Physim

Computer Science Non-Examination Assessment

Alex Bury

Contents

Section 1 - Analysis

1.1 Background to project

In physics classes I noted the use of outdated and unreliable simulation software for virtual demonstrations used by the teachers to present new ideas to students. Often, the physics simulations are tedious for the teacher to navigate and are lacking in features so have to be supported by other means such as graphs sketched by the teacher.

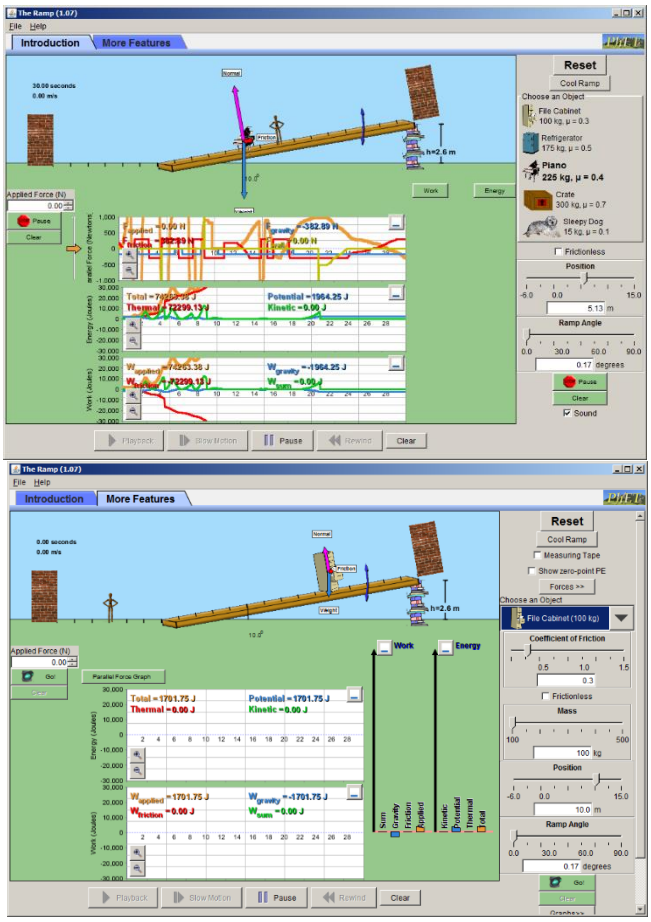
I plan to create a user-friendly solution which is appropriate for the simplest and the most complicated of demonstrations by designing an intuitive and simple user interface system and enabling the teachers to use the software to bounce ideas back and forth from the students. I plan to implement this in such a way so that the program is very versatile in its abilities and to ensure that this versatility doesn't detract from the intuitive design and easy to understand nature.

I'd like to include a 'Simulation Builder' to allow teachers to construct their own simulation and save it to ensure that the simulation is entirely in context with their lesson and allow teachers to pick and choose which elements of the simulation (for example, Velocity of an object) they want to be displayed on a graph.

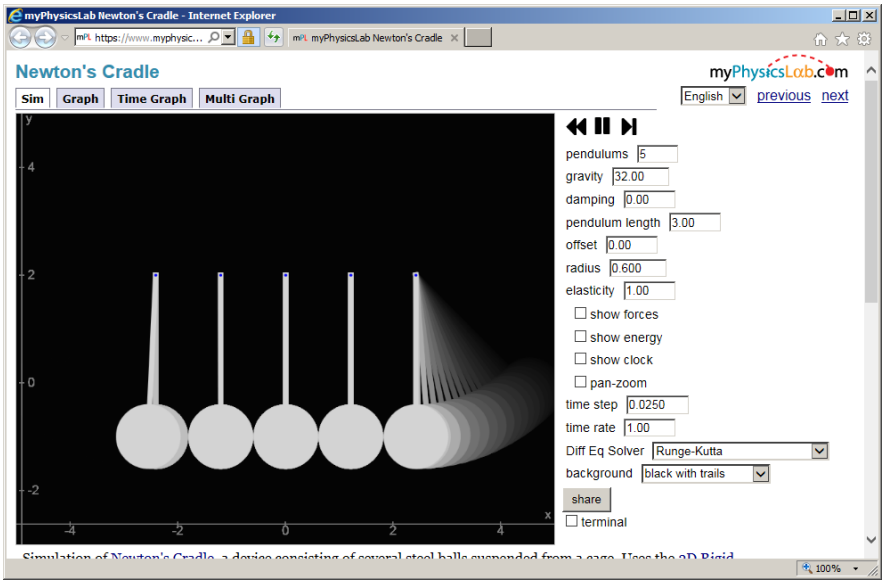
The target audience for my solution is teachers who would use the software to demonstrate principles to GCSE and A-Level students during lessons on a projector screen attached to a desktop computer running Windows 7-10. I will have to ensure that my solution is appropriate so that all text is readable on a 2 m diagonal screen from the back of a large physics laboratory.

1.2 Current system

Some of the physics teachers use the PhET simulations published for free by the University of Colorado online however many of them are outdated and rely on an old and unstable versions of adobe shockwave and flash. Additionally, the versatility of the programs is limited as they are designed to demonstrate just one task, and most of them are in 2D which limits the effectiveness of the demonstration. An example of this is 'The Ramp' simulation to the right. In this user is able to apply force on an object to push it up and down a slope, using different preset masses (eg: Piano, Fridge) and can observe the different associated factors about moving the object on graphs. The 'More Features' tab, as pictured to the right, adds more control over the simulation and this is something I'd like to include in my own simulation to allow the more complex factors to be ignored when teachers are explaining new ideas to younger/lesser ability students who would be otherwise confused by all the different aspects observed in this 'advanced' mode.



Other simulations such as the ones on myPhysicsLab online offer another source for teachers to demonstrate ideas using technology. For example, the 'Newton's Cradle' simulation as pictured below is an example of a detailed solution but there are a couple of areas which I think could be improved. One of these areas is ease of use, as I found it was difficult to make the simulation behave in such a way I intended. For example, If I wanted to 'grab' the ball on the end to hold the ball and release it by my own accord, I cannot because I can only apply a limited amount of force to one ball at a time. This limits the usability of the program as you cannot reset the momentum of the balls to 0. The second area I believe could be improved is the Implementation of the graphs. Initially it is difficult to understand what the graphs are and what they're trying to demonstrate which means teachers would have to explain them to students, using valuable lesson time whereas if they were intuitive and easy to understand, the students would be able to grasp the idea of the simulation relatively quickly.



Another thing to mention is that the simulations on myPhysicsLab are generally all web browser based and don't require anything to be downloaded by the user. Although this is advantageous as the user doesn't have to have anything stored on their local machine and it makes the software more portable, it means that the program has to continually mantained and updated to function correctly with all internet browsers, so I will not be taking this approach for my solution.

1.3 End-User Interveiw

Mr Mumford is the head of department for Physics at Clitheroe Royal Grammar School and has experience using simulatons to present ideas to students aged 11-18. He kindly agreed to an interveiw where I asked the following questions:

Q: What is the most difficult aspect of mechanics to demonstrate to students?

A: "Simple Harmonic Motion I find is one of the more difficult ones to demonstrate, however circular motion is certainly the most difficult to demostrate to students as I haven't to this day found a robust simulation which demonstrates the ideas of circular motion in respect to a car doing a vertical loop-the-loop."

Q: What's the most fustrating aspect of the simulations you use to aid your teaching?

A: "Relating them back to the lesson as the flexibility is often limited."

Q: If the simulations were easier to use and presented ideas more clearly, would you use them more often?

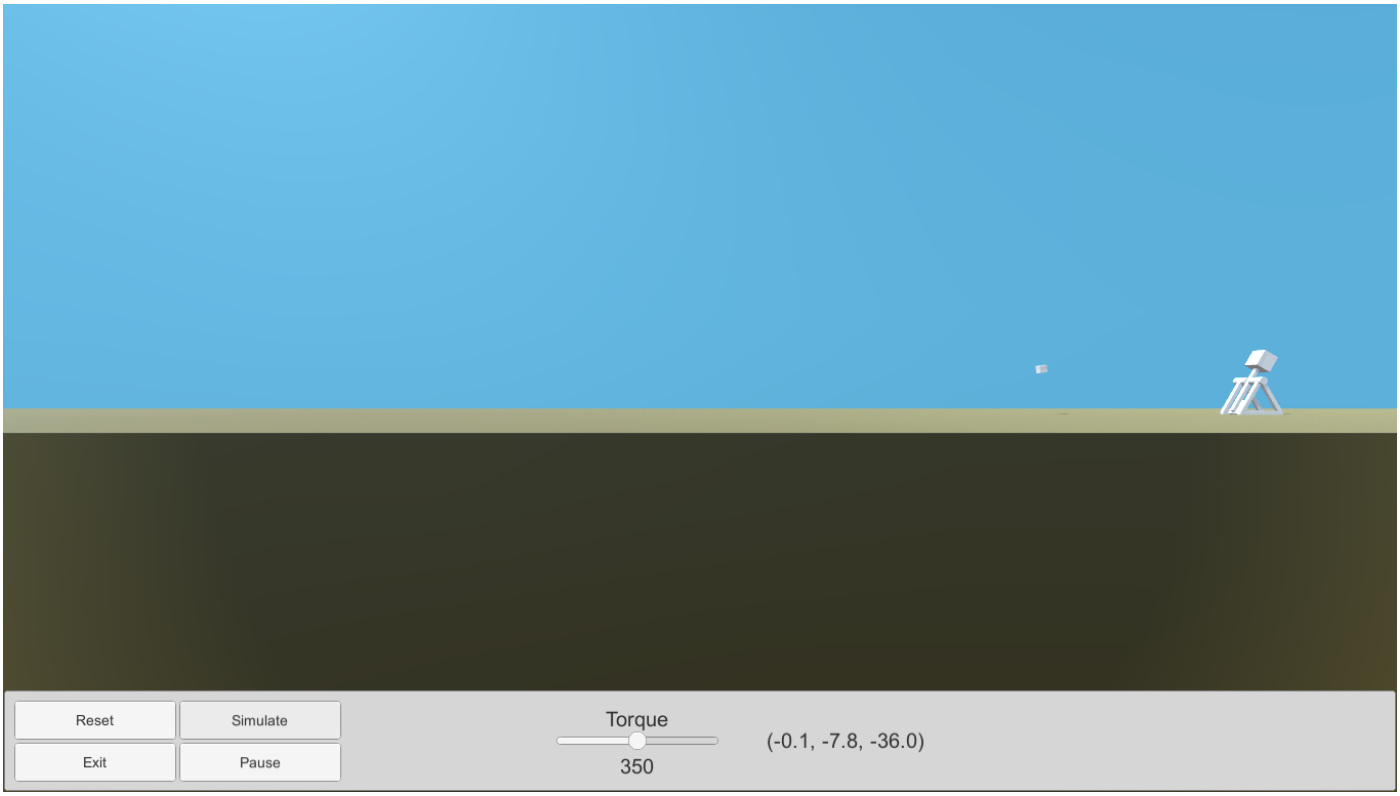
A: "Yes, certainly."

Q: What is the most highly desired feature of a physics simulation from your perspective?

A: "The ability for the students to visualise abstract ideas, because sometimes I think that my students find it difficult to relate the ideas from the simulation back to the ideas in the textbook."

1.4 Prototyping

I created a simple trebochet simulation in a couple of hours as a prototype as pictured below. This simulation involves setting the torque on a motor which swings the trebochet arm and the 3 components of the projectile's velocity is given in real time as it flys through the air.



I presented this to Mr Mumford and he gave the following feedback:

- The displacement of the projectile needs to be listed.
- You should be able to change the mass of the counterweight on the trebochet rather than the torque of the motor to allow for further calculations about transfer of energy.
- The program should be more intuitive as its difficult to know how to work it without knowing how to before hand.

In addition to this feedback, I'd like to include different camera angles, a 2D mode and the ability to plot the factors on graphs in real time.

1.5 Data Volumetrics

The prototype program I made has a filesize of 41MB. This project only has 1 scene, very few assets and is very simple overall. However, the final solution will contian many more scripts, assets and textures across multiple scenes, potentially with the ability to save scenes as listed in my secondary objectives. This means that the filesize will likely be much greater than 41MB. To get an Idea for the scale of this, I compiled the following empty Unity projects:

Program	Number Of Scenes	Filesize (MB)	Increase In Filesize (KB)
1	1	41.3	N/A
2	2	41.4	100
3	3	41.6	200

4	4	41.7	100
5	5	41.8	100

The mean increase in filesize per empty scene is 125KB.

The solution without the ability to save your own scenes (see secondary objectives) would contain 3 scenes in total. This would be at least 41.6MB, as seen from the table above. With the ability to save scenes, there could be a total of up to 23 different scenes (Being limited to 20 custom scenes) which would result in a minimum file size of 44.1MB. Obviously the minimum file size isn't of much use to predicting the size of the program, so I considered one of my other unity projects, which is unrelated to the physics simulation.

This other Unity project has 2 scenes, quite a few textures and lots of separate scripts the total compliled file size is approximately 51.03MB with 13.52MB of data stored. This means that there is approximately 10MB of assets, scripts and textures in this program across 2 scenes, alongside 13.5MB of extra data stored on the device. This allows me to estimate the amount of extra data needed per scene in the final solution to be around 5MB.

Worst Case Scenario

23 Empty scenes would be 44.1MB. To account for error, I'll double the amount of extra data stored per scene, so this would be 10MB per scene. This results in a total of 274.1MB for a compiled and fully featured solution.

The average modern desktop computer has a hard drive capacity of around 500GB to 1000GB and so the worst case of my solution would only use around 0.05% of the capacity of a 500GB hard drive.

This means that my solution will be within the acceptible range of file size for use on a desktop computer.

1.6 Implementation

I plan to use Unity3D and C# to create my solution and Unity has built in ridgidbodies which can be applied to gameobjects and most of the aspects of the simulation that I require such as the velocity of a projectile. However I will have to derive some of the aspects myself such as displacement of a projectile and the forces acting on it.

For this I can use $s = p_1 - p_0$ (Displacement is the change in distance) $F = ma$, to calculate the forces acting on an object.

For calclulating circular motion, I will have to derive more factors using the the following formulas:

- Force, $F = ma$
- Linear Velocity $v = 2\pi r$

1.7 Objectives

The solution should:

1. Demonstrate the ideas of projectiles and simple harmonic motion through a single solution with the versitility to observe any of displacement, velocity, acceleration and force while the simulation is running.
    a. You should be able to turn off the irrelevent factors of the simulation (eg: only show the velocities of a projectile, not the acceleration or forces being acted on it).
    b. Veiw the corresponding points on graphs of velocity, acceleration, force and displacement against time.
    c. Graphs should be drawn in real time as the simulation runs.

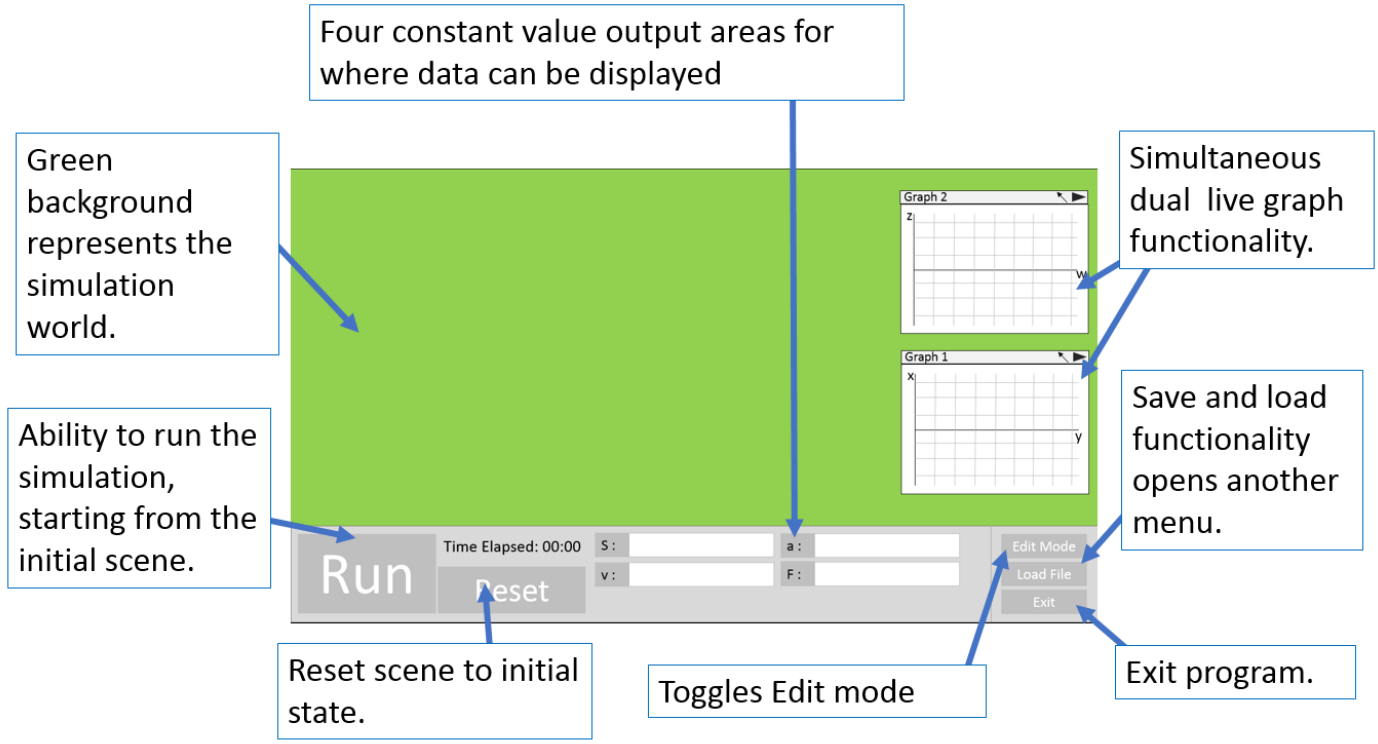
2. Be a versatile solution which is easily understood and is within context of the A-Level and GCSE physics courses.
    a. Ensure the simulation remains within the scope of the A-Level course so only includes relevant factors including Displacement, velocity, acceleration and force all given as vectors.
    b. Ensure that the scenes observe the laws of physics very similar to that in real life.
3. Be easy for teachers to use and present to a class.
    a. The teacher should be able to load preset simulations to show their students.
    b. The program should be readable on a 2 m diagonal screen from the back of a large physics laboratory.
4. Be able to plot graphs relating to velocity, acceleration, force and displacement against time.
5. Be a reliable and optimised solution which runs on a Windows 7+ classroom-level computer.
6. Including a 'Simulation Builder' which enables teachers to construct their own simulations to be more relevent to their lessons, and the ability to save this as a scene to the local machine.
    a. The teacher should be able to add structre elements and projectiles
    b. They should be able to connect these together using specified connection points
    c. They should be able to delete placed objects.
    d. The teacher should be able to add pivots, split joints and anchors to structural components

Section 2 - Design

2.1 Graphical User Interface

2.1.1 In-Simulation Controls:

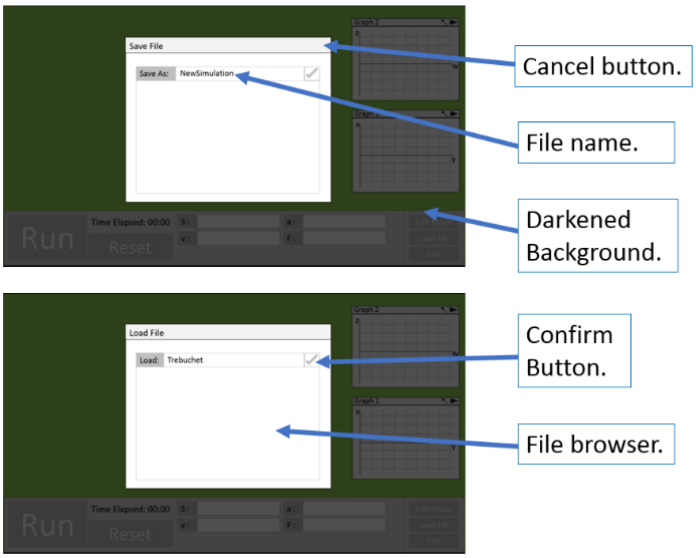
The simulation part of the program will consist of a user interface of a similar stlyle I used in the prototype except with significantly more detail and controls and will follow a similar layout to the model detailed in the diagram below:



During the running of the program, the replay options will be greyed out and the run button will become a stop button, and when the user is in edit mode, the bar at the bottom of the screen will change to a toolbox with the placeable objects which can be instantiated into the scene, and the graphs attributes can be altered.

2.1.2 Loading and Saving

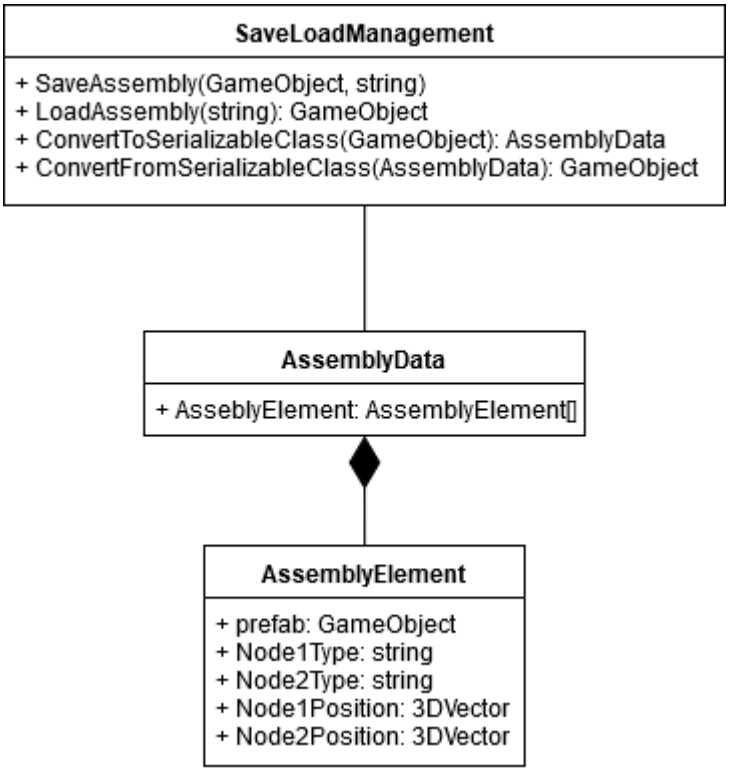
The Loading and saving windows will have separate windows which will look similar to the models on the right. The files will be presented in the file browser as buttons, and when clicked their name will autofill into the file name input box. This allows easy loading and saving as the user does not have to type each name individually.



Each save instance will be saved as a JSON text file with the .phys extension in the AppData folder on the computer. This process requires the abstraction and translation into a serialisable class of the main assembly since references to GameObjects cannot be stored in JSON format in Unity. This means that only the placeable objects, their node's positions and the node's types have to be stored as the MainAssembly can be reconstructed from this information only by instantiating the relevent prefab

(see technical solution documentation for more detail on prefabs), moving the nodes to their correct positions and setting their node type temporarily. Then the program can iterate through the list of nodes and if any node shares the same position as another, then they are connected and their node types can be set using the temp node type. This works in effectively the same way the user would manually construct a simulation except completed by the program itself.

This system consists of 3 classes. These classes are all standard C# classes (not of type MonoBehaviour, which is seldom required in Unity, as the JSON system cannot be placed within one). The first contains procedures which serialise and deserialise the main assembly into and from the abstraction of the class, the second contains an array of up to 100 placeable objects (a list would be preferred but lists cannot be serialised in Unity) and the third contains the two node positions, the two node types and the prefab of the respective object. This is shown in the UML diagram below:

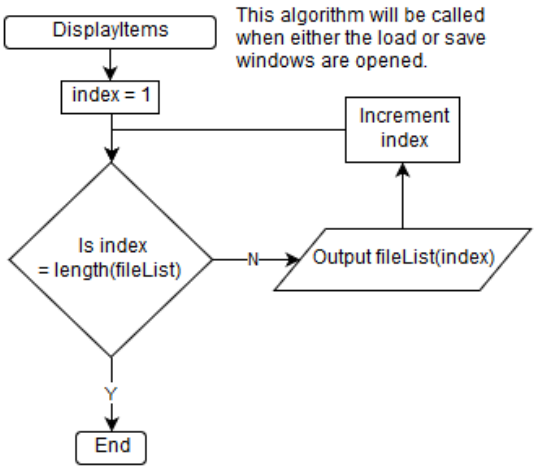


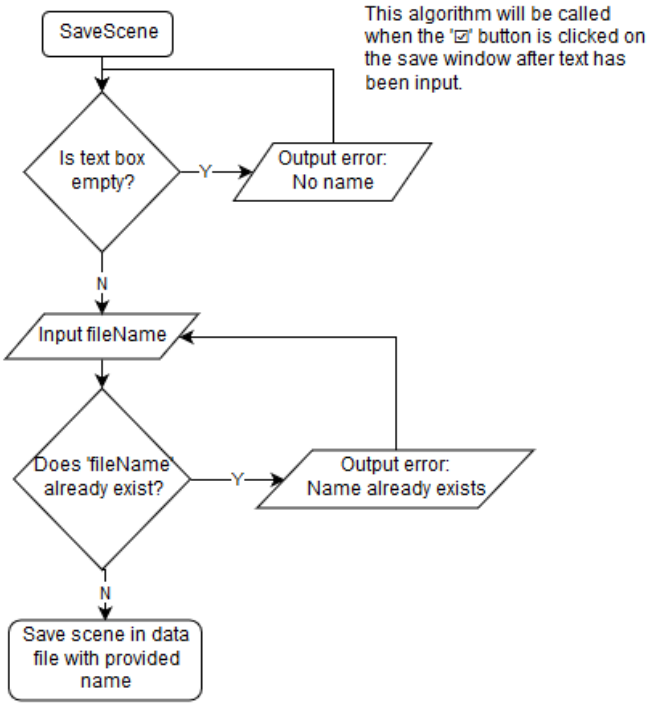
2.1 Fundamental Algorithms

Loading and Saving

The saving and loading functionality will consist of a script attached to the parent object of each corresponding file browser window. To implement this the program will create a folder in the user's documents and all of the simulation assembilies will be stored here, and the program discover and order any files in this specific file location. Any unexpected files will be ignored if they do not have the .phys extension and not detected by the program. In the event of an error in loading or saving a program, an error message will be displayed with the option to cancel or try again.

The systems will make use of the following algorithms:





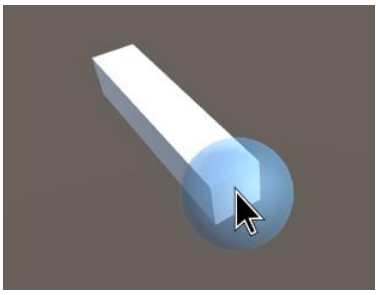
SaveLoadManagement

```
procedure SaveAssembly(MainAssemblyObject : GameObject, fileName : String)
    AssemblyData <- ConvertToSerializableClass(MainAssemblyObject)
    Jsontxt <- Json.ToJson(AssemblyData)
    stream <- StreamWriter(Application.persistentDataPath & "/" & fileName & ".phys")
    stream.Write(Jsontxt)
    stream.Close()
End

Procedure LoadAssembly(fileName : String) : GameObject
    stream <- StreamReader(Application.persistentDataPath & "/" & fileName & ".phys")
    JsonTxt <- stream.ReadToEnd()
    stream.Close()
    AssemblyData <- Json.FromJson(JsonTxt)
    NewMainAssembly : GameObject <- ConvertFromSerializableClass(AssemblyData)
    Return newMainAssembly
End

Procedure ConvertToSerializableClass(Assy : GameObject) : AssemblyData
    For Each assyEle In AssyElements
        If assyEle.name <- "Structure Element(Clone)" Then
            AssyElemnt.prefab <- GameObject.Load(Structure Element)
        ElseIf assyEle.name <- "Projectile(Clone)" Then
            AssyElemnt.prefab <- GameObject.Load(Projectile)
        Else
        End If
        AssyElemnt.Node1Type <- GetNodeType(NodeObj01)
        AssyElemnt.Node2Type <- GetNodeType(NodeObj02)
        AssyElemnt.Node1Position <- NodeConnectionScript.NodeObj01.transform.position
        AssyElemnt.Node2Position <- NodeConnectionScript.NodeObj02.transform.position
        MainAssy.AssemblyElements(i) <- AssyElemnt
        i <- i + 1
    EndFor
    Return MainAssy
End

Procedure ConvertFromSerializableClass(mainAssy : AssemblyData) : GameObject
    newMainAssembly <- GameObject.Instantiate(MainAssembly)
    For Each AssyElemnt In mainAssy.AssemblyElements
```

```
                newAssyEle <- GameObject.Instantiate(AssyElemnt.prefab)
                newAssyEle.transform.parent <- newMainAssembly.transform
                NodeObj01.transform.position <- AssyElemnt.Node1Position
                NodeObj02.transform.position <- AssyElemnt.Node2Position
                setTempNodeType(NCS.NodeObj01, AssyElemnt.Node1Type)
                setTempNodeType(NCS.NodeObj02, AssyElemnt.Node2Type)
                AssemblyManagementScript.AddSceneObject(newAssyEle)
                objectScript.UpdateLocalPosition()
        EndFor
        Return newMainAssembly
    End
```

2.2 The Simulation Builder

2.2.1 The Building System Overview



An example of what a node may look like when a mouse is over it.

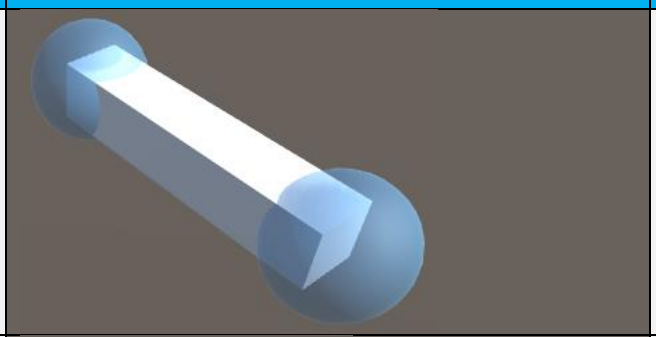
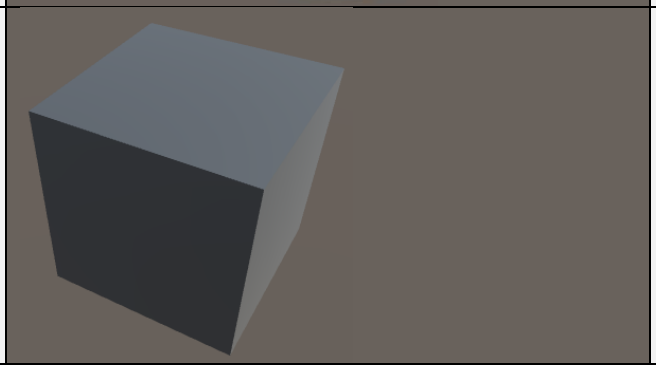
The solution will consist of an 'Edit Mode' where the user can create, modify and delete scenes using a modular construction system based around placeable objects which have different purposes. This is detailed in the following section; '2.2 Placeable Objects'.

The system will rely on the different objects connecting to each other through 'nodes' which are highlighted by the mouse flying over them. Each object will have nodes in preset locations and it is at these points that they will snap together, and the user will have to click and drag the nodes to scale the object or move it around, depending on its properties. This enables the user to ensure that the simulation is very relevent to the idea that they are teaching, and is a fairly straightforward concept to grasp.

After constructing a simulation, the user will be able to save the file if they wish, to preserve it for later.

2.2.2 Placeable Objects

The Following table details the objects available to the user via a toolbox displayed while in edit mode.

Object	Attributes / unit	Appearance	Description
Structure Element	- -		A ridgid bar which has nodes at either end used for building structures and supports
Projectile	Velocity / ms^{-1} Acceleration / ms^{-2} Displacement / m Force / N		An object which can be launched by a system

Each of the objects listed will behave differently if they are interacted with, however the node system is the fundamental idea behind all the connections made between different objects. The 'Main Assembly' is the parent

object of all the individual objects and the main script for the node system is attached to this object. How the object deals with one of its nodes being moved is entirely down to a script attached to itself, and is independent of the node system. This is shown in the entity relationship diagram in Section 3 Technical Solution 3.1.

About The Attributes

Velocity, acceleration, displacement and force will all be given as vectors and one component of these attributes will be able to be plotted on the graph as these will vary throughout the simulation.

2.2.3 Types Of Nodes

The user will be able to click any node in the scene to change it's type. The following table contains the types of nodes, their purposes and their limitations.

Type	Colour	Description	Limitations
Standard	Blue	The standard node can be dragged around the world using the curser while in build mode which allows the movement of placeable objects. Any connections between these types of nodes are ridgid. This is the default node type that elements have when spawned into the scene.	None
Pivot	Yellow	The pivot acts as a link between two subassembilies, on one of which a pivot component is attached and the other subassembly is referenced within this component.	Nodes can only be made into pivots if they are at a connection between 2 nodes.
Split Joint	Red	The split joints acts as a link between two subassembilies, on one of which a fixed joint component is attached and the other subassembly is referenced within this component.	Nodes can only be made into split joints if they are at a connection between 2 nodes.
Anchor	Black	The anchor is a node to lock the enitre subassembly into a fixed position in 3D space, where no amount of force can move them.	The Anchor can be placed on any node with any number of connections.

2.2.4 Camera Management

In edit mode, the user should be able to move the camera around in 3D space to allow for easy construction. The camera will be able to snap to a single plane which will limit free movement of any object when moving it around from this angle, as the original distance of each point away from the camera will be retained as it is dragged around unless it snaps to another node. In addition to this, the user will also be able to set the camera to free-veiw where the assembly can be seen from an 45 degree angle. The following table shows the keybindings and their functions whilst in edit mode:

Type	Function	Keybindings
Snap to veiw	Top-Down view	7
	Front view	8
	Rear view	2
	Left view	4
	Right view	6
	Free-view	5

2.2 Fundamental Algorithms

Main Assembly Algorithm

The aim of this algorithm is to make any node clicked on in the scene follow the curser, and make it snap to any other node in the scene, providing that the other node is not part of the same object. This script is named “userInputScript” and is attached to the “Main Assembly” object in the scene.

```
GetEditMode()
        return EditMode

Start()
        EditMode <- true
        CheckCurserPosition()

UpdateNodeList(node)
        Node.Add(node)
        MouseDownFlag.Add(false)
        MouseOverFlag.Add(false)

UpdateProjectileList(projec)
        Projectile.Add(projec)

Coroutine CheckCurserPosition()
        while EditMode = true and Moving = false
                for i <- 0 to i = Node.Count - 1 do
                    MousePos <- Input.mousePosition
                    NodePos <- Camera.WorldToScreenPoint(Node[i].position)
                    intersectionRange <- RangeMultiplier/(Camera.WorldToScreenPoint(Node[i].position).z)
                    if (NodePos.x - intersectionRange) < (MousePos.x) and (MousePos.x) <
                                    (NodePos.x + intersectionRange) and (NodePos.y - intersectionRange) <
                                    (MousePos.y) and (MousePos.y) < (NodePos.y + intersectionRange) then
                            for y <- 0 to y = MouseDownFlag.Count - 1 do
                                    if MouseDownFlag[y] = false then
                                            MouseDownFlag[i] <- true
                                            Moving <- true
                                            MouseDownOnNode(Node[i], i)
                                            halt procedure
                                    else
                                    endif
                                    y = y + 1
                            endfor
                        else
                            Moving <- false
                            MouseDownFlag[i] <- false
                            if MouseOverFlag[i] = false then
                                    MouseOverFlag[i] <= true
                                    MouseOverNode(Node[i], i)
                            elseif
                            endif
                        endif
                    else
                        MouseOverFlag[i] <- false
                        if MouseDownFlag[i] = false then
                            Node[i].SetActive(false)
                    endif
                   i = i + 1
                endfor
        endwhile
        halt procedure

MouseDownOnNode(node, flagPos)
        FollowCurser(node,flagPos)

MouseOverNode(node, flagPos)
        node.SetActive(true)

coroutine FollowCurser( node, flagPos)
        StructureElement <- node.transform.parent.parent.gameObject
```

```
InitalObjectPosition <- StructureElement.position
structEleScript <- StructureElement.GetComponent(StructEleScript)
initialPosRelToCam <- Camera.WorldToScreenPoint(StructureElement.position).z
InitialObjectLength <- StructureElement.transform.localScale.z
x <- 0
for z <- 0 to z = Node.Count - 1 do
    if node.parent <> Node[z].parent
        OtherNodes.Add(Node[z])
        snapFlag.Add(false)
   z = z + 1
endfor
while MouseDownFlag[flagPos] = true and EditMode = true
    for a <- 0 to a = OtherNodes.Count - 1 then
        NodePos <- Camera.WorldToScreenPoint(node.transform.position)
        OtherNodePos <- Camera.WorldToScreenPoint(OtherNodes[a].position)
        mousePos.x <- Input.mousePosition.x
        mousePos.y <- Input.mousePosition.y
        intersectionRange <- RangeMultiplier / OtherNodePos.z
        if (OtherNodePos.x - intersectionRange) < mousePos.x and mousePos.x <
                    (OtherNodePos.x + intersectionRange) and (OtherNodePos.y -intersectionRange)
                    < mousePos.y and mousePos.y < (OtherNodePos.y + intersectionRange) then
            if snapFlag[a] = false then
                newPosition <- OtherNodes[a].position
                node.position <- newPosition
                snapFlag[a] <- true
                x <- a
           else
           endif
        else
            snapFlag[a] <- false
            if snapFlag[x] = false then
                node.position <- Camera.ScreenToWorldPoint(mousePos.x, mousePos.y,
                                                                    initialPosRelToCam)
            else
           endif
       endif
       a = a + 1
    endfor
    structEleScript.UpdateLocalPosition()
    if Input.GetMouseButtonUp(0) then
        MouseDownFlag[flagPos] <- false
       Halt
    else
    end
endwhile
Moving <- false
OtherNodes.Clear()
CheckCurserPosition()
```

Variables And Lists

Name	Type	Contents
Node	List of game objects	All nodes in the scene
Projectile	List of game objects	All projectiles in the scene
MouseDownFlag	List of Boolean	Mouse is down on a node
MouseOverFlag	List of Boolean	Mouse is over a node
EditMode	Boolean	The program is in edit mode
Moving	Boolean	The user is moving a node
MousePos	2D Vector	The position of the mouse on the screen
NodePos	2D Vector	The position of the current node on the screen
intersectionRange	Integer	A radius around a given point in 2D
RangeMultiple	Float	The multiplier of the radius
StructureElement	Game object	The placeable object in the scene
InitialObjectPosition	3D Vector	Initial Object Position
structEleScript	Script	The script attached to the placeable object

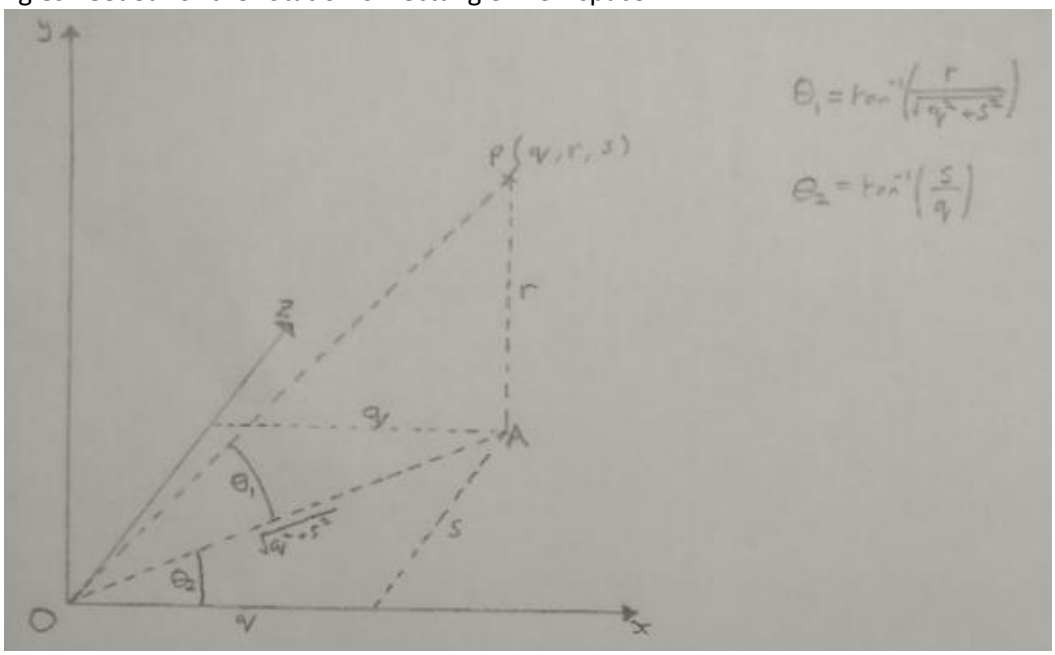
initialPosRelToCam	Float	The distance the object is away from the camera
InitialObjectLength	Float	The scale of the object before it was moved
x	Integer	The location of the clicked object in the list
SnapFlag	List of boolean	The node, a, has been snapped to
node	Game object	The node that is following the curser
flagPos	Integer	The position of the node in the array
OtherNodes	List of Game Objects	All the nodes in the scene which can be snapped to by the current node
OtherNodePos	Game object	The node that the current node may snap to
newPosition	3D vector	The position of the other node when snapping

Procedures and Coroutines

Type	Name	Purpose	Input Parameters	Return
Procedure	Start	Initiailises the Script	-	-
	GetEditMode	Gets the edit mode	-	EditMode
	UpdateNodeList	Updates the list of nodes with new nodes	node (Game Object)	-
	MouseDownOnNode	Is called once when a node is clicked	node (Game Object) flagPos (Integer)	-
	MouseOverNode	Is called once when the mouse moved over the node	node (Game Object) flagPos (Integer)	-
Coroutine	CheckCurserPosition	Checks if the mouse is over a node in the scene, and if it is being dragged	-	-
	FollowCurser	Makes the node in the scene follow the curser	node (Game Object) flagPos (Integer)	-

Structure Element Algorithm

This algorithm makes the rectangle always meet the two nodes at either end. When one node is moved, the other stays in the same place and the rectangle meets both at either end. The image below shows the diagram used to determine the angles needed for the rotation of rectangle in 3D space.




```
Start()
          mainAssemblyObject <- FindGameObject("MainAssembly")
          userInputScript <- mainAssemblyObject.GetComponent(UserInputScript)
          userInputScript.UpdateNodeList(NodeObj01)
          userInputScript.UpdateNodeList(NodeObj02)
          Rectangle <- FindLocalGameObject("Rectangle")

UpdateLocalPosition()
          NodePoint01 <- NodeObj01.position
          NodePoint02 <- NodeObj02.position
          RectangleScale <- NodeObj01.Scale.x
          Rectangle.position <- (NodePoint02.x+NodePoint01.x)/2,
                                    (NodePoint02.y + NodePoint01.y)/2,(NodePoint02.z + NodePoint01.z)/2)
          if (NodePoint01.x < NodePoint02.x) OR (NodePoint01.x = NodePoint02.x) then

                    rotation <- (Arctan((NodePoint01.y - NodePoint02.y) /
                         Sqrt((NodePoint01.z - NodePoint02.z)^2) + (NodePoint02.x - NodePoint01.x)^2)),
                         90 + Arctan((NodePoint01.z - NodePoint02.z) / (NodePoint02.x - NodePoint01.x)), 0)
        else
                    rotation <- (-Arctan((NodePoint01.y - NodePoint02.y) /
                         Sqrt((NodePoint01.z - NodePoint02.z)^2) + (NodePoint02.x - NodePoint01.x)^2)),
                         90 + Arctan((NodePoint01.z - NodePoint02.z) / (NodePoint02.x - NodePoint01.x)), 0)
        endif
        Rectangle.rotation <- Quaternion(rotation)
        Rectangle.localScale <- (RectangleScale, RectangleScale, Sqrt((NodePoint02.x - NodePoint01.x)^2 +
                                 (NodePoint01.y - NodePoint02.y)^2 + (NodePoint01.z - NodePoint02.z)^2)))
```

Notes:

- The function Start() is called at the instantiaion of an instance of an object, wether it be during runtime or on initial startup.
- UpdateLocalPosition() is called from the MainAssemblyAlgorithm while the mouse is clicked down on one of the object's nodes.
- UpdateNodeList() is a function in the MainAssemblyAlgorithm which keeps a list of all the nodes in the scene.

Projectile Algorithm

This algorithm needs to calculate the Acceleration (using change in velocity devided by time), displacement and resultant force (Using Newton's second law of motion, F=ma). The velocity is a native attribute to any gameobject with a ridgidbody attached to it and so it is not required to caluclate this. OnRun() will be called when the "Run" button is clicked by the user.

```
Simulate()
         initialDisplacement <- this.location
GetAcceleration()
         initialVelocity <- this.velocity
         wait 1ms
         accn <- (this.velocity - initialVelocity) / 0.001
         return accn

GetDisplacement()
         dis <- initialDisplacement - this.location
         return dis

GetForce()
         force <- GetAcceleration() * this.mass
FixedUpdate()
    If simulating = True Then
        Displacement <- Vector3(Rectangle.transform.position.x - InitialPosition.x,
Rectangle.transform.position.y - InitialPosition.y, Rectangle.transform.position.z - InitialPosition.z)
        Acceleration <- (GetVelocityAtPoint(Rectangle.transform.position) - Velocity) / DeltaTime
        Velocity <- GetVelocityAtPoint(Rectangle.transform.position)
        Force <- (mass * Acceleration)

        If ShowOnReadouts = True Then
            SceneManagementAlgorithm.SetDisplacementText(Displacement)
            SceneManagementAlgorithm.SetVelocityText(Velocity)
            SceneManagementAlgorithm.SetAccelerationText(Acceleration)
            SceneManagementAlgorithm.SetForceText(Force)
        End If
    End If
End
procedure UpdateLocalPosition()
    If NodeMoved(NodeObj01) = True Then
        If NCS.getNodeObjConnection(2).Count > 0 Then
            AssemblyManagementScript.RemoveConnections(NodeObj02, NCS.getNodeObjConnection(2)(0), NCS)
        End If
        Rectangle.transform.position <- 3DVector(NodeObj01.transform.position.x,
NodeObj01.transform.position.y, NodeObj01.transform.position.z + 0.25F)
        NodeObj02.transform.position <- 3DVector(NodeObj01.transform.position.x,
NodeObj01.transform.position.y, NodeObj01.transform.position.z + 0.5F)
    ElseIf NodeMoved(NodeObj02) = True Then
        If NCS.getNodeObjConnection(1).Count > 0 Then
            AssemblyManagementScript.RemoveConnections(NodeObj01, NCS.getNodeObjConnection(1)(0), NCS)
        End If
        Rectangle.transform.position <- 3DVector(NodeObj02.transform.position.x,
NodeObj02.transform.position.y, NodeObj02.transform.position.z - 0.25F)
        NodeObj01.transform.position <- 3DVector(NodeObj02.transform.position.x,
NodeObj02.transform.position.y, NodeObj02.transform.position.z - 0.5F)
    Else
    End If

    Node1ExpectedPosition <- NodeObj01.transform.position
    Node2ExpectedPosition <- NodeObj02.transform.position
End
```

Notes:

- The function Simulate() is called upon the 'start' button being clicked.
- UpdateLocalPosition() is called from the MainAssemblyAlgorithm while the mouse is clicked down on one of the object's nodes.
- FixedUpdate() is called once per frame

NodeConnection Algorithm

This algorithm manages the two nodes attached to any placeable object and exists as a component alongside the Projectile/ Structure element algorithms as this is not object specific.

```
    Procedure UpdateNodeType(node : GameObject,type : String)
        If node = NodeObj01 Then
            oldType <- Node1Type
            If type = "Anchor" Then
                If type <> Node1Type Then
                    SetNodeType(type, node, oldType)
                    ForEach otherObject In NodeObj01Connection
                        NodeConnectionScript.SetNodeType(type, otherObject, oldType)
                    EndFor
                Else
                    SetNodeType("Standard", node, oldType)

                    ForEach otherObject In NodeObj01Connection
                        NodeConnectionScript.SetNodeType("Standard", otherObject, oldType)
                    EndFor
                End If
            ElseIf NodeObj01Connection.Count = 1 Then
                If type <> Node1Type Then
                    SetNodeType(type, node, oldType)
                    NodeConnectionScript.SetNodeType(type, NodeObj01Connection(0), oldType)
                    Node1AssocitatedObject <- NodeObj01Connection(0)
                Else
                    SetNodeType("Standard", node, oldType)
                    NodeConnectionScript.SetNodeType("Standard", NodeObj01Connection(0), oldType)
                    Node1AssocitatedObject <- Nothing
                End If
            Else
                If type <> "Standard" And type <> "Anchor" Then
                    NotifyUser("Error: Pivots and split joints must only be between 2 nodes.  Try
Reconnecting them.")
                End If
            End If
        ElseIf node = NodeObj02 Then
            oldType <- Node2Type
            If type = "Anchor" Then
                If type <> Node2Type Then
                    SetNodeType(type, node, oldType)
                    ForEach otherObject In NodeObj02Connection
                        NodeConnectionScript.SetNodeType(type, otherObject, oldType)
                    EndFor
                Else
                    SetNodeType("Standard", node, oldType)
                    ForEach otherObject In NodeObj02Connection
                        NodeConnectionScript.SetNodeType("Standard", otherObject, oldType)
                    ForEach
                End If
            ElseIf NodeObj02Connection.Count = 1 Then

                If type <> Node2Type Then
                    SetNodeType(type, node, oldType)
                    NodeConnectionScript.SetNodeType(type, NodeObj02Connection(0), oldType)
                    Node2AssocitatedObject <- NodeObj02Connection(0)
                Else
                    SetNodeType("Standard", node, oldType)
                    NodeConnectionScript.SetNodeType("Standard", NodeObj02Connection(0), oldType)
                    Node2AssocitatedObject <- Nothing
                End If
            ElseIf type <> "Standard" And type <> "Anchor" Then
                NotifyUser("Error: Pivots and split joints must only be between 2 nodes. Try Reconnecting
them.")
            End If
        Else
        End If
```

```
    End
    Procedure LinearSearch(list : List(GameObject), SearchFor : GameObject) : Integer
        If list.Count <> 0 Then
            For i <- 0 To list.Count - 1
                If list(i) = SearchFor Then
                    Return i
                End If
            EndFor
        Else
        Return -1
        End If
    End
    Coroutine UpdateConnection(node : GameObject, ConnectedNode : GameObject, isConnected : Boolean)
        If node = NodeObj01 Then
            If isConnected = True Then
                If (LinearSearch(NodeObj01Connection, ConnectedNode) = -1) And (ConnectedNode <> node)
Then
                    NodeObj01Connection.Add(ConnectedNode)
                End If
            Else
                If NodeObj01Connection.Count <> 0 Then
                    locationIndex <- LinearSearch(NodeObj01Connection, ConnectedNode)
                    If locationIndex <> -1 Then
                        NodeObj01Connection.RemoveAt(locationIndex)
                        NodeObj01Connection.TrimExcess()
                    End If
                End If
            End If
        ElseIf node = NodeObj02 Then
            If isConnected = True Then
                If (LinearSearch(NodeObj02Connection, ConnectedNode) = -1) And (ConnectedNode <> node)
Then
                    NodeObj02Connection.Add(ConnectedNode)
                End If
            Else
                If NodeObj02Connection.Count <> 0 Then
                    locationIndex <- LinearSearch(NodeObj02Connection, ConnectedNode)
                    If locationIndex <> -1 Then
                        NodeObj02Connection.RemoveAt(locationIndex)
                        NodeObj02Connection.TrimExcess()
                    End If
                End If
            End If
        Else
        End If
    End
```

Notes:

- The procedure UpdateNodeType() is called by the AssemblyManagement algorithm when either a connection is broken or a node type is set.
- The function LinearSearch() locates a pivot or split joint in a list of 2 element arrays which contain each side of the pivot/ split joint.
- The procedure UpdateConnection() is called by the AssemblyManagement algorithm when a connection is made or broken.

2.3 The Simulation

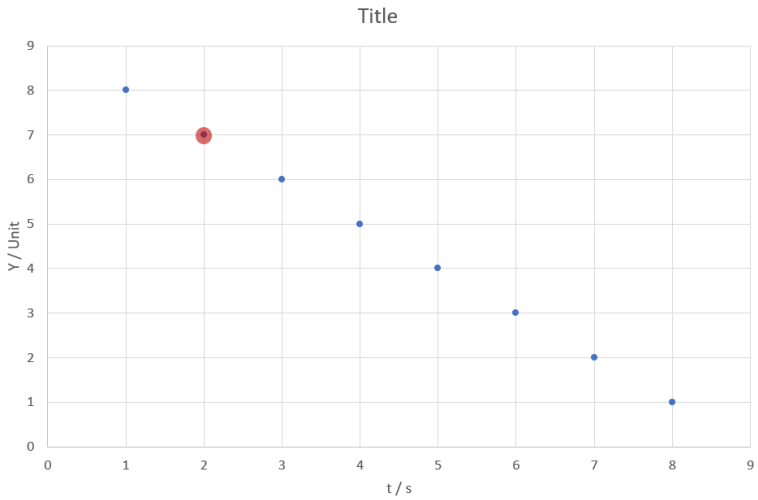
2.3.1 Data Handling

The two elements will consist of the x followed by y values which can be passed from the projectile algorithm to the graphing algorithm and plotted on a graph. The x values will be populated with velocity, acceleration, force or displacement and the y values will be populated with the time elapsed from the start of the simulation to the

moment that the point was recorded. When the scene is reset, the graphs will also reset and the lists will be cleared. Data will stop being recorded once the simulation has been stopped by the user pressing the 'stop' button.

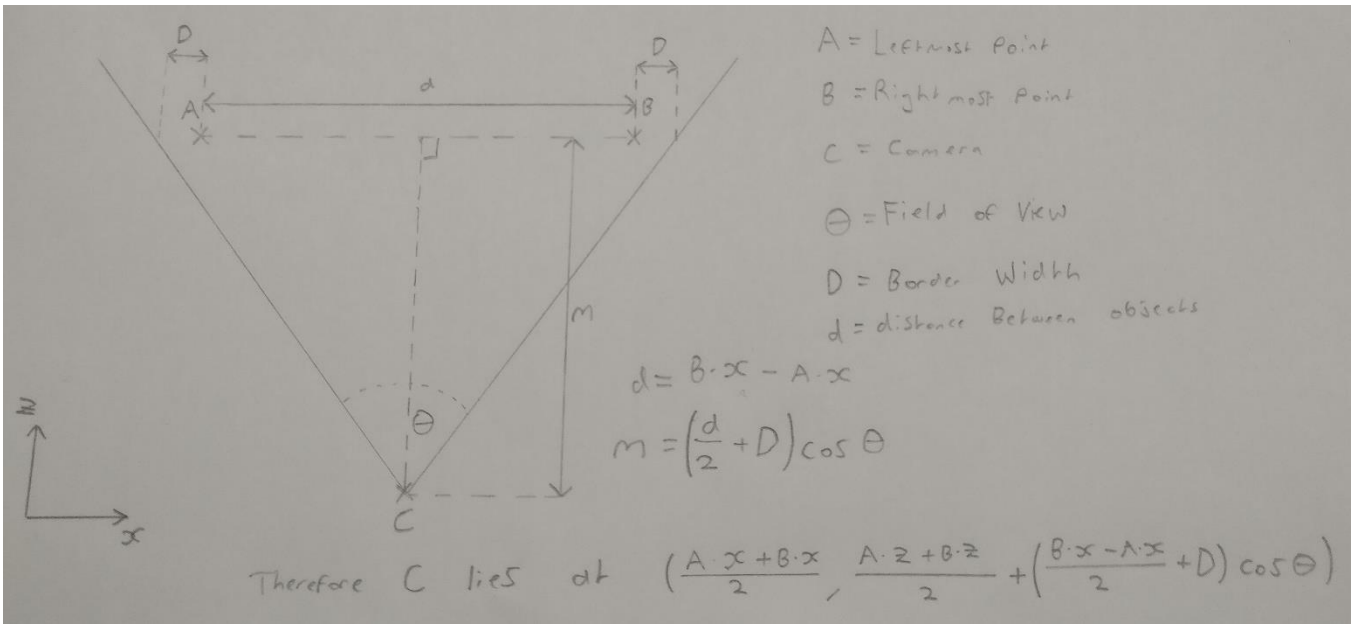
2.3.2 Graphing System

Each graph will have a settings menu where the y values can be set when in edit mode. The graphs should populate themselves with points and scale automatically so that all the points lie within view on the graph, and should automatically update the graph title which corresponds to the attribute being plotted. This will look something like the graph to the right. The graph system should also minimise the amount of instantiation calls for plotting points and so should instantiate points initially then as the graph translates to the left, the points that lie outside of the graphing window should be recycled by translating them to a new position at the front of the graph. This is effectively a circular queue where the front of the queue is moved to the back when the front lies outside of veiw.



2.3.3 Camera Management

The camera in the simulation will be positioned similarly to the camera in the protoype I made and the system should ensure that both the initial structure and any projectiles lie comfortably within the frame of the camera. This means that the camera will track towards and away from the simulation plane, keeping all objects in the simulation in frame whilst doing so. This will be achived by finding the leftmost and rightmost nodes or projectiles then finding the midpoint between them and. To ensure that the camera movement does not become nauseating or confusing, the movement should be damped so it is smooth and gentle. This will be achived by applying a thrust backwards and forwards to the camera whenever any of the objects in the scene are out of range, or are clustered towards the centre of the field of view. The image below shows how the position of the camera relative to the right and leftmost points was determined.



2.3 Fundamental Algorithms

Simulation Manager Algorithm

This algorithm manages the running of the simulation. It will exist in one instance for the entire execution of the program and therefore stores any globally important data.

```
Procedure replaceAssembly(old : GameObject) : AssemblyManagementScript
    If currentAssembly <> Nothing Then
        Destroy(old)
        currentAssembly.SetActive(True)
        MainAssembly <- currentAssembly
        MainAssembly.name <- "MainAssembly"
    End If
    Return assemblyManagementScript
End

Procedure EditMode()
    If simulating <> True Then
        editing <- True
        StartCoroutine(WaitForNextFrame(AMS, False))
    Else
        NotifyUser("You must stop the simulation before you can go into edit mode.")
    End If
End

coroutine WaitForNextFrame(AMS : AssemblyManagementScript,isResetting : Boolean)
    event.toggleEditMode()
    AssemblyManagementScript.OnEditMode()
    If isResetting = True Then
        SimulationMode()
    End If
End

Procedure SimulationMode()
    event.toggleSimulationMode()
    editing <- False
    EditPanel.SetActive(False)
    SimulationPanel.SetActive(True)
    currentAssembly <- Instantiate(MainAssembly)
    currentAssembly.SetActive(False)
    assemblyManagementScript.OnSimulationMode()
End

Procedure instantiateObject(placeableObject : GameObject)
    assemblyManagementScript.SetEditMode(False)
    instantiatedObject <- Instantiate(placeableObject)
    NodeConnectionScript.ClearAllLists()
    NodeConnectionScript.InitialiseNodeTypes()
    instantiatedObject.transform.parent <- MainAssembly.transform
    StartCoroutine(assemblyManagementScript.SimpleFollowCurser(instantiatedObject))
    assemblyManagementScript.OnEditMode()
    assemblyManagementScript.AddSceneObject(instantiatedObject)
End

Procedure ChangeMode(calledFrom : GameObject)

    Case options(Index).text
        Case "Build"
            mode = "Build"
        Case "Select Pivot"
            mode = "Pivot"
        Case "Select Anchor"
            mode = "Anchor"
        Case "Select Split Joint"
            mode = "SplitJoint"
        Case "Delete"
```

```
                mode = "Delete"
            Case Else
                Log("Fatal Error: Mode not found")
        End Case
    End
```

Notes:

- The procedure replaceAssembly() is called when switching into edit mode from simulation mode as the old assembly is made up of subassembilies which cannot be modified.
- The procedure EditMode() is called upon the user switching into edit mode.
- The procedure SimulationMode() is called upon the user switching to simulation mode.
- The procedure InstantiateObject() is called upon the user clicking a button to add a new placeable object.
- The procedure ChangeMode() is called upon the user changing the value of the mode drop down in edit mode.

Camera Management Algorithm

The purpose of this algorithm is to ensure that all the objects in the scene are within the horizontal frame of the camera horizontally.

```
Start()
         mainAssemblyObject <- FindGameObject("MainAssembly")
         userInputScript <- mainAssemblyObject.GetComponent(UserInputScript)
         borderWidth <- 2
         FOV <- camera.FieldOfView
         smoothTime <- 0.5

CalculateTargetPosition()
         targetX <- (minX + maxX) / 2
         targetY <- (minY + maxY) / 2
         targetZ <- (((maxX - minX) / 2) + borderWidth) * cos(FOV)
         targetPosition <- 3Dvector(targetX, targetY, targetZ)
return targetPosition

GetFringeNodes()
         for i <- 0 to userInputScript.Node.Length - 1 do
                  if userInputScript.Node[i].position.x > maxX then
                           maxX <- userInputScript.Node[i].position.x
                  else
                           if userInputScript.Node[i].position.x < minX then
                           minX <- userInputScript.Node[i].position.x
                           else
                           endif
                  endif
                  if userInputScript.Node[i].position.y > maxY then
                           maxY <- userInputScript.Node[i].position.y
                  else
                           if userInputScript.Node[i].position.y < minY then
                           minY <- userInputScript.Node[i].position.y
                           else
                           endif
                  endif
         endfor
         for i <- 0 to userInputScript.projectile.length - 1 do
                  if userInputScript.projectile[i].position.x > maxX then
                           maxX <- userInputScript.projectile[i].position.x
                  else
                           if userInputScript.projectile[i].position.x < minX then
                           minX <- userInputScript.projectile[i].position.x
                           else
                           endif
                  endif
                  if userInputScript.projectile[i].position.y > maxY then
                           maxY <- userInputScript.projectile[i].position.y
                  else
```

```
                        if userInputScript.projectile[i].position.y < minY then
                        minY <- userInputScript.projectile[i].position.y
                        else
                        endif
                endif
        endfor

coroutine AutoCamera()
        GetFringeNodes()
        newPosition <- CalculateTargetPosition()
        camera.position <- translateToTarget.Damp(camera.position, newPosition, (0,0,0), smoothTime)
```

Notes:

- The procedure Start() is called at the beginning of the running of the program.
- The procedure CalculateTargetPosition() is called if the camera needs to be moved to fit all the scene objects to within the frame of the camera and calculates where the camera needs to be to do so.
- The procedure GetFringeNodes() gets the outermost nodes in the scene which will be the furthest object that the camera must have in its frame.
- The coroutine AutoCamera() moves the camera smoothly to a position where all the nodes lie within the frame of the camera.

Graph Management Algorithm

This algorithm manages the graph by plotting the points, recycling points, scaling the graph and moving horizontally.

```
    Procedure PlotPoint(Value : float, time : float)
        If (time * GraphScaleX) / GraphSizeX > 1 Then
            change <- ((time * GraphScaleX) / GraphSizeX) * GraphScaleX
            Difference <- Difference + change
            BackgroundRect.sizeDelta <- New Vector2(Difference, 0)
            BackgroundRect.transform.Translate(2DVector((-change / 2), 0))
            GraphSizeX <- BackgroundRect.width
            RedundantPoint <- RedundantPoint.Concat(LocateRedundantPoints(time * GraphScaleX)).ToList()
        End If
        If Math.Abs((Value * GraphScaleY) / (GraphSizeY / 2)) > 1 Then
            GraphScaleY <- ScaleVertically(GraphScaleY * (1 / Mathf.Abs((Value * GraphScaleY) /
(GraphSizeY / 2))), GraphScaleY)
        End If
        If RedundantPoint.Count <> 0 Then
            NewPoint <- RedundantPoint(0)
            RedundantPoint.Remove(NewPoint)
            RedundantPoint.TrimExcess()
        Else
            NewPoint <- Instantiate(PlottingPointPrefab)
            NewPoint.transform.SetParent(graph background)
            Point.Add(NewPoint)
        End If
        PointTransform <- NewPoint.GetComponent(Transform)()
        Position <- New Vector2((time * GraphScaleX), (Value * GraphScaleY))
        PointTransform.anchoredPosition <- Position
    End

    Procedure ScaleVertically(newScale : float, currentScale : float) : float
        For i <- 0 To Point.Count - 1
            PntTrn <- Point(i).GetComponent(Transform)()
            PntTrn.anchoredPosition = 2DVector(PntTrn.anchoredPosition.x, (PntTrn.anchoredPosition.y /
currentScale) * newScale)
        EndFor
        Return newScale
    End

    Procedure LocateRedundantPoints(currentTime : float) : List(Of GameObject)
        RP : List(GameObject) = New List(GameObject)
        CP : List(GameObject) = New Point
        For i <- 0 To Point.Count - 1
```

```
            If RedundantPoint.Contains(Point(i)) Then
                CP.Remove(Point(i))
            End If
        EndFor
        CP.TrimExcess()
        For i <- 0 To CP.Count - 1
            RT <- Point(i).GetComponent(Of RectTransform)()
            If RT.anchoredPosition.x < (currentTime - InitialBackgroundSize.x) Then
                RP.Add(CP(i))
            End If
        EndFor
        Return RP
    End

    Coroutine PlotPoints()
        StartTime <- Time.fixedTime
        If ProjScript <> Nothing Then
            While Simulating = True
                Case ProjAttribute
                    Case "Displacement"
                        PlotPoint(ProjScript.GetDisplacement(Axis), Time.fixedTime - StartTime)
                    Case "Velocity"
                        PlotPoint(ProjScript.GetVelocity(Axis), Time.fixedTime - StartTime)
                    Case "Acceleration"
                        PlotPoint(ProjScript.GetAcceleration(Axis), Time.fixedTime - StartTime)
                    Case "Force"
                        PlotPoint(ProjScript.GetForce(Axis), Time.fixedTime - StartTime)
                    Case Else
                        Log("Fatal Error: Mode not found")
                End Case
            End While
        Else
        End If
        Title.SetActive(False)
    End
```

Notes:

- The procedure PlotPoint() is called when a new point needs to be plotted on the graph.
- The procedure ScaleVertically() calculates how much the graph needs to adjust its vertical scale to allow all points to fit within the viewing window.
- The procedure LocateRedundantPoints() finds all points which lie outside of veiw and so can be recycled by moving them to the front to avoid expensive instantiate calls.

Projectile Drop Down Algorithm

This algorithm ensures that the dropdowns for the two graphs and the readouts are updated and that they show which projectile is which using a numbering system which is displayed over each projectile.

```
    Procedure ShowProjectileNumbers(ProjectilesList : List(GameObject))
        For i <- 0 To ProjectilesList.Count - 1
            If ProjectilesList(i) <> Nothing Then
                Num = Instantiate(ProjectileNumber)
                Num.transform.SetParent(gUIScript.gameObject.transform)
                Num.GetComponent(RectTransform).Position =
MainCamera.WorldToScreenPoint(ProjectilesList(i).transform.GetChild(1).transform.position)
                Num.GetComponentInChildren(Of Text)().text = i.ToString()
                CurrentProjectileNumbers.Add(Num)
                DisplayingNumbers <- True
            End If
        Next
    End

    Procedure OnPointerExit()
        HideProjectileNumbers()
        If ProjectilesList.Count > DD.value Then
```

```
            CurrentProjectile <- ProjectilesList(DD.value)
        End If
        UpdateGraphs()
    End
    Procedure OnPointerEnter()
        ProjectilesList.Clear()
        UpdateDropdownOptions()
    End
    Procedure UpdateDropdownOptions()
        DD.ClearOptions()
        ProjectilesList.Clear()
        ProjectileLabel.Clear()
        MainAssembly <- gUIScript.getMainAssembly()
        ProjectilesList <- MainAssembly.GetComponent(AssemblyManagementScript).getProjectileList()
        ShowProjectileNumbers(ProjectilesList)
        For i <- 0 To ProjectilesList.Count - 1
            ProjectileLabel.Add("Projectile " & i.ToString())
        EndFor
        DD.AddOptions(ProjectileLabel)
        DD.RefreshShownValue()
        If ProjectileLabel.Count > 0 Then
            CurrentProjectile <- ProjectilesList(DD.value)
        End If
    End
    Procedure RefreshNumberLocations()
        If DisplayingNumbers = True Then
            HideProjectileNumbers()
            ShowProjectileNumbers(AssemblyManagementScript.getProjectileList())
        End If
    End

    Procedure UpdateGraphs()
        ProjectilesList.Tidy()
        If ProjectilesList.Count <> 0 Then
            If Me.gameObject.transform.parent.name = "GraphingPanel1Settings" Then
                GraphScript = GameObject.Find("/Canvas/GraphingPanel1").GetComponent(GraphScript)
                GraphScript.SetProjectile(CurrentProjectile)
            ElseIf this.gameObject.parent.name = "GraphingPanel2Settings" Then
                GraphScript = GameObject.Find("/Canvas/GraphingPanel2").GetComponent(Of GraphScript)()
                GraphScript.SetProjectile(CurrentProjectile)
            Else
                For Each proj : GameObject In ProjectilesList

                    If proj <> Nothing Then
                        PS <- proj.GetComponent(ProjectileScript)()
                        PS.showReadouts(False)
                    End If
                EndFor

                ProjectileScript <- CurrentProjectile.GetComponent(ProjectileScript)()
                ProjectileScript.showReadouts(True)
            End If
        End If
    End
```

Notes:

- The procedure ShowProjectileNumbers() places UI boxes with numbers in over each projectile in the scene to show which one is which.
- The procedure OnPointerExit() is called when the curser leaves the dropdown UI element box.
- The procedure OnPointerEnter() is called when the curser enters the dropdown UI element box.
- The procedure UpdateDropDownOptions() displays each projectile as a listing in the drop down where they can be selected.
- The procedure UpdateGraphs() forwards which projectile is currently selected to its corresponding graphing class.

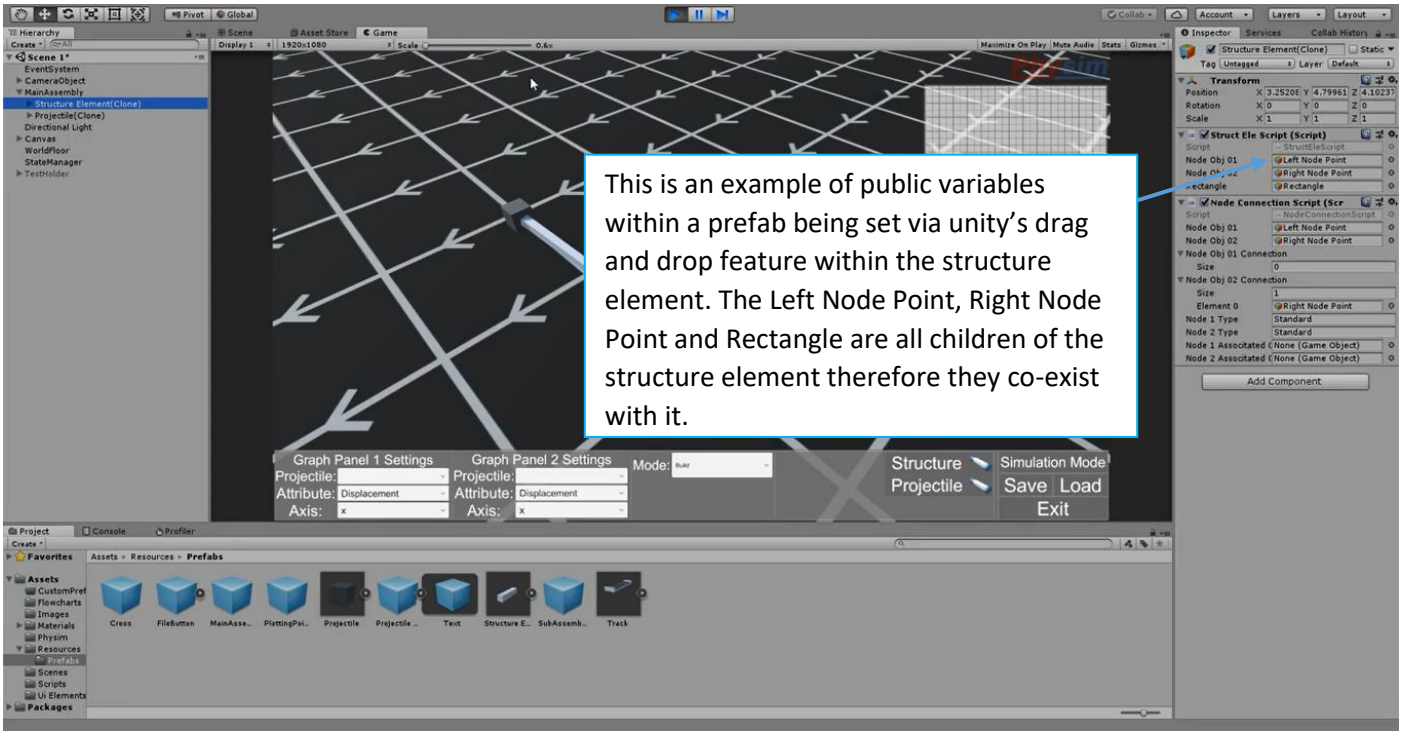
Section 3 - Technical Solution

3.1 Implementation Using UnityEngine

For my program, I have decided to use the 3D graphics engine Unity as it is a powerful and versatile system with similar to real life physics.

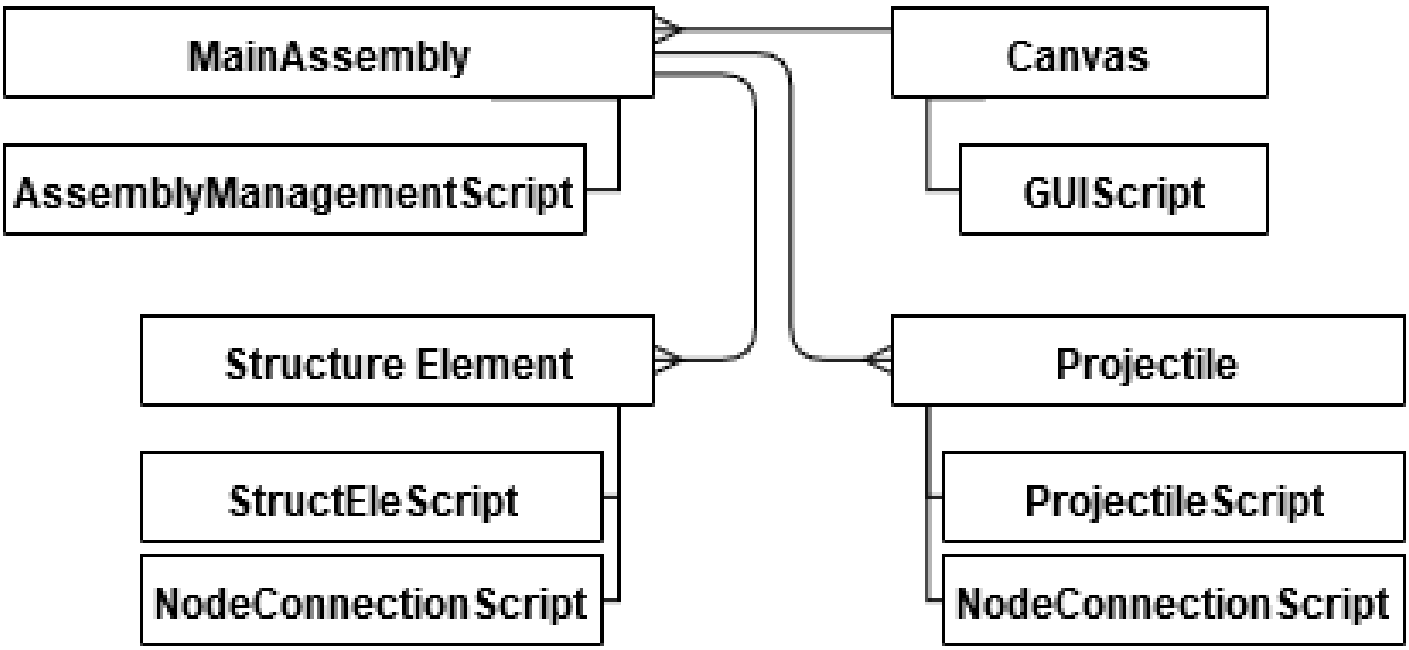
The class structure in Unity is such that GameObjects in the hierarchy act as contianers for components (classes) such as 3D meshes, box colliders and custom C# scripts. The scripts (classes) attached to each gameobject must be a class of MonoBehaviour, and since they can be attached to many components they can have many instances of the same class executing at the same time.

In addition, Unity has a drag-and-drop feature for public variables within monobehaviours which I have made use of to decrease the amount of expensive Find calls for objects existing throughout the entire execution of the program. This has been used in Prefabs, which are pre-made complex GameObjects with preset variable values to allow easy instantiation of multiple instances of the same object (eg: plotting points on the graph). Below is an example of this.



In the photo above, you can also see the hierarchy to the left of the screen. Each object can have a parent or a child, however the child only inherits the movement from the parent, not any other classes. The parent therefore acts as a container for it's child classes.

Due to the way that classes in unity can be set as components of a GameObject, multiple instances of the same classes can exist at any one time within a scene. The following entity relationship diagram shows how this works in Physim, where MainAssembly, Canvas, Structure Element and projectile are all GameObjects and the rest are C# classes which inheret monobehaviour (so are effectively components of their respective GameObjects).



3.2 Complex Programming Index

All the complex programming locations listed below are highlighted in bright green within the Source Code itself for easy reference.

3.3.1 Contains:

- Recursive depth first graph traversal.
- Linear search of a list of 2 element arrays.
- FollowCurser coroutine which makes a clicked node follow the curser.

3.3.2 Contains:

- GameState events which can be subscribed to by any procedure globally.

3.3.4 Contains:

- The UpdateLocalPosition procedure which makes the cubioid of the structure element translate, scale and rotate to meed each node at either end.

3.3.5 Contains:

- The UpdateLocalPosition procedure which makes the cubioid and the other node of the projectile translate to meet the other node at the other side of the cuboid.
- The FixedUpdate procedure which contains the calculation of Displacement, Velocity, Acceleration and force.

3.3.6 Contains:

- The UsePhysics procedure which sets up all the Pivots and SplitJoints and makes anchored subassembilies anchored.

3.3.7 Contains:

- The Orient coroutine which moves the camera to a specific location given an input.
- The Simulate coroutine which makes the camera move to fit all of the nodes within the frame during a simulation.

3.3.8 Contains:

- The LocateRedundantPoints function which returns all points that lie outside of veiw to be recycled.

3.3.9 Contains:

- The ShowProjectileNumbers procedure which displays which projectile is which when the mouse is hovering over a projectile drop down.

3.3.10 Contains:

- The ConvertToSerialisableClass which creates an abstraction of the MainAssembly into AssemblyData and AssemblyElement classes which can then be serialised into JSON format.
- The ConvertFromSerialisableClass which uses AssemblyData and AssemblyElement classes from JSON format to reconstruct the MainAssembly.

3.3.11 & 3.3.12 Contain:

- The UpdateFileList procedure which displays all files with the .phys extension in a persistant data location in appData on a machine as buttons in a scroll view GUI object.

3.3 SourceCode

3.3.1 AssemblyManagementScript

This class is of type MonoBehaviouir and is attached to the MainAssembly gameobject. It manages the construction and destruction of simulation assembilies.

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

[System.Serializable]
public class AssemblyManagementScript : MonoBehaviour
{
    private bool EditMode = false;
    [SerializeField]
    private List<GameObject> Node = new List<GameObject>();
    [SerializeField]                                        // These variables must be serialized to ensure that their
value is preserved when the MainAssembly class is cloned.
    private List<bool> MouseDownFlag = new List<bool>();
    [SerializeField]
    private List<bool> MouseOverFlag = new List<bool>();
    private Vector3 NodePos;
    private Vector3 MousePos;
    private float intersectionRange;
    private const int RangeMultiplier = 100;
    private bool Moving;
    private List<bool> removedFlag = new List<bool>();
    private bool MouseUp;
    private bool BreakFlag;
    private GameObject Canvas;
    private GUIScript gUIScript;
    private bool JustMoved = false;
    [SerializeField]
    private List<GameObject[]> Pivots = new List<GameObject[]>();   // Pivots are strictly between 2 nodes only,
therefore they are stored in a list of 2 element arrays.
    [SerializeField]
    private List<GameObject> Anchors = new List<GameObject>();
    [SerializeField]
    private List<GameObject[]> SplitJoints = new List<GameObject[]>();  // Split joints are strictly between 2 nodes
only, therefore they are stored in a list of 2 element arrays.
    private GameObject OtherNode;
    private bool firstNode = true;
    private List<GameObject> Projectiles = new List<GameObject>();
    private string mode;
    [SerializeField]
```

```
    private List<GameObject> AssemblyElements = new List<GameObject>();
    private int locationIndex = -1;
    private bool snapFlag;
    private List<GameObject> Visited = new List<GameObject>();
    private List<GameObject> SubAssembly = new List<GameObject>();

    public Camera VeiwCamera;   // These variables are public so that they can be assigned in the inspector
    public GameObject SubAssemblyObject;

    public void AddSceneObject(GameObject obj)  // Called to add a scene object
    {
        AssemblyElements.Add(obj);
    }

    public void RemoveAnchors(GameObject obj)   // Removes a given anchor from the list of anchors
    {
        Anchors.Remove(obj);
        Anchors.TrimExcess();   // Sorts the list, shuffling it along to fill any spaces and deletes excess.
    }

    public void DeleteSceneObject(GameObject obj)   // Called when a scene object needs to be deleted.
    {
        AssemblyElements.Remove(obj);
        obj.GetComponent<NodeConnectionScript>().OnDelete(this);    // Calls the OnDelete() procedure within the
NodeConnectionScript attached to the gameObject.
        Destroy(obj);   // Physically deletes the object.
    }

    public void UpdateProjectilesList(GameObject Proj, bool isInScene)  // Updates the projeciles list with a given
projectile depending whether its active or not.
    {
        if (isInScene == true)  // If its active
        {
            Projectiles.Add(Proj);
        }
        else   // If its inactive
        {
            if (Projectiles.IndexOf(Proj) != -1)    // If the projectile exists in the list of projectiles
            {
                Projectiles.RemoveAt(Projectiles.IndexOf(Proj));    // Remove the projectile from the list.
            }
        }
        Projectiles.TrimExcess();
    }

    public List<GameObject> getObjList()    // Returns the list of all the nodes and the projectiles together for use
by the CameraManagementScript.
    {
        List<GameObject> objs = Node.Concat<GameObject>(Projectiles).ToList<GameObject>();
        return objs;
    }

    public void RemoveNode(GameObject node) // Is called when a node needs to be removed after an object is deleted.
    {
        Node.Remove(node);
        Node.TrimExcess();
    }

    public void UpdateTypeList(GameObject node, string type, string oldType)    // Updates the Pivots, Anchors and
Split joint arrays with the node type data. This function is called twice from the two associated nodes of split
joints and pivots, therefore the order in which this is called is monumental.
    {
        if (type == "SplitJoint" || oldType == "SplitJoint" || type == "Pivot" || oldType == "Pivot")   // Used to
keep track of the nodes listed in the lists of arrays.
        {
            if (firstNode == true)  // If it's the first node it will put it in a variable and wait for the second
node.
            {
                OtherNode = node;    // Keeps track of the first node.
                firstNode = false;
            }
            else
            {
                if (oldType == "Pivot") // Removes the old Pivots from the lists.
                {
                    try
```

```
                {
                    Pivots.RemoveAt(FindListIndex(Pivots, node, OtherNode));
                }
                catch
                {
                    Debug.Log("Fatal Error: Unable to locate Pivot");
                }
            }
            else if (oldType == "SplitJoint") // Removes the old SplitJoints from the lists.
            {
                try
                {
                    SplitJoints.RemoveAt(FindListIndex(SplitJoints, node, OtherNode));
                }
                catch
                {
                    Debug.Log("Fatal Error: Unable to locate SplitJoint");
                }
            }

            if (type == "Pivot")    // Adds the Pivots to the list.
            {
                GameObject[] pvt = { node, OtherNode };
                Pivots.Add(pvt);
            }
            else if (type == "SplitJoint")    // Adds the SplitJoints to the list.
            {
                GameObject[] sjt = { node, OtherNode };
                SplitJoints.Add(sjt);
            }
            firstNode = true;
        }
    }
    if (oldType == "Anchor")
    {
        Anchors.Remove(node);
    }
    if (type == "Anchor")
    {
        Anchors.Add(node);
    }
}
    private int FindListIndex(List<GameObject[]> list, GameObject node, GameObject other)   // Returns the index of a pivot/split joint in an array. If it's not present, -1 is returned.
    {
        int index = -1;
        int i = 0;
        if (other != null)
        {
            foreach (GameObject[] ary in list)  // Iterates through the list
            {
                if (node == ary[0] && other == ary[1])  // If they appear in this order
                {
                    index = i;   // The list index of the item being searched for = i.
                    break;
                }
                else if (other == ary[0] && node == ary[1]) // Or if they appear in the other order
                {
                    index = i;  // The list index of the item being searched for = i.
                    break;
                }
                i++;
            }
        }
        else            // Allows only one node to be passed to the procedure.
        {
            foreach (GameObject[] ary in list)
            {
                if (node == ary[0])
                {
                    index = i;
                    break;
                }
                else if (node == ary[1])
                {
                    index = i;
```

```
                    break;
                }
                i++;
            }
        }
        return index;
    }

    private void AddConnections(GameObject node, GameObject otherNode, NodeConnectionScript nodeConnectionScript)
// Is called upon a connection between nodes being made. Adds the other node to a given node's connected list as well
as the other nodes connected nodes & vice versa.
    {
        StartCoroutine(nodeConnectionScript.UpdateConnection(node, otherNode, true));   // Updates this node.
        GameObject OtherObject = otherNode.transform.parent.parent.gameObject;
        NodeConnectionScript otherNodeConnectionScript = OtherObject.GetComponent<NodeConnectionScript>();
        List<GameObject> otherNodeList = otherNodeConnectionScript.GetNodeList(otherNode);  // The nodes connected to
the other node.
        for (int i = 0; i < otherNodeList.Count; i++)
        {
            GameObject tempObject = otherNodeList[i].transform.parent.parent.gameObject;
            NodeConnectionScript tempNodeConnectionScript = tempObject.GetComponent<NodeConnectionScript>();
            StartCoroutine(nodeConnectionScript.UpdateConnection(node, otherNodeList[i], true));
            StartCoroutine(tempNodeConnectionScript.UpdateConnection(otherNodeList[i], node, true));
        }
        StartCoroutine(otherNodeConnectionScript.UpdateConnection(otherNode, node, true));  // Updates the other node.
    }

    public void RemoveConnections(GameObject node, GameObject otherNode, NodeConnectionScript nodeConnectionScript) //
Is called upon a node in the scene being clicked. Removes the other node from a given node's connected list as well as
the other nodes connected nodes & vice versa.
    {
        nodeConnectionScript.UpdateNodeType(node, "Standard");  // Turns the node back into a standard node since it
has been disconnected.
        GameObject OtherObject = otherNode.transform.parent.parent.gameObject;
        NodeConnectionScript otherNodeConnectionScript = OtherObject.GetComponent<NodeConnectionScript>();
        StartCoroutine(otherNodeConnectionScript.UpdateConnection(otherNode, node, false)); // Updates the other node.
        List<GameObject> otherNodeList = otherNodeConnectionScript.GetNodeList(otherNode);  // The nodes connected to
the other node.
        for (int i = 0; i < otherNodeList.Count; i++)
        {
            GameObject tempObject = otherNodeList[i].transform.parent.parent.gameObject;
            NodeConnectionScript tempNodeConnectionScript = tempObject.GetComponent<NodeConnectionScript>();
            StartCoroutine(tempNodeConnectionScript.UpdateConnection(otherNodeList[i], node, false));
        }
        StartCoroutine(nodeConnectionScript.ClearConnectionList(node));
    }

    private void OnEnable() // Is called once upon the object being enabled in the Higherarchy
    {
        snapFlag = false;
        VeiwCamera = Camera.main;
        Canvas = GameObject.Find("Canvas");
        gUIScript = Canvas.GetComponent<GUIScript>();
        mode = gUIScript.GetMode();
    }

    public void OnEditMode()    // Is called once when in edit mode by GUIScript.
    {
        ShowAllSpecialNodes();  // Makes sure all the special nodes are visible.
        EditMode = true;
        StartCoroutine(CheckCurserPosition());  // Starts checking if the curser is over any node in the scene.
    }

    public void OnSimulationMode() // Is called once when in simulation mode by GUIScript.
    {
        EditMode = false;
        Assemble(); // To build the structure into subassembilies
        applyPhysics(); // Make the structure use physics
    }

    public void SetEditMode(bool set)   // Called to update the current status of EditMode locally
    {
        EditMode = set;
    }

    public void clearLists()    // Clears the lists with stored information ready to be used.
```

```
    {
        Node.Clear();
        MouseDownFlag.Clear();
        MouseOverFlag.Clear();
    }
    public void UpdateNodeList(GameObject parent, GameObject node1, GameObject node2)   // Called to intialise the
nodes when a new placeable object is instantiated into the scene
    {
        Node.Add(node1);
        MouseDownFlag.Add(false);   // MouseDownFlag and MouseOVerFlag act as lists parallel to the list of nodes.
        MouseOverFlag.Add(false);
        Node.Add(node2);
        MouseDownFlag.Add(false);
        MouseOverFlag.Add(false);
    }

    private IEnumerator CheckCurserPosition()  // Checks if the current curser is over any node in the scene and
whether it has been clicked
    {
        BreakFlag = false;
        while (EditMode == true && Moving == false)
        {
            if (mode != "Delete")
            {
                for (int i = 0; i < Node.Count; i++)    // Iterates through the list of nodes
                {
                    Vector3 MousePos = Input.mousePosition;
                    NodePos = VeiwCamera.WorldToScreenPoint(Node[i].transform.position);    // Gets the point of the
current node on the screen
                    intersectionRange = RangeMultiplier /
(VeiwCamera.WorldToScreenPoint(Node[i].transform.position).z);
                    if ((NodePos.x - intersectionRange) < (MousePos.x) && (MousePos.x) < (NodePos.x +
intersectionRange) && (NodePos.y - intersectionRange) < (MousePos.y) && (MousePos.y) < (NodePos.y +
intersectionRange)) // If the curser lies within a range of values around the node.
                    {
                        if (Input.GetMouseButtonDown(0))    // If right mouse is down
                        {
                            for (int y = 0; y < MouseDownFlag.Count; y++)   // Iterates through each MouseDownFlag
value, which corresponds to each node (parallel to the list of nodes).
                            {
                                if (MouseDownFlag[y] == false)  // If the mouse isnt already down on a node
                                {
                                    MouseDownFlag[i] = true;    // Make the mouse down on this node
                                    Moving = true;
                                    StartCoroutine(MouseDownOnNode(Node[i], i));    // Called once when the mouse is
down on a node
                                    BreakFlag = true;   // Signal to break the first iteration after a node is
clicked.
                                    yield break;
                                }
                            }
                        }
                        else   // If right mouse is not down
                        {
                            Moving = false; // Not moving
                            MouseDownFlag[i] = false;   // The mouse is not down
                            if (MouseOverFlag[i] == false)  // If the mouse wasnt over the node before
                            {
                                MouseOverFlag[i] = true;
                                MouseOverNode(Node[i]);  // The mouse is now recognised as being over the node
                            }
                        }
                    }
                    else   // If the mouse lies outside the intersection range of the node
                    {
                        MouseOverFlag[i] = false;   // The mouse is not over the node
                        if (MouseDownFlag[i] == false)  // If the mouse is not down on the node
                        {
                            GameObject placeableObj = Node[i].transform.parent.parent.gameObject;
                            NodeConnectionScript NCS = placeableObj.GetComponent<NodeConnectionScript>();   // Gets
the node's associated NodeConnectionScript attached to its parent placeable object.
                            if (NCS.GetNodeType(Node[i]) == "Standard")
                            {
                                Node[i].SetActive(false);   // Hide the node's sphere.
                            }
                        }
```

```
                    }
                    if (BreakFlag == true)  // If the iteration should break
                    {
                        yield break;    // Break the iteration
                    }
                }
            }
            else  // If in delete mode
            {
                for (int i = 0; i < AssemblyElements.Count; i++)    // Iterate through the list of assembly elements
                {
                    Vector3 MousePos = Input.mousePosition;
                    Vector3 ObjPosition =
VeiwCamera.WorldToScreenPoint(AssemblyElements[i].transform.GetChild(1).position);  // Finds the body of the assembly
object (the mesh is attached to the second child in all cases), then gets the on-screen position of this.
                    if ((ObjPosition.x - intersectionRange) < (MousePos.x) && (MousePos.x) < (ObjPosition.x +
intersectionRange) && (ObjPosition.y - intersectionRange) < (MousePos.y) && (MousePos.y) < (ObjPosition.y +
intersectionRange))    // If the mouse lies within a range around the object.
                    {
                        if (Input.GetMouseButtonDown(0))    // If the mouse is clicked
                        {
                            DeleteSceneObject(AssemblyElements[i]); // Delete the GameObject
                        }
                    }
                    yield return null;
                }
            }
            mode = gUIScript.GetMode(); // Gets the current mode from GUIScript and updates this script's variable.
            yield return null;
        }
        yield return null;
    }

    private IEnumerator MouseDownOnNode(GameObject node, int flagPos)  // Is called once when a node is clicked and
decides what to do based on the current mode.
    {
        GameObject Canvas = GameObject.Find("Canvas");
        GUIScript gUIScript = Canvas.GetComponent<GUIScript>();
        GameObject placeableObj = node.transform.parent.parent.gameObject;
        NodeConnectionScript NCS = placeableObj.GetComponent<NodeConnectionScript>();
        string mode = gUIScript.GetMode();
        if (mode == "Build")
        {
            StartCoroutine(FollowCurser(node, flagPos)); // Starts the coroutine to make the node follow the curser
        }
        else   // If selecting pivot/ split joint/ anchor
        {
            NCS.UpdateNodeType(node, mode); // Update the type of node based on the current mode
            Moving = false;
            MouseUp = false;
            yield return new WaitUntil(() => MouseUp = true);   // Waits until left mouse is released
            StartCoroutine(CheckCurserPosition());  // Starts checking the position of the curser once again
        }
    }
    private void OnMouseUp()    // Is called by UnityEngine when left mouse is lifted.
    {
        MouseUp = true;
    }

    private void MouseOverNode(GameObject node)    // Is called once when the mouse is over a node.
    {
        node.SetActive(true);   // Makes the node visible
    }

    public IEnumerator FollowCurser(GameObject node, int flagPos)    // Makes whichever node has been clicked follow
the curser.
    {
        GameObject PlaceableObject = node.transform.parent.parent.gameObject;   // Gets the parent Placeable Object of
the node.
        NodeConnectionScript nodeConnectionScript = PlaceableObject.GetComponent<NodeConnectionScript>();   // Locates
the NodeConnectionScript attached to this placeable object
        float initialPosRelToCam = VeiwCamera.WorldToScreenPoint(node.transform.position).z;    // This is the
perpendicular distance from the camera object to the node's position.
        Vector2 mousePos = new Vector2();   // Initialises lists
        List<GameObject> OtherNodes = new List<GameObject>();
```

```
        if (Node.Count != 0)    // If there is any nodes in the scene
        {
            for (int i = 0; i < Node.Count; i++)    // Iterate through the nodes
            {
                if (node.transform.parent.gameObject != Node[i].transform.parent.gameObject)    // If the node current
node isnt part of the same placeable object as the node thats been clicked.
                {
                    OtherNodes.Add(Node[i]);    // Populates a list of all nodes in the scene other than the given
node and the one attached to the same object.
                    removedFlag.Add(true);
                }
            }
        }
        while ((MouseDownFlag[flagPos] == true) && (EditMode == true))  // While the mouse is down on the node and the
simulation is in edit mode.
        {
            NodePos = VeiwCamera.WorldToScreenPoint(node.transform.position);   // Finds the location of the node on
the screen.
            mousePos.x = Input.mousePosition.x;
            mousePos.y = Input.mousePosition.y;
            if (OtherNodes.Count != 0)  // If there is any snappable nodes in the scene.
            {
                for (int i = 0; i < OtherNodes.Count; i++)
                {
                    Vector3 OtherNodePos = VeiwCamera.WorldToScreenPoint(OtherNodes[i].transform.position); // Finds
the location of the other node on the screen
                    intersectionRange = RangeMultiplier / (OtherNodePos.z); // Calculates on the intersection range
based on the distance the other node is from the camera object
                    if ((OtherNodePos.x - intersectionRange) < mousePos.x && mousePos.x < (OtherNodePos.x +
intersectionRange) && (OtherNodePos.y - intersectionRange) < mousePos.y && mousePos.y < (OtherNodePos.y +
intersectionRange))   // If the mouse lies within a range around the other node
                    {
                        NodeConnectionScript NCS =
Node[i].transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>(); // Locates the NCS of the other node
object
                        if (snapFlag == false && (NCS.GetNodeType(Node[i]) != "Pivot" && NCS.GetNodeType(Node[i]) !=
"SplitJoint"))    // Makes sure that the node cannot snap to any pivots or split joints.
                        {
                            Vector3 newPosition = OtherNodes[i].transform.position;
                            node.transform.position = newPosition;  // Sets the current node's position to the other
node so that they snap together.
                            snapFlag = true;
                            AddConnections(node, OtherNodes[i], nodeConnectionScript);  // Add the connection between
the two snapped nodes
                            locationIndex = i;
                        }
                    }
                    else    // If the node lies outside the snapping range
                    {
                        if (i == locationIndex) // If the node used to be connected to the other node
                        {
                            locationIndex = -1;
                            RemoveConnections(node, OtherNodes[i], nodeConnectionScript);   // Remove the connections
between the two nodes
                            StartCoroutine(WaitUntilCurserFree(OtherNodePos));  // Starts a coroutine which waits
until the curser has left a range larger than needed to snap to the other node to prevent flickering between being
connected and not.
                        }
                        if (snapFlag == false)  // If the node is not snapped to anything else
                        {
                            node.transform.position = VeiwCamera.ScreenToWorldPoint(new Vector3(mousePos.x,
mousePos.y, initialPosRelToCam));  // Make the node follow the position of the curser, maintaining it's perpendicular
distance away from the camera
                        }
                    }
                }
            }
            else  // If there are no other snappable nodes
            {
                node.transform.position = VeiwCamera.ScreenToWorldPoint(new Vector3(mousePos.x, mousePos.y,
initialPosRelToCam));  // Make the node follow the position of the curser, maintaining it's perpendicular distance
away from the camera
            }
            if (node.transform.parent.parent.gameObject.GetComponent<StructEleScript>() != null)    // If the node is
attached to a structure element
            {
```

```
                StructEleScript ObjectScript = PlaceableObject.GetComponent<StructEleScript>();
                ObjectScript.UpdateLocalPosition(); // Tells the Object-specific script attached to the placeable
object to account for the movement of the node
            }
            else if (node.transform.parent.parent.gameObject.GetComponent<ProjectileScript>() != null)  // If its a
projectile
            {
                ProjectileScript ObjectScript = PlaceableObject.GetComponent<ProjectileScript>();
                ObjectScript.UpdateLocalPosition(); // Tells the Object-specific script attached to the placeable
object to account for the movement of the node
            }
            else     // This should never be reached, and if it is, the program is set up incorrectly within Unity
Editor.
            {
                Debug.Log("Fatal error: Object type not found!");
            }
            if (Input.GetMouseButtonUp(0))  // If the left mouse button is up
            {
                MouseDownFlag[flagPos] = false; // Updates the MouseDownFlag list to show this.
            }
            yield return null;
        }
        snapFlag = false;
        Moving = false;
        OtherNodes.Clear(); // Clears the list of other nodes to tidy up
        StartCoroutine(CheckCurserPosition());  // Start checking if the curser is over any node in the scene once
again.
    }

    private IEnumerator WaitUntilCurserFree(Vector3 OtherNodePos)   // This coroutine waits until the mouse has left a
region which is greater than the snapping region of the node to signal that the node is no longer snapped, to prevent
flickering between being snapped and not.
    {
        Vector2 mousePos = new Vector2();
        mousePos.x = Input.mousePosition.x;
        mousePos.y = Input.mousePosition.y;
        while ((OtherNodePos.x - (intersectionRange + 2)) < mousePos.x && mousePos.x < (OtherNodePos.x +
(intersectionRange + 2)) && (OtherNodePos.y - (intersectionRange + 2)) < mousePos.y && mousePos.y < (OtherNodePos.y +
(intersectionRange + 2)))    // While the mouse lies within a slightly larger range than needed to snap.
        {
            mousePos.x = Input.mousePosition.x;
            mousePos.y = Input.mousePosition.y;
            yield return null;
        }
        snapFlag = false;   // Effectively un-snaps the node, as this variable is checked in every iteration inside
the previous coroutine.
        yield return null;
    }
    public IEnumerator SimpleFollowCurser(GameObject gameObject)    // Any object follow the curser until it it
clicked. Used when instantiating objects.
    {
        Vector3 mousePos = new Vector3();
        while (Input.GetMouseButtonDown(0) == false)    // While left mouse is not clicked.
        {
            mousePos.x = Input.mousePosition.x;
            mousePos.y = Input.mousePosition.y;
            mousePos.z = Input.mousePosition.z;
            gameObject.transform.position = VeiwCamera.ScreenToWorldPoint(new Vector3(mousePos.x, mousePos.y, 10f));
// The object follows the curser.
            yield return null;
        }
        yield return null;
    }

    public void ShowAllSpecialNodes()   // This procedure sets all the non-standard nodes to active.
    {
        foreach (GameObject[] piv in Pivots)    // Iterates through the list of arrays of pivots, and sets each one as
active so they can be seen.
        {
            piv[0].SetActive(true);
            piv[1].SetActive(true);
        }
        foreach (GameObject[] SJ in SplitJoints)    // Iterates through the list of arrays of split joints, and sets
each one as active so they can be seen.
        {
            SJ[0].SetActive(true);
```

```
            SJ[1].SetActive(true);
        }
        foreach (GameObject an in Anchors)  // Iterates through the list of anchors and sets each one as active so
they can be seen.
        {
            an.SetActive(true);
        }
    }

    public void Assemble()  // Assembles the MainAssembly into subassemblies.
    {
        List<GameObject> sceneObject = new List<GameObject>();
        List<GameObject> unvisitedObject = new List<GameObject>();
        int numberOfObjects = this.transform.childCount;
        Visited.Clear();
        for (int i = 0; i < numberOfObjects; i++)
        {
            sceneObject.Add(transform.GetChild(i).gameObject);    // Adds all the placeable objects in the scene into
a list.
        }
        List<GameObject> VisitedTemp = new List<GameObject>();
        unvisitedObject = sceneObject;  // All the objects are initially unvisited.
        while (unvisitedObject.Count != 0)   // Finds and allocates each individual subassembly.
        {
            VisitedTemp.Clear();
            VisitedTemp = DepthFirstTrav(unvisitedObject[0], Visited);  // Conducts a Depth-First graph traversal of
the structure, starting from the first element in the list.
            CreateSubAssembly(VisitedTemp); // Creates a subassembly containg all the visited nodes in that
subassembly
            unvisitedObject = unvisitedObject.Except(VisitedTemp).ToList(); // Removes any now visited items from the
unvisited list.
            unvisitedObject.TrimExcess();   // Tidies up list and removes empty elements.
        }
    }

    private void CreateSubAssembly(List<GameObject> Child)  // Creates a new sub assembly with the given nodes as it's
children.
    {
        GameObject SubAssObj = Instantiate(SubAssemblyObject, new Vector3(0, 0, 0), new Quaternion(0, 0, 0, 0));    //
Instantiates a new SubAssembly object in the scene.
        SubAssObj.transform.parent = this.transform;    // Makes the new SubAssembly a child of the MainAssembly
object
        SubAssembly.Add(SubAssObj); // Adds the new SubAssembly to the list of subassembilies in the scene.
        for (int i = 0; i < Child.Count; i++)
        {
            Child[i].transform.parent = SubAssObj.transform;    // Makes all given GameObjects the child of the new
SubAssembly
        }
    }
    private List<GameObject> DepthFirstTrav(GameObject Obj, List<GameObject> VisitedList)    // Perfomes a recursive
Depth-First-Traversal of the connections to a given node and returns all the connections.
    {
        List<GameObject> connectedObj = new List<GameObject>();
        NodeConnectionScript NCS = Obj.GetComponent<NodeConnectionScript>();
        connectedObj = NCS.getConnectionList();
        VisitedList.Add(Obj);
        for (int i = 0; i < connectedObj.Count; i++)
        {
            if (VisitedList.Contains(connectedObj[i]) == false) // If the node hasn't already been visited
            {
                VisitedList.Concat(DepthFirstTrav(connectedObj[i], VisitedList));    // Finds the objects indirectly
connected to the given one (Cousin objects).
            }
        }
        return VisitedList;
    }

    private void applyPhysics()  // Makes the subassembilies use physics by adding components.
    {
        for (int i = 0; i < Pivots.Count; i++)  // Iterates through the list of pivots.
        {
            SubAssemblyScript SAS =
Pivots[i][0].transform.parent.parent.parent.gameObject.GetComponent<SubAssemblyScript>();
            SAS.AddPivot(Pivots[i][0], Pivots[i][1].transform.parent.parent.parent.gameObject);    // Tells the
subAssbembly where it's pivots are if it has any.
        }
```

```
        for (int i = 0; i < SplitJoints.Count; i++)
        {
            SubAssemblyScript SAS =
SplitJoints[i][0].transform.parent.parent.parent.gameObject.GetComponent<SubAssemblyScript>();
            SAS.AddSplitJoint(SplitJoints[i][0], SplitJoints[i][1].transform.parent.parent.parent.gameObject);    //
Tells the subAssbembly where it's split joints are if it has any.
        }
        for (int i = 0; i < Anchors.Count; i++)
        {
            SubAssemblyScript SAS =
Anchors[i].transform.parent.parent.parent.gameObject.GetComponent<SubAssemblyScript>();
            SAS.IsAnchored(true);   // Tells the subAssembly if it's anchored.
        }
        foreach (GameObject subAssy in SubAssembly)
        {
            SubAssemblyScript SAS = subAssy.GetComponent<SubAssemblyScript>();
            SAS.UsePhysics();   // Tells each SubAssembly to use physics depending on its properties.
        }
    }

    public void ReconstructConnections()    // Called to reconstruct connetctions between a newly loaded main
assembly.
    {
        foreach (GameObject node in Node)
        {
            NodeConnectionScript NCS = node.transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
            if (node != null)   // Makes sure that the current node exists and therefore hasn't been visited before
            {
                foreach (GameObject otherNode in Node)
                {
                    //NodeConnectionScript otherNCS =
otherNode.transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                    if (node != otherNode)
                    {
                        if (node.transform.position == otherNode.transform.position)    // If they share the same
position then they are connected.
                        {
                            AddConnections(node, otherNode, NCS);   // Add the connection between the two nodes
                        }
                    }
                }
            }
            if (NCS.GetNodeType(node) != NCS.getTempNodeType(node)) // If the node type hasnt already been changed via
a connection to another node.
            {
                NCS.UpdateNodeType(node, NCS.getTempNodeType(node));    // Set the node type.
            }
        }
    }

    public GameObject getProjectile(int i)  // Returns a projectile given it's index in the projectiles list.
    {
        if (Projectiles.Count != 0)
        {
            return Projectiles[i];
        }
        else
        {
            return null;
        }
    }

    public List<GameObject> getProjectileList() // Returns the list of currently active projectiles
    {
        return Projectiles;
    }
}
```

3.3.2 GUIScript

This class is of type MonoBehaviouir and is attached to the Canvas gameObject (plane on which the UI is displayed). It contains the algorithms to act as the SimulationManager, but is named GUIScript as it interfaces between the GUI and the world.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class GUIScript : MonoBehaviour
{

    private GraphScript GraphingPanel1Script;
    private AssemblyManagementScript assemblyManagementScript;
    private GameObject SimulationPanel;
    private GameObject EditPanel;
    private GameObject currentAssembly;
    private GameObject StartButton;
    private GameObject SavePanel;
    private GameObject LoadPanel;
    private GameObject NoticePanel;
    private string mode;
    private bool simulating;
    private bool editing;
    private Dropdown DD1;
    private Dropdown DD2;
    private GameObject ProjectileNumber;
    private List<GameObject> CurrentProjectileNumbers;

    public GameObject MainAssembly; // These are assigned in the UnityEditor
    public Camera MainCamera;

    public delegate void ToggleEditMode();  // These must be public as they are events
    public static event ToggleEditMode toggleEditMode;
    public delegate void ToggleSimulationMode();
    public static event ToggleSimulationMode toggleSimulationMode;
    public delegate void Simulate();
    public static event Simulate OnSimulate;
    public delegate void SimulateEnd();
    public static event SimulateEnd OnSimulateEnd;


    void Start()    // Called at the start of the program's execution
    {
        GraphingPanel1Script = GameObject.Find("/Canvas/GraphingPanel1").GetComponent<GraphScript>();
        DD1 = GameObject.Find("/Canvas/EditPanel/GraphingPanel1Settings/ProjectileDropdown").GetComponent<Dropdown>();
        DD2 = GameObject.Find("/Canvas/EditPanel/GraphingPanel2Settings/ProjectileDropdown").GetComponent<Dropdown>();
        editing = false;
        mode = "Build";
        assemblyManagementScript = MainAssembly.GetComponent<AssemblyManagementScript>();
        assemblyManagementScript.clearLists();
        SimulationPanel = GameObject.Find("SimulationPanel");
        SimulationPanel.SetActive(true);
        EditPanel = GameObject.Find("EditPanel");
        EditPanel.SetActive(false);
        SavePanel = GameObject.Find("SavePanel");
        SavePanel.SetActive(false);
        LoadPanel = GameObject.Find("LoadPanel");
        LoadPanel.SetActive(false);
        NoticePanel = GameObject.Find("NoticePanel");
        NoticePanel.SetActive(false);

    }

    public AssemblyManagementScript replaceAssembly(GameObject old) // Replaces the current assembly with a new one
when going back into edit mode
    {
        if (currentAssembly != null)
        {
            Destroy(old);
            currentAssembly.SetActive(true);
            MainAssembly = currentAssembly;
            MainAssembly.name = "MainAssembly";
            assemblyManagementScript = MainAssembly.GetComponent<AssemblyManagementScript>();
        }
        return assemblyManagementScript;
    }

    public void SimulationRun(GameObject StartBtn)  // Globally called once when "Start" button is clicked
    {
```

```
        simulating = true;
        OnSimulate();   // Triggers the OnSimulate event.
        StartCoroutine(GraphingPanel1Script.setElapsedTimeText());  // Starts setting the readout values.
        StartButton = StartBtn;
        StartButton.SetActive(false);   // Hides the start button so that the stop button is shown, as the stop button
is behind it.
        Debug.Log("Simulating");
    }

    public void SimulationStop(GameObject StopBtn)  // Globally called once when "Stop" button is clicked
    {
        simulating = false;
        StartButton.SetActive(true);    // Shows the start button.
        OnSimulateEnd();    // Triggers the OnSimulateEnd event.
    }

    public void ResetSimulation()  // Globally called once when "Reset" button is clicked. Switches to edit mode then
back to simulation mode in the next frame.
    {
        if (simulating != true)
        {
            AssemblyManagementScript AMS = replaceAssembly(MainAssembly);
            editing = true; // Switches back to edit mode
            SimulationPanel.SetActive(false);
            EditPanel.SetActive(true);
            StartCoroutine(WaitForNextFrame(AMS, true));
        }
        else
        {
            Debug.Log("You must stop the simulation before you can go into edit mode!.");
            NotifyUser("You must stop the simulation before you can go into edit mode.");
        }
    }

    public bool IsSimulating()  // Returns if the simulation is currently in simulation mode
    {
        return simulating;
    }
    public void EditMode()  // Called once when "Edit Mode" button is clicked
    {
        if (simulating != true)
        {
            editing = true;
            SimulationPanel.SetActive(false);
            EditPanel.SetActive(true);
            AssemblyManagementScript AMS = replaceAssembly(MainAssembly);
            StartCoroutine(WaitForNextFrame(AMS, false));
        }
        else
        {
            Debug.Log("You must stop the simulation before you can go into edit mode!.");
            NotifyUser("You must stop the simulation before you can go into edit mode.");
        }
    }

    public bool isEditing() // Returns if the simulation is currently in edit mode
    {
        return editing;
    }

    private IEnumerator WaitForNextFrame(AssemblyManagementScript AMS, bool isResetting)  // Needs to wait for the
next frame as the gameobject will not be destroyed until the end of the current frame.
    {
        yield return new WaitForFixedUpdate();
        if (toggleEditMode != null) // If the toggleEditMode event if it's not already being triggered.
        {
            toggleEditMode();   // Initiates the toggleEditMode event.
        }
        AMS.OnEditMode();
        if (isResetting == true)    // If it needs to go back to simulation mode afterwards.
        {
            SimulationMode();   // Initiates the SimulationMode event.
        }
    }

    public void SimulationMode()  // Called once when "Simulation Mode" button is clicked
```

```
    {
        if (toggleSimulationMode != null)
        {
            toggleSimulationMode();
        }
        editing = false;
        EditPanel.SetActive(false);
        SimulationPanel.SetActive(true);
        currentAssembly = Instantiate(MainAssembly, new Vector3(0, 0, 0), new Quaternion(0, 0, 0, 0));  // Clones the
mian assembly before it's split into subassemblies.
        currentAssembly.SetActive(false);
        if (MainAssembly != null)
        {
            assemblyManagementScript.OnSimulationMode();
        }
    }

    public void ExitProgram()  // Called once when "Exit" button is clicked
    {
        Application.Quit();
    }

    public void instantiateObject(GameObject placeableObject)   // Is called directly by the GUI instantiate buttons.
    {
        assemblyManagementScript.SetEditMode(false);
        GameObject instantiatedObject = Instantiate(placeableObject, new Vector3(0, 0, 0), Quaternion.identity);
        NodeConnectionScript NCS = instantiatedObject.GetComponent<NodeConnectionScript>();
        NCS.ClearAllLists();
        NCS.InitialiseNodeTypes();
        instantiatedObject.transform.parent = MainAssembly.transform;
        NCS.OnObjectLoaded();
        StartCoroutine(assemblyManagementScript.SimpleFollowCurser(instantiatedObject));
        assemblyManagementScript.OnEditMode();
        if (assemblyManagementScript == null)   // Makes sure its the most recent version of the main assembly.
        {
            assemblyManagementScript = GameObject.Find("MainAssembly").GetComponent<AssemblyManagementScript>();
        }
        assemblyManagementScript.AddSceneObject(instantiatedObject);
    }

    public void OnLoadScene()   // Is called directly by the load buttons
    {
        if (simulating != true)
        {
            LoadPanel.SetActive(true);
        }
        else
        {
            Debug.Log("You must stop this simulation before you can load a new one.");
            NotifyUser("You must stop the simulation before you can load a new one.");
        }
    }

    public void OnSaveScene()   // Is called directly by the save button
    {
        SavePanel.SetActive(true);
    }

    public GameObject getMainAssembly() // Returns the current MainAssembly object
    {
        return MainAssembly;
    }

    public void setMainAssembly(GameObject MA)  // Sets the current MainAssembly object
    {
        MainAssembly = MA;
        assemblyManagementScript = MainAssembly.GetComponent<AssemblyManagementScript>();
    }
    public void ChangeMode(GameObject calledFrom)   // Accessed by the OnValueChanged function of dropdown. Changes
the current building mode.
    {
        int Index = calledFrom.GetComponent<Dropdown>().value;
        List<Dropdown.OptionData> options = calledFrom.GetComponent<Dropdown>().options;
        switch (options[Index].text)
        {
            case "Build":
```

```
            mode = "Build";
            break;
        case "Select Pivot":
            mode = "Pivot";
            break;
        case "Select Anchor":
            mode = "Anchor";
            break;
        case "Select Split Joint":
            mode = "SplitJoint";
            break;
        case "Delete":
            mode = "Delete";
            break;
        default:
            Debug.Log("Fatal Error: Mode not found");
            break;
    }
}

public string GetMode() // Returns the current mode.
{
    return mode;
}

public void SetDisplacementText(Vector3 dis)    // Is called to set the displacement readout text.
{
    try
    {
        GameObject.Find("/Canvas/SimulationPanel/Displacement/Values").GetComponent<Text>().text = dis.ToString();
    }
    catch
    {
    }
}

public void SetVelocityText(Vector3 vel)    // Is called to set the velocity readout text.
{
    try
    {
        GameObject.Find("/Canvas/SimulationPanel/Velocity/Values").GetComponent<Text>().text = vel.ToString();
    }
    catch
    {
    }
}

public void SetAccelerationText(Vector3 Acc)    // Is called to set the acceleration readout text.
{
    try
    {
        GameObject.Find("/Canvas/SimulationPanel/Acceleration/Values").GetComponent<Text>().text = Acc.ToString();
    }
    catch
    {
    }
}

public void SetForceText(Vector3 frc)   // Is called to set the force readout text.
{
    try
    {
        GameObject.Find("/Canvas/SimulationPanel/Force/Values").GetComponent<Text>().text = frc.ToString();
    }
    catch
    {
    }
}

public void NotifyUser(string message)  // Opens a dialogue box with a given message.
{
    Text txt = NoticePanel.transform.GetChild(0).GetChild(2).gameObject.GetComponent<Text>();
    txt.text = message;
    NoticePanel.SetActive(true);
}
```

```
    public void CloseNoticePanel()  // Is called directly by the 'Continue' button in the noticePanel.
    {
        NoticePanel.SetActive(false);
    }

}
```

3.3.3 NodeConnectionScript

This class is of type MonoBehaviouir and is attached to each individual gameObject. It keeps track of the nodes' connections and their types.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class NodeConnectionScript : MonoBehaviour
{
    [SerializeField]                    // These variables must be serialized to ensure that their value is preserved
when the MainAssembly is cloned.
    private List<GameObject> NodeObj01Connection = new List<GameObject>();
    [SerializeField]
    private List<GameObject> NodeObj02Connection = new List<GameObject>();
    [SerializeField]
    private string Node1Type;
    [SerializeField]
    private string Node2Type;
    [SerializeField]
    private GameObject Node1AssocitatedObject;
    [SerializeField]
    private GameObject Node2AssocitatedObject;
    private GameObject Canvas;
    private GUIScript gUIScript;
    private String TempNode1Type;
    private String TempNode2Type;

    public GameObject NodeObj01;       // This variable is public because it is assigned manually in the unity editor.
    public GameObject NodeObj02;       // This variable is public because it is assigned manually in the unity editor.
    [HideInInspector]                  // Hides the following public class in the unity editor Inspector window.
    public Material Pivot;             // This field must be public to be accessed by reflection.
    [HideInInspector]                  // Hides the following public class in the unity editor Inspector window.
    public Material Anchor;            // This field must be public to be accessed by reflection.
    [HideInInspector]                  // Hides the following public class in the unity editor Inspector window.
    public Material SplitJoint;        // This field must be public to be accessed by reflection.
    [HideInInspector]                  // Hides the following public class in the unity editor Inspector window.
    public Material Standard;          // This field must be public to be accessed by reflection.

    public List<GameObject> getNodeObjConnection(int node)  // Returns the list of connected nodes to a given node.
    {
        if (node == 1)
        {
            return NodeObj01Connection;
        }
        else if (node == 2)
        {
            return NodeObj02Connection;
        }
        else
        {
            Debug.Log("Fatal Error: Node not found!");
            return null;
        }
    }

    public void setTempNodeType(GameObject node, string type)   // Is used by the SaveLaodScript to temporarily store
the node type
    {
        if (node == NodeObj01)
        {
            TempNode1Type = type;
        }
        else if (node == NodeObj02)
        {
            TempNode2Type = type;
```

```
        }
        else
        {
            Debug.Log("Fatal Error: node not found!");
        }
    }

    public string getTempNodeType(GameObject node)  // Gets the node type of a given node
    {
        if (node == NodeObj01)
        {
            return TempNode1Type;
        }
        else if (node == NodeObj02)
        {
            return TempNode2Type;
        }
        else
        {
            Debug.Log("Fatal Error: node not found!");
            return null;
        }
    }

    private void OnEnable() // Is called by UnityEngine when the gameobject this script is attached to is enabled
    {
        gUIScript = GameObject.Find("Canvas").GetComponent<GUIScript>();
        GUIScript.toggleEditMode += OnEditMode; // Adds the OnEditMode procedure to the trigger event.
        Pivot = Resources.Load<Material>("PivotPointMaterial"); // Locates the different types of pivot materials
        Anchor = Resources.Load<Material>("AnchorPointMaterial");
        SplitJoint = Resources.Load<Material>("SplitJointMaterial");
        Standard = Resources.Load<Material>("NodePointMaterial");
    }

    private void OnDisable()   // Is called by UnityEngine when the gameobject this script is attached to is disabled
    {
        GUIScript.toggleEditMode -= OnEditMode; // Removes the OnEditMode procedure from the trigger event.
    }

    public void OnDelete(AssemblyManagementScript AMS)  // Is called upon delting the gameObject to tidy up all the
variables.
    {
        foreach (GameObject obj in NodeObj01Connection) // Hides all the attached nodes to node 1.
        {
            obj.SetActive(false);
        }
        foreach (GameObject obj in NodeObj02Connection) // Hides all the attached nodes to node 2.
        {
            obj.SetActive(false);
        }
        AMS.RemoveNode(NodeObj01);  // Removes these nodes from the AMS's lists.
        AMS.RemoveNode(NodeObj02);  // ^^
        if (NodeObj01Connection.Count > 0)
        {
            AMS.RemoveConnections(NodeObj01, NodeObj01Connection[0], this);  // Removes both node's connections when
their object is destroyed.
        }
        if (NodeObj02Connection.Count > 0)
        {
            AMS.RemoveConnections(NodeObj02, NodeObj02Connection[0], this);  // ^^
        }
        if (Node1Type == "Anchor")
        {
            AMS.RemoveAnchors(NodeObj01);
        }
        if (Node2Type == "Anchor")
        {
            AMS.RemoveAnchors(NodeObj02);
        }
    }
    public void OnEditMode()    //  Called once when put into edit mode
    {
        AssemblyManagementScript AMS = gUIScript.getMainAssembly().GetComponent<AssemblyManagementScript>();
        if (this.gameObject.activeInHierarchy == true)  // If the gameObject is currently active in the scene.
        {
            if (Node1AssocitatedObject != null)
```

```
            {
                AMS.UpdateTypeList(NodeObj01, Node1Type, "Standard");       // Must be called in this manner to ensure
that the UdateTypeList procedure recieves both sides of the joint one after another.
                AMS.UpdateTypeList(Node1AssocitatedObject, Node1Type, "Standard");
            }
            if (Node2AssocitatedObject != null)
            {
                AMS.UpdateTypeList(NodeObj02, Node2Type, "Standard");
                AMS.UpdateTypeList(Node2AssocitatedObject, Node2Type, "Standard");
            }
        }
    }

    public string GetNodeType(GameObject node)  // Returns the type of node when passed the node.
    {
        if (node == NodeObj01)
        {
            if (Node1Type == "Pivot")
            {
                return "Pivot";
            }
            else if (Node1Type == "Anchor")
            {
                return "Anchor";
            }
            else if (Node1Type == "SplitJoint")
            {
                return "SplitJoint";
            }
            else
            {
                return "Standard";
            }
        }
        else if (node == NodeObj02)
        {
            if (Node2Type == "Pivot")
            {
                return "Pivot";
            }
            else if (Node2Type == "Anchor")
            {
                return "Anchor";
            }
            else if (Node2Type == "SplitJoint")
            {
                return "SplitJoint";
            }
            else
            {
                return "Standard";
            }
        }
        else
        {
            Debug.Log("Fatal Error: Node not found.");
            return null;
        }
    }

    public void InitialiseNodeTypes()   // Is called once when the object is instantiated to setup the type of node.
    {
        Node1Type = "Standard";
        Node2Type = "Standard";
    }

    private void SetNodeType(string type, GameObject node, string oldType)   // Sets the type of node
    {
        Material NewMaterial = (Material)this.GetType().GetField(type).GetValue(this);  // Finds the material
associated with this type by name and assigns it to NewMaterial.
        node.GetComponentInChildren<MeshRenderer>().material = NewMaterial;     // Assigns the relevant material to
the node.
        AssemblyManagementScript AMS = this.transform.parent.gameObject.GetComponent<AssemblyManagementScript>();   //
Finds the assembly management script.
        AMS.UpdateTypeList(node, type, oldType); // Tells the AMS to update it's lists.
        if (node == NodeObj01)
```

```
        {
            Node1Type = type;
        }
        else if (node == NodeObj02)
        {
            Node2Type = type;
        }
        else
        {
            Debug.Log("Fatal Error: Node not found.");
        }
    }

    public void UpdateNodeType(GameObject node, string type)    // Is called once upon clicking on a node to change
it's type.
    {
        string oldType;
        if (node == NodeObj01)
        {
            oldType = Node1Type;
            if (type == "Anchor")
            {
                if (type != Node1Type)  // If its a different type, change the type.
                {
                    SetNodeType(type, node, oldType);    // Sets the status of this node
                    foreach (GameObject otherObject in NodeObj01Connection)
                    {
                        NodeConnectionScript NCS =
otherObject.transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                        NCS.SetNodeType(type, otherObject, oldType); // Sets the status of the other nodes
                    }
                }
                else            // If its the same type, revert to standard.
                {
                    SetNodeType("Standard", node, oldType); // Resets the status of this node
                    foreach (GameObject otherObject in NodeObj01Connection)
                    {
                        NodeConnectionScript NCS =
otherObject.transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                        NCS.SetNodeType("Standard", otherObject, oldType); // Resets the status of the other nodes
                    }
                }
            }
            else if (NodeObj01Connection.Count == 1) // If it's a pivot or split joint, but not an anchor.
            {
                if (type != Node1Type)  // If a new type has been given.
                {
                    SetNodeType(type, node, oldType);    // Sets the status of this node
                    NodeConnectionScript NCS =
NodeObj01Connection[0].transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                    NCS.SetNodeType(type, NodeObj01Connection[0], oldType); // Sets the status of the other node to
the same
                    Node1AssocitatedObject = NodeObj01Connection[0];
                }
                else  // If the special type is to be removed.
                {
                    SetNodeType("Standard", node, oldType); // Resets the status of this node
                    NodeConnectionScript NCS =
NodeObj01Connection[0].transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                    NCS.SetNodeType("Standard", NodeObj01Connection[0], oldType); // Resets the status of the other
node to the same
                    Node1AssocitatedObject = null;
                }
            }
            else
            {
                if (type != "Standard" && type != "Anchor")
                {
                    Debug.Log("Error: Pivots and split joints must only be between 2 nodes.  Try Reconnecting them.");
                    gUIScript.NotifyUser("Error: Pivots and split joints must only be between 2 nodes.  Try
Reconnecting them.");
                }
            }
        }
        else if (node == NodeObj02)
        {
```

```
            oldType = Node2Type;
            if (type == "Anchor")
            {
                if (type != Node2Type)
                {
                    SetNodeType(type, node, oldType);    // Sets the status of this node
                    foreach (GameObject otherObject in NodeObj02Connection)
                    {
                        NodeConnectionScript NCS =
otherObject.transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                        NCS.SetNodeType(type, otherObject, oldType); // Sets the status of the other nodes
                    }
                }
                else
                {
                    SetNodeType("Standard", node, oldType); // Resets the status of this node
                    foreach (GameObject otherObject in NodeObj02Connection)
                    {
                        NodeConnectionScript NCS =
otherObject.transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                        NCS.SetNodeType("Standard", otherObject, oldType); // Resets the status of the other nodes
                    }
                }
            }
            else if (NodeObj02Connection.Count == 1) // If it's a pivot or split joint, but not an anchor.
            {
                if (type != Node2Type)
                {
                    SetNodeType(type, node, oldType);    // Sets the status of this node
                    NodeConnectionScript NCS =
NodeObj02Connection[0].transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                    NCS.SetNodeType(type, NodeObj02Connection[0], oldType); // Sets the status of the other node to
the same
                    Node2AssocitatedObject = NodeObj02Connection[0];
                }
                else
                {
                    SetNodeType("Standard", node, oldType); // Resets the status of this node
                    NodeConnectionScript NCS =
NodeObj02Connection[0].transform.parent.parent.gameObject.GetComponent<NodeConnectionScript>();
                    NCS.SetNodeType("Standard", NodeObj02Connection[0], oldType); // Resets the status of the other
node to the same
                    Node2AssocitatedObject = null;
                }
            }
            else if (type != "Standard" && type != "Anchor")
            {
                Debug.Log("Error: Pivots and split joints must only be between 2 nodes. Try Reconnecting them.");
                gUIScript.NotifyUser("Error: Pivots and split joints must only be between 2 nodes. Try Reconnecting
them.");
            }
        }
        else
        {
            Debug.Log("Fatal Error: Node not found.");
        }
    }
    public void ClearAllLists()
    {
        NodeObj01Connection.Clear();
        NodeObj02Connection.Clear();
    }   // Clears the list of connections for each nodes

    private static int LinearSearch(List<GameObject> list, GameObject SearchFor)    // A linear search of a given list
for a given object. Returns -1 if not present.
    {
        if (list.Count != 0)
        {
            for (int i = 0; i < list.Count; i++)
            {
                if (list[i] == SearchFor)
                {
                    return i;
                }
            }
        }
```

```
        return -1;
    }

    public IEnumerator ClearConnectionList(GameObject node) // Clears all the attached nodes from this node.
    {
        if (node == NodeObj01)
        {
            NodeObj01Connection.Clear();
        }
        else if (node == NodeObj02)
        {
            NodeObj02Connection.Clear();
        }
        else
        {
            Debug.Log("Fatal Error: Node not found.");
        }
        yield return null;
    }

    public IEnumerator UpdateConnection(GameObject node, GameObject ConnectedNode, bool isConnected)   // Updates the
NodeObjXXConnection variable when a node in the scene is conencted to or removed from another node.
    {
        int locationIndex;
        if (node == NodeObj01)
        {
            if (isConnected == true) // If the nodes are connected
            {
                if ((LinearSearch(NodeObj01Connection, ConnectedNode) == -1) && (ConnectedNode != node)) // Checks
that the node isn't already connected and that the node isnt itself.
                {
                    NodeObj01Connection.Add(ConnectedNode); // Adds the other node to this node's connection list.
                }
            }
            else   // If the nodes are not connected
            {
                if (NodeObj01Connection.Count != 0)
                {
                    locationIndex = LinearSearch(NodeObj01Connection, ConnectedNode);   // Find the location of the
node in the list of nodes
                    if (locationIndex != -1)    // If the node is present in the list of nodes
                    {
                        NodeObj01Connection.RemoveAt(locationIndex);    // Remove the node
                        NodeObj01Connection.TrimExcess();   // Tidy the list.
                    }
                }
            }
        }
        else if (node == NodeObj02)
        {
            if (isConnected == true) // If the nodes are connected
            {
                if ((LinearSearch(NodeObj02Connection, ConnectedNode) == -1) && (ConnectedNode != node)) // Checks
that the node isn't already connected and that the node isnt itself.
                {
                    NodeObj02Connection.Add(ConnectedNode);
                }
            }
            else   // If the nodes are not connected
            {
                if (NodeObj02Connection.Count != 0)
                {
                    locationIndex = LinearSearch(NodeObj02Connection, ConnectedNode);    // Find the location of the
node in the list of nodes
                    if (locationIndex != -1)     // If the node is present in the list of nodes
                    {
                        NodeObj02Connection.RemoveAt(locationIndex);   // Remove the node
                        NodeObj02Connection.TrimExcess();   // Tidy the list
                    }
                }
            }
        }
        else
        {
            Debug.Log("Fatal Error: node not found.");
        }
```

```
        yield return null;
    }
    public List<GameObject> GetNodeList(GameObject node)    // Returns a list of connected nodes given a specific
node.
    {
        List<GameObject> nodeList = new List<GameObject>();
        if (node == NodeObj01)
        {
            nodeList = NodeObj01Connection;
        }
        else if (node == NodeObj02)
        {
            nodeList = NodeObj02Connection;
        }
        else
        {
            Debug.Log("ERROR: Node not found.");
            nodeList = null;
        }
        return nodeList;
    }

    public List<GameObject> getConnectionList() // Provides a list of all the directly connected GameObjects which are
connected through nodes.
    {
        List<GameObject> objList = new List<GameObject>();
        if (Node1Type != "Pivot" && Node1Type != "SplitJoint")  // Ignores any pivots or split joints as these are
ignored in the graph traversal.
        {
            for (int x = 0; x < NodeObj01Connection.Count; x++)
            {
                objList.Add(NodeObj01Connection[x].transform.parent.parent.gameObject); // Add the placeable object
which is parent of the other node
            }
        }
        if (Node2Type != "Pivot" && Node2Type != "SplitJoint")  // Ignores any pivots or split joints as these are
ignored in the graph traversal.
        {
            for (int x = 0; x < NodeObj02Connection.Count; x++)
            {
                objList.Add(NodeObj02Connection[x].transform.parent.parent.gameObject); // Add the placeable object
which is parent of the other node
            }
        }
        return objList;
    }

    public void OnObjectLoaded()    // Called manually to initialise the the nodes in the AssemblyMangementScript
after its main assembly has been loaded
    {
        AssemblyManagementScript assemblyManagementScript =
this.gameObject.transform.parent.gameObject.GetComponent<AssemblyManagementScript>();   // Gets the current version of
the MainAssembly object and therefore the most current instance of the AssemblyManagementScript.
        assemblyManagementScript.UpdateNodeList(gameObject, NodeObj01, NodeObj02);
        OnEditMode();
    }
}
```

3.3.4 StructEleScript

This class is of type MonoBehaviouir and is attached to each structure element gameObject. It contiains object-specific procedures and so scales the rectangle to meet the node at either end.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class StructEleScript : MonoBehaviour
{
    public GameObject NodeObj01;    // These variables are public as they are assigned in the Editor
    public GameObject NodeObj02;
    public GameObject Rectangle;

    private Vector3 NodePoint01;
    private Vector3 NodePoint02;
```

```
    private Vector3 rotation;
    private GameObject mainAssemblyObject;
    private AssemblyManagementScript assemblyManagementScript;
    private BoxCollider boxCollider;

    void OnEnable() // This is called when the object is enabled in the hierarchy
    {
        mainAssemblyObject = GameObject.Find("MainAssembly");
        assemblyManagementScript = mainAssemblyObject.GetComponent<AssemblyManagementScript>();
        boxCollider = Rectangle.GetComponent<BoxCollider>();
        if ((NodeObj01 == null) || (NodeObj02 == null))
        {
            Debug.Log("ERROR: Variable has null value.");   // This error should never be reached providing that this
object is set up correctly within the editor.
        }
    }

    public void UpdateLocalPosition()   // Makes the rectangle meet the two nodes at either end.
    {
        NodePoint01 = NodeObj01.transform.position;
        NodePoint02 = NodeObj02.transform.position;
        float RectangleScale = Rectangle.transform.localScale.x;
        Rectangle.transform.position = new Vector3((NodePoint02.x+NodePoint01.x)/2,(NodePoint02.y +
NodePoint01.y)/2,(NodePoint02.z + NodePoint01.z)/2);    // Sets the position of the rectangle. This is inherited by
the box collider.
        if (NodePoint01.x <= NodePoint02.x) // Different conditions are required depending on which is lower because
the tan function does not accout for in which direction the rotation is.
        {
            rotation = new Vector3(Mathf.Rad2Deg * Mathf.Atan((NodePoint01.y - NodePoint02.y) /
Mathf.Sqrt(Mathf.Pow(NodePoint01.z - NodePoint02.z, 2) + Mathf.Pow(NodePoint02.x - NodePoint01.x, 2))), 90 +
Mathf.Rad2Deg * Mathf.Atan((NodePoint01.z - NodePoint02.z) / (NodePoint02.x - NodePoint01.x)), 0);
        }
        else
        {
            rotation = new Vector3(-Mathf.Rad2Deg * Mathf.Atan((NodePoint01.y - NodePoint02.y) /
Mathf.Sqrt(Mathf.Pow(NodePoint01.z - NodePoint02.z, 2) + Mathf.Pow(NodePoint02.x - NodePoint01.x, 2))), 90 +
Mathf.Rad2Deg * Mathf.Atan((NodePoint01.z - NodePoint02.z) / (NodePoint02.x - NodePoint01.x)), 0);
        }
        Rectangle.transform.rotation = Quaternion.Euler(rotation);  // Sets the rotation of the rectangle.
        boxCollider.transform.rotation = Quaternion.Euler(rotation);    // Sets the rotation of the box collider
attached to the rectangle
        Rectangle.transform.localScale = new Vector3(RectangleScale, RectangleScale,
Mathf.Sqrt(Mathf.Pow((NodePoint02.x - NodePoint01.x),2) + Mathf.Pow((NodePoint01.y - NodePoint02.y),2) +
Mathf.Pow((NodePoint01.z - NodePoint02.z), 2)));  // Sets the scale of the rectangle
        boxCollider.transform.localScale = new Vector3(RectangleScale, RectangleScale,
Mathf.Sqrt(Mathf.Pow((NodePoint02.x - NodePoint01.x), 2) + Mathf.Pow((NodePoint01.y - NodePoint02.y), 2) +
Mathf.Pow((NodePoint01.z - NodePoint02.z), 2)));  // Sets the scale of the box collider
    }
}
```

3.3.5 ProjectileScript

This class is of type MonoBehaviouir and is attached to each projectile gameObject. It contiains object-specific procedures and so makes sure that the box follows either node at each side.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ProjectileScript : MonoBehaviour
{

    private Vector3 Displacement;
    private Vector3 InitialPosition;
    private Vector3 Velocity;
    private Vector3 InitialVelocity;
    private float Speed;
    private Vector3 Acceleration;
    private Vector3 Force;
    private bool simulating;
    private NodeConnectionScript NCS;
    private GameObject Canvas;
    private GUIScript gUIScript;
    private Vector3 Node1ExpectedPosition;
    private Vector3 Node2ExpectedPosition;
```

```
    private AssemblyManagementScript AMS;
    private float StartTime;
    private GraphScript GS;
    private GUIScript gUIS;
    private Rigidbody RB;
    private bool ShowOnReadouts;

    public GameObject Rectangle;
    public GameObject NodeHolder;
    public GameObject NodeObj01;
    public GameObject NodeObj02;

    private void OnEnable() // Is called every time the object is enabled.
    {
        GS = GameObject.Find("/Canvas/GraphingPanel1").GetComponent<GraphScript>();
        Node1ExpectedPosition = NodeObj01.transform.position;   // A position that the node is known to have been at.
        Node2ExpectedPosition = NodeObj02.transform.position;
        NCS = this.gameObject.GetComponent<NodeConnectionScript>(); // Gets the projectile's nodeConnectionScript.
        Canvas = GameObject.Find("Canvas"); // Finds the canvas gameobject
        gUIScript = Canvas.GetComponent<GUIScript>();   // Sets the GUIScript component attached to the canvas to a
variable.
        simulating = false; //The projectile can only be loaded when the program isnt simulating.
        AMS = gUIScript.getMainAssembly().GetComponent<AssemblyManagementScript>(); // Gets the
AssemblyManagementScript which is attached to the current MainAssembly returned by GUIScript.
        AMS.UpdateProjectilesList(this.gameObject, true);    // Adds this projectile to the AMS's list of projectiles.
        GUIScript.OnSimulate += OnSimulation;   // Subscribes the OnSimulation procedure to the OnSimulate trigger
event attached to the GUIScript.
        GUIScript.OnSimulateEnd += OnEnd;   // Subscribes the OnEnd procedure to the OnSimulateEnd trigger attached to
the GUIScript.
    }

    private void OnSimulation() // Is called by the OnSimulate event on the GUIScript.
    {
        if (this.gameObject.activeInHierarchy == true)  // If this gameobject is currently being used.
        {
            simulating = true;  // This projectile is part of the simulation
            InitialPosition = Rectangle.transform.position;   // Sets the initial position of this object to allow the
future calculation of displacement.
            InitialVelocity = new Vector3(0, 0, 0);
            gUIS = GameObject.Find("Canvas").GetComponent<GUIScript>();
            RB = this.gameObject.transform.parent.gameObject.GetComponent<Rigidbody>(); // Locates the rigidbody
attached to the projectile's subassembly.
        }
    }

    private void OnEnd()    // Is called by the OnSimulationEnd event on the GUIScript
    {
        simulating = false;
    }

    public void UpdateLocalPosition()   //Updates the projectile's and other node's position to account for the moved
node.
    {
        AMS = gUIScript.getMainAssembly().GetComponent<AssemblyManagementScript>();    // Retrieves the most recent
instance of the mainAssembly and its ManagementScript.
        if (NodeMoved(NodeObj01) == true)   // If NodeObj01 is the one that has been moved.
        {
            if (NCS.getNodeObjConnection(2).Count > 0)  //if the node is connected to anything.
            {
                AMS.RemoveConnections(NodeObj02, NCS.getNodeObjConnection(2)[0], NCS);    // Breaks the connections of
the other node. The connections for the NodeObj01 will have alreay been broken at this stage hence why they dont need
to be broken here.
            }
            Rectangle.transform.position = new Vector3(NodeObj01.transform.position.x, NodeObj01.transform.position.y,
NodeObj01.transform.position.z + 0.25f);
            NodeObj02.transform.position = new Vector3(NodeObj01.transform.position.x, NodeObj01.transform.position.y,
NodeObj01.transform.position.z + 0.5f);
        }
        else if (NodeMoved(NodeObj02) == true) // If NodeObj02 is the one that has been moved.
        {
            if (NCS.getNodeObjConnection(1).Count > 0)  //if the node is connected to anything.
            {
                AMS.RemoveConnections(NodeObj01, NCS.getNodeObjConnection(1)[0], NCS);    // Breaks the connections of
the other node. The connections for the NodeObj02 will have alreay been broken at this stage hence why they dont need
to be broken here.
            }
```

```
            Rectangle.transform.position = new Vector3(NodeObj02.transform.position.x, NodeObj02.transform.position.y,
NodeObj02.transform.position.z - 0.25f);
            NodeObj01.transform.position = new Vector3(NodeObj02.transform.position.x, NodeObj02.transform.position.y,
NodeObj02.transform.position.z - 0.5f);
        }
        else
        {

        }
        Node1ExpectedPosition = NodeObj01.transform.position;        // Stores the location of both nodes after they've
been moved.
        Node2ExpectedPosition = NodeObj02.transform.position;
    }

    private bool NodeMoved(GameObject node) // Checks if a given node has been moved out of place.
    {
        if (node == NodeObj01)
        {
            if (NodeObj01.transform.position == Node1ExpectedPosition)  // If the node has been moved out of its last
know position by the curser.
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        else if (node == NodeObj02)
        {
            if (NodeObj02.transform.position == Node2ExpectedPosition)  // If the node has been moved out of its last
know position by the curser.
            {
                return false;
            }
            else
            {
                return true;
            }
        }
        else
        {
            Debug.Log("Fatal Error: Node not found!");
            return false;
        }
    }

    private void OnDisable()    // Is called upon this object being disabled (Including its parents)
    {
        GUIScript.OnSimulate -= OnSimulation;      // Unsubscribes the OnSimulation procedure to the OnSimulate
trigger event attached to the GUIScript.
        GUIScript.OnSimulateEnd -= OnEnd;      // Unsubscribes the OnEnd procedure to the OnSimulateEnd trigger
attached to the GUIScript.
        AMS.UpdateProjectilesList(this.gameObject, false);  // Tells the current AssemblyManagementScript to remove
this projectile from its list of projectiles.
    }

    public float GetDisplacement(char Direction)    // Returns the Displacement for a given direction of the
projectile.
    {
        switch (Direction)
        {
            case 'x':
                return (Displacement.x);
            case 'y':
                return (Displacement.y);
            case 'z':
                return (Displacement.z);
            default:
                Debug.Log("Fatal Error: Direction not found");
                return 0;
        }
    }

    public float GetVelocity(char Direction)    // Returns the Velocity for a given direction of the projectile.
    {
```

```
        switch (Direction)
        {
            case 'x':
                return (Velocity.x);
            case 'y':
                return (Velocity.y);
            case 'z':
                return (Velocity.z);
            default:
                Debug.Log("Fatal Error: Direction not found");
                return 0;
        }
    }

    public float GetAcceleration(char Direction)    // Returns the Acceleration for a given direction of the
projectile.
    {
        switch (Direction)
        {
            case 'x':
                return (Acceleration.x);
            case 'y':
                return (Acceleration.y);
            case 'z':
                return (Acceleration.z);
            default:
                Debug.Log("Fatal Error: Direction not found");
                return 0;
        }
    }

    public float GetForce(char Direction)    // Returns the Force for a given direction of the projectile.
    {
        switch (Direction)
        {
            case 'x':
                return (Force.x);
            case 'y':
                return (Force.y);
            case 'z':
                return (Force.z);
            default:
                Debug.Log("Fatal Error: Direction not found");
                return 0;
        }
    }

    private void FixedUpdate()  // Called once per frame
    {
        if (simulating == true) // If the simulation is currently running
        {
            Displacement = new Vector3(Rectangle.transform.position.x - InitialPosition.x,
Rectangle.transform.position.y - InitialPosition.y, Rectangle.transform.position.z - InitialPosition.z);   //
Calculates the displacement of the projectile at the time when it is called.
            Acceleration = (RB.GetPointVelocity(Rectangle.transform.position) - Velocity) / Time.fixedDeltaTime;    //
Acceleration = change in velocity / time for 1 frame.
            Velocity = RB.GetPointVelocity(Rectangle.transform.position); // Gets the velocity of the point of this
object's rectangle within the ridgidbody attached to the parent
            Force = ((RB.mass * Acceleration));  // F = MA
            if (ShowOnReadouts == true) // If this projectile outputs it's values to the readout.
            {
                gUIScript.SetDisplacementText(Displacement);
                gUIScript.SetVelocityText(Velocity);
                gUIScript.SetAccelerationText(Acceleration);
                gUIScript.SetForceText(Force);
            }
        }
    }

    public void showReadouts(bool show) // Called to set the ShowReadouts variable.
    {
        ShowOnReadouts = show;
    }
}
```

3.3.6 SubAssemblyScript

This class is of type MonoBehaviouir and is attached to the SubAssembly prefab. It enables the creation of pivots and split joints between SubAssembly objects.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SubAssemblyScript : MonoBehaviour
{

    private List<Pivot> Pivots = new List<Pivot>();
    private List<SplitJoint> splitJoints = new List<SplitJoint>();
    private bool anchored;

    class Pivot // The Pivot class is used in the pivot list which is populated when creating subassembilies.
    {
        GameObject pivotPoint;
        GameObject foreignBody;
        public void setPivot(GameObject Obj, GameObject foreignObj)
        {
            pivotPoint = Obj;
            foreignBody = foreignObj;
        }
        public Vector3 getPivotPt()
        {
            Vector3 location = pivotPoint.transform.position;
            return location;
        }
        public Rigidbody getRidgidbody()
        {
            Rigidbody RB = foreignBody.GetComponent<Rigidbody>();
            return RB;
        }
    }

    class SplitJoint    // The SplitJoint class is used in the SplitJoint list which is populated when creating
subassembilies.
    {
        GameObject SplitJointPoint;
        GameObject foreignBody;
        FixedJoint fixedJoint;
        public void setSplitJoint(GameObject Obj, GameObject foreignObj)
        {
            SplitJointPoint = Obj;
            foreignBody = foreignObj;
        }
        public Vector3 getSplitJointPt()
        {
            Vector3 location = SplitJointPoint.transform.position;
            return location;
        }
        public Rigidbody getRidgidbody()
        {
            Rigidbody RB = foreignBody.GetComponent<Rigidbody>();
            return RB;
        }
        public void setFixedJoint(FixedJoint FJ)
        {
            fixedJoint = FJ;
        }
        public FixedJoint getFixedJoint()
        {
            return fixedJoint;
        }
    }

    private void Start() // This is called on the start of the simulation or when the object is instantiated.
    {
        GUIScript.toggleEditMode += ClearLists;
        GUIScript.OnSimulate += OnSimulation;
    }

    public void AddSplitJoint(GameObject locationObj, GameObject foreignSubAssembly)  // Adds the location of a given
split joint to the list of split joint clasees.
    {
```

```
        SplitJoint split = new SplitJoint();
        split.setSplitJoint(locationObj, foreignSubAssembly);
        splitJoints.Add(split);
    }

    public void AddPivot(GameObject locationObj, GameObject foreignSubAssembly)  // Adds the location of a given pivot
to the list of pivot classes.
    {
        Pivot pivot = new Pivot();
        pivot.setPivot(locationObj, foreignSubAssembly);
        Pivots.Add(pivot);
    }

    public void IsAnchored(bool set)    // Sets if this subassembily is anchored or not.
    {
        anchored = set;
    }

    public void UsePhysics()    // Sets up the pivots, split joints and anchors
    {
        Rigidbody subAssyRB = this.GetComponent<Rigidbody>();
        if (anchored == true)
        {
            subAssyRB.useGravity = false;
            subAssyRB.isKinematic = true;
        }
        else
        {
            subAssyRB.useGravity = true;
            subAssyRB.isKinematic = false;
        }
        foreach (Pivot obj in Pivots)   // Sets up all the pivots attached to this SubAssembly.
        {
            ConfigurableJoint piv = this.gameObject.AddComponent<ConfigurableJoint>();
            piv.anchor = obj.getPivotPt();
            piv.connectedBody = obj.getRidgidbody();
            piv.xMotion = ConfigurableJointMotion.Locked;
            piv.yMotion = ConfigurableJointMotion.Locked;
            piv.zMotion = ConfigurableJointMotion.Locked;
        }
        foreach (SplitJoint obj in splitJoints)
        {
            FixedJoint Splt = this.gameObject.AddComponent<FixedJoint>();   // Split joints use fixed joint components
which is to be broken when the split joint breaks.
            Splt.connectedBody = obj.getRidgidbody();
            obj.setFixedJoint(Splt);
        }
    }

    public void ClearLists()    //   Is called by the toggleEditMode event within the GUIScript
    {
        Pivots.Clear();
        splitJoints.Clear();
    }

    public void OnSimulation() // Is called by the OnSimulate event within the GUIScript
    {
        foreach (SplitJoint Obj in splitJoints)
        {
            Destroy(Obj.getFixedJoint());   // Breaks the split joints when running the simulation.
        }
    }

}
```

3.3.7 CameraManagementScript

This class is of type MonoBehaviouir and is attached to the camera in the scene. It enables the movement of the camera both whilst in edit mode and the tracking of the camera when running the simulation.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraManagementScript : MonoBehaviour
```

```
{

    public GameObject ProjectileDropDown1;  // These are public as they are manually assigned in the Editor.
    public GameObject ProjectileDropDown2;
    public GameObject ProjectileDropDown3;

    private GameObject MainCamera;
    private const int TransitionSpeed = 10;
    private bool CanSwitch;
    private ProjectileDropDownScript PDS1;
    private ProjectileDropDownScript PDS2;
    private ProjectileDropDownScript PDS3;

    private Vector3 TopPosition;    // Ideally these would be declared as constants, however C# only allows C# native
types to be declated as a constant
    private Quaternion TopRotation;
    private Vector3 BottomPosition;
    private Quaternion BottomRotation;
    private Vector3 RightPosition;
    private Quaternion RightRotation;
    private Vector3 LeftPosition;
    private Quaternion LeftRotation;
    private Vector3 FrontPosition;
    private Quaternion FrontRotation;
    private Vector3 BackPosition;
    private Quaternion BackRotation;
    private Vector3 FreePosition;
    private Quaternion FreeRotation;
    private Vector3 initialPosition;
    private Quaternion initialRotation;
    const int borderWidth = 5;
    bool simulating;
    GameObject MainAssembly;
    AssemblyManagementScript AMS;
    float XMax = 0;
    float YMax = 0;
    float XMin = 0;
    float YMin = 0;
    List<GameObject> node;
    Vector3 TargetPos;

    private void Start()    // This is called at the start of the execution of the program.
    {
        PDS1 = ProjectileDropDown1.GetComponent<ProjectileDropDownScript>();      // Locates the 3 projectile dd
scripts.
        PDS2 = ProjectileDropDown2.GetComponent<ProjectileDropDownScript>();
        PDS3 = ProjectileDropDown3.GetComponent<ProjectileDropDownScript>();

        MainCamera = GameObject.Find("CameraObject/MainCamera");

        CanSwitch = false;  // Initially the simulation starts in simulation mode and therefore the camera cannot
switch immediately after startup.

        TopPosition = new Vector3(0, 10, 0);
        TopRotation = Quaternion.Euler(90, 90, 0);

        BottomPosition = new Vector3(0, -10, 0);
        BottomRotation = Quaternion.Euler(-90, -90, 0);

        LeftPosition = new Vector3(0, 0, 10);
        LeftRotation = Quaternion.Euler(0, -180, 0);

        RightPosition = new Vector3(0, 0, -10);
        RightRotation = Quaternion.Euler(0, 0, 0);

        FrontPosition = new Vector3(-10, 0, 0);
        FrontRotation = Quaternion.Euler(0, 90, 0);

        BackPosition = new Vector3(10, 0, 0);
        BackRotation = Quaternion.Euler(0, -90, 0);

        FreePosition = new Vector3(10, 10, 10);
        FreeRotation = Quaternion.Euler(45, 225, 0);

        GUIScript.toggleEditMode += OnEditMode; // Subscribes these procedures to the events within the GUIScript.
        GUIScript.toggleSimulationMode += OnSimulationMode;
```

```
        GUIScript.OnSimulate += Simulate;
        GUIScript.OnSimulateEnd += OnSimulationEnd;
    }

    private void OnEditMode()   // Is called by the toggleEditMode event.
    {
        CanSwitch = true;
        simulating = false;
    }

    private void OnSimulationMode() // Is called by the toggleSimulationMode event.
    {
        XMax = 0;   // Resets the simulation camera's position.
        YMax = 0;
        XMin = 0;
        YMin = 0;
        StartCoroutine(FreeCam(FreePosition, FreeRotation, false));  // Moves the camera to free veiw.
    }

    private void OnSimulationEnd()  // Is called by the OnSimulateEnd delegate
    {
        simulating = false;
        StartCoroutine(FreeCam(FreePosition, FreeRotation, false)); // Moves the camera back to free view position
        CanSwitch = false;
    }

    private void Update()        // Update is called once per frame
    {
        if (Input.GetKeyDown("[7]"))    // The square brackets around a number signify that these are numpad keys.
        {
            OrientTop();
        }
        if (Input.GetKeyDown("[1]"))
        {
            OrientBottom();
        }
        if (Input.GetKeyDown("[4]"))
        {
            OrientLeft();
        }
        if (Input.GetKeyDown("[6]"))
        {
            OrientRight();
        }
        if (Input.GetKeyDown("[2]"))
        {
            OrientFront();
        }
        if (Input.GetKeyDown("[8]"))
        {
            OrientBack();
        }
        if (Input.GetKeyDown("[5]"))
        {
            OrientFree();
        }
    }
    private IEnumerator Orient(Vector3 Position, Quaternion Rotation, bool leaveInState)   // Orients the camera into a plane and sets it to orthographic projection.
    {
        CanSwitch = false;
        MainCamera.GetComponent<Camera>().orthographic = false;
        while (Vector3.Distance(MainCamera.transform.position, Position) > 0.5f)    // While the camera is at a distance greater than 0.5 units, interpolate the camera from its current position to its target position. Without this, the camera would never actually reach the point, it would just get closer to it at an increasingly slower rate.
        {
            MainCamera.transform.position = Vector3.Slerp(MainCamera.transform.position, Position, TransitionSpeed * Time.deltaTime);
            MainCamera.transform.rotation = Quaternion.Slerp(MainCamera.transform.rotation, Rotation, TransitionSpeed * Time.deltaTime);
            yield return new WaitForEndOfFrame();   // Must wait for end of frame to allow the interpolation to take place.
        }
        MainCamera.GetComponent<Camera>().orthographic = true;
        MainCamera.transform.position = Position;   // Sets the cameras position to the actual precise position.
        MainCamera.transform.rotation = Rotation;
```

```
        CanSwitch = leaveInState;
        PDS1.RefreshNumberLocations();  // Tells the ProjectileDropdownScripts to update the location of their numbers as they will now be outdated since the camera has moved.
        PDS2.RefreshNumberLocations();
        PDS3.RefreshNumberLocations();
        yield return 0;
    }

    private IEnumerator FreeCam(Vector3 Position, Quaternion Rotation, bool leaveInState)  // Orients the camera in a free-veiw mode and sets it to perspective projection.
    {
        CanSwitch = false;
        MainCamera.GetComponent<Camera>().orthographic = false;
        while (Vector3.Distance(MainCamera.transform.position, Position) > 1f)   // While the camera is at a distance greater than 1 unit, interpolate the camera from its current position to its target position. Without this, the camera would never actually reach the point, it would just get closer to it at an increasingly slower rate.
        {
            MainCamera.transform.position = Vector3.Slerp(MainCamera.transform.position, Position, TransitionSpeed * Time.deltaTime);
            MainCamera.transform.rotation = Quaternion.Slerp(MainCamera.transform.rotation, Rotation, TransitionSpeed * Time.deltaTime);
            yield return new WaitForEndOfFrame();   // Must wait for end of frame to allow the interpolation to take place.
        }
        MainCamera.transform.position = Position;
        MainCamera.transform.rotation = Rotation;
        CanSwitch = leaveInState;
        PDS1.RefreshNumberLocations();  // Tells the PDSs to update the location of their numbers as they will now be outdated since the camera has moved.
        PDS2.RefreshNumberLocations();
        PDS3.RefreshNumberLocations();
        yield return 0;
    }

    public void Simulate()  // Is called by the OnSimulate event attached to the GUIScript.
    {
        simulating = true;
        MainAssembly = GameObject.Find("MainAssembly");
        AMS = MainAssembly.GetComponent<AssemblyManagementScript>();     // Updates the instance of the AssemblyManagementScript.
        node = AMS.getObjList();
        TargetPos = new Vector3(0, 0, 0);
        StartCoroutine(SimulateCam());  // Switches to the simulation camera
    }

    private IEnumerator SimulateCam()    // Is used in simulations to ensure that all the nodes in the scene lie within the frame of the camera.
    {
        MainCamera.transform.rotation = Quaternion.Euler(5, 180, 0);
        while (simulating == true)
        {
            for (int i = 0; i < node.Count; i++)     // Iterates through all the nodes, comparing each with a minimum and maximum to find the nodes with the minimum and maximum distance from the initial position.
            {
                if (node[i] != null)
                {
                    Vector3 position = node[i].transform.position;
                    if (position.x > XMax)
                    {
                        XMax = position.x;
                    }
                    else if (position.x < XMin)
                    {
                        XMin = position.x;
                    }
                    if (position.y > YMax)
                    {
                        YMax = position.y;
                    }
                    else if (position.y < YMin && position.y > 5)
                    {
                        YMin = position.y;
                    }
                }
            }
            TargetPos.x = (XMin + XMax) / 2;
```

```
            TargetPos.y = (YMin + YMax) / 2;
            TargetPos.z = (((XMax - XMin) / 2) + borderWidth) /
Mathf.Tan((MainCamera.GetComponent<Camera>().fieldOfView * Mathf.Deg2Rad) / 2);
            MainCamera.transform.position = Vector3.Slerp(TargetPos, this.transform.position, TransitionSpeed *
Time.deltaTime / 1000);
            yield return new WaitForSeconds(0.01f);
            yield return null;
        }
        yield return null;
    }

    public void OrientTop() // Orients the camera to the top.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(Orient(TopPosition, TopRotation, true));
        }
    }

    private void OrientBottom() // Orients the camera to the bottom.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(Orient(BottomPosition, BottomRotation, true));
        }
    }

    private void OrientLeft()  // Orients the camera to the left.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(Orient(LeftPosition, LeftRotation, true));
        }
    }

    private void OrientRight()  // Orients the camera to the right.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(Orient(RightPosition, RightRotation, true));
        }
    }

    private void OrientFront()   // Orients the camera to the front.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(Orient(FrontPosition, FrontRotation, true));
        }
    }

    private void OrientBack()    // Orients the camera to the back.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(Orient(BackPosition, BackRotation, true));
        }
    }

    private void OrientFree()    // Orients the camera to the freeView position.
    {
        if (CanSwitch == true)
        {
            StartCoroutine(FreeCam(FreePosition, FreeRotation, true));
        }
    }
}
```

3.3.8 GraphScript

This class is of type MonoBehaviouir and is attached to each graph in the canvas. It enables the plotting of points on a graph continuously while running a simulation. One of the graphs also manages the readouts.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
using System.Linq;
using UnityEngine.UI;

public class GraphScript : MonoBehaviour
{

    private GameObject GraphBackground;
    private float GraphSizeX;
    private float GraphSizeY;
    private float GraphScaleX;
    private float GraphScaleY;
    private float Difference;
    private List<GameObject> Point = new List<GameObject>();
    private List<GameObject> RedundantPoint = new List<GameObject>();
    private Vector2 InitialBackgroundPosition;
    private Vector2 InitialBackgroundSize;
    private ProjectileScript ProjScript;
    private RectTransform BackgroundRect;
    private string ProjAttribute;
    private float StartTime;
    private bool Simulating;
    private char Axis;
    private RectTransform GraphMaskWindow;
    private GUIScript gUIScript;

    public GameObject Title;    // These variables are public because they are assigned in the UnityEditor.
    public GameObject PlottingPointPrefab;
    public GameObject ElapsedTimeText;

    private void OnEnable() // Called by UnityEngine when this GameObject is set as active in the higherarchy.
    {
        Title.SetActive(false);
        gUIScript = GameObject.Find("/Canvas").GetComponent<GUIScript>();
        Simulating = false;
        GUIScript.OnSimulate += StartPlotting;  // Subscribes the StartPlotting procedure to the OnSimulate event.
        GUIScript.OnSimulateEnd += OnEnd;   // Subscribes the OnEnd procedure to the OnSimulateEnd event.
        GraphMaskWindow = this.gameObject.transform.GetChild(0).gameObject.GetComponent<RectTransform>();   // Locates
the Mask component (a window which only renders the portion of the background that lies inside it)
        GraphBackground = this.gameObject.transform.GetChild(0).GetChild(0).gameObject; // Locates background on which
the points are plotted
        BackgroundRect = GraphBackground.GetComponent<RectTransform>();
        InitialBackgroundPosition = BackgroundRect.position;
        GraphScaleX = 40.0f;
        GraphScaleY = 20.0f;
        InitialBackgroundSize = new Vector2(GraphMaskWindow.rect.width - 10, GraphMaskWindow.rect.height - 10);   //
Sets the width of the veiwable graph.
        ProjAttribute = "Displacement"; // Sets the default attribute
        Axis = 'x'; // Sets the default axis
    }

    private void OnDisable()// Called by UnityEngine when this GameObject is set as inactive in the higherarchy.
    {
        GUIScript.OnSimulate -= StartPlotting;  // Unsubscribes from event.
        GUIScript.OnSimulateEnd -= OnEnd;
    }

    public void OnEnd() // Is called by the OnSimulateEnd event attached to the GUIScript.
    {
        Simulating = false;
    }

    private void PlotPoint(float Value, float time) // Plots a point given its value (y axis) and time (x axis)
    {
        GameObject NewPoint;
        if ((time * GraphScaleX) / GraphSizeX > 1) // If the time is greater than the current width of the box.
        {
            float change = ((time * GraphScaleX) / GraphSizeX) * GraphScaleX;   // Calculates the change in size
            Difference = Difference + change;   // Calculates the difference in size from original
            BackgroundRect.sizeDelta = new Vector2(Difference, 0);  // Resizes the graph background to allow the new
points to be plotted.
            BackgroundRect.transform.Translate(new Vector2((-change / 2), 0)); // Moves the background to the left by
half the difference in width
            GraphSizeX = BackgroundRect.rect.width; // Sets the new width of the graph
            RedundantPoint = RedundantPoint.Concat(LocateRedundantPoints(time * GraphScaleX)).ToList(); //Gets all the
new points outside the veiw window which can be re used and adds them to the list of redundant points.
        }
```

```
        if (Mathf.Abs((Value * GraphScaleY) / (GraphSizeY / 2)) > 1)
        {
            GraphScaleY = ScaleVertically(GraphScaleY * (1 / Mathf.Abs((Value * GraphScaleY) / (GraphSizeY / 2))),
GraphScaleY);
        }
        if (RedundantPoint.Count != 0)  // If ther are any points outside the view window.
        {
            NewPoint = RedundantPoint[0];   // Take the redundant point from the front of the list
            RedundantPoint.Remove(NewPoint);    // Remove this point from the front of the redundant points list.
            RedundantPoint.TrimExcess();    // Shuffle the list along to fill the empty space.
        }
        else
        {
            NewPoint = Instantiate(PlottingPointPrefab); // Creates a new point if there are no redundant points
available.
            NewPoint.transform.SetParent(this.gameObject.transform.GetChild(0).GetChild(0));    // Makes the new point
a child of the graphing background.
            Point.Add(NewPoint);    // Adds this newly instantiated point to the list of points
        }
        RectTransform PointTransform = NewPoint.GetComponent<RectTransform>();  // Locates the Trasform component of
the point.
        Vector2 Position = new Vector2((time * GraphScaleX), (Value * GraphScaleY));    // Calculates the point's new
position
        PointTransform.anchoredPosition = Position; // Sets the points new position
    }

    private float ScaleVertically(float newScale, float currentScale)    // Is called to change the vertical scale of
the graph
    {
        for (int i = 0; i < Point.Count; i++)   // Iterates through all the points on the graph
        {
            RectTransform PntTrn = Point[i].GetComponent<RectTransform>();  // Gets the transform attached to each
point
            PntTrn.anchoredPosition = new Vector2(PntTrn.anchoredPosition.x, (PntTrn.anchoredPosition.y /
currentScale) * newScale);    // Calculates the new position of the point and moves it there.
        }
        return newScale;
    }

    private List<GameObject> LocateRedundantPoints(float currentTime)   // Returns a list of points which lie outside
of the mask window
    {
        List<GameObject> RP = new List<GameObject>();   // A new list to be populated with new redundant points
        List<GameObject> CP = new List<GameObject>(Point);   // A new list to be populated with all the points
        for (int i = 0; i < Point.Count; i++)    // Iterates through Point and removes any from CP if it is also
already known to be a redundant point.
        {
            if (RedundantPoint.Contains(Point[i]))    // If the point is already known to be redundant
            {
                CP.Remove(Point[i]);  // Remove the point from the list of points to check.
            }
        }
        CP.TrimExcess(); // Tidies up the CP list, removing any blank spaces.
        for (int i = 0; i < CP.Count; i++)  // Iterates through the CP list
        {
            RectTransform RT = Point[i].GetComponent<RectTransform>();  // Gets the transform attached to each point
            if (RT.anchoredPosition.x < (currentTime - InitialBackgroundSize.x))    // If the point lies outside the
mask (Cannot be seen)
            {
                RP.Add(CP[i]);  // Add it to the list of newly redundant points.
            }
        }
        return RP;  // Return the list of newly redundant points.
    }

    public void StartPlotting() // Initialises the plotting procerdure
    {
        Title.GetComponentInChildren<Text>().text = ProjAttribute;
        Title.SetActive(true);
        ClearGraph();
        Simulating = true;
        Difference = 0; // The change in size of the graph horizontally.
        StartCoroutine(PlotPoints());
    }
    private void ClearGraph()   // Clears the graph and resets the background position and size.
    {
```

```
        foreach (GameObject pt in Point)
        {
            Destroy(pt);    // Destroys the point gameObject
        }
        Point.Clear();
        RedundantPoint.Clear();
        GraphBackground.GetComponent<RectTransform>().position = InitialBackgroundPosition;
        BackgroundRect.sizeDelta = new Vector2(0, 0); // Resets the graph background's size to its initial size.
        GraphSizeX = GraphMaskWindow.rect.width;
        GraphSizeY = GraphMaskWindow.rect.height;
    }

    private IEnumerator PlotPoints()    // Plots the points on the graph corresponding to this class's Projectile,
Attribute and axis.
    {
        StartTime = Time.fixedTime;    // Sets the time that the simulation started to a variable to allow the
calculation of time since the simulation started.
        Debug.Log("Plotting Points on Graph " + this.gameObject.name[13]);
        yield return new WaitForEndOfFrame();   // Must wait until this script has it's Projectile Script assigned.
        if (ProjScript != null)
        {
            while (Simulating == true)
            {
                switch (ProjAttribute)
                {
                    case "Displacement":
                        PlotPoint(ProjScript.GetDisplacement(Axis), Time.fixedTime - StartTime);
                        break;
                    case "Velocity":
                        PlotPoint(ProjScript.GetVelocity(Axis), Time.fixedTime - StartTime);
                        break;
                    case "Acceleration":
                        PlotPoint(ProjScript.GetAcceleration(Axis), Time.fixedTime - StartTime);
                        break;
                    case "Force":
                        PlotPoint(ProjScript.GetForce(Axis), Time.fixedTime - StartTime);
                        break;
                    default:
                        Debug.Log("Fatal Error: Mode not found");
                        break;
                }
                yield return new WaitForSeconds(0.1f);
                yield return null;
            }
        }
        else
        {
            Debug.Log("Error " + this.gameObject.name[13]);
        }
        Title.SetActive(false);
        yield return null;
    }

    public void SetProjectile(GameObject projectile)    // Sets the projectile refernce to a given one.
    {
        if (projectile != null)
        {
            ProjScript = projectile.GetComponent<ProjectileScript>();
        }
        else
        {
            Debug.Log("Fatal Error");
        }
    }

    public void SetAttribute(GameObject dropdown)   // Sets the attribute based on the graph's attributes dropdown's
value.
    {
        ProjAttribute =
dropdown.GetComponent<Dropdown>().options[dropdown.GetComponent<Dropdown>().value].text.ToString();
    }

    public void SetAxis(GameObject dropdown)   // Sets the axis based on the graph's axis dropdown's value.
    {
        Debug.Log(dropdown.GetComponent<Dropdown>().captionText.ToString());
```

```
        Axis = dropdown.GetComponent<Dropdown>().options[dropdown.GetComponent<Dropdown>().value].text.ToString()[0];
// Finds the first char of the currently selected string.
    }

    public IEnumerator setElapsedTimeText() // Used by one of the graphs to update the ElapsedTimeText as the
simulation is running
    {
        Text ELTtext = ElapsedTimeText.GetComponent<Text>();
        while (Simulating == true)
        {
            ELTtext.text = (Mathf.Round((Time.fixedTime - StartTime) * 100) / 100).ToString();    // Rounds the time
to 2 sf and displays it.
            yield return null;
        }
        yield return null;
    }
}
```

3.3.9 ProjectileDropDownScript

This class is of type MonoBehaviouir and is attached to each dropdown to select projectiles in the canvas. It keeps the graphs up to date on which projectile they should be plotting and enables all the projectiles in the scene to be displayed with numbers to show which one is which.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

public class ProjectileDropDownScript : MonoBehaviour, IPointerExitHandler, IPointerEnterHandler
{
    public GameObject ProjectileNumber; // These variables are public as they are assigned in the Editor.
    public GameObject CameraObject;

    private GameObject MainAssembly;
    private Camera MainCamera;
    private List<GameObject> CurrentProjectileNumbers = new List<GameObject>();
    private GUIScript gUIScript;
    private Dropdown DD;
    private bool DisplayingNumbers;
    private GameObject CurrentProjectile;
    private List<GameObject> ProjectilesList = new List<GameObject>();
    private List<string> ProjectileLabel = new List<string>();

    private void ShowProjectileNumbers(List<GameObject> ProjectilesList)    // This displays numbers over each
projectile in the scene to show which is which.
    {
        for (int i = 0; i < ProjectilesList.Count; i++)
        {
            if (ProjectilesList[i] != null)
            {
                GameObject Num = Instantiate(ProjectileNumber); // Creates a new number gameobject.
                Num.transform.SetParent(gUIScript.gameObject.transform);    // Moves  the number under the canvas so
the camera can render it.
                Num.GetComponent<RectTransform>().anchoredPosition =
MainCamera.WorldToScreenPoint(ProjectilesList[i].transform.GetChild(1).transform.position);    // Locates the point of
the box attached to the projectile and puts the projectile number over it.
                Num.GetComponentInChildren<Text>().text = i.ToString();    // Sets the text of the projectile number.
                CurrentProjectileNumbers.Add(Num);  // To keep track of the projectile numbers for when they need to
be removed.
                DisplayingNumbers = true;
            }
        }
    }

    private void OnEnable() // Called upon the object being made active in the hierarchy.
    {
        GUIScript.OnSimulate += OnSimulation;
        MainCamera = CameraObject.GetComponent<Camera>();
        gUIScript = GameObject.Find("Canvas").GetComponent<GUIScript>();
        DD = this.gameObject.GetComponent<Dropdown>();
        int oldValue = DD.value;    // Gets the value of the projectile drop down before.
        UpdateDropdownOptions();
        HideProjectileNumbers();
```

```
        if (ProjectilesList.Count > 0)
        {
            CurrentProjectile = ProjectilesList[oldValue];
            UpdateGraphs();
        }
    }

    public void OnSimulation()  // Called by the OnSimulate event attached to the GUIScript.
    {
        UpdateDropdownOptions();
        UpdateGraphs();
        HideProjectileNumbers();
    }

    private void HideProjectileNumbers()    // Hides the numbers over each projectile
    {
        foreach (GameObject Num in CurrentProjectileNumbers)
        {
            Destroy(Num);
        }
        CurrentProjectileNumbers.Clear();
        DisplayingNumbers = false;
    }

    public void OnPointerExit(PointerEventData eventData)   // Called by UnityEngine when the pointer leaves the drop
down box.
    {
        HideProjectileNumbers();
        if (ProjectilesList.Count > DD.value)
        {
            CurrentProjectile = ProjectilesList[DD.value];
        }
        UpdateGraphs();
    }

    public void OnPointerEnter(PointerEventData eventData)  // Called by UnityEngine when the pointer enters the drop
down box.
    {
        ProjectilesList.Clear();
        UpdateDropdownOptions();
    }

    private void UpdateDropdownOptions()    // Gets all the pivots in the scene and displays them in the drop down
    {
        DD.ClearOptions();  // Resets the dropdown listings ready to be reloaded.
        ProjectilesList.Clear();
        ProjectileLabel.Clear();
        MainAssembly = gUIScript.getMainAssembly();
        ProjectilesList = new
List<GameObject>(MainAssembly.GetComponent<AssemblyManagementScript>().getProjectileList());
        ShowProjectileNumbers(ProjectilesList);
        for (int i = 0; i < ProjectilesList.Count; i++)
        {
            ProjectileLabel.Add("Projectile " + i.ToString());  // Adds each projectile option to a list of projectile
options
        }
        DD.AddOptions(ProjectileLabel); // Adds this new list to the options in the projectile drop down.
        DD.RefreshShownValue();
        if (ProjectileLabel.Count > 0)
        {
            CurrentProjectile = ProjectilesList[DD.value];  // Sets the current projectile to the one that was
selected before.
        }
    }

    public void RefreshNumberLocations()    // Used to refresh the location of the numbers if the camera is moved.
    {
        if (DisplayingNumbers == true)
        {
            HideProjectileNumbers();

ShowProjectileNumbers(gUIScript.getMainAssembly().GetComponent<AssemblyManagementScript>().getProjectileList());
        }
    }

    public void UpdateGraphs()  // Tells the graph which projectile's points to plot.
```

```
    {
        ProjectilesList.TrimExcess();
        if (ProjectilesList.Count != 0)
        {
            if (this.gameObject.transform.parent.name == "GraphingPanel1Settings")  // If this is attached to the
dropdown under the settings panel for graphing panel 1.
            {
                GraphScript GS = GameObject.Find("/Canvas/GraphingPanel1").GetComponent<GraphScript>();
                GS.SetProjectile(CurrentProjectile);
            }
            else if (this.gameObject.transform.parent.name == "GraphingPanel2Settings")  // If this is attached to the
dropdown under the settings panel for graphing panel 2.
            {
                GraphScript GS = GameObject.Find("/Canvas/GraphingPanel2").GetComponent<GraphScript>();
                GS.SetProjectile(CurrentProjectile);
            }
            else     // If this is attached to the dropdown for the readouts.
            {
                ProjectileScript PS;
                foreach (GameObject proj in ProjectilesList)
                {
                    if (proj != null)
                    {
                        PS = proj.GetComponent<ProjectileScript>();
                        PS.showReadouts(false); // Tells all the projectile scripts to not sent their values to the
readouts.
                    }
                }
                PS = CurrentProjectile.GetComponent<ProjectileScript>();
                PS.showReadouts(true);  // Tells the currently selected projectile to send its values to the readouts.
            }
        }
    }
}
```

3.3.10 SaveLoadManagement

This class a regular C# script which contains 3 classes, the first of which which manages the conversion between a saveable JSON file and a MainAssembly GameObject and vice versa. The second and third classes are serialiasable abstractions of MainAssembly GameObjects and AssemblyElements respectively.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;
using System;
using UnityEngine.UI;

public static class SaveLoadManagement
{

    public static void SaveAssembly(GameObject MainAssemblyObject, string fileName)  //Saves the current MainAssembly
to file given a filename.
    {
        AssemblyData AD = ConvertToSerializableClass(MainAssemblyObject);   // Converts the MainAssembly gameObject
into an AssemblyData class which is an abstraction and can be serialised.
        string Jsontxt = JsonUtility.ToJson(AD);    // Converts this AssemblyData class into JSON format and puts it
in a string.
        StreamWriter stream = new StreamWriter(Application.persistentDataPath + "/" + fileName + ".phys");  // Creates
a new file at the given location, or overwrites a current file.
        stream.Write(Jsontxt);  // Populates this file with the JSON string.
        stream.Close(); // Closes the file.
    }

    public static GameObject LoadAssembly(string fileName)  // Loads a saved MainAssembly given its filename.
    {
        StreamReader stream = new StreamReader(Application.persistentDataPath + "/" + fileName + ".phys");  //
Initialises a new stream from the given file location.
        string JsonTxt = stream.ReadToEnd();    // Emptys the entire contents of the file into a string.
        stream.Close(); // Closes the file.
        AssemblyData AD = JsonUtility.FromJson<AssemblyData>(JsonTxt);  // Creates a new AssemblyData class using the
json string.
        GameObject newMainAssembly = ConvertFromSerializableClass(AD);  // Converts this class into a gamobject class
and instantiates it into the scene.
        return newMainAssembly; // Returns the newMainAssembly Object.
```

```
    }

    private static AssemblyData ConvertToSerializableClass(GameObject Assy) // Generates an AssemblyData instance for a given MainAssembly object.
    {
        AssemblyData MainAssy = new AssemblyData(); // Creates a new instance of the MainAssemblyClass class which is ready to be populated.
        int i = 0;
        foreach (Transform assyEle in Assy.transform) // Iterates through the children of the main assebly (IE: the assembly elements). This sets up each structure element's serialisable class.
        {
            NodeConnectionScript NCS = assyEle.GetComponent<NodeConnectionScript>();    // Locates the NodeConnectionScript component attached to the current child of the MainAssemblyClass
            AssemblyElement AssyElemnt = new AssemblyElement(); // Creates a new instance of the AssemblyElement class which is ready to be populated.
            if (assyEle.name == "Structure Element(Clone)")
            {
                AssyElemnt.prefab = Resources.Load("Prefabs/Structure Element") as GameObject;    // Locates the structure element prefab.
            }
            else if (assyEle.name == "Projectile(Clone)")
            {
                AssyElemnt.prefab = Resources.Load("Prefabs/Projectile") as GameObject;    // Locates the projectile prefab.
            }
            else if (assyEle.name == "Track(Clone)")
            {
                AssyElemnt.prefab = Resources.Load("Prefabs/Track") as GameObject;    // Locates the track prefab.
            }
            else
            {
                Debug.Log("Add alternative conditions here");
            }
            AssyElemnt.Node1Type = NCS.GetNodeType(NCS.NodeObj01);
            AssyElemnt.Node2Type = NCS.GetNodeType(NCS.NodeObj02);
            AssyElemnt.Node1Position = NCS.NodeObj01.transform.position;
            AssyElemnt.Node2Position = NCS.NodeObj02.transform.position;
            MainAssy.AssemblyElements[i] = AssyElemnt;      // Adds this element to the array of elements within the new main assembly.
            i = i + 1;
        }
        return MainAssy;
    }

    private static GameObject ConvertFromSerializableClass(AssemblyData mainAssy)    // Returns a new MainAssembly object when given an AssemblyData instanvce.
    {
        GameObject newMainAssembly = GameObject.Instantiate(Resources.Load("Prefabs/MainAssembly") as GameObject); // Creates the new MainAssemblyClass object which is to be parented by the assebly elements.
        newMainAssembly.name = "MainAssembly";  // Renames this object "MainAssembly"
        GUIScript gUIScript = GameObject.Find("Canvas").GetComponent<GUIScript>();  // Locates the GUIScript and puts a reference of it in gUIScript
        gUIScript.setMainAssembly(newMainAssembly); //Tells the GUIScript to set the current mainAssembly to the new instantiated one.
        foreach (AssemblyElement AssyElemnt in mainAssy.AssemblyElements)   // Iterates through each assembly element which is in the main assembly
        {
            if (AssyElemnt.prefab != null) // If this array element is populated.
            {
                try
                {
                    GameObject newAssyEle = GameObject.Instantiate(AssyElemnt.prefab);  // Instantiates a new object in the scene using the prefab referenced in AssyElemnt.
                    newAssyEle.transform.parent = newMainAssembly.transform;    // Sets the parent of this new placeable object to the new MainAssembly
                    NodeConnectionScript NCS = newAssyEle.GetComponent<NodeConnectionScript>(); // Locates the NodeConnectionScript which is attached to the newly instantiated assembly element.
                    NCS.NodeObj01.transform.position = AssyElemnt.Node1Position;    // Moves the nodes in this assembly to their proper positions using the data stored in the AssyElemnt class.
                    NCS.NodeObj02.transform.position = AssyElemnt.Node2Position;    // ^^
                    NCS.setTempNodeType(NCS.NodeObj01, AssyElemnt.Node1Type);    // Sets the temporary node type of each node for later reference when setting up the node types.
                    NCS.setTempNodeType(NCS.NodeObj02, AssyElemnt.Node2Type);    // ^^
                    NCS.OnObjectLoaded();   // Calls the OnObjectLoaded subprocedure attached to this newly instantiated assembly element.
```

```
                    newMainAssembly.GetComponent<AssemblyManagementScript>().AddSceneObject(newAssyEle);     // Adds this assembly element to the MainAssembilie's list of scene objects.
                    if (newAssyEle.name == "Structure Element(Clone)")  // Locates the object specific script and tells it to update it's local position (move the body of the object between the nodes).
                    {
                        StructEleScript SES = newAssyEle.GetComponent<StructEleScript>();
                        SES.UpdateLocalPosition();
                    }
                    else if (newAssyEle.name == "Projectile(Clone)")
                    {
                        ProjectileScript PJS = newAssyEle.GetComponent<ProjectileScript>();
                        PJS.UpdateLocalPosition();
                    }
                    else if (newAssyEle.name == "Track(Clone")
                    {
                        TrackScript TRS = newAssyEle.GetComponent<TrackScript>();
                        TRS.UpdateLocalPosition();
                    }
                    else
                    {
                        Debug.Log("Fatal Error: Assembly Element Script Not found.");
                    }
                }
                catch
                {
                    gUIScript.NotifyUser("Error: Unable to load assembly. Incompatable version.");
                }
            }
            else    // if the end of the array has been reached.
            {
            }
        }
        return newMainAssembly;
    }

}

[Serializable]
public class AssemblyData
{
    public AssemblyElement[] AssemblyElements = new AssemblyElement[100]; // An array must be used as a list is a generic data type which cannot be serialised by unity's built in serialiser.
}

[Serializable]
public class AssemblyElement
{
    public GameObject prefab;   // These must be public to allow the JsonUtility to access them.
    public string Node1Type;    // The type of node1
    public string Node2Type;    // The type of node2
    public Vector3 Node1Position;   // The position of the node1
    public Vector3 Node2Position;   // The position of the node2
}
```

3.3.11 SaveScript

This script contains a class of type MonoBehaviour which is attached to the SaveWindow GUI object and interfaces between this and the SaveLoadManagement classes.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.IO;
using System.Text.RegularExpressions;

public class SaveScript : MonoBehaviour
{

    public GameObject TextInputBox;     // These are public as they have been used to reference the UI elements in the Unity IDE, which saves inefficient find functions.
    public GameObject ScrollView;
    public GameObject FileButtonPrefab;
    const int ButtonHeight = 60;

    GameObject canvas;
```

```
    GUIScript gUIScript;

    private void OnEnable()
    {
        canvas = GameObject.Find("Canvas"); // Finds the canvas gameobject in the scene
        gUIScript = canvas.GetComponent<GUIScript>();
        UpdateFileList();
    }

    private void UpdateFileList()   // This procedure presents all the files as buttons in a scrollVeiw object in the scene.
    {
        foreach (Transform child in ScrollView.transform.GetChild(0).GetChild(0))
        {
            GameObject.Destroy(child.gameObject);           // Makes sure that there are no buttons in the scroll view before it starts adding them.
        }
        Vector2 previousAnchorMin = new Vector2(0, 1);  // Sets the location for the first anchor max.
        DirectoryInfo directory = new DirectoryInfo(Application.persistentDataPath + "/");  // Locates the directory and stores it in a variable.
        FileInfo[] listOfFiles = directory.GetFiles();  // Gets all the files in the directory and stores them in an arry
        List<FileInfo> files = new List<FileInfo>();    // Initialises a new list which will contain just the files with the .phys extension.
        files.Clear();  // Makes sure this list is ready to be used.
        foreach (FileInfo file in listOfFiles)
        {
            if (file.Extension == ".phys")  // Filters out any unwanted files.
            {
                files.Add(file);    // Adds all the files with the .phys extension to the files list.
            }
        }
        ScrollView.transform.GetChild(0).GetChild(0).gameObject.GetComponent<RectTransform>().sizeDelta = new Vector2(0, ButtonHeight * files.Count);   // Sets the height of the container for all the buttons based on how many there are and their individual heights.
        foreach (FileInfo file in files)
        {
            GameObject NewButton = Instantiate(FileButtonPrefab);   // Creates a new button using the prefab.
            NewButton.transform.SetParent(ScrollView.transform.GetChild(0).GetChild(0));    // Makes the new button a child of the container.
            NewButton.transform.GetChild(0).gameObject.GetComponent<Text>().text = Path.GetFileNameWithoutExtension(file.Name); // sets the text on the button to the name of the file.
            RectTransform RT = NewButton.GetComponent<RectTransform>(); // Gets the RectTransform component attached to the button so its position can be set.
            RT.anchorMax = new Vector2(1, previousAnchorMin.y); // Puts the top of this button at the bottom of the button above it.
            RT.anchorMin = new Vector2(0, RT.anchorMax.y - (1.0f / files.Count)); // Sets the bottom of the button at a fixed distance away from the top of the button.
            RT.offsetMax = new Vector2(-5, -5); // Adds a border around the button so they're not all touching.
            RT.offsetMin = new Vector2(5, 5);   //  Adds a border around the button so they're not all touching.
            NewButton.GetComponent<Button>().onClick.AddListener(delegate { OnItemClicked(NewButton); }); // Attaches the OnItemClicked procedure to the buttons OnClick event so that it calls this function when clicked.
            previousAnchorMin = RT.anchorMin;   // Sets this buttons minimum point for the next button to use.
        }
    }

    public void OnSubmit()  // This is called directly by the OnClick event attached to the submit button. It takes the text from the input box and saves the main assembly to that filename.
    {
        string input = TextInputBox.GetComponent<InputField>().text;
        if (input != "")    // Makes sure that the input field isnt blank. If a duplicate filename is provided the files will be overwritten.
        {
            SaveLoadManagement.SaveAssembly(gUIScript.getMainAssembly(), input);    // Tells SaveLoadManagement to save the assembly stored in the MainAssembly variable attached to the GUIScript under the name of the input text.
            UpdateFileList();   // Rearranges the files to account for this.
            this.gameObject.SetActive(false);   // Hides the save window after a successful save.
        }
        else    // The input is invalid and an error message should be shown.
        {
            gUIScript.NotifyUser("Error: Invalid Input");
            Debug.Log("Invalid input");
        }
    }
```

```
    public void OnCancel()   // This is called directly by the OnClick event attached to the Cancel button. It closes the save window.
    {
        this.gameObject.SetActive(false);
    }

    public void OnItemClicked(GameObject CalledFrom)    // This is called directly by the OnClick event attached to any button representing a file. It passes itself and then it's text is used to identify a file.
    {
        string text = CalledFrom.transform.GetChild(0).gameObject.GetComponent<Text>().text.ToString(); // Finds the text on the button
        TextInputBox.GetComponent<InputField>().text = text;    //Presents this text in the TextInputBox for easy saving to an assembly in development.
    }
}
```

3.3.12 LoadScript

This script contains a class of type MonoBehaviour which is attached to the Load window GUI object and interfaces between this and the SaveLoadManagement classes.

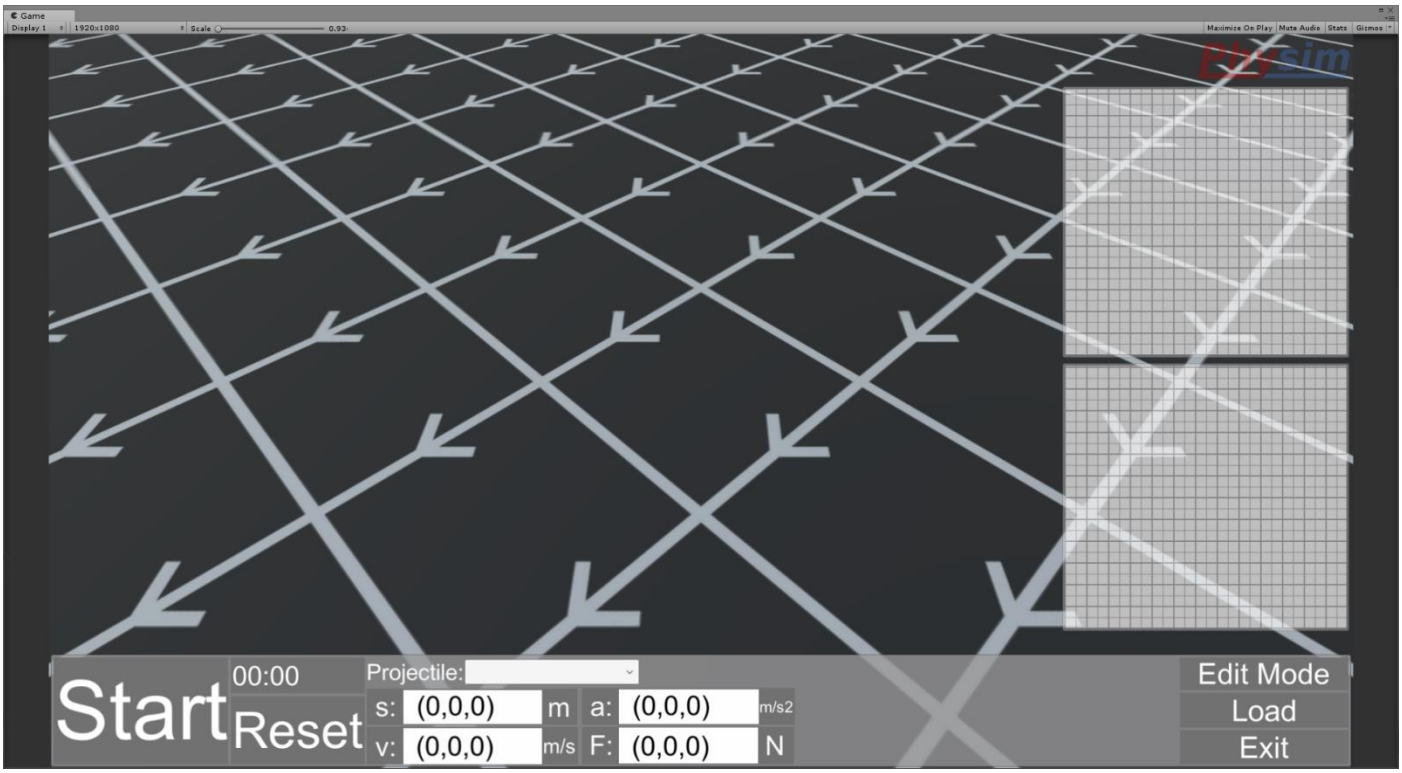
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System.IO;
using System.Text.RegularExpressions;

public class LoadScript : MonoBehaviour
{

    public GameObject TextInputBox;     // These are public as they have been used to reference the UI elements in the Unity IDE, which saves on inefficient find functions.
    public GameObject ScrollView;
    public GameObject FileButtonPrefab;

    const int ButtonHeight = 60;
    List<string> FileNames = new List<string>();
    GameObject canvas;
    GUIScript gUIScript;

    private void OnEnable() // Called upon the object being enabled in the higherarchy.
    {
        canvas = GameObject.Find("Canvas"); // Finds the canvas gameobject in the scene
        gUIScript = canvas.GetComponent<GUIScript>();
        UpdateFileList();
    }

    private void UpdateFileList()   // This procedure presents all the files as buttons in a scrollVeiw object in the scene.
    {
        foreach (Transform child in ScrollView.transform.GetChild(0).GetChild(0))
        {
            GameObject.Destroy(child.gameObject);           // Makes sure that there are no buttons in the scroll view before it starts adding them.
        }
        Vector2 previousAnchorMin = new Vector2(0, 1);  // Sets the location for the first anchor max.
        DirectoryInfo directory = new DirectoryInfo(Application.persistentDataPath + "/");  // Locates the directory and stores it in a variable.
        FileInfo[] listOfFiles = directory.GetFiles();  // Gets all the files in the directory and stores them in an arry
        List<FileInfo> files = new List<FileInfo>();    // Initialises a new list which will contain just the files with the .phys extension.
        files.Clear();  // Makes sure this list is ready to be used.
        FileNames.Clear();
        foreach (FileInfo file in listOfFiles)
        {
            if (file.Extension == ".phys")  // Filters out any unwanted files.
            {
                files.Add(file);    // Adds all the files with the .phys extension to the files list.
                FileNames.Add(file.Name);
            }
        }
```

```
        ScrollView.transform.GetChild(0).GetChild(0).gameObject.GetComponent<RectTransform>().sizeDelta = new Vector2(0, ButtonHeight * files.Count);   // Sets the height of the container for all the buttons based on how many there are and their individual heights.
        foreach (FileInfo file in files)
        {
            GameObject NewButton = Instantiate(FileButtonPrefab);   // Creates a new button using the prefab.
            NewButton.transform.SetParent(ScrollView.transform.GetChild(0).GetChild(0));    // Makes the new button a child of the container.
            NewButton.transform.GetChild(0).gameObject.GetComponent<Text>().text = Path.GetFileNameWithoutExtension(file.Name); // sets the text on the button to the name of the file.
            RectTransform RT = NewButton.GetComponent<RectTransform>(); // Gets the RectTransform component attached to the button so its position can be set.
            RT.anchorMax = new Vector2(1, previousAnchorMin.y); // Puts the top of this button at the bottom of the button above it.
            RT.anchorMin = new Vector2(0, RT.anchorMax.y - (1.0f / files.Count)); // Sets the bottom of the button at a fixed distance away from the top of the button.
            RT.offsetMax = new Vector2(-5, -5); // Adds a border around the button so they're not all touching.
            RT.offsetMin = new Vector2(5, 5);   //  Adds a border around the button so they're not all touching.
            NewButton.GetComponent<Button>().onClick.AddListener(delegate { OnItemClicked(NewButton); }); // Attaches the OnItemClicked procedure to the buttons OnClick event so that it calls this function when clicked.
            previousAnchorMin = RT.anchorMin;   // Sets this buttons minimum point for the next button to use.
        }
    }

    public void OnSubmit()  // This is called directly by the OnClick event attached to the submit button. It takes the text from the input box and saves the main assembly to that filename.
    {
        string input = TextInputBox.GetComponent<InputField>().text;
        if (input != "")    // Makes sure that the input field isnt blank. If a duplicate filename is provided the files will be overwritten.
        {
            if (FileNames.Contains(input + ".phys"))
            {
                Destroy(gUIScript.getMainAssembly());
                GameObject MainAssembly = SaveLoadManagement.LoadAssembly(input);    // Tells SaveLoadManagement to save the assembly stored in the MainAssembly variable attached to the GUIScript under the name of the input text.
                MainAssembly.GetComponent<AssemblyManagementScript>().ReconstructConnections();
                bool switchBackToSimulation = false;
                if (gUIScript.isEditing() == false) // If it's in edit mode.
                {
                    MainAssembly.GetComponent<AssemblyManagementScript>().OnEditMode();
                    switchBackToSimulation = true;
                }
                gUIScript.setMainAssembly(MainAssembly);    // Tells the GUIScript to update its MainAssembly reference to the most recent one.
                UpdateFileList();   // Rearranges the files to account for this.
                if (switchBackToSimulation == true) // Only needs to reset the simulation if its not in edit mode becuase the subassembilies will be generated automatically when they switch back to simulation mode.
                {
                    gUIScript.SimulationMode();    // Resets the simulations which re-builds the subassembilies (or in this case ,creates them).
                }
                MainAssembly.GetComponent<AssemblyManagementScript>().ShowAllSpecialNodes();
                this.gameObject.SetActive(false);   // Hides the save window after a successful load.
            }
            else
            {
                Debug.Log("Error: file does not exist.");
                gUIScript.NotifyUser("Error: file does not exist.");
            }
        }
        else    // The input is invalid and an error message should be shown.
        {
            Debug.Log("Invalid input");
            gUIScript.NotifyUser("Error: Invalid Input");
        }
    }

    public void OnCancel()   // This is called directly by the OnClick event attached to the Cancel button. It closes the save window.
    {
        this.gameObject.SetActive(false);
    }

    public void OnItemClicked(GameObject CalledFrom)    // This is called directly by the OnClick event attached to any button representing a file. It passes itself and then it's text is used to identify a file.
```

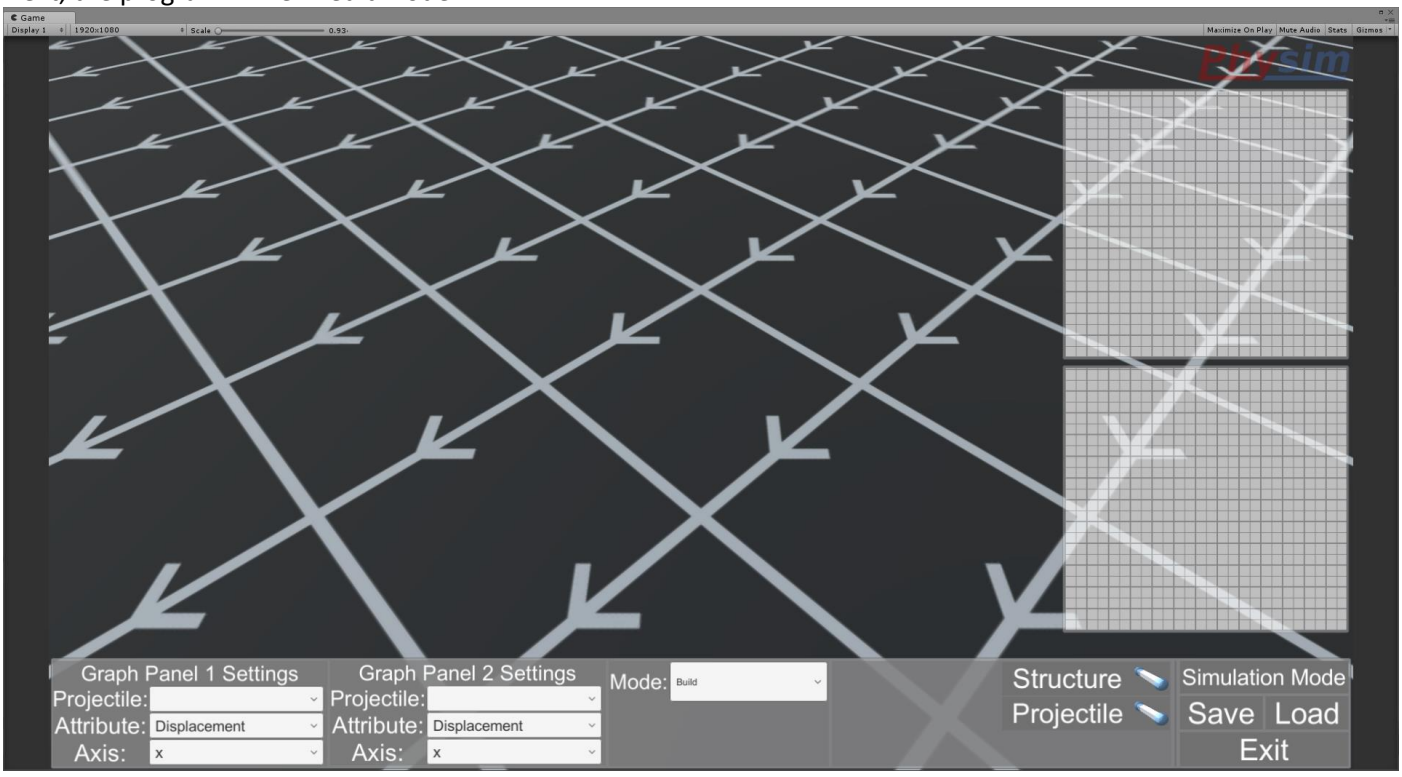
```
    {
        string text = CalledFrom.transform.GetChild(0).gameObject.GetComponent<Text>().text.ToString(); // Finds the text on the button
        TextInputBox.GetComponent<InputField>().text = text;    //Presents this text in the TextInputBox for easy saving to an assembly in development.
    }
}
```

3.4 Graphical User Interface

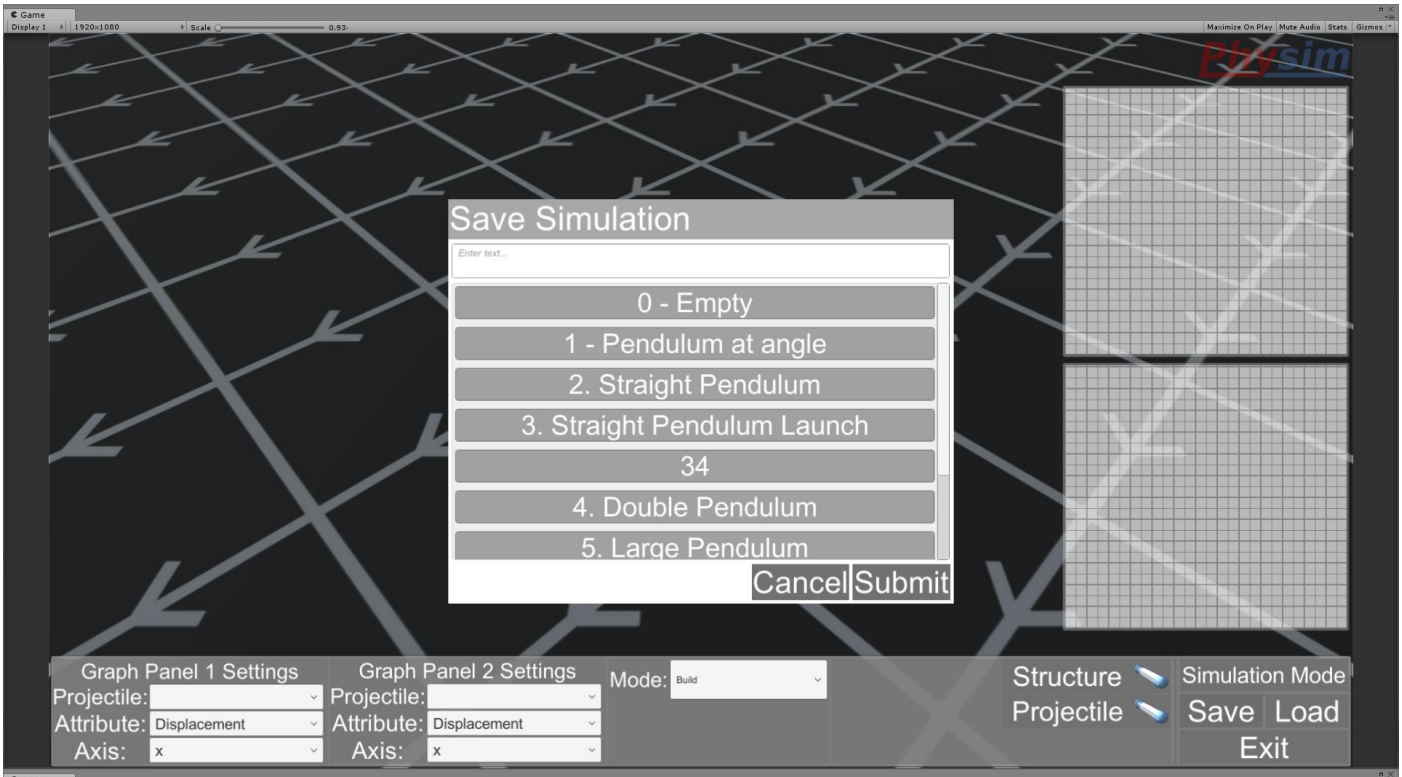
Below is the GUI displayed when in simulation mode. This is the mode that the program starts in when it is run.

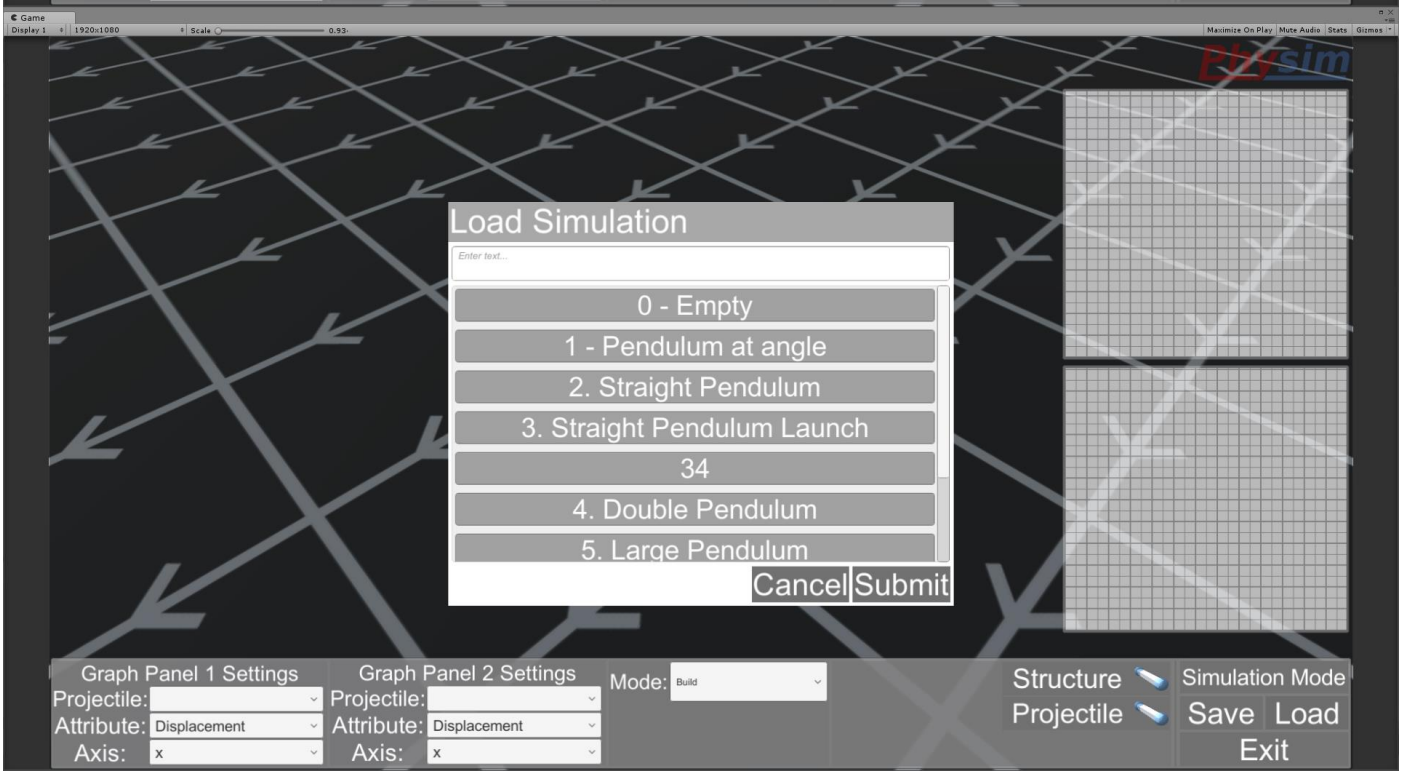


Next, the program while in edit mode.

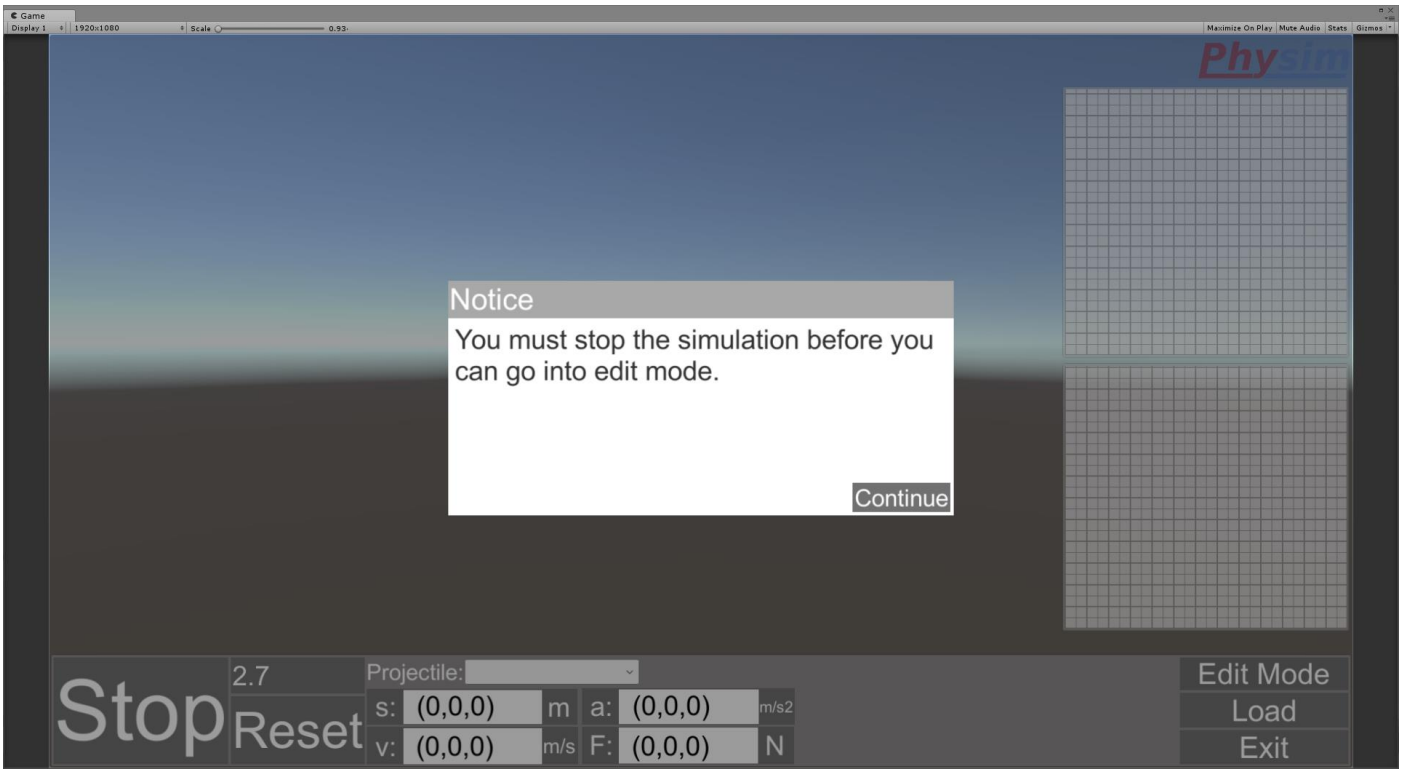


The next two photos are of the save and load windows respectively.

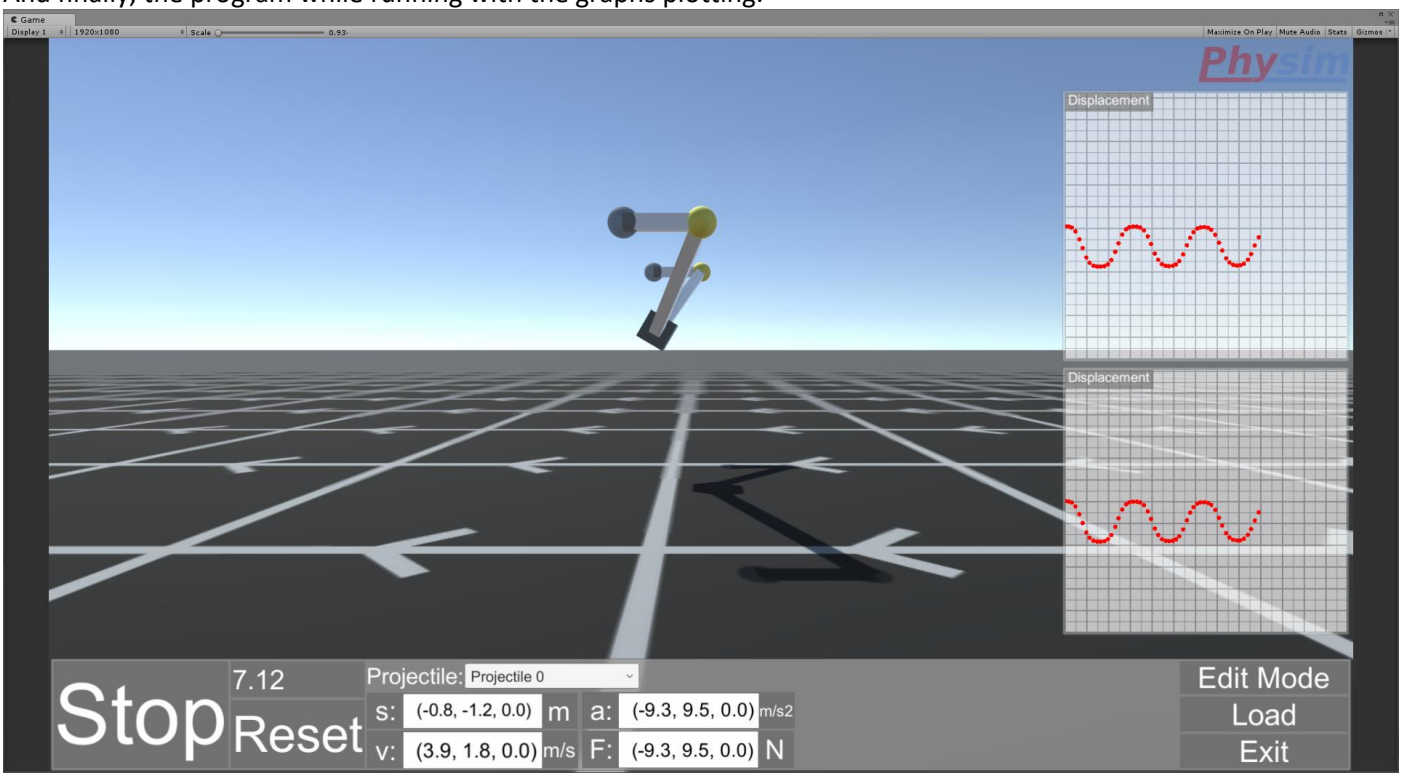




The notice window.



And finally, the program while running with the graphs plotting.



Section 4 – Testing

4.0 Introduction

My testing strategy for Physim is to perform all the tests on the final or near-final build of the program as all of the different systems are reliant on each other for the function of the program to be as desired. This also means that there is overlap between the tests and that it will likely be immediately obvious if there is an issue. Due to the nature of the program, with effectively infinate possabilities because of the simulation builder, it is virtually impossible for me to show every combination of test. Therefore I will show the systems working with both complex and simple random structures to ensure that the majority of cases are tested.

NOTE : All the testing videos contain commentary so please ensure that your audio is working.

4.1 The Simulation Builder

The following tests for section 1 are found in the video at this URL:
https://youtu.be/3-W88qNMrdI

All the following tests are to be completed in Edit Mode unless otherwise stated.

Test No	Test Description	Action of Input	Expected Result	Actual Result	Video Timestamp
1.1	Spawn a placeable assembly element into the scene and move it around	Click any of the placeable object buttons in Edit mode and move mouse over the scene then click right mouse.	The placeable object follows the curser and is released when the left mouse is clicked.	The placeable object follows the curser and is released when the left mouse is clicked.	00:00
1.2	Click on nodes on assembly elements to move them around	In edit mode, after selecting 'Build' under the mode section, click on the end of a placeable object so that the node is highlighted	The placable object follows the node and either scales or moves to meet the node at one of its ends.	The placable object follows the node and either scales or moves to meet the node at one of its ends.	00:38
1.3	Connect multiple nodes together	Click on a node to move it around (like test 1.2), then drag the node over the position of another node.	The node snaps to the node you're hovering over and so share its position. In addition, both NodeConnectionScripts attached to each object have their connected node lists updated.	The node snaps to the node you're hovering over and so share its position. In addition, both NodeConnectionScripts attached to each object have their connected node lists updated.	01:28
1.4	Set the types of nodes	In edit mode, click on the mode dropdown	The colour of the node changes depending on the type of connection (Pivot ⇨	The colour of the node changes depending on the type of connection (Pivot ⇨	2:20

		and choose 'Select Pivot', 'Select Split Joint' and Select Anchor to change the mode. Then click on the positions of nodes in the scene.	yellow, Split Joint ⇨ red, Anchor ⇨ Black) In Addition, the Node Type variable attached to each Node Connection Script changes depending on the node type. A connection between anything other than 2 nodes will not become a pivot or split joint as these are strictly restricted for connections between 2 objects. Also, the AssemblyManagementScript attached to the main assembly object will be made aware of this change and updates its lists to do so.	yellow, Split Joint ⇨ red, Anchor ⇨ Black) In Addition, the Node Type variable attached to each Node Connection Script changes depending on the node type. A connection between anything other than 2 nodes will not become a pivot or split joint as these are strictly restricted for connections between 2 objects. Also, the AssemblyManagementScript attached to the main assembly object is made aware of this change and updates its lists to do so.	
1.5	Convert all the components of the main assembly into subassembilies	After creating a complex structure, click the 'Simulation Mode' button to go back into simulation mode.	The current Main Assembly will be cloned and set to inactive for future use. Then the program goes through the placeable objects via a Depth-First graph traversal algorithm using the connected nodes between them. Any pivots or split joint nodes will be ignored and their connection will be disregarded. All the objects in this graph traversal will be put into a newly instantiated sub assembly object. Any still unvisited nodes/structures at this point will be put into their own subassembily as the first was. Then Hinge/ Fixed joint components will be added to the respective subassembilies and the other subassembly will be set as the foreign body of the component.	The current Main Assembly is cloned and set to inactive for future use. Then the program goes through the placeable objects via a Depth-First graph traversal algorithm using the connected nodes between them. Any pivots or split joint nodes will be ignored and their connection will be disregarded. All the objects in this graph traversal will be put into a newly instantiated sub assembly object. Any still unvisited nodes/structures at this point will be put into their own subassembily as the first was. Then Hinge/ Fixed joint components are added to the respective subassembilies and the other subassembly will be set as the foreign body of the component.	07:22
1.6	Delete components	After creating a structure, select 'delete' mode whilst in edit mode then click the centre of the gameObject.	The object is destroyed and its connections are removed.	The object is destroyed and its connections are removed.	10:20

4.2 Saving And loading

The following tests for section 2 are found in the video at this URL:
https://youtu.be/um_nIQSfcyA

Test No	Test Description	Action of Input	Expected Result	Actual Result	Video Timestamp
2.1	Open save & load windows	Click on 'save' button in Edit mode and 'load' button in simulation mode.	The save panel opens when the save button is clicked and the load panel when the load button is clicked.	The save panel opens when the save button is clicked and the load panel when the load button is clicked.	00:00
2.2	Show current files	Done automatically when save and load windows are opened	Files in the persistant data path file are represented as buttons in a scroll veiw, providing their extension is '.phys'.	Files in the persistant data path file are represented as buttons in a scroll veiw, providing their extension is '.phys'.	00:21
2.3	Select a file	Click on a button on the save or load panel	The name of the file appears in the text input box at the top of the window.	The name of the file appears in the text input box at the top of the window.	00:59
2.4	Save a file	Click on the 'submit' button in the save window with valid text in the text input box.	A file in the persistant data path location should be created under the name given with the .phys extension.	A file in the persistant data path location is created under the name given with the .phys extension.	02:02
2.5	Attempting to save a file with no name	Click on the 'submit' button in the save window with no text in the text input box	An error message is displayed.	An error message is displayed.	2:48
2.6	Loading a file while in Simulation Mode	Click on the 'Load' button whilst in simulation mode, then click on a button, then click submit.	An active main assembly is instantiated which immediately uses physics and another inactive main assembly is created as a future reference.	An active main assembly is instantiated which immediately uses physics and another inactive main assembly is created as a future reference.	02:58
2.7	Attempting to load a file which does not exist.	Click on the 'Load' button whilst in simulation mode then click submit with the text input box empty.	An error message is displayed	An error message is displayed	03:18

2.8	Checking a simulation has loaded correctly	Load a simulation, as in 2.6.	The NodeConnection lists contained within the node connection scripts attached to each placeable object are populated with the correct nodes, the types are correct and the Assembly Management Script is made aware of the types	The NodeConnection lists contained within the node connection scripts attached to each placeable object are populated with the correct nodes, the types are correct and the Assembly Management Script is made aware of the types	03:37
2.9	Leading on from 2.8, checking the simulation works as expected	Load a simulation, as in 2.6.	The placeable elements are correctly assigned to subassembilies, pivots, split joints and anchors are assigned and set to the right place and setup correctly.	The placeable elements are correctly assigned to subassembilies, pivots, split joints and anchors are assigned and set to the right place and setup correctly.	04:30
2.10	Similar to 2.7, although in edit mode	Click on the 'Load' button whilst in edit mode then click submit with the text input box empty.	A single active main assembly is instantiated which can immediately be modified. The NodeConnection lists contained within the node connection scripts attached to each placeable object are populated with the correct nodes, the types are correct and the Assembly Management Script is made aware of the types	A single active main assembly is instantiated which can immediately be modified. The NodeConnection lists contained within the node connection scripts attached to each placeable object are populated with the correct nodes, the types are correct and the Assembly Management Script is made aware of the types	04:51

4.3 Camera Management

The following tests for section 3 are found in the video at this URL:
https://youtu.be/xy-oRgPe7nA

Test No	Test Description	Action of Input	Expected Result	Actual Result	Video Timestamp
3.1	Test Edit mode camera	Whilst in 'edit mode' click numpad 1,2,4,5,6,7,8	The camera pans to the angles listed in the table in design section 2.3 in perspective mode, then switchs to orthographic view afterwards for each angle other than freeview (num 5).	The camera pans to the angles listed in the table in design section 2.3 in perspective mode, then switchs to orthographic view afterwards for each angle other than freeview (num 5).	00:00

3.2	Check camera returns to free view position when switching from edit mode to simulation mode.	Whilst in edit mode after switching camera to any other position that freeview (num 5), click the "Simulation mode".	The camera switches to perspective view then pans to free view angle.	The camera switches to perspective view then pans to free view angle.	01:31
3.3	Check camera correctly moves to fit all the projectiles/nodes in the scene within its frame whilst simulating	Load an assembly into the scene which contains a moving element which has a large displacement, then click "start" to begin	The camera moves to the side and as the system moves, the camera should move backwards to ensure that the entire system remains within the scene.	The camera moves to the side and as the system moves, the camera should move backwards to ensure that the entire system remains within the scene.	01:48
3.4	Check that you can only move the camera wilst in edit mode	Select 'edit mode' and use the numpad keys as in 3.1. Then switch to simulation mode and press the same numpad keys once again. Finally, click 'start' and press the keys for a third time.	The camera pans corresponding to the tabe in design section 2.3 whilst in edit mode, and the camera remains in its current position whilst in simulation mode or whilst running the simulation.	The camera pans corresponding to the tabe in design section 2.3 whilst in edit mode, however you can still move the camera whilst in simulation mode after running the simulation	02:07
3.5	Check that the movement of the camera in Simulation Mode has been fixed (see corrective action below table)	Select 'edit mode' and use the numpad keys as in 3.1. Then switch to simulation mode and press the same numpad keys once again. Finally, click 'start' and press the keys for a third time.	The camera pans corresponding to the tabe in design section 2.3 whilst in edit mode, and the camera remains in its current position whilst in simulation mode or whilst running the simulation.	The camera pans corresponding to the tabe in design section 2.3 whilst in edit mode, and the camera remains in its current position whilst in simulation mode or whilst running the simulation. The camera does no longer move when pressing numpad keys after running a simulation	02:41

Test 3.4 Corrective Action

Change the value of the boolean `CanSwitch` to `false` from `true` within the `OnSimulationEnd()` procedure in the Camera Management Script, and change `OrientFree();` to `StartCoroutine(FreeCam(FreePosition, FreeRotation, false));` which will ignore the state of `CanSwitch`, and will set `CanSwitch`'s value to false.

4.4 Running A Simulation

The following tests for section 4 are found in the video at this URL:
https://youtu.be/xhgEIFHm1TQ

Test No	Test Description	Action of Input	Expected Result	Actual Result	Video Timestamp
4.1	Check the time starts counting	Click on the 'start' button	The time starts counting	The time starts counting	00:00
4.2	Check the split joint breaks	Click on the 'start' button	The split joints break their connections	The split joints break their connections	00:17
4.3	Check both graphs start plotting points	Click on the 'start' button	Both graphs show red dots plotted against time	Graph 2 starts plotting however graph 1 does not.	00:38
4.4	Check both graphs now start plotting points (See corrective action below the table)	Click on the 'start' button	Both graphs show red dots plotted against time	Both graphs show red dots plotted against time	01:07
4.6	Check you cannot reset the simulation, load a simulation or go into edit mode whilst the simulation is running.	Click on the 'start' button, then click on 'Reset' , 'Edit Mode' and 'Load'	An error message is displayed telling the user that they must stop the simulation before they can Reset the simulation, load a simulation or go back into edit mode.	An error message is displayed telling the user that they must stop the simulation before they can Reset the simulation, load a simulation or go back into edit mode.	01:29

Test 4.3 Corrective Action

Add the line `yield return new WaitForEndOfFrame();` Into the Coroutine `PlotPoints()` within the GraphScript class before it does a presence check on the `ProjScript` variable as the ProjectileDropDownScript class must be allowed time to update this variable to it's correct value.

4.5 The Graphing System

The following tests for section 5 are found in the video at this URL:
https://youtu.be/6KVvwvz3lUg

Test No	Test Description	Action of Input	Expected Result	Actual Result	Video Timestamp
5.1	Check that all projectiles are shown as an option in the projectile drop downs.	Add a projectile to the scene, then click on a projectile drop down, then add another, then click on the projectile drop down again.	The first time there is 1 projectile listed, the second time there is two.	The first time there is 1 projectile listed, the second time there is two.	00:00
5.2	Check that projectiles are shown as the	Move the curser over a projectile	The projectiles have a number presented in from of them which	The projectiles have a number presented in from of them which	00:00

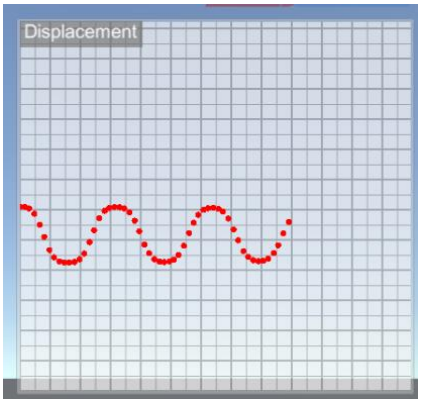
	correct one when hovering over the projectile drop down box.	drop down, then move it away.	corresponds to their position in the Assembly Management Script's list of projectiles. When the curser is moved away, these numbers are hidden.	corresponds to their position in the Assembly Management Script's list of projectiles. When the curser is moved away, these numbers are hidden.	
5.3	Check that the graph stops plotting points when the simulation stops	Run a simulation while plotting points, then stop the simulation	The graph plots points when the simulation is running then stops plotting points when the simulation stops, but the points remain in their position.	The graph plots points when the simulation is running then stops plotting points when the simulation stops, but the points remain in their position.	00:58
5.4	Check that the graph moves to the right correctly and that the point recycling system is running	Run a simulation and wait for the points to reach the other side.	The graph moves to the left to create room for the new points, which are taken from the back of the graph off the screen.	The graph moves to the left to create room for the new points, which are taken from the back of the graph off the screen.	01:15
5.5	Check that the vertical scale of the graph shrinks when points of a greater magnitude than the graph need to be plotted	Run the simulation, with one of the projectiles having a very large displacement which is shown on the graph.	The graph scales vertically to ensure that all plotted points are visible within the graphing window.	The graph scales vertically to ensure that all plotted points are visible within the graphing window.	02:08
5.6	Check that the correct points are plotted from the correct projectile on the correct graph	Create a simulation which contains multiple projectiles, and in edit mode, set the Projectile, attribute and Axis boxes to different projectiles and run the simulation.	The first graph plots the points of the first projectile using the attribute given on the appropriate axis, and the second does the same projectile.	The first graph plots the points of the first projectile using the attribute given on the appropriate axis, and the second does the same for its set projectile.	02:52

Section 5 - Evaluation

5.1 Achived Objectives

Below I have responded to each objective individually and reflected on third party feeback for each objective.

5.1.1 Does the simulation demonstrate the ideas of projectiles and simple harmonic motion through a single solution with the versitility to observe any of displacement, velocity, acceleration and force while the simulation is running?



The graphing system allows the choice of plotting between displacement, velocity, acceleration and force acting on any projectile against time in the scene in which the system is moved by the acceleration due to gravity. While the simulation is running the graphs plot accurate displacement against time curves (which are sine/ cosine curves, pictured to the right). Mr Mumford stated that " Projectile simulation is useful as an extension to the pendulum SHM theory. As a teacher I could use this as students could predict the graph." [Appendix B 1], and when asked if he thinks his students could understand what the simulation is trying to show, he replied "Yes – the software clearly shows the mathematical (graphical) analysis of SHM." [Appendix B 3]
In addition to the graphs, the variable readouts also display the displacement, velocity, acceleration and force while running the simulation.
This means that objective 1 is achived.

5.1.2 Is the simulation a versatile solution which is easily understood and is within context of the A-Level and GCSE physics courses?

The attributes in the simulation are limited to displacement, velocity, acceleration and force given as vectors and so remains entirely within the A-Level course.
In addition, the system uses a constant force for g (the gravitational constant) and so simulations behave similarly to what you would expect them too If they were in real life. Also, UnityEngine has damping inbuilt which means that the amplitude of displacement of a pendulum decreases exponentially over a long period of time, which is also similar to real life and is relevant to the A-Level physics course.
This means that objective 2 has been entirely achived.

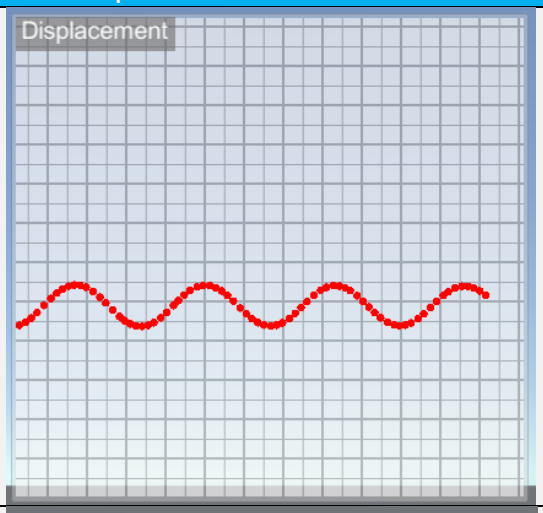
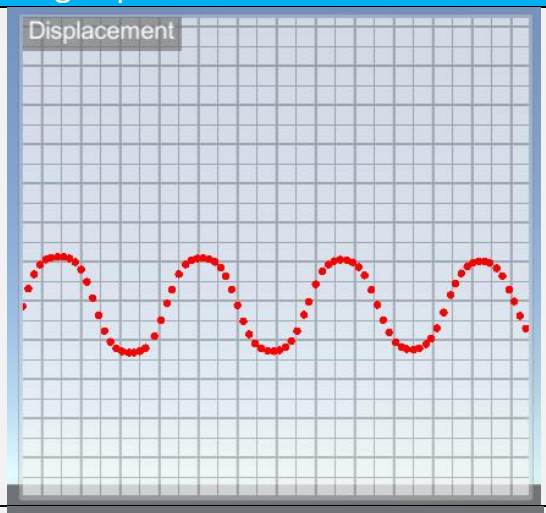
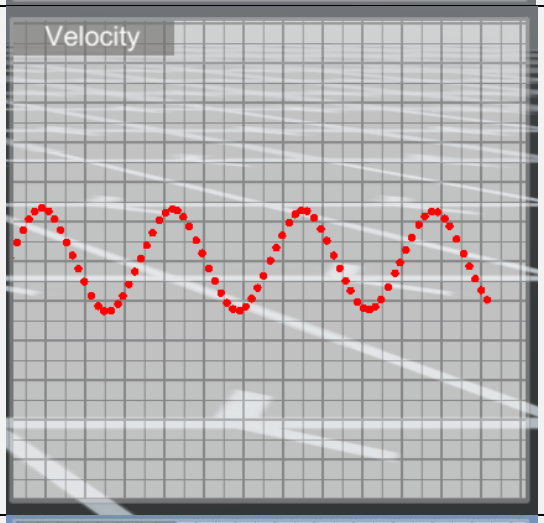
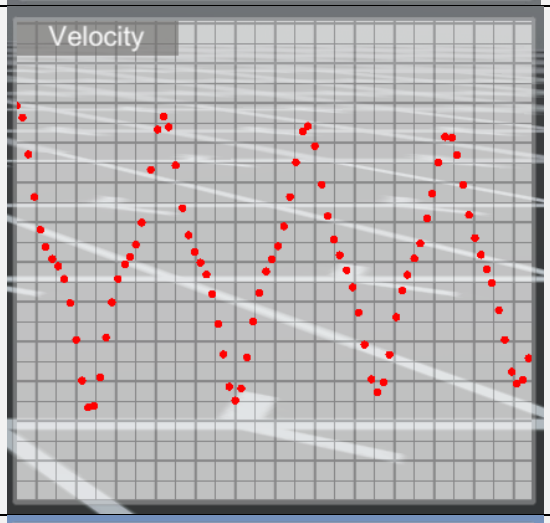
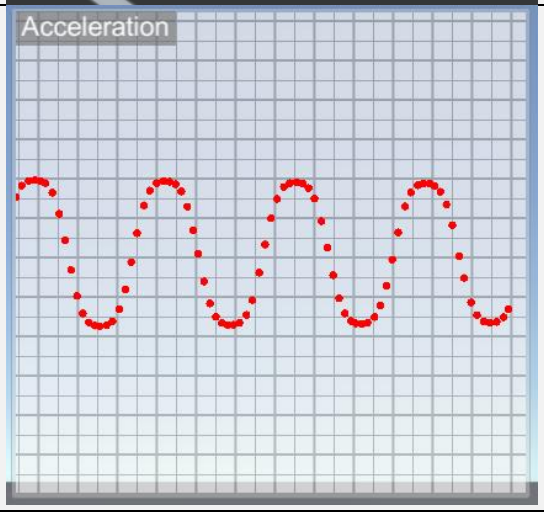
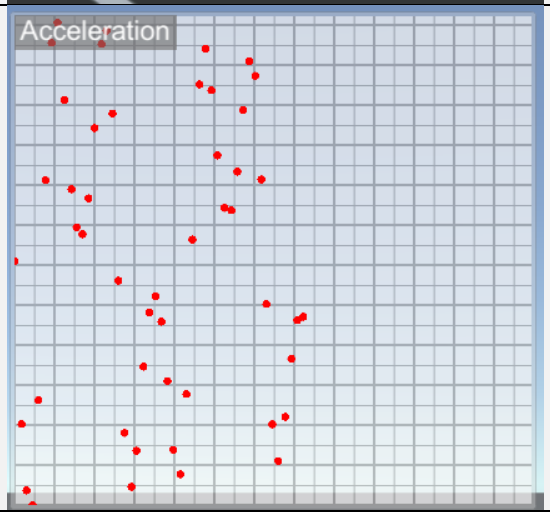
5.1.3 Is the simulation easy for teachers to use and present to a class?

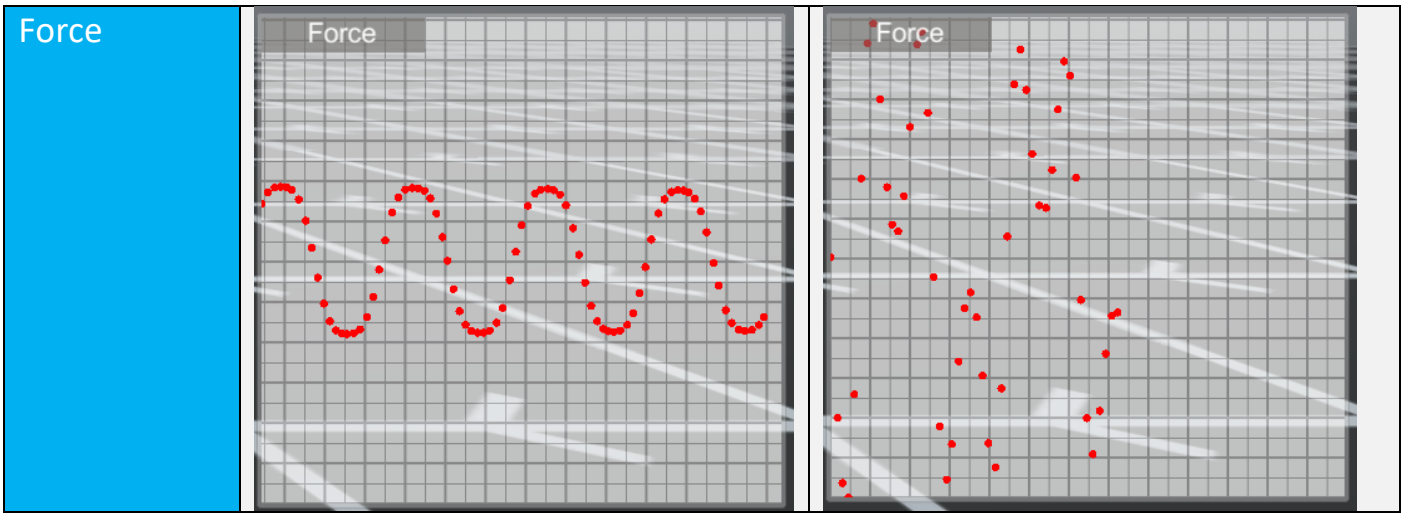
Mr Mumford was very enthusiastic about the program however he did mention outside of the written feedback that he thought it would be useful to make which mode you're currently in more clear. In the written feedback he also mentioned that it would be great to enable students to perform calculations before the simulation is completed to see if they can get the correct result. He also mentioned that "As a teacher I would mainly use this a simulation to be used in lessons." [Appendix B 4]
Below is a picture of the simulation running on the whiteboard of a physics laboratory, running on a windows 10 classroom-level computer.

I would say that objective 3 has been almost entirely achived. With minor improvements such as a label to show which mode you're currently in, this objective could be entirely achived.

5.1.4 Is the user able to plot graphs relating to velocity, acceleration, force and displacement against time?

Yes, although the graphs of velocity, acceleration and force do not follow the exact path they should when the projectile is at high speed as picutred below. I believe this issue to be down to Unity's inbuilt rigidbody system using a rounded velocity which is not accurate enough to plot consistent points and this is used by my algorithm to calculate acceleration using change in velocity per frame hence why the acceleration and force graphs are undesirable at high speed. I could get around this by calculating the velocity of projectiles in my own algorithm which would be designed to be able to calculate highly accurate velocities and in turn accelerations and forces.
Mr Mumford also expressed he found some trouble getting the velocity and acceleration graphs to begin plotting consistently, however I have been unable to recreate this error. It may be that the corrective action taken place in the testing has caused another issue to arise, so this is something I would look into further given the opportunity. This means that objective 4 is partially achived.

	Small Speed	Large Speed
Displacement		
Velocity		
Acceleration		



5.1.5 Does the program include a Simulation Builder which enables teachers to construct their own simulations to be more relevent to their lessons, and the ability to save this as a scene to the local machine?

The program has an EditMode which enables the placing, movement and destruction of structure elements and projectiles. Additionally, these placable objects can be connected together through a system of nodes which can be set as rigid joints, pivots, split joints or anchors. This is documented in testing section 1, The Simulation Builder. Also, these can be saved as preset simulations, stored on the local machine and loaded at a later date. This is documented in testing section 2, Saving And Loading.
When asked if the simulation builder was a useful tool to create custom simulations, Mr Mumford replied with: “As a teacher I would mainly use this a simulation to be used in lessons. As an extension students could build there custom simulation and predict the results e.g. displacement of the projectile.” [Appendix B 4]
This means that Objective 5 is fully achived.

5.2 Areas To Improve

Below is a list of things I would change/ fix immediately given the opportunity.

- Find the cause of Mr Mumford’s struggles to get velocity and acceleration graphs showing properly by running the simulation with him and trying all possible combinations of events to attempt to recreate the issue.
- Add a mode indicator to the top of the screen to show whether the user is currently in edit mode or simulation mode.
- Populate the pre-made simulations into the app data persistant data path location at runtime so the user does not have to do it manually.

Next, a list of things I would like to add or change in the long term if given the opportunity.

- Add the ability to give the students an opportunity to ‘fill in the blanks’ before the simulation completes, as this is something Mr Mumford suggested would be desirable.
- Create documentation and a user guide for how to use the simulation as some people may find it tricky to use without guidance.
- Add tips, descriptions and other visual aids to assist in the construction of a simulation as some people may not be immediately aware of what to do.
- Add a ‘Student Mode’ for teachers to hand over control to their students and let them use the software. This is based off feedback from Mr Mumford where he suggested that students may benefit from having hands-on experience with a simulation.

5.3 Conclusion

Overall, I would say that the program meets the requirements and objectives set out very well, with a comprehensive simulation builder which allows teachers to create custom simulations which can be saved and loaded for use in lessons. The simulation runs at over 30 frames per second on a windows 10 classroom computer which means it's not uncomfortable or nauseating to use and can be resolved from the back of a large physics laboratory when displayed on a projector screen. In addition to this, the versatile graphing system allows for the simultaneous display of two attributes of a projectile which is plotted in real time. With a few minor improvements, as suggested in section 5.2, I believe this program could be a near-perfect solution.

Appendix A

The following sources were used in section 1 Analysis.

University Of Colorado PhET Physics Simulations [Online] Available at: https://phet.colorado.edu/en/simulations/category/physics [Date Accessed: 28/09/2018]

University Of Colorado PhET 'The Ramp' Simulation [Online] Available at: https://phet.colorado.edu/en/simulation/legacy/the-ramp [Date Accessed: 28/09/2018]

Erik Neumann, myPhysicsLab [Online] available at: https://www.myphysicslab.com/ [Date Accessed: 01/10/2018]

Erik Neumann, myPhysicsLab Newton's Cradle Simulation [Online] available at: https://www.myphysicslab.com/engine2D/newtons-cradle-en.html [Date Accessed: 01/10/2018]

Simon Mumford, Head Of Physics, Clitheroe Royal Grammar School [Date Interveiwed: 04/10/18]

Multiple Contributers at Wikipedia.org, SUVAT and circular motion photo, [Online] available at: https://en.wikipedia.org/wiki/Equations_of_motion [Date accessed: 11/10/18]

Appendix B

The following responses are feedback from my end-user, Mr Mumford.

1 What are your favourite features of the program?

"Graphics are excellent, the user can clearly see what the simulation represents. Software quickly loads and it is easy to select the different modes. Interface is easy to use.

Diplacement graph displays a good sine curve to allow students to view the mathematical link.

Projectile simulation is useful as an extension to the pendulum SHM theory. As a teacher I could use this as students could predict the graph."

2 What whould you like to change about the program?

"Is it possible to include different amount of damping?

A graph of vel-time dual display with displacement time.

A cursor to select vel, displacement values from the graph. Especially useful with the projectile simulation."

3 Do you think that your students could understand what the simulation is trying to show?

"Yes – the software clearly shows the mathematical (graphical) analysis of SHM."

4 Is the Simulation Builder a useful tool to create custom simulations?

"As a teacher I would mainly use this a simulation to be used in lessons. As an extension students could build there custom simulation and predict the results e.g. displacement of the projectile."

5 Does the simulation Builder impede on the simulation's clarity and ease of use?

"Yes and No. I feel it could changed to a different 'mode' i.e. click for custom mode. As a teacher uses the software they would only want to use the simulation."

6 Overall, what would you rate the usefulness of the program on a scale of 1-10 (where 10 is perfect) and why?

"It is clear you have a great effort on a challenge software project – well done. I give you a 8."

Signature