z - d) /
Math.Sqrt(a * a + b * b + c * c);
        z = (Math.Pow(a, 2) * point.z - a * c * point.x + Math.Pow(b, 2)
* point.z + c * d - b * c * point.y) / (Math.Pow(a, 2) + Math.Pow(b, 2) +
Math.Pow(c, 2));
        x = a * (z - point.z) / c + point.x;
        y = (d - a * x - c * z) / b;
        adjacent = Math.Sqrt(Math.Pow(camera.position.x - x, 2) +
Math.Pow(camera.position.y - y, 2) + Math.Pow(camera.position.z - z, 2));
        length = adjacent * tanfov;

        if (upOrDown(point, a, b, c, d) == true)
        {
            return (-dist / length);
        }
        else
        {
            return (dist / length);
        }
    }
}

public static int denormaliseX(double x, double width)
{
    return Convert.ToInt32((width / 2 * (x + 1)));
}
public static int denormaliseY(double y, double height)
{
    return Convert.ToInt32((height / 2 * (y + 1)));
}

```

---

//Camera Rotation

---



---

```

        public static Vector rotateYaw(double angle, Vector vect) //Rotates a
vector left by some angle
    {
        double newX = vect.x * Math.Cos(angle) + vect.z * Math.Sin(angle);
        double newZ = -vect.x * Math.Sin(angle) + vect.z * Math.Cos(angle);
        return new Vector(newX, vect.y, newZ);
    }
    public static Vector rotatePitch(double angle, Vector vect) //Rotates a
vector up by some angle
    {
        if (camera.direction.z > 0)
        {
            angle = -angle;
        }
        Vector k = getCamHorPerpUnitVect(camera);
        Vector v = vect;
        double cosine = Math.Cos(angle);
        double sine = Math.Sin(angle);

        Vector term1 = new Vector(v.x * cosine, v.y * cosine, v.z * cosine);
        Vector crossProd = crossProduct(k, v);
        Vector term2 = new Vector(crossProd.x * sine, crossProd.y * sine,
crossProd.z * sine);
        double dotProd = dotProduct(k, v);
        Vector term3 = new Vector(k.x * dotProd * (1 - cosine), k.y * dotProd
* (1 - cosine), k.z * dotProd * (1 - cosine));
        Vector dummyVector = new Vector(term1.x + term2.x + term3.x, term1.y
+ term2.y + term3.y, term1.z + term2.z + term3.z);
        if (dummyVector.x > 0 && vect.x < 0 || dummyVector.x < 0 && vect.x >
0 || dummyVector.z > 0 && vect.z < 0 || dummyVector.z < 0 && vect.z > 0)
            //vector is not in the same quadrant before or after
        {
            return vect; //Do nothing
        }
        else
        {
            return dummyVector; //the cameras direction is changed to this
new vector
        }
    }

    -----
}

public static bool leftOrRight(Vector point)
{
    if (camera.direction.z > 0) //camera is facing right
    {
        if (point.x < camera.direction.x * point.z / camera.direction.z +
camera.position.x - camera.direction.x * camera.position.z / camera.direction.z)
        {
            return (true);
        }
        else
        {
            return (false);
        }
    }
}

```

```

        }
        else //vector is facing left (facing due north or south was already
covered by special cases)
        {
            if (point.x < camera.direction.x * point.z / camera.direction.z +
camera.position.x - camera.direction.x * camera.position.z / camera.direction.z)
            {
                return (false);
            }
            else
            {
                return (true);
            }
        }
    }
    public static bool upOrDown(Vector point, double a, double b, double c,
double d)
{
    if (point.y < (d - a * (point.x) - c * point.z) / b)
    {
        return true;
    }
    else
    {
        return false;
    }
}
public static double dotProduct(Vector vect1, Vector vect2) //computes
the dot product of two vectors
{
    return (vect1.x * vect2.x + vect1.y * vect2.y + vect1.z * vect2.z);
}
public static Vector crossProduct(Vector vect1, Vector vect2) //computes
the cross product of two vectors
{
    double xComp = vect1.y * vect2.z - vect1.z * vect2.y;
    double yComp = vect1.z * vect2.x - vect1.x * vect2.z;
    double zComp = vect1.x * vect2.y - vect1.y * vect2.x;
    return (new Vector(xComp, yComp, zComp));
}
public static Vector getCamHorPerpUnitVect(Camera camera) //gets the
vector about which the camera vector will be rotated for a pitch rotation
{
    double vectMag = Math.Sqrt(Math.Pow(camera.direction.x, 2) +
Math.Pow(camera.direction.z, 2));
    double zComp = camera.direction.x / vectMag;
    double xComp = -camera.direction.z / vectMag;
    return (new Vector(xComp, 0, zComp));
}

public static Vector getCamHorUnitVect(Camera camera) //same as the above
but not the perpendicular vector
{
    double xComp = camera.direction.x;
    double zComp = camera.direction.z;
    double vectMag = Math.Sqrt(Math.Pow(xComp, 2) + Math.Pow(zComp, 2));
    return new Vector(xComp / vectMag, 0, zComp / vectMag);
}

//Culling
public static bool checkRender(Camera camera, Vector point, double
vertFov)

```

```

{
    Vector dummyCameraDir;
    if (camera.direction.z < 0)
    {
        dummyCameraDir = new Vector(camera.direction.x, -
camera.direction.y, camera.direction.z);
    }
    else
    {
        dummyCameraDir = camera.direction;
    }
    double cosTheta = dotProduct(dummyCameraDir,
vectBetweenPoints(camera.position, point))/distBetweenPoints(camera.position,
point);
    if (cosTheta <= 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}

```

#### //Camera Movement-----

```

public static void moveUp(double displacement)
{
    camera.position.y = camera.position.y - displacement;
}
public static void moveDown(double displacement)
{
    camera.position.y = camera.position.y + displacement;
}
public static void moveForward(double displacement)
{
    Vector camHorUnitVect = getCamHorUnitVect(camera);
    Camera.position.x = camera.position.x + displacement *
camHorUnitVect.x;
    Camera.position.z = camera.position.z + displacement *
camHorUnitVect.z;
}
public static void moveBackward(double displacement)
{
    Vector camHorUnitVect = getCamHorUnitVect(camera);
    Camera.position.x = camera.position.x - displacement *
camHorUnitVect.x;
    Camera.position.z = camera.position.z - displacement *
camHorUnitVect.z;
}
public static void moveRight(double displacement)
{
    Vector camHorPerpUnitVect = getCamHorPerpUnitVect(camera);
    Camera.position.x = camera.position.x + displacement *
camHorPerpUnitVect.x;
    Camera.position.z = camera.position.z + displacement *
camHorPerpUnitVect.z;
}
public static void moveLeft(double displacement)
{
    Vector camHorPerpUnitVect = getCamHorPerpUnitVect(camera);

```

```

        Camera.position.x = camera.position.x - displacement *
camHorPerpUnitVect.x;
        Camera.position.z = camera.position.z - displacement *
camHorPerpUnitVect.z;
    }

//User input:

//Creating bools for whether each important key is pressed or not which
can be changed and viewed anywhere else in the program.
private static bool wHeld = false;
public static bool WHeld
{
    get { return wHeld; }
    set { wHeld = value; }
}
private static bool aHeld = false;
public static bool AHeld
{
    get { return aHeld; }
    set { aHeld = value; }
}
private static bool sHeld = false;
public static bool SHeld
{
    get { return sHeld; }
    set { sHeld = value; }
}
private static bool dHeld = false;
public static bool DHeld
{
    get { return dHeld; }
    set { dHeld = value; }
}
private static bool upHeld = false;
public static bool UpHeld
{
    get { return upHeld; }
    set { upHeld = value; }
}
private static bool downHeld = false;
public static bool DownHeld
{
    get { return downHeld; }
    set { downHeld = value; }
}
private static bool leftHeld = false;
public static bool LeftHeld
{
    get { return leftHeld; }
    set { leftHeld = value; }
}
private static bool rightHeld = false;
public static bool RightHeld
{
    get { return rightHeld; }
    set { rightHeld = value; }
}
private static bool spaceHeld = false;
public static bool SpaceHeld
{
    get { return spaceHeld; }
    set { spaceHeld = value; }
}

```

```

        }
        private static bool eHeld = false;
        public static bool EHeld
        {
            get { return eHeld; }
            set { eHeld = value; }
        }
        private static bool shiftHeld = false;
        public static bool ShiftHeld
        {
            get { return shiftHeld; }
            set { shiftHeld = value; }
        }
        private static bool qHeld = false;
        public static bool QHeld
        {
            get { return qHeld; }
            set { qHeld = value; }
        }
    public static void cameraControl(double displacement, double angle,
Camera camera)
    {
        if (wHeld)
        {
            moveForward(displacement);
        }
        if (aHeld)
        {
            moveLeft(displacement);
        }
        if (sHeld)
        {
            moveBackward(displacement);
        }
        if (dHeld)
        {
            moveRight(displacement);
        }
        if (spaceHeld || eHeld)
        {
            moveUp(displacement);
        }
        if (shiftHeld || qHeld)
        {
            moveDown(displacement);
        }
        if (upHeld)
        {
            camera.direction = rotatePitch(angle, camera.direction);
        }
        if (downHeld)
        {
            camera.direction = rotatePitch(-angle, camera.direction);
        }
        if (leftHeld)
        {
            Vector dummyVector = rotateYaw(angle, camera.direction);
            //preview" the vector before the rotation takes place
            if (Math.Sign(dummyVector.z) == Math.Sign(camera.direction.z))
                { //The vector is on the same size of the x axis before and
after, rotate the same as usual
                    camera.direction = rotateYaw(angle, camera.direction);
            //rotate yaw the same as usual
        }
    }
}

```

```

        }
        else
            { //For some reason (I don't know why) the vertical angle of the
camera's direction seems to change sign when the z-component of the camera's
direction switches
                camera.direction = rotateYaw(angle, camera.direction);
                camera.direction.y = -camera.direction.y;
            }
        }
        if (rightHeld)
        {
            Vector dummyVector = rotateYaw(-angle, camera.direction);
            if (Math.Sign(dummyVector.z) == Math.Sign(camera.direction.z))
            {
                camera.direction = rotateYaw(-angle, camera.direction);
            }
            else
            {
                camera.direction = rotateYaw(-angle, camera.direction);
                camera.direction.y = -camera.direction.y;
            }
        }
    }
//-----
-----
```

```

public static double getBrightnessLevel(Vector vect, double vectMag)
{
    double angle = dotProduct(vect, lightVector);
    angle = angle / (vectMag * lightMag);
    angle = Math.Acos(angle);
    return (angle / Math.PI);
}
```

```
//Vector operations-----
```

```

public static double distBetweenPoints(Vector point1, Vector point2)
{
    return Math.Sqrt(Math.Pow(point1.x - point2.x, 2) + Math.Pow(point1.y
- point2.y, 2) + Math.Pow(point1.z - point2.z, 2));
}
public static Vector averagePoint(params Vector[] points) //returns the
average of a set of points (e.g., centre of a face or cube)
{
    double xTotal = 0;
    double yTotal = 0;
    double zTotal = 0;
    int len = points.Length;
    for (int i = 0; i < len; i++)
    {
        xTotal = xTotal + points[i].x;
        yTotal = yTotal + points[i].y;
        zTotal = zTotal + points[i].z;
    }
    return new Vector(xTotal / len, yTotal / len, zTotal / len);
}
public static Vector vectBetweenPoints(Vector point1, Vector point2)
//returns the vector from point1 to point2
{
```

```

        double xComp = point2.x - point1.x;
        double yComp = point2.y - point1.y;
        double zComp = point2.z - point1.z;
        return new Vector(xComp, yComp, zComp);
    }
    public static int closestPoint(Mesh mesh, Camera camera)
    {
        double dist1 = distBetweenPoints(mesh.vertices[0], camera.position);
        double dist2 = distBetweenPoints(mesh.vertices[1], camera.position);
        double dist3 = distBetweenPoints(mesh.vertices[2], camera.position);
        double dist4 = distBetweenPoints(mesh.vertices[3], camera.position);
        double dist5 = distBetweenPoints(mesh.vertices[4], camera.position);
        double dist6 = distBetweenPoints(mesh.vertices[5], camera.position);
        double dist7 = distBetweenPoints(mesh.vertices[6], camera.position);
        double dist8 = distBetweenPoints(mesh.vertices[7], camera.position);

        double shortestDist = Math.Min(dist1, dist2);
        shortestDist = Math.Min(shortestDist, dist3);
        shortestDist = Math.Min(shortestDist, dist4);
        shortestDist = Math.Min(shortestDist, dist5);
        shortestDist = Math.Min(shortestDist, dist6);
        shortestDist = Math.Min(shortestDist, dist7);
        shortestDist = Math.Min(shortestDist, dist8);

        if (shortestDist == dist1)
        { return 1; }
        else if (shortestDist == dist2)
        { return 2; }
        else if (shortestDist == dist3)
        { return 3; }
        else if (shortestDist == dist4)
        { return 4; }
        else if (shortestDist == dist5)
        { return 5; }
        else if (shortestDist == dist6)
        { return 6; }
        else if (shortestDist == dist7)
        { return 7; }
        else
        { return 8; }
    }
    public static Vector closestPointCoords(Mesh mesh, Camera camera) //gets
the distance of the closest point
    {
        int index = closestPoint(mesh, camera); //e.g., if index = 2, the
closest point is mesh.point2
        switch (index)
        {
            case 1:
                return mesh.vertices[0];
                break;
            case 2:
                return mesh.vertices[1];
                break;
            case 3:
                return mesh.vertices[2];
                break;
            case 4:
                return mesh.vertices[3];
                break;
            case 5:
                return mesh.vertices[4];
                break;
        }
    }
}

```

```

        case 6:
            return mesh.vertices[5];
            break;
        case 7:
            return mesh.vertices[6];
            break;
        case 8:
            return mesh.vertices[7];
            break;
    }
    return new Vector(0, 0, 0); //This is just to prevent "not all paths
return a value error" but this line will never be called
}

public static Vector vectorTimesScalar(Vector v, double s)
{
    return (new Vector(s * v.x, s * v.y, s * v.z));
}

public static Vector vectorAddVector(Vector v1, Vector v2)
{
    return (new Vector(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z));
}
public static double vectMag(Vector v)
{
    return (Math.Sqrt(Math.Pow(v.x, 2) + Math.Pow(v.y, 2) + Math.Pow(v.z,
2)));
}

public static double angleBetweenVectors(Vector v1, Vector v2)
{
    return (Math.Acos(dotProduct(v1, v2)/(vectMag(v1) * vectMag(v2))));

public static double distBetweenPointAndFace(Vector point, Face face)
{
    Vector normal = face.normal;
    double d = -dotProduct(normal, face.vertices[0]);
    double distance = Math.Abs(dotProduct(normal, point) + d) /
vectMag(normal);
    return distance;
}
public static Vector normalOfClosestFace(Vector point, Face[] faces)
{
    List<double> distances = new List<double>();
    for (int i = 0; i < faces.Length; i++)
    {
        double distance = distBetweenPointAndFace(point, faces[i]);
        distances.Add(distance);
    }
    int indexOfShortest = distances.IndexOf(distances.Min());
    Vector normal = faces[indexOfShortest].normal;
    return normal;
}
public static Vector projectVector(Vector u, Vector v) //v is a unit
vector
{
    return vectorTimesScalar(v, dotProduct(u, v));
}

//-----

```

```

//Face normal
public static Vector faceNormal(Face face, Mesh mesh)
{
    Vector faceCentre = Global.averagePoint(face.vertices[0],
face.vertices[1], face.vertices[2], face.vertices[3]);
    Vector meshCentre = Global.averagePoint(mesh.vertices[0],
mesh.vertices[1], mesh.vertices[2], mesh.vertices[3], mesh.vertices[4],
mesh.vertices[5], mesh.vertices[6], mesh.vertices[7]);
    Vector perpVect = new Vector(faceCentre.x - meshCentre.x,
faceCentre.y - meshCentre.y, faceCentre.z - meshCentre.z);
    return perpVect;
}

----- //Physics implementation -----
----- //Motion
public static void updateAcceleration(Entity entity)
{
    entity.rigidBody.acceleration =
vectorTimesScalar(entity.rigidBody.force, 1 / entity.rigidBody.mass);
}

public static void updateVelocity(Entity entity)
{
    entity.rigidBody.velocity =
vectorAddVector(entity.rigidBody.velocity,
vectorTimesScalar(entity.rigidBody.acceleration, frameTime));
}
public static void displaceObject(Entity entity, Vector displacement)
{
    entity.rigidBody.centreOfMass =
vectorAddVector(entity.rigidBody.centreOfMass, displacement);
    for (int i = 0; i < entity.mesh.vertices.Length; i++) //Loops through
every vertex of the
    {
        entity.mesh.vertices[i] =
vectorAddVector(entity.mesh.vertices[i], displacement);
    }
}

//Rotation
public static void updateAngularVelocity(Entity entity)
{
    entity.rigidBody.angularVelocity =
vectorAddVector(entity.rigidBody.angularVelocity,
vectorTimesScalar(entity.rigidBody.angularAcceleration, frameTime));
}
public static void rotateObject(Entity obj, Vector angle)
{
    double aX = angle.x;
    double aY = angle.y;
    double aZ = angle.z;
    for (int i = 0; i < obj.mesh.vertices.Length; i++)
    {
        obj.mesh.vertices[i] =
vectBetweenPoints(obj.rigidBody.centreOfMass, obj.mesh.vertices[i]);
        double x = obj.mesh.vertices[i].x;
        double y = obj.mesh.vertices[i].y;

```

```

        double z = obj.mesh.vertices[i].z;
        obj.mesh.vertices[i].x = x * Math.Cos(aY) * Math.Cos(aZ) + y *
Math.Cos(aY) * Math.Sin(aZ) - z * Math.Sin(aY);
        obj.mesh.vertices[i].y = x * (Math.Sin(aX) * Math.Sin(aY) *
Math.Cos(aZ) - Math.Cos(aX) * Math.Sin(aZ)) + y * (Math.Sin(aX) * Math.Sin(aY) *
Math.Sin(aZ) + Math.Cos(aX) * Math.Cos(aZ)) + z * Math.Sin(aX) * Math.Cos(aY);
        obj.mesh.vertices[i].z = x * (Math.Sin(aX) * Math.Sin(aZ) +
Math.Cos(aX) * Math.Sin(aY) * Math.Cos(aZ)) + y * (Math.Cos(aX) * Math.Sin(aY) *
Math.Sin(aZ) - Math.Sin(aX) * Math.Cos(aZ)) + z * Math.Cos(aX) * Math.Cos(aY);
        obj.mesh.vertices[i] = vectorAddVector(obj.mesh.vertices[i],
obj.rigidBody.centreOfMass);
    }
}
//Collisions
//Collision Detection
public static bool boundingSpheresCheck(Entity obj1, Entity obj2)
{
    double radius1 = distBetweenPoints(obj1.rigidBody.centreOfMass,
obj1.mesh.vertices[0]);
    double radius2 = distBetweenPoints(obj2.rigidBody.centreOfMass,
obj2.mesh.vertices[0]);
    double distance = distBetweenPoints(obj1.rigidBody.centreOfMass,
obj2.rigidBody.centreOfMass);
    if (radius1 + radius2 < distance)
    {
        return (false);
    }
    else
    {
        return (true);
    }
}
public static bool checkPointAgainstFace(Vector point, Face face)
{
    Vector pointVect = vectBetweenPoints(face.vertices[0], point);
    double angle = angleBetweenVectors(pointVect, face.normal);
    if (0 <= angle && angle < Math.PI / 2)
    {
        return (false);
    }
    else
    {
        return (true);
    }
}
public static bool checkPointAgainstMesh(Vector point, Entity obj)
{
    for(int i = 0; i < obj.mesh.faces.Length; i++)
    {
        if (checkPointAgainstFace(point, obj.mesh.faces[i]) == false)
        {
            return (false);
        }
    }
    return (true);
}
public static Tuple<bool, Vector, Entity, Entity>
checkMeshAgainstMesh(Entity obj1, Entity obj2)
{
    List<Vector> points = new List<Vector>();
    for (int i = 0; i < obj1.mesh.vertices.Length; i++)
    {
        if (checkPointAgainstMesh(obj1.mesh.vertices[i], obj2) == true)

```

```

        {
            points.Add(obj1.mesh.vertices[i]);
        }
    }

    if (points.Count == 0)
    {
        return new Tuple<bool, Vector, Entity, Entity>(false, new
Vector(0, 0, 0), obj1, obj2);
    }
    else
    {
        return new Tuple<bool, Vector, Entity, Entity>(true,
averagePoint(points.ToArray()), obj1, obj2);
    }
}
public static Tuple<bool, Vector, Entity, Entity>
checkCollisionPrecise(Entity obj1, Entity obj2)
{
    Tuple<bool, Vector, Entity, Entity> check1 =
checkMeshAgainstMesh(obj1, obj2);
    if (check1.Item1 == true)
    {
        return (check1);
    }
    Tuple<bool, Vector, Entity, Entity> check2 =
checkMeshAgainstMesh(obj2, obj1);
    if (check2.Item1 == true)
    {
        return (check2);
    }
    else
    {
        var t = new Tuple<bool, Vector, Entity, Entity>(false, new
Vector(0, 0, 0), obj1, obj2);
        return (t);
    }
}
public static Tuple<bool, Vector, Entity, Entity> checkCollision(Entity
obj1, Entity obj2)
{
    if (boundingSpheresCheck(obj1, obj2) == true)
    {
        Tuple<bool, Vector, Entity, Entity> preciseCheck =
checkCollisionPrecise(obj1, obj2);
        if (preciseCheck.Item1 == true)
        {
            return (preciseCheck);
        }
    }
    var t = new Tuple<bool, Vector, Entity, Entity> (false, new Vector(0,
0, 0), obj1, obj2);
    return t;
}
public static List<Tuple<Vector, Entity, Entity>> checkAllCollisions()
{
    List<Tuple<Vector, Entity, Entity>> collisions = new
List<Tuple<Vector, Entity, Entity>>();
    for (int i = 0; i < Entities.Count - 1; i++)
    {
        for (int j = 0; j < Entities.Count - i - 1; j++)
        {

```

```

        Tuple<bool, Vector, Entity, Entity> check =
checkCollision(Entities[i], Entities[i + j + 1]);
        if (check.Item1 == true)
        {
            Tuple<Vector, Entity, Entity> collision = new
Tuple<Vector, Entity, Entity>(check.Item2, check.Item3, check.Item4);
            collisions.Add(collision);
        }
    }
    return collisions;
}

//Collision Response
public static void collisionResponse(Entity obj1, Entity obj2, Vector
point, Vector normal)
{
    Vector n = vectorTimesScalar(normal, 1 / vectMag(normal));
    double e = (obj1.rigidBody.CoR + obj2.rigidBody.CoR)/2;
    double m1 = obj1.rigidBody.mass; double m2 = obj2.rigidBody.mass;
    Vector u1 = obj1.rigidBody.velocity; Vector u2 =
obj2.rigidBody.velocity;
    Vector omegaInitial1 = obj1.rigidBody.angularVelocity; Vector
omegaInitial2 = obj2.rigidBody.angularVelocity;
    Vector rp1 = vectBetweenPoints(obj1.rigidBody.centreOfMass, point);
    Vector rp2 = vectBetweenPoints(obj2.rigidBody.centreOfMass, point);
    Vector up1 = vectorAddVector(u1, crossProduct(omegaInitial1, rp1));
    Vector up2 = vectorAddVector(u2, crossProduct(omegaInitial2, rp2));
    double part1 = -((e + 1)*dotProduct(vectBetweenPoints(up2, up1),
n));
    Vector part2 = vectorTimesScalar(n, 1 / m1 + 1 / m2);
    Vector v = crossProduct(rp1, n); //v is to be multiplied by the
tensor
    Vector tensor1 = obj1.rigidBody.inertiaInv;
    Vector part3 = crossProduct(new Vector(tensor1.x * v.x, tensor1.y *
v.y, tensor1.z * v.z), rp1);
    Vector tensor2 = obj2.rigidBody.inertiaInv;
    v = crossProduct(rp2, n);
    Vector part4 = crossProduct(new Vector(tensor2.x * v.x, tensor2.y *
v.y, tensor2.z * v.z), rp2);
    Vector j = vectorTimesScalar(n, Math.Abs(part1 /
dotProduct(vectorAddVector(vectorAddVector(part2, part3), part4), n)));
    Vector v1 = vectorAddVector(u1, vectorTimesScalar(j, 1 / m1));
    Vector v2 = vectorAddVector(u2, vectorTimesScalar(j, -1 / m2));
    v = crossProduct(rp1, j);
    Vector omegaFinal1 = vectorAddVector(omegaInitial1, new
Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z));
    v = crossProduct(j, rp2);
    Vector omegaFinal2 = vectorAddVector(omegaInitial2, new
Vector(tensor2.x * v.x, tensor2.y * v.y, tensor2.z * v.z));

    obj1.rigidBody.velocity = v1;
    obj2.rigidBody.velocity = v2;
    obj1.rigidBody.angularVelocity = omegaFinal1;
    obj2.rigidBody.angularVelocity = omegaFinal2;
}
public static Tuple<Vector, Vector> getNormalReactions(Entity obj1,
Entity obj2, Vector normal)
{
    Vector n = vectorTimesScalar(normal, 1 / vectMag(normal));
    Vector reaction1 =
vectorTimesScalar(projectVector(obj1.rigidBody.force, n), -1);

```

```

        Vector reaction2 =
vectorTimesScalar(projectVector(obj2.rigidBody.force, n), -1);
        return (new Tuple<Vector, Vector>(reaction1, reaction2));
    }

//Physics for the ground
public static List<Vector> groundCollisionDetection(Mesh mesh)
{
    List<Vector> points = new List<Vector>();
    for(int i = 0; i < mesh.vertices.Length; i++)
    {
        if (mesh.vertices[i].y > ground.height)
        {
            points.Add(mesh.vertices[i]);
        }
    }
    return points;
}
public static void groundCollisionResponse(Entity obj, Vector point)
{
    Vector n = new Vector(0, -1, 0);
    double e = obj.rigidBody.CoR;
    double m = obj.rigidBody.mass;
    Vector u = obj.rigidBody.velocity;
    Vector omegaInitial = obj.rigidBody.angularVelocity;
    Vector rp = vectBetweenPoints(obj.rigidBody.centreOfMass, point);
    Vector up = vectorAddVector(u, crossProduct(omegaInitial, rp));
    double part1 = -((e + 1) * dotProduct(up, n));
    Vector part2 = vectorTimesScalar(n, 1 / m);
    Vector v = crossProduct(rp, n);
    Vector tensor1 = obj.rigidBody.inertiaInv;
    Vector part3 = crossProduct(new Vector(tensor1.x * v.x, tensor1.y *
v.y, tensor1.z * v.z), rp);
    Vector j = vectorTimesScalar(n, Math.Abs(part1 /
dotProduct(vectorAddVector(part2, part3), n)));
    Vector v1 = vectorAddVector(u, vectorTimesScalar(j, 1 / m));
    v = crossProduct(rp, j);
    Vector omegaFinal = vectorAddVector(omegaInitial, new
Vector(tensor1.x * v.x, tensor1.y * v.y, tensor1.z * v.z));

    obj.rigidBody.velocity = v1;
    obj.rigidBody.angularVelocity = omegaFinal;
}

public static Vector regExInterp(string str)
{
    string pattern = @"^(\()?( -)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
    Match match = Regex.Match(str, pattern);
    double v1 = Convert.ToDouble(match.Groups[2].Value +
match.Groups[3].Value + match.Groups[4].Value);
    double v2 = Convert.ToDouble(match.Groups[6].Value +
match.Groups[7].Value + match.Groups[8].Value);
    double v3 = Convert.ToDouble(match.Groups[10].Value +
match.Groups[11].Value + match.Groups[12].Value);
    return (new Vector(v1, v2, v3));
}
}
}

```

Menu:

```
namespace ComputerScienceAlevelProject
{
    public partial class Menu : Form
    {
        public Menu()
        {
            InitializeComponent();
        }

        private void Menu_Load(object sender, EventArgs e)
        {
            Global.main_menu = this;
        }

        private void Open_Settings_Click(object sender, EventArgs e)
        {
            Settings settings = new Settings();
            settings.ShowDialog();
        }

        private void Play_Sim1_Click(object sender, EventArgs e)
        {
            RigidBodyCollisions sim = new RigidBodyCollisions();
            sim.WindowState = FormWindowState.Maximized;
            sim.FormBorderStyle = FormBorderStyle.None;
            sim.Show();
            this.Hide();
        }

        private void Play_Sim2_Click(object sender, EventArgs e)
        {
            ProjectileMotion sim = new ProjectileMotion();
            sim.WindowState = FormWindowState.Maximized;
            sim.FormBorderStyle = FormBorderStyle.None;
            sim.Show();
            this.Hide();
        }

        private void Play_Sim3_Click(object sender, EventArgs e)
        {
            Orbit sim = new Orbit();
            sim.WindowState = FormWindowState.Maximized;
            sim.FormBorderStyle = FormBorderStyle.None;
            sim.Show();
            this.Hide();
        }
    }
}
```

Settings:

```
namespace ComputerScienceAlevelProject
{
    public partial class Settings : Form
    {
        public Settings()
        {
            InitializeComponent();
        }
}
```

```

private void Field_of_View_Scroll(object sender, EventArgs e)
{
    textBox1.Text = Field_of_View.Value.ToString();
}

private void Settings_Load(object sender, EventArgs e)
{
    //Set field values to their actual program values.
    Camera_Speed.Value = Global.camSpeed;

    Camera_Sensitivity.Value = Global.camSensitivity;

    Field_of_View.Value = Convert.ToInt32(Global.Camera.fov * 180 /
Math.PI);
    textBox1.Text = Convert.ToInt32(Global.Camera.fov * 180 /
Math.PI).ToString();

    Maximum_Framerate.Value = Global.fps;
}

private void Save_Button_Click(object sender, EventArgs e)
{
    Global.camSpeed = Convert.ToInt32(Camera_Speed.Value);
    Global.camSensitivity = Convert.ToInt16(Camera_Sensitivity.Value);
    Global.Camera.fov = Field_of_View.Value * Math.PI / 180;
    Global.fps = Convert.ToInt16(Maximum_Framerate.Value);
}

private void Exit_Settings_Click(object sender, EventArgs e)
{
    this.Close();
}
}
}
}

```

RigidBodyCollisions:

```

using System.Data;
using System.Diagnostics;
using System.Text.RegularExpressions;

namespace ComputerScienceAlevelProject
{
    public partial class RigidBodyCollisions : Form
    {
        public RigidBodyCollisions()
        {
            InitializeComponent();
            this.KeyPreview = true;
        }

        private async void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            int formWidth = this.Width;
            int formHeight = this.Height;
            this.DoubleBuffered = true;
            double vertFov = 2 * Math.Atan(formHeight *
Math.Tan(Global.Camera.fov / 2) / formWidth);
            double vertTanFov = Math.Tan(vertFov / 2);
        }
    }
}

```

```

        double tanfov = Math.Tan(Global.Camera.fov / 2);

        //Rendering individual face:
        void renderFace(Face face, Vector colour)
        {
            //Colour determination
            Vector perpVect = face.normal;
            double vectMag = Global.vectMag(perpVect);
            double shade = Global.getBrightnessLevel(perpVect, vectMag);
            Brush brush = new
SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade),
Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade))); //Can be
changed for different colours

            Point point1 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[0],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[0], vertTanFov), formHeight));
            Point point2 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[1],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[1], vertTanFov), formHeight));
            Point point3 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[2],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[2], vertTanFov), formHeight));
            Point point4 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[3],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[3], vertTanFov), formHeight));
            Point[] points = { point1, point2, point3, point4 };
            g.FillPolygon(brush, points);
        }

        void renderGround(Ground ground, Camera camera)
        {
            double tanTheta = camera.direction.y /
Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2));
            double H;
            if (camera.direction.z >= 0)
            {
                H = -tanTheta / vertTanFov; //Calculates H based on the
derived formula
            }
            else
            {
                H = tanTheta / vertTanFov;
            }
            H = Global.denormaliseY(H, formHeight); //denormalises this value
            double shade = Global.getBrightnessLevel(new Vector(0, -1, 0),
1);
            Vector colour = ground.colour; //colour of the ground
            Brush brush = new
SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade),
Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade)));
            Point point1 = new Point(0, Convert.ToInt16(H)); //brush and
points are created so that the shape can be rendered
            Point point2 = new Point(formWidth, Convert.ToInt16(H));
            Point point3 = new Point(formWidth, formHeight);
            Point point4 = new Point(0, formHeight);
            Point[] points = { point1, point2, point3, point4 };
            g.FillPolygon(brush, points);
        }
    }
}

```

```

//Render mesh
void renderMesh(Mesh mesh, Camera camera)
{
    //check all mesh vertices
    List<Vector> vertices = new List<Vector>();
    vertices.Add(mesh.vertices[0]);
    vertices.Add(mesh.vertices[1]);
    vertices.Add(mesh.vertices[2]);
    vertices.Add(mesh.vertices[3]);
    vertices.Add(mesh.vertices[4]);
    vertices.Add(mesh.vertices[5]);
    vertices.Add(mesh.vertices[6]);
    vertices.Add(mesh.vertices[7]);
    bool passCheck = true;
    for (int i = 0; i < 8; i++)
    {
        if (Global.checkRender(camera, vertices[i], vertFov) ==
false)
        {
            passCheck = false;
            break;
        }
    }

    if (passCheck == true)
    {
        int closestPoint = Global.closestPoint(mesh, camera);
        Face topFace = mesh.faces[0];
        Face bottomFace = mesh.faces[1];
        Face leftFace = mesh.faces[2];
        Face rightFace = mesh.faces[3];
        Face frontFace = mesh.faces[4];
        Face backFace = mesh.faces[5];
        switch (closestPoint)
        {
            case 1:
                renderFacesInOrder(topFace, leftFace, frontFace,
camera, mesh);
                break;
            case 2:
                renderFacesInOrder(topFace, rightFace, frontFace,
camera, mesh);
                break;
            case 3:
                renderFacesInOrder(topFace, rightFace, backFace,
camera, mesh);
                break;
            case 4:
                renderFacesInOrder(topFace, leftFace, backFace,
camera, mesh);
                break;
            case 5:
                renderFacesInOrder(bottomFace, leftFace, frontFace,
camera, mesh);
                break;
            case 6:
                renderFacesInOrder(bottomFace, rightFace, frontFace,
camera, mesh);
                break;
            case 7:

```

```

        renderFacesInOrder(bottomFace, rightFace, backFace,
camera, mesh);
            break;
        case 8:
            renderFacesInOrder(bottomFace, leftFace, backFace,
camera, mesh);
            break;
    }
}

void renderFacesInOrder(Face face1, Face face2, Face face3, Camera
camera, Mesh mesh)
{
    Vector point1 = Global.averagePoint(face1.vertices[0],
face1.vertices[1], face1.vertices[2], face1.vertices[3]);
    Vector point2 = Global.averagePoint(face2.vertices[0],
face2.vertices[1], face2.vertices[2], face2.vertices[3]);
    Vector point3 = Global.averagePoint(face3.vertices[0],
face3.vertices[1], face3.vertices[2], face3.vertices[3]);
    double dist1 = Global.distBetweenPoints(point1, camera.position);
    double dist2 = Global.distBetweenPoints(point2, camera.position);
    double dist3 = Global.distBetweenPoints(point3, camera.position);
    //distance of each face from the camera

    List<Face> faces = new List<Face>();

    if (dist1 < dist2 && dist1 < dist3) //dist1 is the smallest
    {
        faces.Add(face1);
        if (dist2 < dist3) //dist2 is the next smallest
        {
            faces.Add(face2);
            faces.Add(face3); //Add face2 then face3
        }
        else //dist3 > dist2 so face3 is added, then face2
        {
            faces.Add(face3);
            faces.Add(face2);
        }
    }
    else if (dist2 < dist1 && dist2 < dist3) //dist2 is the smallest
    {
        faces.Add(face2);
        if (dist1 < dist3)
        {
            faces.Add(face1);
            faces.Add(face3);
        }
        else
        {
            faces.Add(face3);
            faces.Add(face1);
        }
    }
    else //dist3 is the smallest
    {
        faces.Add(face3);
        if (dist1 < dist2)
        {
            faces.Add(face1);
            faces.Add(face2);
        }
    }
}

```

```

        else
        {
            faces.Add(face2);
            faces.Add(face1);
        }
    }
    //Render faces in order:
    for (int i = 2; i > -1; i--) //render from furthest to closest
    {
        renderFace(faces[i], mesh.colour);
    }
}

//Define mesh faces
void setMeshFaces(Mesh mesh)
{
    Vector v = new Vector(0, 0, 0);
    Face face1 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[1], mesh.vertices[2], mesh.vertices[3] }, v);
    Face face2 = new Face(new Vector[] { mesh.vertices[4],
mesh.vertices[5], mesh.vertices[6], mesh.vertices[7] }, v);
    Face face3 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[3], mesh.vertices[7], mesh.vertices[4] }, v);
    Face face4 = new Face(new Vector[] { mesh.vertices[1],
mesh.vertices[2], mesh.vertices[6], mesh.vertices[5] }, v);
    Face face5 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[1], mesh.vertices[5], mesh.vertices[4] }, v);
    Face face6 = new Face(new Vector[] { mesh.vertices[2],
mesh.vertices[3], mesh.vertices[7], mesh.vertices[6] }, v);
    face1.normal = Global.faceNormal(face1, mesh);
    face2.normal = Global.faceNormal(face2, mesh);
    face3.normal = Global.faceNormal(face3, mesh);
    face4.normal = Global.faceNormal(face4, mesh);
    face5.normal = Global.faceNormal(face5, mesh);
    face6.normal = Global.faceNormal(face6, mesh);
    Face[] faces = { face1, face2, face3, face4, face5, face6 };
    mesh.faces = faces;
}

//RENDER ALL:
void renderAll(List<Entity> entities, Camera camera)
{
    if (Global.Ground.toggled)
    {
        renderGround(Global.Ground, camera);
    }
    entities = entities.OrderByDescending(x =>
Global.vectMag(Global.projectVector(Global.vectBetweenPoints(x.rigidBody.centreOfMass, camera.position), camera.direction))).ToList(); //list is order based on closest point's distance (descending order)
    for (int i = 0; i < entities.Count; i++)
    {
        setMeshFaces(entities[i].mesh);
        renderMesh(entities[i].mesh, camera);
    }
}

void gameLoop()
{
    if (Global.Ground.toggled)
    {
        if (Global.Camera.position.y > Global.Ground.height)
        {

```

```

                Global.Camera.position.y = Global.Ground.height;
            }
        }
        Global.cameraControl(Global.camSpeed * Global.frameTime,
Convert.ToDouble(Global.camSensitivity) / 50 * Math.PI * Global.frameTime,
Global.Camera);
        renderAll(Global.Entities, Global.Camera);
        physics();
    }

    //Physics-----
-----
```

---

```

        void doCollisionResponse(Tuple<Vector, Entity, Entity> check)
{
    Vector normal = Global.normalOfClosestFace(check.Item1,
check.Item3.mesh.faces);
    Global.collisionResponse(check.Item2, check.Item3, check.Item1,
normal);
}

void physics()
{
    //Collision Detection
    List<Tuple<Vector, Entity, Entity>> check =
Global.checkAllCollisions();
    for (int i = 0; i < Global.Entities.Count; i++)
    {
        //Add weight to resultant force.
        Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0,
Global.Entities[i].rigidBody.mass * Global.gravity, 0));
        //Checking collision flags
        for (int j = 0; j < check.Count(); j++) //for each current
intersection
        {
            bool found = false;
            for (int k = 0; k < Global.collisionFlags.Count(); k++)
//for each collision flag
            {
                if ((Global.collisionFlags[k].Item2 == check[j].Item2
&& Global.collisionFlags[k].Item3 == check[j].Item3)
                || (Global.collisionFlags[k].Item2 ==
check[j].Item3 && Global.collisionFlags[k].Item3 == check[j].Item2))
                {
                    //There is a match, the registered collision
pairing already has a flag
                    found = true;
                    if (Global.collisionFlags[k].Item1 == false)
{
                        Global.collisionFlags[k] = new Tuple<bool,
Entity, Entity>(true, Global.collisionFlags[k].Item2,
Global.collisionFlags[k].Item3);
                        doCollisionResponse(check[j]);
}
                    //else, there is already a collision flag set to
true, meaning the response was already handled the previous frame(s).
                }
            }
            if (found == false)
{
```

```

                //check[j] is a collision which is occurring, but does
not exist as a flag, create one.
                Global.collisionFlags.Add(new Tuple<bool, Entity,
Entity>(true, check[j].Item2, check[j].Item3));
                doCollisionResponse(check[j]);
            }
        }

        //Checking if there are any flags which need to be updated
from true to false.
        for (int k = 0; k < Global.collisionFlags.Count(); k++)
        {
            if (Global.collisionFlags[k].Item1)
            {
                bool found = false;
                for (int j = 0; j < check.Count(); j++)
                {
                    if ((Global.collisionFlags[k].Item2 ==
check[j].Item2 && Global.collisionFlags[k].Item3 == check[j].Item3)
                        || (Global.collisionFlags[k].Item2 ==
check[j].Item3 && Global.collisionFlags[k].Item3 == check[j].Item2))
                    {
                        found = true;
                    }
                }
                if (found == false)
                {
                    //There was a flag set to true, for which there
was no collision this frame, set the flag to false.
                    Global.collisionFlags[k] = new Tuple<bool,
Entity, Entity>(false, Global.collisionFlags[k].Item2,
Global.collisionFlags[k].Item3);
                }
            }
        }

        //Check if the ground is enabled
        if (Global.Ground.toggled)
        {
            //Collision detection and response with the ground

            List<Vector> points =
Global.groundCollisionDetection(Global.Entities[i].mesh);
            bool groundCollided = false;

            if (Global.Entities[i].rigidBody.groundFlag == false)
            {
                if (points.Count != 0)
                {
                    //The flag is false and there was an
intersection:
                    //Do collision response and set flag to true.
                    Vector position =
Global.averagePoint(points.ToArray());

                    Global.groundCollisionResponse(Global.Entities[i], position);
                    groundCollided = true;
                    Global.Entities[i].rigidBody.groundFlag = true;
                }
            }
            else
            {
                //The flag was false and there was no
intersection:
            }
        }
    }
}

```

```

                //Do nothing
            }
        }
    }
    else
    {
        if (points.Count != 0)
        {
            //The flag is true and there was an intersection:
            groundCollided = true;
        }
        else
        {
            //The flag was true but there was no
intersection:
            //Set flag to false:
            Global.Entities[i].rigidBody.groundFlag = false;
        }
    }

    //Find with which object and normal (if any) this object
collided with each frame and apply normal reaction.
//This is done regardless of collision flags.
Vector reaction = new Vector(0, 0, 0);
for (int j = 0; j < check.Count; j++)
{
    if (Global.Entities[i] == check[j].Item2)
    {
        reaction = Global.vectorAddVector(reaction,
Global.getNormalReactions(check[j].Item2, check[j].Item3,
Global.normalOfClosestFace(check[j].Item1, check[j].Item3.mesh.faces)).Item1);
    }
    if (Global.Entities[i] == check[j].Item3)
    {
        reaction = Global.vectorAddVector(reaction,
Global.getNormalReactions(check[j].Item2, check[j].Item3,
Global.normalOfClosestFace(check[j].Item1, check[j].Item3.mesh.faces)).Item2);
    }
    if (groundCollided)
    {
        reaction = Global.vectorAddVector(reaction, new
Vector(0, -Global.Entities[i].rigidBody.force.y, 0));
    }

    Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force, reaction);
    Global.updateAcceleration(Global.Entities[i]);
    Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force,
Global.vectorTimesScalar(reaction, -1));

    Global.updateVelocity(Global.Entities[i]);
    Global.displaceObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity,
Global.frameTime));

    Global.updateAngularVelocity(Global.Entities[i]);
    Global.rotateObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity,
Global.frameTime));
}
else

```

```

        {
            Global.updateAcceleration(Global.Entities[i]);
            Global.updateVelocity(Global.Entities[i]);
            Global.displaceObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity,
Global.frameTime));
            Global.updateAngularVelocity(Global.Entities[i]);
            Global.rotateObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity,
Global.frameTime));
        }
        //Take away gravity
        Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0, -Global.Entities[i].rigidBody.mass * Global.gravity, 0));
    }
}

//FrameRate-----
-----
```

```

if (Global.paused == false)
{
    this.BackColor = Global.bgColour;
    for (int i = 0; i < this.Controls.Count; i++)
    {
        this.Controls[i].Visible = false;
        this.Controls[i].Enabled = false;
    }
    Stopwatch timer = new Stopwatch();
    timer.Start();
    gameLoop();
    timer.Stop();
    Global.frameTime = timer.Elapsed.TotalSeconds;

    if (Global.frameTime < 1.0 / Global.fps)
    {
        Stopwatch timer2 = new Stopwatch();
        timer2.Start();

        while (timer2.Elapsed.TotalSeconds < 1.0 / Global.fps -
Global.frameTime)
        {
            timer2.Stop();
            Global.frameTime = 1.0 / Global.fps;
        }
    }
    else
    {
        this.BackColor = Color.Gray;
        for (int i = 0; i < this.Controls.Count; i++)
        {
            if (Controls[i] != ValidationLabel && Controls[i] !=
ValidationLabel2 && Controls[i] != ValidationLabel3 && Controls[i] !=
ValidationLabel4)
            {
                this.Controls[i].Visible = true;
            }
        }
    }
}
```

```
                this.Controls[i].Enabled = true;
            }
        }
    }
    Invalidate();
}

private void Form1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = true;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = true;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = true;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = true;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = true;
    }
    if (e.KeyCode == Keys.ShiftKey)
    {
        Global.ShiftHeld = true;
    }
    if (e.KeyCode == Keys.E)
    {
        Global.EHeld = true;
    }
    if (e.KeyCode == Keys.Q)
    {
        Global.QHeld = true;
    }
    if (e.KeyCode == Keys.Up)
    {
        Global.UpHeld = true;
    }
    if (e.KeyCode == Keys.Down)
    {
        Global.DownHeld = true;
    }
    if (e.KeyCode == Keys.Left)
    {
        Global.LeftHeld = true;
    }
    if (e.KeyCode == Keys.Right)
    {
        Global.RightHeld = true;
    }
    if (e.KeyCode == Keys.Escape)
    {
        Global.paused = !Global.paused;
        massInput2.Clear();
        corInput2.Clear();
        positionInput2.Clear();
    }
}
```

```
        velocityInput2.Clear();
        angularVelocityInput2.Clear();
        colourInput2.Clear();
        forceInput2.Clear();
    }
    Invalidate();
}

private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = false;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = false;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = false;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = false;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = false;
    }
    if (e.KeyCode == Keys.ShiftKey)
    {
        Global.ShiftHeld = false;
    }
    if (e.KeyCode == Keys.E)
    {
        Global.EHeld = false;
    }
    if (e.KeyCode == Keys.Q)
    {
        Global.QHeld = false;
    }
    if (e.KeyCode == Keys.Up)
    {
        Global.UpHeld = false;
    }
    if (e.KeyCode == Keys.Down)
    {
        Global.DownHeld = false;
    }
    if (e.KeyCode == Keys.Left)
    {
        Global.LeftHeld = false;
    }
    if (e.KeyCode == Keys.Right)
    {
        Global.RightHeld = false;
    }
    Invalidate();
}

private void Form1_Load(object sender, EventArgs e)
{
```

```

        Global.Ground.toggled = false;
        Global.paused = false;
        Global.Entities = new List<Entity>();
        Global.collisionFlags = new List<Tuple<bool, Entity, Entity>>();
        Global.Camera = new Camera(new Vector(-1, 0, 0), new Vector(1, 0.01,
0.01), 7 * Math.PI / 18);
        Global.gravity = 9.81;
    }

    private void Open_Settings_Click(object sender, EventArgs e)
{
    Settings settings = new Settings();
    settings.ShowDialog();
}

private void Exit_Click(object sender, EventArgs e)
{
    this.Close();
    Global.main_menu.Show();
}

private void Create_Click(object sender, EventArgs e)
{
    //Validate user inputs
    bool check = true;
    string pattern = @"^(\()?( -)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?(\))?$";
    string sMass = massInput.Text;
    string sCor = corInput.Text;
    Vector dimensions = new Vector(0, 0, 0);
    Vector colour = new Vector(0, 0, 0);
    if (double.TryParse(sMass, out double mass))
    {
        if (mass <= 0) { check = false; }
        //entered mass is negative which is not allowed
    }
    else { check = false; }//entered mass is not a number

    if (double.TryParse(sCor, out double cor))
    {
        if (cor < 0 || cor > 1) { check = false; }
        //checks that entered CoR is between 0 and 1
    }
    else { check = false; }

    //make sure the rest are of the form (x,y,z) or similar
    string sPosition = positionInput.Text;
    if (!Regex.IsMatch(sPosition, pattern))
    { check = false; }
    string sVelocity = velocityInput.Text;
    if (!Regex.IsMatch(sVelocity, pattern))
    { check = false; }
    string sDimensions = dimensionsInput.Text;
    if (!Regex.IsMatch(sDimensions, pattern))
    { check = false; }
    else
    {
        dimensions = Global.regExInterp(sDimensions);
        if (!(dimensions.x > 0 && dimensions.y > 0 && dimensions.z > 0))
        { check = false; }
    }
    string sColour = colourInput.Text;
    if (!Regex.IsMatch(sColour, pattern))
}

```

```

{ check = false; }
else
{
    colour = Global.regExInterp(sColour);
    if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y &&
colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
        { check = false; }
}
string sForce = forceInput.Text;
if (!Regex.IsMatch(sForce, pattern))
{ check = false; }

if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel.Visible = false;
    ValidationLabel.Enabled = false;

    //Assign object creation variables
    Vector position = Global.regExInterp(sPosition);
    Vector velocity = Global.regExInterp(sVelocity);
    Vector force = Global.regExInterp(sForce);

    //Create Object
    //Create vertices
    double depth = dimensions.x / 2;
    double height = dimensions.y / 2;
    double width = dimensions.z / 2;
    //I have divided these by 2 because I need the distance
    //from centre to face, not face to face.
    Vector v1 = new Vector(-depth, -height, -width);
    Vector v2 = new Vector(-depth, -height, width);
    Vector v3 = new Vector(depth, -height, width);
    Vector v4 = new Vector(depth, -height, -width);
    Vector v5 = new Vector(-depth, height, -width);
    Vector v6 = new Vector(-depth, height, width);
    Vector v7 = new Vector(depth, height, width);
    Vector v8 = new Vector(depth, height, -width);
    Vector[] vertices = { v1, v2, v3, v4, v5, v6, v7, v8 };

    //Create mesh:
    Mesh mesh = new Mesh(vertices, dimensions, colour);

    //Create Tensor and RigidBody
    Vector v0 = new Vector(0, 0, 0); //zero vector
    Vector tensor = Global.vectorTimesScalar(new Vector(1 / (height *
height + width * width), 1 / (depth * depth + width * width), 1 / (depth * depth
+ height * height)), 12 / mass);
    RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, v0,
force, cor, tensor, false);

    //Create name:
    string sName = nameInput.Text;
    string name = "";
    if (sName == "")
    {
        name = "Cube" + Global.Entities.Count;
    }
    else
    {
        name = sName;
    }
}

```

```

        //Create object
        Entity obj = new Entity(mesh, rb, name);
        Global.displaceObject(obj, position);
        Global.Entities.Add(obj);
        SelectObject.Items.Add(obj);
    }
    else
    {
        //Display validation label
        ValidationLabel.Visible = true;
        ValidationLabel.Enabled = true;
    }
}

private void MakeChanges_Click(object sender, EventArgs e)
{
    bool check = true;
    string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
    string sGrav = gravityInput.Text;
    string sColour = bgColourInput.Text;
    double grav;
    Vector colour = new Vector(0, 0, 0);

    if (sGrav == "") //if sGrav is empty it should pass the test
immediately
    {
        grav = 9.81;
    }
    else if (!double.TryParse(sGrav, out grav))
    {
        check = false;
    }

    //make sure colour is of the correct form or is an empty string
    else if (sColour != "" && !Regex.IsMatch(sColour, pattern))
    { check = false; }

    //if (check == true)
    if (check == true)
    {
        //Remove validation label (in case it is there)
        ValidationLabel2.Visible = false;
        ValidationLabel2.Enabled = false;

        if (sColour != "")
        {
            colour = Global.regExInterp(sColour);
            //apply background colour
            Global.bgColour = Color.FromArgb(Convert.ToInt16(colour.x),
Convert.ToInt16(colour.y), Convert.ToInt16(colour.z));
        }

        //apply gravity
        Global.gravity = grav;
    }
    else
    {
        //Display validation label
        ValidationLabel2.Visible = true;
        ValidationLabel2.Enabled = true;
    }
}

```

```

}

private void GroundEnabled_CheckedChanged(object sender, EventArgs e)
{
    if (GroundEnabled.Checked)
    {
        Global.Ground.toggled = true;
    }
    else
    {
        Global.Ground.toggled = false;
    }
}

private void Alter_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem == null)
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
    else
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;

        //Validate user inputs
        bool check = true;
        string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
        string sMass = massInput2.Text;
        string sCor = corInput2.Text;
        Vector colour = new Vector(0, 0, 0);

        double mass;
        if (double.TryParse(sMass, out mass))
        {
            if (mass <= 0) { check = false; }
            //entered mass is negative which is not allowed
        }
        else if (sMass != "")
        { check = false; } //entered mass is not a number and is non-
empty

        double cor;
        if (double.TryParse(sCor, out cor))
        {
            if (cor < 0 || cor > 1) { check = false; }
            //checks that entered CoR is between 0 and 1
        }
        else if (sCor != "")
        { check = false; }

        //make sure the rest are of the form (x,y,z) or similar
        string sPosition = positionInput2.Text;
        if (!Regex.IsMatch(sPosition, pattern) && sPosition != "")
        { check = false; }

        string sVelocity = velocityInput2.Text;
        if (!Regex.IsMatch(sVelocity, pattern) && sVelocity != "")
        { check = false; }

        string sAngularVelocity = angularVelocityInput2.Text;
    }
}

```

```

        if (!Regex.IsMatch(sAngularVelocity, pattern) && sAngularVelocity
!= "") 
{ check = false; }

        string sColour = colourInput2.Text;
if (!Regex.IsMatch(sColour, pattern) && sColour != "") 
{ check = false; }
else
{
    if (sColour != "") 
    {
        colour = Global.regExInterp(sColour);
    }
    if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y &&
colour.y <= 255 && 0 <= colour.z && colour.z <= 255)) 
        { check = false; }
}

        string sForce = forceInput2.Text;
if (!Regex.IsMatch(sForce, pattern) && sForce != "") 
{ check = false; }

//if (check == true)
if (check == true)
{
    //Remove validation label (in case it is there)
ValidationLabel3.Visible = false;
ValidationLabel3.Enabled = false;

//Get object from SelectedItems (the DomainUpDown Control)

Entity obj = SelectObject.SelectedItem as Entity;

if (sMass != "") 
{
    obj.rigidBody.mass = mass;
}

if (sCor != "") 
{
    obj.rigidBody.CoR = cor;
}

if (sColour != "") 
{
    obj.mesh.colour = colour;
}

if (sPosition != "") 
{
    Vector position = Global.regExInterp(sPosition);
    //To set position, I must displace it to the origin
    //and then displace it to the desired location
    Global.displaceObject(obj,
Global.vectorTimesScalar(obj.rigidBody.centreOfMass, -1));
    Global.displaceObject(obj, position);
}

if (sVelocity != "") 
{
    Vector velocity = Global.regExInterp(sVelocity);
    obj.rigidBody.velocity = velocity;
}

```

```

        if (sForce != "")
        {
            Vector force = Global.regExInterp(sForce);
            obj.rigidBody.force = force;
        }

        if (sAngularVelocity != "")
        {
            Vector angularVelocity =
Global.regExInterp(sAngularVelocity);
            obj.rigidBody.angularVelocity = angularVelocity;
        }

    }
else
{
    //Display validation label
    ValidationLabel3.Visible = true;
    ValidationLabel3.Enabled = true;
}
}

private void Delete_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem != null)
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;
        Entity obj = SelectObject.SelectedItem as Entity;
        Global.Entities.Remove(obj);
        SelectObject.Items.Remove(obj);
    }
else
{
    ValidationLabel4.Visible = true;
    ValidationLabel4.Enabled = true;
}
}
}
}

```

### ProjectileMotion:

```

using System.Data;
using System.Diagnostics;
using System.Text.RegularExpressions;

namespace ComputerScienceAlevelProject
{
    public partial class ProjectileMotion : Form
    {
        public ProjectileMotion()
        {
            InitializeComponent();
            this.KeyPreview = true;
        }

        private void ProjectileMotion_Paint(object sender, PaintEventArgs e)
        {

```

```

    Graphics g = e.Graphics;
    int formWidth = this.Width;
    int formHeight = this.Height;
    this.DoubleBuffered = true;
    double vertFov = 2 * Math.Atan(formHeight *
Math.Tan(Global.Camera.fov / 2) / formWidth);
    double vertTanFov = Math.Tan(vertFov / 2);
    double tanfov = Math.Tan(Global.Camera.fov / 2);

    //Rendering individual face:
    void renderFace(Face face, Vector colour)
    {
        //Colour determination
        Vector perpVect = face.normal;
        double vectMag = Global.vectMag(perpVect);
        double shade = Global.getBrightnessLevel(perpVect, vectMag);

        Brush brush = new
SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade),
Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade))); //Can be
changed for different colours

        Point point1 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[0],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[0], vertTanFov), formHeight));
        Point point2 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[1],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[1], vertTanFov), formHeight));
        Point point3 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[2],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[2], vertTanFov), formHeight));
        Point point4 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[3],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[3], vertTanFov), formHeight));
        Point[] points = { point1, point2, point3, point4 };
        g.FillPolygon(brush, points);
    }

    void renderGround(Ground ground, Camera camera)
    {
        double tanTheta = camera.direction.y /
Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2));
        double H;
        if (camera.direction.z >= 0)
        {
            H = -tanTheta / vertTanFov; //Calculates H based on the
derived formula
        }
        else
        {
            H = tanTheta / vertTanFov;
        }
        H = Global.denormaliseY(H, formHeight); //denormalises this value
        double shade = Global.getBrightnessLevel(new Vector(0, -1, 0),
1);
        Vector colour = ground.colour; //colour of the ground
        Brush brush = new
SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade),
Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade)));
    }
}

```

```

        Point point1 = new Point(0, Convert.ToInt16(H)); //brush and
points are created so that the shape can be rendered
        Point point2 = new Point(formWidth, Convert.ToInt16(H));
        Point point3 = new Point(formWidth, formHeight);
        Point point4 = new Point(0, formHeight);
        Point[] points = { point1, point2, point3, point4 };
        g.FillPolygon(brush, points);
    }

//Render mesh
void renderMesh(Mesh mesh, Camera camera)
{
    //check all mesh vertices
    List<Vector> vertices = new List<Vector>();
    vertices.Add(mesh.vertices[0]);
    vertices.Add(mesh.vertices[1]);
    vertices.Add(mesh.vertices[2]);
    vertices.Add(mesh.vertices[3]);
    vertices.Add(mesh.vertices[4]);
    vertices.Add(mesh.vertices[5]);
    vertices.Add(mesh.vertices[6]);
    vertices.Add(mesh.vertices[7]);
    bool passCheck = true;
    for (int i = 0; i < 8; i++)
    {
        if (Global.checkRender(camera, vertices[i], vertFov) ==
false)
        {
            passCheck = false;
            break;
        }
    }

    if (passCheck == true)
    {
        int closestPoint = Global.closestPoint(mesh, camera);
        Face topFace = mesh.faces[0];
        Face bottomFace = mesh.faces[1];
        Face leftFace = mesh.faces[2];
        Face rightFace = mesh.faces[3];
        Face frontFace = mesh.faces[4];
        Face backFace = mesh.faces[5];
        switch (closestPoint)
        {
            case 1:
                renderFacesInOrder(topFace, leftFace, frontFace,
camera, mesh);
                break;
            case 2:
                renderFacesInOrder(topFace, rightFace, frontFace,
camera, mesh);
                break;
            case 3:
                renderFacesInOrder(topFace, rightFace, backFace,
camera, mesh);
                break;
            case 4:
                renderFacesInOrder(topFace, leftFace, backFace,
camera, mesh);
                break;
            case 5:

```

```

        renderFacesInOrder(bottomFace, leftFace, frontFace,
camera, mesh);
            break;
        case 6:
            renderFacesInOrder(bottomFace, rightFace, frontFace,
camera, mesh);
            break;
        case 7:
            renderFacesInOrder(bottomFace, rightFace, backFace,
camera, mesh);
            break;
        case 8:
            renderFacesInOrder(bottomFace, leftFace, backFace,
camera, mesh);
            break;
    }
}
}

void renderFacesInOrder(Face face1, Face face2, Face face3, Camera
camera, Mesh mesh)
{
    Vector point1 = Global.averagePoint(face1.vertices[0],
face1.vertices[1], face1.vertices[2], face1.vertices[3]);
    Vector point2 = Global.averagePoint(face2.vertices[0],
face2.vertices[1], face2.vertices[2], face2.vertices[3]);
    Vector point3 = Global.averagePoint(face3.vertices[0],
face3.vertices[1], face3.vertices[2], face3.vertices[3]);
    double dist1 = Global.distBetweenPoints(point1, camera.position);
    double dist2 = Global.distBetweenPoints(point2, camera.position);
    double dist3 = Global.distBetweenPoints(point3, camera.position);
    //distance of each face from the camera

    List<Face> faces = new List<Face>();

    if (dist1 < dist2 && dist1 < dist3) //dist1 is the smallest
    {
        faces.Add(face1);
        if (dist2 < dist3) //dist2 is the next smallest
        {
            faces.Add(face2);
            faces.Add(face3); //Add face2 then face3
        }
        else //dist3 > dist2 so face3 is added, then face2
        {
            faces.Add(face3);
            faces.Add(face2);
        }
    }
    else if (dist2 < dist1 && dist2 < dist3) //dist2 is the smallest
    {
        faces.Add(face2);
        if (dist1 < dist3)
        {
            faces.Add(face1);
            faces.Add(face3);
        }
        else
        {
            faces.Add(face3);
            faces.Add(face1);
        }
    }
}

```

```

        else //dist3 is the smallest
    {
        faces.Add(face3);
        if (dist1 < dist2)
        {
            faces.Add(face1);
            faces.Add(face2);
        }
        else
        {
            faces.Add(face2);
            faces.Add(face1);
        }
    }
    //Render faces in order:
    for (int i = 2; i > -1; i--) //render from furthest to closest
    {
        renderFace(faces[i], mesh.colour);
    }
}

//Define mesh faces
void setMeshFaces(Mesh mesh)
{
    Vector v = new Vector(0, 0, 0);
    Face face1 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[1], mesh.vertices[2], mesh.vertices[3] }, v);
    Face face2 = new Face(new Vector[] { mesh.vertices[4],
mesh.vertices[5], mesh.vertices[6], mesh.vertices[7] }, v);
    Face face3 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[3], mesh.vertices[7], mesh.vertices[4] }, v);
    Face face4 = new Face(new Vector[] { mesh.vertices[1],
mesh.vertices[2], mesh.vertices[6], mesh.vertices[5] }, v);
    Face face5 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[1], mesh.vertices[5], mesh.vertices[4] }, v);
    Face face6 = new Face(new Vector[] { mesh.vertices[2],
mesh.vertices[3], mesh.vertices[7], mesh.vertices[6] }, v);
    face1.normal = Global.faceNormal(face1, mesh);
    face2.normal = Global.faceNormal(face2, mesh);
    face3.normal = Global.faceNormal(face3, mesh);
    face4.normal = Global.faceNormal(face4, mesh);
    face5.normal = Global.faceNormal(face5, mesh);
    face6.normal = Global.faceNormal(face6, mesh);
    Face[] faces = { face1, face2, face3, face4, face5, face6 };
    mesh.faces = faces;
}

//RENDER ALL:
void renderAll(List<Entity> entities, Camera camera)
{
    if (Global.Ground.toggled)
    {
        renderGround(Global.Ground, camera);
    }
    entities = entities.OrderByDescending(x =>
Global.vectMag(Global.projectVector(Global.vectBetweenPoints(x.rigidBody.centreOfMass,
camera.position), camera.direction))).ToList(); //list is order based on
closest point's distance (descending order)
    for (int i = 0; i < entities.Count; i++)
    {
        setMeshFaces(entities[i].mesh);
        renderMesh(entities[i].mesh, camera);
    }
}

```

```

        }

        void gameLoop()
        {
            if (Global.Ground.toggled)
            {
                if (Global.Camera.position.y > Global.Ground.height)
                {
                    Global.Camera.position.y = Global.Ground.height;
                }
            }
            Global.cameraControl(Global.camSpeed * Global.frameTime,
Convert.ToDouble(Global.camSensitivity) / 50 * Math.PI * Global.frameTime,
Global.Camera);
            renderAll(Global.Entities, Global.Camera);
            physics();
        }

        //Physics-----
-----
```

---

```

        void physics()
        {
            //Collision Detection
            List<Tuple<Vector, Entity, Entity>> check =
Global.checkAllCollisions();
            for (int i = 0; i < Global.Entities.Count; i++)
            {
                //Add weight to resultant force.
                Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0,
Global.Entities[i].rigidBody.mass * Global.gravity, 0));

                //Don't need to check if the ground is enabled because it
always is.

                //Collision detection and response with the ground

                List<Vector> points =
Global.groundCollisionDetection(Global.Entities[i].mesh);
                bool groundCollided = false;

                if (Global.Entities[i].rigidBody.groundFlag == false)
                {
                    if (points.Count != 0)
                    {
                        //The flag is false and there was an intersection:
                        //Do collision response and set flag to true.
                        Vector position =
Global.averagePoint(points.ToArray());
                        Global.groundCollisionResponse(Global.Entities[i],
position);
                        groundCollided = true;
                        Global.Entities[i].rigidBody.groundFlag = true;
                    }
                    else
                    {
                        //The flag was false and there was no intersection:
                        //Do nothing
                    }
                }
            }
        }
    
```

```

        {
            if (points.Count != 0)
            {
                //The flag is true and there was an intersection:
                groundCollided = true;
            }
            else
            {
                //The flag was true but there was no intersection:
                //Set flag to false:
                Global.Entities[i].rigidBody.groundFlag = false;
            }
        }

        //Find with which object and normal (if any) this object
        collided with each frame and apply normal reaction.
        //This is done regardless of collision flags.
        Vector reaction = new Vector(0, 0, 0);
        if (groundCollided)
        {
            reaction = Global.vectorAddVector(reaction, new Vector(0,
-Global.Entities[i].rigidBody.force.y, 0));
        }

        Global.Entities[i].rigidBody.force =
        Global.vectorAddVector(Global.Entities[i].rigidBody.force, reaction);
        Global.updateAcceleration(Global.Entities[i]);
        Global.Entities[i].rigidBody.force =
        Global.vectorAddVector(Global.Entities[i].rigidBody.force,
        Global.vectorTimesScalar(reaction, -1));

        Global.updateVelocity(Global.Entities[i]);
        Global.displaceObject(Global.Entities[i],
        Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity,
        Global.frameTime));

        Global.updateAngularVelocity(Global.Entities[i]);
        Global.rotateObject(Global.Entities[i],
        Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity,
        Global.frameTime));

        //Take away gravity
        Global.Entities[i].rigidBody.force =
        Global.vectorAddVector(Global.Entities[i].rigidBody.force, new Vector(0, -
        Global.Entities[i].rigidBody.mass * Global.gravity, 0));
    }
}

//FrameRate-----
-----
```

---

```

if (Global.paused == false)
{
    this.BackColor = Global.bgColour;
    for (int i = 0; i < this.Controls.Count; i++)
    {
        this.Controls[i].Visible = false;
        this.Controls[i].Enabled = false;
    }
    Stopwatch timer = new Stopwatch();
}
```

```

        timer.Start();
        gameLoop();
        timer.Stop();
        Global.frameTime = timer.Elapsed.TotalSeconds;

        if (Global.frameTime < 1.0 / Global.fps)
        {
            Stopwatch timer2 = new Stopwatch();
            timer2.Start();

            while (timer2.Elapsed.TotalSeconds < 1.0 / Global.fps -
Global.frameTime)
            {

            }
            timer2.Stop();
            Global.frameTime = 1.0 / Global.fps;
        }
    }
    else
    {
        this.BackColor = Color.Gray;
        for (int i = 0; i < this.Controls.Count; i++)
        {
            if (Controls[i] != ValidationLabel && Controls[i] != ValidationLabel2 && Controls[i] != ValidationLabel3 && Controls[i] != ValidationLabel4)
            {
                this.Controls[i].Visible = true;
                this.Controls[i].Enabled = true;
            }
        }
        Invalidate();
    }
}

private void ProjectileMotion_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = true;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = true;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = true;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = true;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = true;
    }
    if (e.KeyCode == Keys.ShiftKey)
    {
        Global.ShiftHeld = true;
    }
    if (e.KeyCode == Keys.E)
}

```

```

        {
            Global.EHeld = true;
        }
        if (e.KeyCode == Keys.Q)
        {
            Global.QHeld = true;
        }
        if (e.KeyCode == Keys.Up)
        {
            Global.UpHeld = true;
        }
        if (e.KeyCode == Keys.Down)
        {
            Global.DownHeld = true;
        }
        if (e.KeyCode == Keys.Left)
        {
            Global.LeftHeld = true;
        }
        if (e.KeyCode == Keys.Right)
        {
            Global.RightHeld = true;
        }
        if (e.KeyCode == Keys.Escape)
        {
            Global.paused = !Global.paused;
            massInput2.Clear();
            corInput2.Clear();
            colourInput2.Clear();
            forceInput2.Clear();
        }
        Invalidate();
    }

    private void ProjectileMotion_KeyUp(object sender, KeyEventArgs e)
    {
        if (e.KeyCode == Keys.W)
        {
            Global.Wheld = false;
        }
        if (e.KeyCode == Keys.A)
        {
            Global.AHeld = false;
        }
        if (e.KeyCode == Keys.S)
        {
            Global.SHeld = false;
        }
        if (e.KeyCode == Keys.D)
        {
            Global.DHeld = false;
        }
        if (e.KeyCode == Keys.Space)
        {
            Global.SpaceHeld = false;
        }
        if (e.KeyCode == Keys.ShiftKey)
        {
            Global.ShiftHeld = false;
        }
        if (e.KeyCode == Keys.E)
        {
            Global.EHeld = false;
        }
    }
}

```

```

        }

        if (e.KeyCode == Keys.Q)
        {
            Global.QHeld = false;
        }
        if (e.KeyCode == Keys.Up)
        {
            Global.UpHeld = false;
        }
        if (e.KeyCode == Keys.Down)
        {
            Global.DownHeld = false;
        }
        if (e.KeyCode == Keys.Left)
        {
            Global.LeftHeld = false;
        }
        if (e.KeyCode == Keys.Right)
        {
            Global.RightHeld = false;
        }
        Invalidate();
    }

    private void ProjectileMotion_Load(object sender, EventArgs e)
    {
        Global.Ground.toggled = true;
        Global.paused = false;
        Global.Entities = new List<Entity>();
        Global.collisionFlags = new List<Tuple<bool, Entity, Entity>>();
        Global.Camera = new Camera(new Vector(-1, 0, 0), new Vector(1, 0.01,
0.01), 7 * Math.PI / 18);
        Global.gravity = 9.81;
    }

    private void Open_Settings_Click(object sender, EventArgs e)
    {
        Settings settings = new Settings();
        settings.ShowDialog();
    }

    private void Exit_Click(object sender, EventArgs e)
    {
        this.Close();
        Global.main_menu.Show();
    }

    private void Create_Click(object sender, EventArgs e)
    {
        //Validate user inputs
        bool check = true;
        string pattern = @"^(\()?( -)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
        string sMass = massInput.Text;
        string sCor = corInput.Text;
        string sHeight = heightInput.Text;
        string sSpeed = speedInput.Text;
        string sAngle = angleInput.Text;
        Vector colour = new Vector(0, 0, 0);

        //Scalars
        if (double.TryParse(sMass, out double mass))

```

```

{
    if (mass <= 0) { check = false; }
    //entered mass is negative which is not allowed
}
else { check = false; }//entered mass is not a number

if (double.TryParse(sCor, out double cor))
{
    if (cor < 0 || cor > 1) { check = false; }
    //checks that entered CoR is between 0 and 1
}
else { check = false; }

if (double.TryParse(sHeight, out double height))
{
    if (height <= 0) { check = false; }
    //entered height is negative which is not allowed
}
else { check = false; }//entered height is not a number

if (double.TryParse(sSpeed, out double speed))
{
    if (speed < 0) { check = false; }
}
else { check = false; }

if (double.TryParse(sAngle, out double angle))
{
    if (!(-90 <= angle && angle <= 90)) { check = false; }
}
else { check = false; }

//Vectors
string sColour = colourInput.Text;
if (!Regex.IsMatch(sColour, pattern))
{ check = false; }
else
{
    colour = Global.regExInterp(sColour);
    if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y &&
colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
        { check = false; }
}
string sForce = forceInput.Text;
if (!Regex.IsMatch(sForce, pattern))
{ check = false; }

if (check == true)
{
    ValidationLabel.Visible = false;
    ValidationLabel.Enabled = false;

    //Assign object creation variables
    Vector position = new Vector(0, 10 - height, 0);
    angle = angle * Math.PI / 180;
    double Ux = speed * Math.Cos(angle);
    double Uy = -speed * Math.Sin(angle);
    Vector velocity = new Vector(0, Uy, Ux);
    Vector force = Global.regExInterp(sForce);

    //Create Object
    Vector v1 = new Vector(-0.5, -0.5, -0.5);
    Vector v2 = new Vector(-0.5, -0.5, 0.5);
}

```

```

        Vector v3 = new Vector(0.5, -0.5, 0.5);
        Vector v4 = new Vector(0.5, -0.5, -0.5);
        Vector v5 = new Vector(-0.5, 0.5, -0.5);
        Vector v6 = new Vector(-0.5, 0.5, 0.5);
        Vector v7 = new Vector(0.5, 0.5, 0.5);
        Vector v8 = new Vector(0.5, 0.5, -0.5);
        Vector[] vertices = { v1, v2, v3, v4, v5, v6, v7, v8 };

        //Create mesh:
        Mesh mesh = new Mesh(vertices, new Vector(1, 1, 1), colour);

        //Create Tensor and RigidBody
        Vector v0 = new Vector(0, 0, 0); //zero vector
        Vector tensor = new Vector(6 / mass, 6 / mass, 6 / mass);
        RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, v0,
        force, cor, tensor, false);

        //Create name:
        string sName = nameInput.Text;
        string name = "";
        if (sName == "")
        {
            name = "Projectile" + Global.Entities.Count;
        }
        else
        {
            name = sName;
        }

        //Create object
        Entity obj = new Entity(mesh, rb, name);
        Global.displaceObject(obj, position);
        Global.Entities.Add(obj);
        SelectObject.Items.Add(obj);
    }
    else
    {
        //Display validation label
        ValidationLabel.Visible = true;
        ValidationLabel.Enabled = true;
    }
}

private void MakeChanges_Click(object sender, EventArgs e)
{
    bool check = true;
    string pattern = @"^(\()?( -)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
    string sGrav = gravityInput.Text;
    string sColour = bgColourInput.Text;
    double grav;
    Vector colour = new Vector(0, 0, 0);

    if (sGrav == "") //if sGrav is empty it should pass the test
immediately
    {
        grav = 9.81;
    }
    else if (!double.TryParse(sGrav, out grav))
    {
        check = false;
    }
}

```

```

//make sure colour is of the correct form or is an empty string
else if (sColour != "" && !Regex.IsMatch(sColour, pattern))
{ check = false; }

//if (check == true)
if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel2.Visible = false;
    ValidationLabel2.Enabled = false;

    if (sColour != "")
    {
        colour = Global.regExInterp(sColour);
        //apply background colour
        Global.bgColour = Color.FromArgb(Convert.ToInt16(colour.X),
Convert.ToInt16(colour.Y), Convert.ToInt16(colour.Z));
    }

    //apply gravity
    Global.gravity = grav;
}
else
{
    //Display validation label
    ValidationLabel2.Visible = true;
    ValidationLabel2.Enabled = true;
}
}

private void Alter_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem == null)
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
    else
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;

        //Validate user inputs
        bool check = true;
        string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?(\))?$";
        string sMass = massInput2.Text;
        string sCor = corInput2.Text;
        Vector colour = new Vector(0, 0, 0);

        double mass;
        if (double.TryParse(sMass, out mass))
        {
            if (mass <= 0) { check = false; }
            //entered mass is negative which is not allowed
        }
        else if (sMass != "")
        { check = false; } //entered mass is not a number and is non-
empty

        double cor;
        if (double.TryParse(sCor, out cor))

```

```

{
    if (cor < 0 || cor > 1) { check = false; }
    //checks that entered CoR is between 0 and 1
}
else if (sCor != "")
{ check = false; }

//make sure the rest are of the form (x,y,z) or similar
string sPosition = positionInput2.Text;
if (!Regex.IsMatch(sPosition, pattern) && sPosition != "")
{ check = false; }

string sVelocity = velocityInput2.Text;
if (!Regex.IsMatch(sVelocity, pattern) && sVelocity != "")
{ check = false; }

string sAngularVelocity = angularVelocityInput2.Text;
if (!Regex.IsMatch(sAngularVelocity, pattern) && sAngularVelocity
!= "")
{ check = false; }

string sColour = colourInput2.Text;
if (!Regex.IsMatch(sColour, pattern) && sColour != "")
{ check = false; }
else
{
    if (sColour != "")
    {
        colour = Global.regExInterp(sColour);
    }
    if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y &&
colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
        { check = false; }
}

string sForce = forceInput2.Text;
if (!Regex.IsMatch(sForce, pattern) && sForce != "")
{ check = false; }

if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel3.Visible = false;
    ValidationLabel3.Enabled = false;

    //Get object from SelectedItems (the DomainUpDown Control)

    Entity obj = SelectObject.SelectedItem as Entity;

    if (sMass != "")
    {
        obj.rigidBody.mass = mass;
    }

    if (sCor != "")
    {
        obj.rigidBody.CoR = cor;
    }

    if (sColour != "")
    {
        obj.mesh.colour = colour;
    }
}

```

```

        if (sPosition != "")
        {
            Vector position = Global.regExInterp(sPosition);
            //To set position, I must displace it to the origin
            //and then displace it to the desired location
            Global.displaceObject(obj,
Global.vectorTimesScalar(obj.rigidBody.centreOfMass, -1));
            Global.displaceObject(obj, position);
        }

        if (sVelocity != "")
        {
            Vector velocity = Global.regExInterp(sVelocity);
            obj.rigidBody.velocity = velocity;
        }

        if (sForce != "")
        {
            Vector force = Global.regExInterp(sForce);
            obj.rigidBody.force = force;
        }

        if (sAngularVelocity != "")
        {
            Vector angularVelocity =
Global.regExInterp(sAngularVelocity);
            obj.rigidBody.angularVelocity = angularVelocity;
        }

    }
    else
    {
        //Display validation label
        ValidationLabel3.Visible = true;
        ValidationLabel3.Enabled = true;
    }
}

private void Delete_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem != null)
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;
        Entity obj = SelectObject.SelectedItem as Entity;
        Global.Entities.Remove(obj);
        SelectObject.Items.Remove(obj);
    }
    else
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
}
}

```

Orbit:

```
using System.Data;
using System.Diagnostics;
using System.Text.RegularExpressions;

namespace ComputerScienceAlevelProject
{
    public partial class Orbit : Form
    {
        public Orbit()
        {
            InitializeComponent();
            this.KeyPreview = true;
        }

        private void Orbit_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            int formWidth = this.Width;
            int formHeight = this.Height;
            this.DoubleBuffered = true;
            double vertFov = 2 * Math.Atan(formHeight *
Math.Tan(Global.Camera.fov / 2) / formWidth);
            double vertTanFov = Math.Tan(vertFov / 2);
            double tanfov = Math.Tan(Global.Camera.fov / 2);

            //Rendering individual face:
            void renderFace(Face face, Vector colour)
            {
                //Colour determination
                Vector perpVect = face.normal;
                double vectMag = Global.vectMag(perpVect);
                double shade = Global.getBrightnessLevel(perpVect, vectMag);
                Brush brush = new
SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade),
Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade))); //Can be
changed for different colours

                Point point1 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[0],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[0], vertTanFov), formHeight));
                Point point2 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[1],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[1], vertTanFov), formHeight));
                Point point3 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[2],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[2], vertTanFov), formHeight));
                Point point4 = new
Point(Global.denormaliseX(Global.calcXCoord(Global.Camera, face.vertices[3],
tanfov), formWidth), Global.denormaliseY(Global.calcYCoord(Global.Camera,
face.vertices[3], vertTanFov), formHeight));
                Point[] points = { point1, point2, point3, point4 };
                g.FillPolygon(brush, points);
            }

            void renderGround(Ground ground, Camera camera)
            {
                double tanTheta = camera.direction.y /
Math.Sqrt(Math.Pow(camera.direction.x, 2) + Math.Pow(camera.direction.z, 2));
            }
        }
    }
}
```

```

        double H;
        if (camera.direction.z >= 0)
        {
            H = -tanTheta / vertTanFov; //Calculates H based on the
derived formula
        }
        else
        {
            H = tanTheta / vertTanFov;
        }
        H = Global.denormaliseY(H, formHeight); //denormalises this value
        double shade = Global.getBrightnessLevel(new Vector(0, -1, 0),
1);
        Vector colour = ground.colour; //colour of the ground
        Brush brush = new
SolidBrush(Color.FromArgb(Convert.ToInt16(colour.x * shade),
Convert.ToInt16(colour.y * shade), Convert.ToInt16(colour.z * shade)));
        Point point1 = new Point(0, Convert.ToInt16(H)); //brush and
points are created so that the shape can be rendered
        Point point2 = new Point(formWidth, Convert.ToInt16(H));
        Point point3 = new Point(formWidth, formHeight);
        Point point4 = new Point(0, formHeight);
        Point[] points = { point1, point2, point3, point4 };
        g.FillPolygon(brush, points);
    }

//Render mesh
void renderMesh(Mesh mesh, Camera camera)
{
    //check all mesh vertices
    List<Vector> vertices = new List<Vector>();
    vertices.Add(mesh.vertices[0]);
    vertices.Add(mesh.vertices[1]);
    vertices.Add(mesh.vertices[2]);
    vertices.Add(mesh.vertices[3]);
    vertices.Add(mesh.vertices[4]);
    vertices.Add(mesh.vertices[5]);
    vertices.Add(mesh.vertices[6]);
    vertices.Add(mesh.vertices[7]);
    bool passCheck = true;
    for (int i = 0; i < 8; i++)
    {
        if (Global.checkRender(camera, vertices[i], vertFov) ==
false)
        {
            passCheck = false;
            break;
        }
    }

    if (passCheck == true)
    {
        int closestPoint = Global.closestPoint(mesh, camera);
        Face topFace = mesh.faces[0];
        Face bottomFace = mesh.faces[1];
        Face leftFace = mesh.faces[2];
        Face rightFace = mesh.faces[3];
        Face frontFace = mesh.faces[4];
        Face backFace = mesh.faces[5];
        switch (closestPoint)
        {
            case 1:

```

```

        renderFacesInOrder(topFace, leftFace, frontFace,
camera, mesh);
        break;
    case 2:
        renderFacesInOrder(topFace, rightFace, frontFace,
camera, mesh);
        break;
    case 3:
        renderFacesInOrder(topFace, rightFace, backFace,
camera, mesh);
        break;
    case 4:
        renderFacesInOrder(topFace, leftFace, backFace,
camera, mesh);
        break;
    case 5:
        renderFacesInOrder(bottomFace, leftFace, frontFace,
camera, mesh);
        break;
    case 6:
        renderFacesInOrder(bottomFace, rightFace, frontFace,
camera, mesh);
        break;
    case 7:
        renderFacesInOrder(bottomFace, rightFace, backFace,
camera, mesh);
        break;
    case 8:
        renderFacesInOrder(bottomFace, leftFace, backFace,
camera, mesh);
        break;
    }
}
}

void renderFacesInOrder(Face face1, Face face2, Face face3, Camera
camera, Mesh mesh)
{
    Vector point1 = Global.averagePoint(face1.vertices[0],
face1.vertices[1], face1.vertices[2], face1.vertices[3]);
    Vector point2 = Global.averagePoint(face2.vertices[0],
face2.vertices[1], face2.vertices[2], face2.vertices[3]);
    Vector point3 = Global.averagePoint(face3.vertices[0],
face3.vertices[1], face3.vertices[2], face3.vertices[3]);
    double dist1 = Global.distBetweenPoints(point1, camera.position);
    double dist2 = Global.distBetweenPoints(point2, camera.position);
    double dist3 = Global.distBetweenPoints(point3, camera.position);
    //distance of each face from the camera

    List<Face> faces = new List<Face>();

    if (dist1 < dist2 && dist1 < dist3) //dist1 is the smallest
    {
        faces.Add(face1);
        if (dist2 < dist3) //dist2 is the next smallest
        {
            faces.Add(face2);
            faces.Add(face3); //Add face2 then face3
        }
        else //dist3 > dist2 so face3 is added, then face2
        {
            faces.Add(face3);
            faces.Add(face2);
        }
    }
}

```

```

        }
    }
    else if (dist2 < dist1 && dist2 < dist3) //dist2 is the smallest
    {
        faces.Add(face2);
        if (dist1 < dist3)
        {
            faces.Add(face1);
            faces.Add(face3);
        }
        else
        {
            faces.Add(face3);
            faces.Add(face1);
        }
    }
    else //dist3 is the smallest
    {
        faces.Add(face3);
        if (dist1 < dist2)
        {
            faces.Add(face1);
            faces.Add(face2);
        }
        else
        {
            faces.Add(face2);
            faces.Add(face1);
        }
    }
    //Render faces in order:
    for (int i = 2; i > -1; i--) //render from furthest to closest
    {
        renderFace(faces[i], mesh.colour);
    }
}

//Define mesh faces
void setMeshFaces(Mesh mesh)
{
    Vector v = new Vector(0, 0, 0);
    Face face1 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[1], mesh.vertices[2], mesh.vertices[3] }, v);
    Face face2 = new Face(new Vector[] { mesh.vertices[4],
mesh.vertices[5], mesh.vertices[6], mesh.vertices[7] }, v);
    Face face3 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[3], mesh.vertices[7], mesh.vertices[4] }, v);
    Face face4 = new Face(new Vector[] { mesh.vertices[1],
mesh.vertices[2], mesh.vertices[6], mesh.vertices[5] }, v);
    Face face5 = new Face(new Vector[] { mesh.vertices[0],
mesh.vertices[1], mesh.vertices[5], mesh.vertices[4] }, v);
    Face face6 = new Face(new Vector[] { mesh.vertices[2],
mesh.vertices[3], mesh.vertices[7], mesh.vertices[6] }, v);
    face1.normal = Global.faceNormal(face1, mesh);
    face2.normal = Global.faceNormal(face2, mesh);
    face3.normal = Global.faceNormal(face3, mesh);
    face4.normal = Global.faceNormal(face4, mesh);
    face5.normal = Global.faceNormal(face5, mesh);
    face6.normal = Global.faceNormal(face6, mesh);
    Face[] faces = { face1, face2, face3, face4, face5, face6 };
    mesh.faces = faces;
}

```

```

//RENDER ALL:
void renderAll(List<Entity> entities, Camera camera)
{
    if (Global.Ground.toggled)
    {
        renderGround(Global.Ground, camera);
    }
    entities = entities.OrderByDescending(x =>
Global.vectMag(Global.projectVector(Global.vectBetweenPoints(x.rigidBody.centreOf
Mass, camera.position), camera.direction))).ToList(); //list is order based on
closest point's distance (descending order)
    for (int i = 0; i < entities.Count; i++)
    {
        setMeshFaces(entities[i].mesh);
        renderMesh(entities[i].mesh, camera);
    }
}

void gameLoop()
{
    if (Global.Ground.toggled)
    {
        if (Global.Camera.position.y > Global.Ground.height)
        {
            Global.Camera.position.y = Global.Ground.height;
        }
        Global.cameraControl(Global.camSpeed * Global.frameTime,
Convert.ToDouble(Global.camSensitivity) / 50 * Math.PI * Global.frameTime,
Global.Camera);
        renderAll(Global.Entities, Global.Camera);
        physics();
    }
}

//Physics-----
-----

void physics()
{
    for (int i = 0; i < Global.Entities.Count; i++)
    {
        //calculate the force of gravity from all other objects
        Vector gravity = new Vector(0, 0, 0);
        for (int j = 0; j < Global.Entities.Count; j++)
        {
            if (i != j) //Checking every *other* object
            {
                Vector vectBetweenObjects =
Global.vectBetweenPoints(Global.Entities[i].rigidBody.centreOfMass,
Global.Entities[j].rigidBody.centreOfMass);
                double r = Global.vectMag(vectBetweenObjects);
                if (r > Global.Entities[i].mesh.dimensions.x / 2 +
Global.Entities[i].mesh.dimensions.x / 2)
                {
                    Vector newGrav =
Global.vectorTimesScalar(vectBetweenObjects, Global.gravConst *
Global.Entities[i].rigidBody.mass * Global.Entities[j].rigidBody.mass /
Math.Pow(r, 3));
                    gravity = Global.vectorAddVector(gravity,
newGrav);
                }
            }
        }
    }
}

```

```

        }

        //Add this force:
        Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force, gravity);

        Global.updateAcceleration(Global.Entities[i]);

        Global.updateVelocity(Global.Entities[i]);
        Global.displaceObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.velocity,
Global.frameTime));

        Global.updateAngularVelocity(Global.Entities[i]);
        Global.rotateObject(Global.Entities[i],
Global.vectorTimesScalar(Global.Entities[i].rigidBody.angularVelocity,
Global.frameTime));

        //Take away gravitational force
        Global.Entities[i].rigidBody.force =
Global.vectorAddVector(Global.Entities[i].rigidBody.force,
Global.vectorTimesScalar(gravity, -1));
    }
}

//FrameRate-----
-----
```

```

if (Global.paused == false)
{
    this.BackColor = Global.bgColour;
    for (int i = 0; i < this.Controls.Count; i++)
    {
        this.Controls[i].Visible = false;
        this.Controls[i].Enabled = false;
    }
    Stopwatch timer = new Stopwatch();
    timer.Start();
    gameLoop();
    timer.Stop();
    Global.frameTime = timer.Elapsed.TotalSeconds;

    if (Global.frameTime < 1.0 / Global.fps)
    {
        Stopwatch timer2 = new Stopwatch();
        timer2.Start();

        while (timer2.Elapsed.TotalSeconds < 1.0 / Global.fps -
Global.frameTime)
        {
            }

        }
        timer2.Stop();
        Global.frameTime = 1.0 / Global.fps;
    }
}
else
{
    this.BackColor = Color.Gray;
    for (int i = 0; i < this.Controls.Count; i++)
```

```
        {
            if (Controls[i] != ValidationLabel && Controls[i] != ValidationLabel2 && Controls[i] != ValidationLabel3 && Controls[i] != ValidationLabel4)
            {
                this.Controls[i].Visible = true;
                this.Controls[i].Enabled = true;
            }
        }
        Invalidate();
    }

private void Orbit_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = true;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = true;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = true;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = true;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = true;
    }
    if (e.KeyCode == Keys.ShiftKey)
    {
        Global.ShiftHeld = true;
    }
    if (e.KeyCode == Keys.E)
    {
        Global.EHeld = true;
    }
    if (e.KeyCode == Keys.Q)
    {
        Global.QHeld = true;
    }
    if (e.KeyCode == Keys.Up)
    {
        Global.UpHeld = true;
    }
    if (e.KeyCode == Keys.Down)
    {
        Global.DownHeld = true;
    }
    if (e.KeyCode == Keys.Left)
    {
        Global.LeftHeld = true;
    }
    if (e.KeyCode == Keys.Right)
    {
        Global.RightHeld = true;
    }
}
```

```

        if (e.KeyCode == Keys.Escape)
    {
        Global.paused = !Global.paused;
        massInput2.Clear();
        positionInput2.Clear();
        velocityInput2.Clear();
        colourInput2.Clear();
    }
    Invalidate();
}

private void Orbit_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.W)
    {
        Global.WHeld = false;
    }
    if (e.KeyCode == Keys.A)
    {
        Global.AHeld = false;
    }
    if (e.KeyCode == Keys.S)
    {
        Global.SHeld = false;
    }
    if (e.KeyCode == Keys.D)
    {
        Global.DHeld = false;
    }
    if (e.KeyCode == Keys.Space)
    {
        Global.SpaceHeld = false;
    }
    if (e.KeyCode == Keys.ShiftKey)
    {
        Global.ShiftHeld = false;
    }
    if (e.KeyCode == Keys.E)
    {
        Global.EHeld = false;
    }
    if (e.KeyCode == Keys.Q)
    {
        Global.QHeld = false;
    }
    if (e.KeyCode == Keys.Up)
    {
        Global.UpHeld = false;
    }
    if (e.KeyCode == Keys.Down)
    {
        Global.DownHeld = false;
    }
    if (e.KeyCode == Keys.Left)
    {
        Global.LeftHeld = false;
    }
    if (e.KeyCode == Keys.Right)
    {
        Global.RightHeld = false;
    }
    Invalidate();
}

```

```

private void Orbit_Load(object sender, EventArgs e)
{
    Global.Ground.toggled = false;
    Global.paused = false;
    Global.Entities = new List<Entity>();
    Global.collisionFlags = new List<Tuple<bool, Entity, Entity>>();
    Global.Camera = new Camera(new Vector(-1, 0, 0), new Vector(1, 0.01,
0.01), 7 * Math.PI / 18);
    Global.gravity = 0;
}

private void Open_Settings_Click(object sender, EventArgs e)
{
    Settings settings = new Settings();
    settings.ShowDialog();
}
private void Exit_Click(object sender, EventArgs e)
{
    this.Close();
    Global.main_menu.Show();
}

private void Create_Click(object sender, EventArgs e)
{
    //Validate user inputs
    bool check = true;
    string pattern = @"^(\()?( -)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?(\))?$";
    string sMass = massInput.Text;
    string sRadius = radiusInput.Text;
    Vector colour = new Vector(0, 0, 0);
    if (double.TryParse(sMass, out double mass))
    {
        if (mass <= 0) { check = false; }
        //entered mass is negative which is not allowed
    }
    else { check = false; } //entered mass is not a number

    if (double.TryParse(sRadius, out double radius))
    {
        if (radius <= 0) { check = false; }
        //entered radius is negative which is not allowed
    }
    else { check = false; } //entered radius is not a number

    //make sure the rest are of the form (x,y,z) or similar
    string sPosition = positionInput.Text;
    if (!Regex.IsMatch(sPosition, pattern))
    { check = false; }
    string sVelocity = velocityInput.Text;
    if (!Regex.IsMatch(sVelocity, pattern))
    { check = false; }
    string sColour = colourInput.Text;
    if (!Regex.IsMatch(sColour, pattern))
    { check = false; }
    else
    {
        colour = Global.regExInterp(sColour);
        if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y &&
colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
        { check = false; }
    }
}

```

```

if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel.Visible = false;
    ValidationLabel.Enabled = false;

    //Assign object creation variables
    Vector position = Global.regExInterp(sPosition);
    Vector velocity = Global.regExInterp(sVelocity);
    Vector dimensions = new Vector(2 * radius, 2 * radius, 2 *
radius);

    //Create Object
    //Create vertices
    double depth = dimensions.x / 2;
    double height = dimensions.y / 2;
    double width = dimensions.z / 2;
    //I have divided these by 2 because I need the distance
    //from centre to face, not face to face.
    Vector v1 = new Vector(-depth, -height, -width);
    Vector v2 = new Vector(-depth, -height, width);
    Vector v3 = new Vector(depth, -height, width);
    Vector v4 = new Vector(depth, -height, -width);
    Vector v5 = new Vector(-depth, height, -width);
    Vector v6 = new Vector(-depth, height, width);
    Vector v7 = new Vector(depth, height, width);
    Vector v8 = new Vector(depth, height, -width);
    Vector[] vertices = { v1, v2, v3, v4, v5, v6, v7, v8 };

    //Create mesh:
    Mesh mesh = new Mesh(vertices, dimensions, colour);

    //Create Tensor and RigidBody
    Vector v0 = new Vector(0, 0, 0); //zero vector
    Vector tensor = Global.vectorTimesScalar(new Vector(1 / (height *
height + width * width), 1 / (depth * depth + width * width), 1 / (depth * depth +
height * height)), 12 / mass);
    RigidBody rb = new RigidBody(mass, v0, velocity, v0, v0, v0, v0,
0, tensor, false);

    //Create name:
    string sName = nameInput.Text;
    string name = "";
    if (sName == "")
    {
        name = "Cube" + Global.Entities.Count;
    }
    else
    {
        name = sName;
    }

    //Create object
    Entity obj = new Entity(mesh, rb, name);
    Global.displaceObject(obj, position);
    Global.Entities.Add(obj);
    SelectObject.Items.Add(obj);
}
else
{
    //Display validation label
}

```

```

        ValidationLabel.Visible = true;
        ValidationLabel.Enabled = true;
    }
}

private void MakeChanges_Click(object sender, EventArgs e)
{
    bool check = true;
    string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
    string sGrav = gravityInput.Text;
    string sColour = bgColourInput.Text;
    double grav;
    Vector colour = new Vector(0, 0, 0);

    if (sGrav == "") //if sGrav is empty it should pass the test
immediately
    {
        grav = 6.67*Math.Pow(10, -11);
    }
    else if (!double.TryParse(sGrav, out grav))
    {
        check = false;
    }

//make sure colour is of the correct form or is an empty string
else if (sColour != "" && !Regex.IsMatch(sColour, pattern))
{ check = false; }

//if (check == true)
if (check == true)
{
    //Remove validation label (in case it is there)
    ValidationLabel2.Visible = false;
    ValidationLabel2.Enabled = false;

    if (sColour != "")
    {
        colour = Global.regExInterp(sColour);
        //apply background colour
        Global.bgColour = Color.FromArgb(Convert.ToInt16(colour.x),
Convert.ToInt16(colour.y), Convert.ToInt16(colour.z));
    }

    //apply gravity
    Global.gravConst = grav;
}
else
{
    //Display validation label
    ValidationLabel2.Visible = true;
    ValidationLabel2.Enabled = true;
}
}

private void Alter_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem == null)
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
    else
    {

```

```

ValidationLabel4.Visible = false;
ValidationLabel4.Enabled = false;

//Validate user inputs
bool check = true;
string pattern = @"^(\()?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,( +)?(-)?(\d+)(.\d+)?,$";
string sMass = massInput2.Text;
Vector colour = new Vector(0, 0, 0);

double mass;
if (double.TryParse(sMass, out mass))
{
    if (mass <= 0) { check = false; }
    //entered mass is negative which is not allowed
}
else if (sMass != "")
{ check = false; } //entered mass is not a number and is non-empty

//make sure the rest are of the form (x,y,z) or similar
string sPosition = positionInput2.Text;
if (!Regex.IsMatch(sPosition, pattern) && sPosition != "")
{ check = false; }

string sVelocity = velocityInput2.Text;
if (!Regex.IsMatch(sVelocity, pattern) && sVelocity != "")
{ check = false; }

string sColour = colourInput2.Text;
if (!Regex.IsMatch(sColour, pattern) && sColour != "")
{ check = false; }
else
{
    if (sColour != "")
    {
        colour = Global.regExInterp(sColour);
    }
    if (!(0 <= colour.x && colour.x <= 255 && 0 <= colour.y && colour.y <= 255 && 0 <= colour.z && colour.z <= 255))
    { check = false; }
}

//if (check == true)
if (check == true)
{
    //Remove validation label (in case it is there)
ValidationLabel3.Visible = false;
ValidationLabel3.Enabled = false;

//Get object from SelectedItems (the DomainUpDown Control)

Entity obj = SelectObject.SelectedItem as Entity;

if (sMass != "")
{
    obj.rigidBody.mass = mass;
}

if (sColour != "")
{
    obj.mesh.colour = colour;
}

```

```

        if (sPosition != "")
        {
            Vector position = Global.regExInterp(sPosition);
            //To set position, I must displace it to the origin
            //and then displace it to the desired location
            Global.displaceObject(obj,
Global.vectorTimesScalar(obj.rigidBody.centreOfMass, -1));
            Global.displaceObject(obj, position);
        }

        if (sVelocity != "")
        {
            Vector velocity = Global.regExInterp(sVelocity);
            obj.rigidBody.velocity = velocity;
        }
    }
    else
    {
        //Display validation label
        ValidationLabel3.Visible = true;
        ValidationLabel3.Enabled = true;
    }
}

private void Delete_Click(object sender, EventArgs e)
{
    if (SelectObject.SelectedItem != null)
    {
        ValidationLabel4.Visible = false;
        ValidationLabel4.Enabled = false;
        Entity obj = SelectObject.SelectedItem as Entity;
        Global.Entities.Remove(obj);
        SelectObject.Items.Remove(obj);
    }
    else
    {
        ValidationLabel4.Visible = true;
        ValidationLabel4.Enabled = true;
    }
}
}

```