Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Pendulum

H446 Component 3

Nishad Deokar

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Analysis

Introduction

Oscillation is an incredibly common phenomenon. A major application of oscillation is the pendulum. The regular motion of pendulums was used for timekeeping, and was the world's most accurate timekeeping technology for 270 years. The pendulum clock achieved an accuracy of about one second per year before it was superseded as a time standard by the quartz clock in the 1930s. To this day pendulums are used in scientific instruments such as accelerometers and seismometers.

My task is to create a simulation of a pendulum. This simulation will be used for educational purposes. As the pendulum is first taught in schools in Year 2 of the A Level Physics course, as well as in the mechanics module of the Further Mathematics A Level course, this simulation will be aimed primarily at older students currently in sixth form. However, that is not to say it cannot appeal to the average person who does not study these subjects and does not know the science behind its motion. For example, chaotic motion exhibited by a double pendulum at large angles is a very unexpected and exciting phenomenon to observe and I believe the ability to experiment with this motion will appeal to people of all levels; from children to experts in the field.

Approaching this project computationally is the best option. A mechanical approach to simulating a pendulum will require apparatus such as a rod, a range of different weights to attach to the end of the rod as well as a pivot from which the rod can swing freely from. While this may not seem like a particularly difficult task for the average Physics student, for a person without such apparatus readily available to them this is an extremely daunting procedure. A computational approach allows for a far more accessible solution that requires only a computer with an Internet connection. Furthermore, a computational solution will allow for greater flexibility as it is far easier to change the mass of the weights, the length of the rod and the amplitude that the pendulum swings through. It also allows for these values to be changed by very small amounts. This makes it beneficial for students who may find it tricky to model certain conditions using only simple apparatus. The following computational methods can be used:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Applicable computational methods

Abstraction

Abstraction is the process of removing unnecessary detail. My simulation will initially use the simple gravity pendulum model. This mathematical model of a pendulum assumes no friction or air resistance. It is used so that the weight will swing back and forth at a constant amplitude and so that the amplitude of the swing does not decline as a result of friction. Abstraction is further used in the 2D modelling of the pendulum. The pendulum in reality is a 3D object. However, we often model it in only 2D as a 2D elevation enables us to see the full oscillation of the pendulum. Hence, abstraction is used.

Decomposition

Decomposition is the process of breaking down a complex problem into smaller, more manageable parts. Decomposition will be used for this problem as the problem must be broken down into two parts one performing relevant calculations using the provided user inputs and two representing these calculations visually in a graphical user interface.

Thinking concurrently

When modelling the pendulum, the angle of the pendulum from equilibrium (θ) and the displacement of the pendulum from equilibrium (x) at a time t must be calculated for every value of t between 0 and the time at which the simulation is terminated. These calculations must be done at once in order for the motion to be completely smooth. Therefore, the program must make use of concurrency.

Thinking procedurally

When calculating each value of θ and x, procedural thinking will be used. However, the procedures used for calculating these values do not necessarily have to be understood at a mathematical level by me as a developer; they are abstracted from me. I only need to be able to know when to use these mathematical procedures in my program.

Stakeholders

As stated earlier, my simulation will primarily be aimed at students studying the Physics A Level as well as any teachers teaching the Physics A Level. This is because while the simulation can be used effectively by almost anyone it is of most benefit to those studying or teaching the Physics

A Level. The features of my simulation will therefore largely be based on the specification of the Physics A Level, although additional features such as friction can be implemented to deepen the understanding of the topic and provide a more robust and exciting solution.

The primary use of the simulation will therefore be to complement the A Level Physics course. However, it can also be used for entertainment purposes when experimenting with different objects other than the standard Sphere at the end of the pendulum. The Physics A Level specification makes the following references to the study of pendulum motion:

- Students must be able to analyse the characteristics of simple harmonic motion.
- Create graphical representations linking the variations of x, v and a with time.
- Required practical 7: Investigation into simple harmonic motion using a mass–spring system and a simple pendulum.

My solution will be able to meet this specification, as it will be able to display the characteristics of simple harmonic motion through the use of a GUI. A corresponding graph can also be implemented into the simulation to meet the requirement of being able to create graphical representations. However, this may not be entirely necessary as a graph can very easily be plotted manually using the results obtained from this simulation. I will therefore not be including a graph plotter in my simulation, as I believe it is often good practise for students to plot graphs manually at this stage of their learning. With the ability to change the length of the rod and the angle between the rod and the downwards vertical, and with the aid of an external stopwatch, Required Practical 7 can be conducted fully using only this simulation.

Existing similar approaches

https://phet.colorado.edu/sims/html/pendulum-lab/latest/pendulum-lab_en.html

- The starting amplitude of the pendulum is set when the pendulum is intuitively dragged by the user to the required angle.
- Other inputs such as length, mass, gravity and friction can be changed using sliders and numerical inputs.
- A ruler and a stopwatch are provided as part of the simulation.
- The animation can be made to run faster or slower. This in combination with the ruler and stopwatch makes this an ideal tool for simulating a practical such as Required Practical 7.

- Only allows up to two pendulums to swing at a time. Does not include a double pendulum. This helps reduce the complexity of the user interface to provide a more focused simulation.

https://www.myphysicslab.com/pendulum/double-pendulum-en.html

- Does not allow you to set exact angles like the simulation above. Therefore, it is more difficult to conduct experiments with.
- Implements the same interactive dragging feature used to control the simulation. Also makes use of sliders and numerical inputs.
- A ruler and a stopwatch are not included. Furthermore, the animation only runs at a certain speed. This suggests that the simulation was not intended for experimental use rather for more recreational use.
- A double pendulum is included here.
- Other features such as the ability to change the colour of the simulation and a live graph are also present. While these features are all very exciting I do not think any of them are essential to my simulation.

https://lsr_lab.gitlab.io/pendulum/main/Code/index.html

- This simulation also includes a drag feature like the previous simulations. It also allows for the length of the rod to be changed using this same click and drag function. It also features a stopwatch that cannot be toggled and is embedded into the simulation.
- This simulation lacks features that were present in the previous two simulations such as the ability to change mass, gravity and friction as well as the ability to take any numerical input. It could also be argued that this simulation has a weaker interface than the other two, in terms of aesthetics.
- However, this simulation has the advantage of having a live graph that traces itself as the pendulum swings. As well as this, it has a live equation that changes value with the position of the pendulum. This clearly has massive educational value and could be a very useful feature to incorporate if implemented correctly.
- Finally, this simulation provides interactive exercises such as trying to match the generated live graph with a sample graph, by changing the values of the amplitude and the length of the rod. It also includes questions at the end of the simulation to test your understanding of the subject. This is beyond the scope of my simulation as while I want my simulation to have educational value I want to create this value in a more subtle way with a simple simulation that can be used to derive equations and graphs rather than a simulation with all the equations and graphs already embedded into it. Although these tracing

features are not essential they should definitely be considered if the project is completed amply before the deadline.

Derived approaches

The simulations above that simulate a single pendulum make use of object oriented programming. Hence, object oriented programming is the approach I will be taking in my project. After using the simulations that include a double pendulum, I have decided to omit this feature from my simulation. This is because while the double pendulum holds great entertainment value it also holds very little educational value in proportion to the amount of additional work it would take to include a double pendulum option. Therefore, as my simulation is designed for educational use first, I will not include a double pendulum option. I will, however, try to include an entertainment option such as the ability to change the object at the end of the pendulum. This is preferable to the double pendulum option as it will be far easier to implement while still giving the simulation a more fun and overall more relaxed vibe. Furthermore, the time saved through this omission can be used in making the single pendulum simulation as robust as possible.

All of the above solutions make use of a GUI. It is impossible to model the motion of a pendulum in a purely numerical way. It is therefore clear that a GUI is essential for my project. I have also decided to make the simulation three-dimensional. This is because I already have some experience using Unity 3D and would be comfortable making a three-dimensional project. Furthermore, none of the above existing solutions made use of three-dimensional features. A three-dimensional simulation would be fundamentally more realistic than a two-dimensional one and I therefore believe that, by making my simulation three-dimensional, I will be able to make an innovative program that truly stands out from its competitors.

Essential features

- Must accurately model the motion of a pendulum
  - The simulation must be a completely accurate representation of real-life pendulum motion in order for it to be considered an educational tool. Therefore, it must use the correct equations of motion in its program.
- The user must be able to set the starting position of the pendulum by clicking and dragging the pendulum into the desired position.
  - This is a common feature seen in most existing solutions, for good reason. It is a highly intuitive way of determining starting positions and allows the user to experiment with different scenarios.
- The user must be able to control the length of the rod.

- This is so that the user can further experiment with different scenarios and observe how these factors affect motion.

- The simulation must run smoothly.
  - The program will have to frequently make large calculations. This could result in the simulation having lag. A significant time delay between the position of the pendulums will result in an unrealistic simulation since the motion will not be smooth. Therefore, lag must be avoided.
- The simulation must include a way of measuring the angle the pendulum is released at.
  - This is so that the user is able to use the simulation in a more practical sense allowing the simulation to be used as a tool from which the user can obtain valuable results from.
- The simulation should include an additional option.
  - This is to make the simulation more fun to use as well as introduce the user to concepts that they otherwise may have been unaware of, e.g. the introduction of friction, an alternative object at the end of the pendulum rod.

While these are the essential features for the project there are also additional features that could be explored, once the essential features have been implemented. Possible additional features include a live graph to accompany the pendulum, an option to toggle animation speed and the ability for the user to change gravity.

Limitations

The simple gravity pendulum model used for this simulation is a limitation, as it assumes no friction or air resistance. This model is used because the effects of air resistance can often be ignored as they do not significantly affect the observed motion of the pendulum. This is a necessary limitation because the effects of air resistance are very hard to incorporate into the program as it makes the mathematics considerably more complicated and will make the program more prone to error for little reward. Air resistance should not be confused with a simplified model of friction, which is far easier to implement. This limitation is, however, only based on the essential features and can be removed once the essential features are implemented as stated earlier.

Requirements

- OS: Windows, macOS, Linux

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

- Processor: 2.0 GHz
- Memory: 2 GB RAM
- Graphics: 128MB
- Storage: N/A

Success criteria

Criterion	Justification	Measurement
The program must demonstrate pendulum motion.	This will determine whether or not the pendulum is demonstrating the relevant equations correctly to the user.	The simulation will display characteristics of a motion that is widely accepted and defined as pendulum motion.
The controls must be intuitive and comfortable to use.	This will make the program efficient to use and easy to learn for new users.	The user should be able to use the simulation without any difficulty and without the need for any prompts outside of the program.
The program must have a practical use i.e. the ability to be used as a tool.	This will fulfil the primary purpose of the program that is to complement the A Level Physics course.	The user should be able to obtain correct calculations and answers using the results obtained from this simulation, linking back to the first success criteria.

The program must allow a range of different lengths and angles to be inputted.	As stated in the essential features section, this is so that the user can experiment with different scenarios and thus gain a deeper understanding of the topic.	The user should be able to run the simulation using any combination of values, provided the values are valid.
The program must be able to deal with all valid, invalid and extreme inputs from the user.	This will make the program more robust and less prone to crashing. This provides the user with a more reliable experience.	The user should not be able to crash the program as a direct result of an invalid or extreme input.
The program can be used from within a web browser.	This will make the program as accessible as possible making it possible for students and teachers to use the program both in lessons and at home.	The user should be able to open the URL and run the program smoothly on any web browser and therefore any operating system.

Summary

The essential features show that the problem can be broken down into the following parts:

- The program must be able to perform the necessary calculations to determine single pendulum motion and display this motion in a basic view.

- The program must have a GUI that allows the user to interact with the simulation and allows the user to determine the starting position of the pendulum and change the length of the rod.
- A protractor must then be embedded into the program.
- It must then contain an additional feature, such as the ability to change friction, gravity or the object suspended at the end of the pendulum rod.

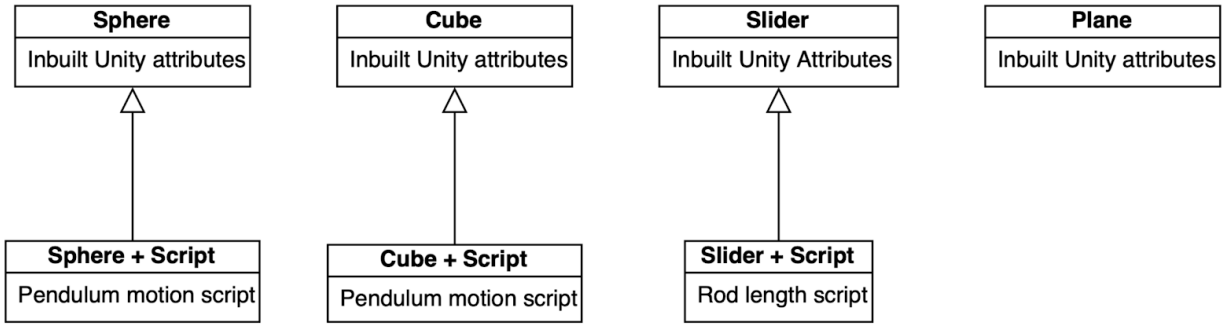
I have made the decision to write my program in C# and use the Unity engine. This is because I have experience with, and am most comfortable, using Unity 3D. As well as this, Unity makes the implementation into a web browser relatively easy when compared to alternative solutions.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Design

Program structure

As my program will be written in C#, I have decided to take an object oriented approach. As a result, my simulation will consist of classes and subclasses. The structure of the program may therefore be illustrated using class diagrams, as follows:



It should be noted that the representation of classes here will likely differ slightly in the actual implementation due to the way in which the inbuilt classes in Unity behave. This model does however help visualise the overall program structure. Below is an explanation of the above classes.

Sphere + Script

This is a subclass of the superclass Sphere. The subclass Sphere + Script therefore inherits all attributes and methods of the superclass Sphere. In practical terms, as the class Sphere is an inbuilt class in Unity which makes an item spherical, all objects of the class Sphere + Script will also be spherical. However, this class also defines its own methods required to perform pendulum motion. This is an example of polymorphism. Instances of the class Sphere + Script will therefore be the mass (pendulum bob) and the pivot, as both of these objects are spherical and undergo pendulum motion in the context of the simulation.

Cube + Script

This is a subclass of the superclass Cube. The subclass Cube + Script therefore inherits all attributes and methods of the superclass Cube. In practical terms, as the class Cube in Unity makes an item cuboid, all objects of the class Cube + Script will also be cuboid. Similar to the Sphere + Script class, this class also defines its own methods required to perform pendulum motion. This is another example of polymorphism. An object of this class will therefore be the

rod that is suspended by the pivot and from which the mass is suspended by. This is because the rod is cuboid and undergoes pendulum motion in the context of the simulation.

Slider + Script

This is a subclass of the superclass Slider. The superclass Slider is an inbuilt class in Unity which creates a standard slider that can be moved between a minimum and maximum value. As Slider + Script is a subclass of this superclass, it inherits all attributes and methods from it and therefore all objects of the class Slider + Script are standard sliders. The class Slider + Script also defines its own methods. These methods are required to scale the size of the object Rod based on the input of the slider hence changing the length of the rod which is ultimately what the purpose of this class is, although this may be extended to change the value of friction and gravity as the project develops.

Plane

There is no external script and therefore no polymorphism associated with this class. This is because an object of the class Plane will at no point be in motion or transformed in any way. In other words, all instances of the Plane class will be stationary so no additional script or subclass is required here. The object of the Plane class in the context of this program is the protractor used to measure the angle at which the pendulum is released from. The protractor object therefore contains all the inbuilt Unity attributes and methods of the Plane class, as it is an instance of this predefined class.

Following this explanation of classes, I will now provide an explanation of the various objects of these classes used within this simulation.

Let a denote class Sphere
- a* denote class Sphere + Script,
- b* denote class Cube + Script,
- c* denote class Slider + Script,
- d denote class Plane, and

Object [class]	Description	Explanation	Justification
Node [a]	This object defines a sphere, which will from now on be referred to as a node, that appears whenever the mouse	This feature is implemented primarily to make the subsequent calculations of the amplitude of the	However, this object also has a practical purpose to the user. The node makes using the protractor object easier, as the

	is clicked.	pendulum easier to visualise as a developer and hence easier to code.	desired angle can be selected with more ease and more precision through the use of the node object.
Pivot [a*]	This object, as its name suggests, defines the pivot from which both the rod and the mass are suspended from.	This object however, while inheriting from the class Sphere + Script, does not have to be spherical at all. In fact, it does not even necessarily have to be visible to the user.	Why include this object at all then? I have included the pivot as an object because as a user it is easier to understand the pendulum suspended from a single point, the pivot, as opposed to viewing the pendulum as a floating object. Hence, the pivot inherits from the class Sphere, as this shape makes the most sense for a pivot.
Mass [a*]	This defines the object suspended at the end of the rod and hence the pivot.	This mass will initially be presented as a spherical object. Hence it inherits from the class Sphere + Script. However, as explained earlier, the mass does not necessarily have to be spherical and its shape could later be changed as the project develops, in order to make a more interesting and entertaining simulation.	However, at this stage, I will model the mass as a spherical object as this is the most standard way of representing it. It is worth noting that the shape of this mass can be changed later on, by creating a new subclass which inherits from the class of the new shape. This new subclass should also contain the methods required to perform pendulum motion.

Rod [b*]	This defines the rod that is suspended by the pivot and from which the mass is suspended by.	The rod will inherit attributes from the class Cube + Script. The cuboid shape inherited from this class will have to be modified such that the rod is as long and as narrow as possible.	This modification of the cuboid shape could be done using methods, included in the script part of Cube + Script, alongside the methods calculating pendulum motion. However, in practise, I will likely perform this modification within Unity itself using no external script. This action would be hard to model in the design phase and I will therefore try to illustrate it during development.
Slider [c*]	This object defines the most essential slider used within the simulation; the slider used to change the length of the rod.	However, as explained earlier, very similar objects can be created for additional sliders used to change friction and/or gravity.	To create these additional sliders, new subclasses would have to be created. These subclasses would also inherit from the superclass Slider. However, unlike Slider + Script, the subclasses would define methods required to change the value of fraction and/or gravity accordingly.
Protractor [d]	This object defines the protractor used to measure the angle at which the pendulum is released from.	The protractor inherits attributes from the class Plane. It is therefore essentially a 2D object in an otherwise 3D simulation. This	For the object to be of any use, it will need to have an image of a protractor embedded within its plane. This could be done by creating a new

		model is ideal, as a protractor, for all intents and purposes, is basically a 2D object when used in practise.	subclass that inherits from the class Plane. This subclass would then have to define its own attributes in order to embed an image of a protractor. The protractor would then instead be defined as an object of this new subclass. However, in practise, I will likely perform this modification within Unity itself using no additional subclass. Similar to the cuboid modification, this action would be hard to model in the design phase and I will therefore try to illustrate it during development.

As you can see, the program does not have too great a number of classes and subclasses. This is largely due to the dedicated purpose it provides. The algorithms defined in the subclasses of the program introduce complexity to the project.

Algorithm design

Below are the algorithms for each element of functionality. They are each as accurate as I can make them, although they will likely be written slightly differently in implementation due to the heavy utilisation of Unity features within these algorithms.

Initialisation()

At the beginning of the program, certain variables will have to be initialised. This algorithm Initialisation() is designed to initialise: the value of the angle at which the pendulum is released

from (θ_0), the angle of the pivot due to θ_0, and the timer. The value of θ_0 at the start of the simulation, before any user input, will be 0° as the pendulum must start at equilibrium. The angle of the pivot due to θ_0 will therefore also be 0°. The timer must be initialised to 0 at the beginning of the simulation as no time has elapsed. The pseudocode for the algorithm may therefore be written as follows:

```
procedure Initialisation()
      maxAngle = 0 //maxAngle is another term for θ0
      pivot.rotation = vector(0, 0, 0)
      currentTime = 0
end procedure
```

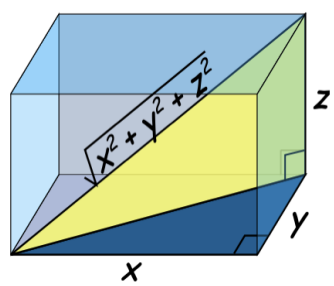
These variables will hold different values as the user begins to use the program.

FindLength()

This algorithm is designed to find the length of the rod. This is achieved by using the Pythagorean theorem to find the distance from the mass to the pivot, hence finding the length of rod. As I am making this project using Unity3D, the following version of the Pythagorean theorem will be used:

$$c^2 = x^2 + y^2 + z^2$$

This result may be illustrated using the following diagram:



The algorithm may therefore be written as follows:

```
function FindLength(mass, pivot)
      length = sqrt((mass.x - pivot.x)^2 + (mass.y - pivot.y)^2 +
(mass.z - pivot.z)^2) //object.x denotes the x coordinate of the
object, and so on.
      return length
end function
```

Note that the algorithm is a function that returns the value of the length of the rod. Hence, when this function is called, the length of the rod is recalculated. This function is essential for the

overall program as the length of the rod is necessary for subsequent calculations, such as the maximum displacement of the pendulum from equilibrium.

SetNodePos()

This algorithm is designed to move the node object to the position of the mouse. More specifically, it sets the position of the node to the position of the mouse. Due to the need for mouse position in this procedure, the actual implementation will likely make use of inbuilt Unity functions in order to obtain and make use of this position.

The pseudocode for this algorithm may be written as follows:

```
procedure setNodePosition()
      node.position = vector(input.MousePosition.x,
input.MousePosition.y, input.MousePosition.z) //the node position is
set using a three-dimensional vector because this is how the three
input mouse positions would be defined in space, in Unity.
end procedure
```

Due to the nature of the simulation, the user sets the maximum angle of the pendulum from equilibrium (θ_0). This is equal to the angle at which the pendulum is released from. θ_0 is therefore calculated using the position of the pivot (which is constant) and the position of the user's mouse. Hence, by moving the node object to the position of the mouse, θ_0 can similarly be calculated using the position of the pivot and the position of the node object. The position of the node object is calculated in the above procedure, setNodePosition(). Therefore, the calculation of θ_0 is how the procedure setNodePosition() fits in with the rest of the program.

CalculateConstants()

This algorithm is designed to calculate the constants required for this simulation. These constants are: the length of the rod, the maximum displacement of the pendulum from equilibrium (x_0) and the period of oscillation.

Length of the rod

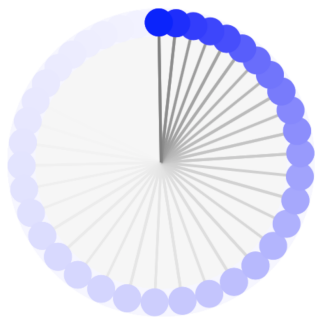
We have already calculated the length of the rod in the function FindLength(). However, this would mean we would have to call this function every time the length of the rod is required. As the length of the rod is required in subsequent calculations it would be preferable to assign the length returned from the function FindLength() to some variable, in order to make the subsequent calculations easier to read and maintain. The following pseudocode illustrates how this could be achieved:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

```
lengthOfRod = FindLength(mass, pivot)
```

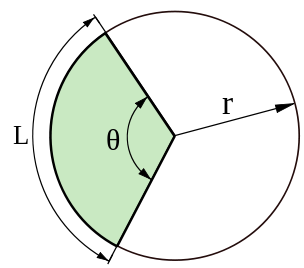
Maximum displacement

We can see through observations of pendulum motion that the pendulum follows a circular path.



Using this information, we can calculate the displacement fairly easily, as follows:

We know through the definition of a radian that $\theta = s / r$, where θ is the subtended angle in radians, s is arc length, and r is radius. This result may illustrated using the following diagram:



In the context of the overall program, θ is θ_0 (defined earlier), r is the length of the rod (calculated from the function FindLength()), and s is the maximum displacement of the pendulum from equilibrium. We are trying to calculate this value of s. Therefore, as we now know the values for r and θ, we can rearrange the equation $\theta = s / r$ to $s = r\theta$ and hence calculate the maximum displacement. The following pseudocode illustrates this calculation:

```
maxDisplacement = maxAngle * lengthOfRod
```

Period of oscillation

The time for one complete cycle, a left swing and a right swing, is what is meant by the period of oscillation. The period of oscillation (T) is:

$$T = 2\pi\sqrt{\frac{L}{G}}$$

In the context of the simulation, L is the length of the rod and g is the local acceleration of gravity. The period of oscillation could therefore be calculated using the following pseudocode:

```
timePeriod = 2 * π * sqrt(lengthOfRod / g)
```

The entire pseudocode for the procedure CalculateConstants() may therefore be written as follows:

```
procedure CalculateConstants()
      lengthOfRod = FindLength(mass, pivot)
      maxDisplacement = maxAngle * lengthOfRod;
      timePeriod = 2 * π * sqrt(lengthOfRod / g);
end procedure
```

The constants calculated in this procedure are frequently used again throughout the rest of the program. For this reason, this procedure is very useful as it makes it quicker and easier to make use of these constants in subsequent calculations.

Timer()

This algorithm is designed to act as a timer by incrementing the current time (initialised at 0) every frame. The pseudocode for this is as follows:

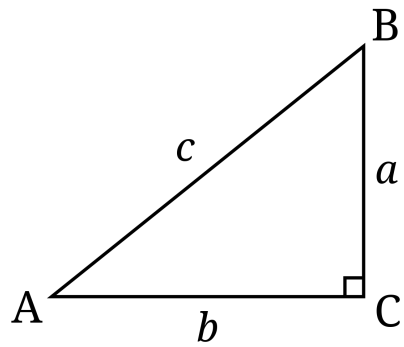
```
procedure Timer()
      currentTime += time since the last frame //this idea of frames is
derived from Unity and as a result the use of inbuilt Unity functions
will be required in actual implementation.
end procedure
```

The current time (t) is fundamentally required to calculate the current displacement of the pendulum from equilibrium at current time t (x), given t is less than the period of oscillation. By incrementing the value of t every frame, we can thereby recalculate the value of x frequently in small intervals. This will in turn result in smooth pendulum motion being exhibited to the user.

IfClicked()

This algorithm works as follows: if the mouse is clicked, the maximum angle of the pendulum from equilibrium (θ_0) is calculated based on the position of the node (calculated in SetNodePos) and the position of the pivot (which is constant). This calculation can be illustrated using trigonometry. Consider the following diagram:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131



Let point A be the position of the node, B be the position of the pivot and length c be the length of the rod.

Therefore, using trigonometry, it can be deduced that the angle at which the pendulum is released from (θ_0) is equal to arctan(b / a). Length b can be calculated by finding the difference between the x coordinates of A and B. Length a can similarly be calculated by finding the difference between the y coordinates of A and B. The calculation for θ_0 can therefore be illustrated in the following pseudocode:

```
maxAngle = arctan((node.y - pivot.y) / (node.x - pivot.x))
```

As well as recalculating the angle at which the pendulum is released from, when the mouse is clicked, the rotation of the pivot should also be recalculated appropriately. As the angle at which the pendulum is released from is equal to θ_0, it follows that the rotation of the pivot should also be set to θ_0. An object can be represented in three-dimensional space in Unity using the following coordinate system:

It therefore follows that, when the mouse is clicked, the pivot must be rotated by angle θ_0 about the z-axis. The following pseudocode illustrates how this could be achieved:

```
pivot.rotation = vector(0, 0, maxAngle) //where vector parameter is in
the format vector(x, y, z)
```

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

We know that the time period (T) is defined as the time taken for one left swing and one right swing. Therefore if the time at which the pendulum is at the maximum angle from equilibrium is equal to 0, then the time taken for the pendulum to oscillate back to this angle is equal to T. It follows that when the pendulum is released from θ_0, and hence when the mouse is clicked, the value of current time t should be set to 0. The entire pseudocode for the procedure IfClicked() may therefore be written as follows:

```
procedure IfClicked()
      SetNodePos() //procedure called to obtain the position of the
node, which is required in order to calculate the maxAngle.
      maxAngle = arctan((node.y - pivot.y) / (node.x - pivot.x))
      pivot.rotation = vector(0, 0, maxAngle)
      currentTime = 0
end procedure()
```

The above algorithm determines the motion of the pendulum when the mouse is clicked.
The below algorithm determines the motion of the pendulum when the mouse is not clicked.

Motion()

This algorithm is designed to calculate: the current displacement of the pendulum from equilibrium at current time t (x), the angle of the pendulum from equilibrium at current time t due to x (θ), and the rotation of the pivot due to θ.

Current displacement (x)

We know through the study of simple harmonic motion that the current displacement (x) at a time t is given by:

$$x = A\cos\omega t$$

where A is the maximum displacement and ω is the angular frequency.
As the maximum displacement is calculated in the function CalculateConstants() and the angular frequency is given by 2π / time period, the current displacement can be calculated using the following pseudocode:

```
currentDisplacement = maxDisplacement * cos((2 * π  / timePeriod) *
currentTime)
```

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Angle of the pendulum from equilibrium (θ)

We know through the definition of a radian that θ = s / r. Therefore, if we let s equal the current displacement and r equal the length of the rod, the angle of the pendulum from equilibrium at current time t due to x (θ) may be calculated as follows:

```
currentAngle = currentDisplacement / lengthOfRod
```

Rotation of the pivot

Using the same logic as earlier, the rotation of the pivot may be calculated using the following pseudocode:

```
pivot.rotation = vector(0, 0, currentAngle) //where vector parameter
is in the format vector(x, y, z)
```

The entire pseudocode for the procedure Motion() may therefore be written as follows:

```
procedure Motion()
     CalculateConstants() //procedure called to obtain the values of
the length of the rod, the maximum displacement and the time
period, all of which are required in the calculations that follow.
     currentDisplacement = maxDisplacement * cos((2 * π  / timePeriod)
* currentTime)
     currentAngle = currentDisplacement / lengthOfRod
     pivot.rotation = vector(0, 0, currentAngle)
end procedure()
```

This algorithm calculates the current displacement of the pendulum from equilibrium at current time t (x). This value of x is then used to calculate the angle of the pendulum from equilibrium at current time t (θ). This value of θ is then finally used to set the rotation of the pivot and hence the pendulum at current time t. As current time t is incremented every frame, in the procedure Timer(), it follows that the simulation will achieve smooth pendulum motion as a result of this algorithm.

Slider()

This algorithm is designed to scale the length of the rod based on the input of the slider.

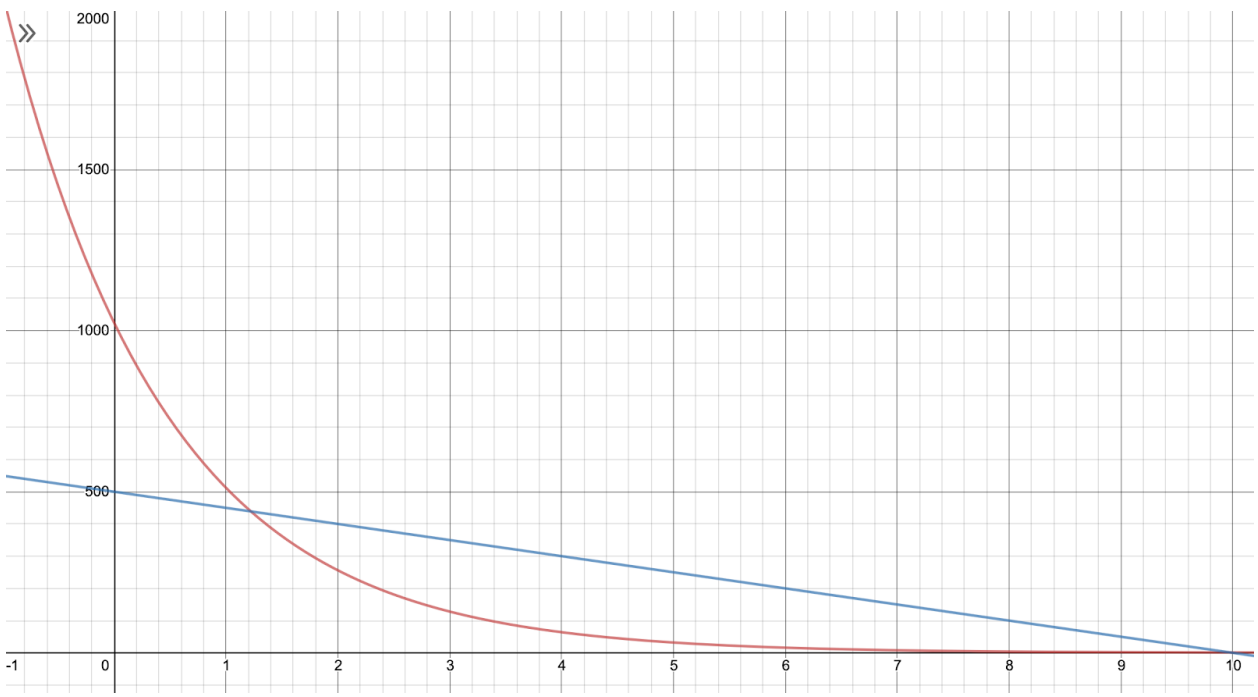
```
procedure Slider()
     new_length = slider.value
```

```
      rod.scale = vector(current length, new_length, current length)
//length determined by y-axis
end procedure
```

Note that unlike all previous routines, this algorithm will be associated with class Slider + Script and will not be associated with classes Sphere + Script and Cube + Script. This is because the procedure Slider() does not contribute in determining pendulum motion.

Damping()

There are two general approaches to how damping may be achieved. One is an arithmetic approach, where the maximum angle of the pendulum from equilibrium (θ_0) is decreased by a fixed amount (the damping factor) every frame. The other is a geometric approach, where (θ_0) is decreased by a fraction (the damping factor) of (θ_0) every frame. The following graph illustrates how the two approaches (geometric in red, arithmetic in blue) tend to 0 over time.



It is clear to me from these graphs that the geometric progression is the more natural of the two. This makes sense since geometric progressions are often observed in natural phenomena, e.g. radioactive decay, bacteria growth. Therefore, in order to simulate the damping that occurs in the natural phenomena of oscillation, I will be using a geometric method of damping. The pseudocode for this method is as follows:

```
procedure Damping()
maxAngle = (maxAngle - (dampingFactor * time since the last frame))
```

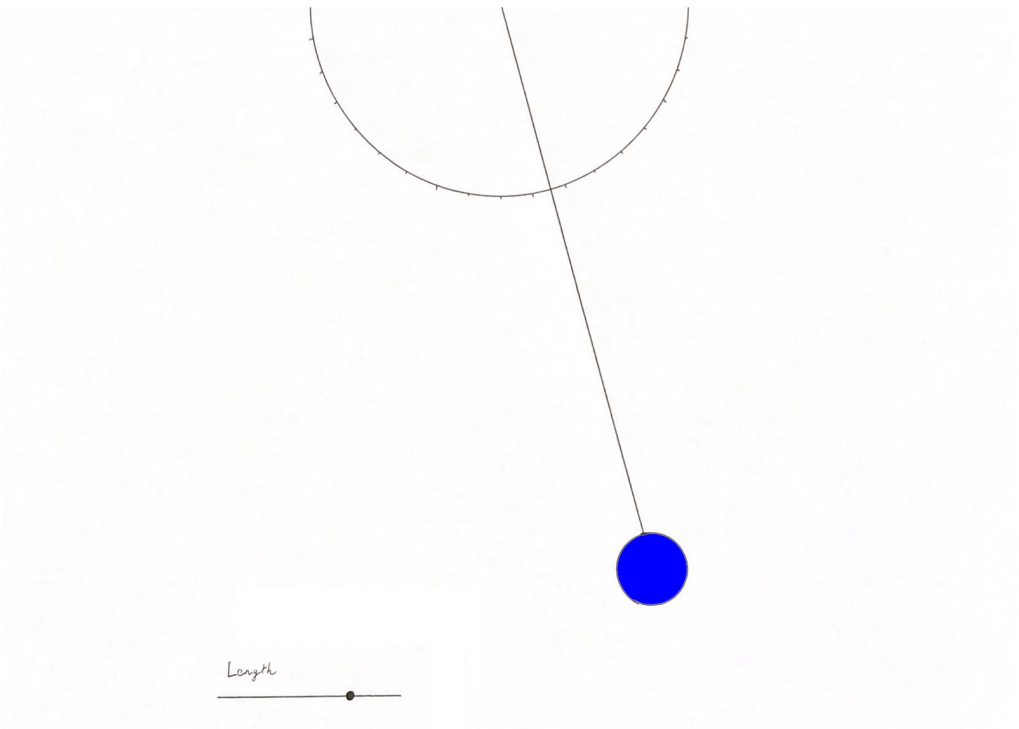
Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

```
end procedure
```

The above method Damping() could be used with the class Slider to make a new subclass of Slider. This would allow the damping factor to be set by the user through the use of a slider. This feature is something I will attempt to implement during development.

Usability features

I have decided to use very simple colours for my simulation, namely blue and white. This is to give the simulation a clean and professional appearance. It should be remembered that this is not only a simulation but also a tool from which experimental results may be drawn from. Therefore a minimalistic design, with as little text and artwork as possible, is preferred. The following diagram shows how this minimalist interface may be manifested:



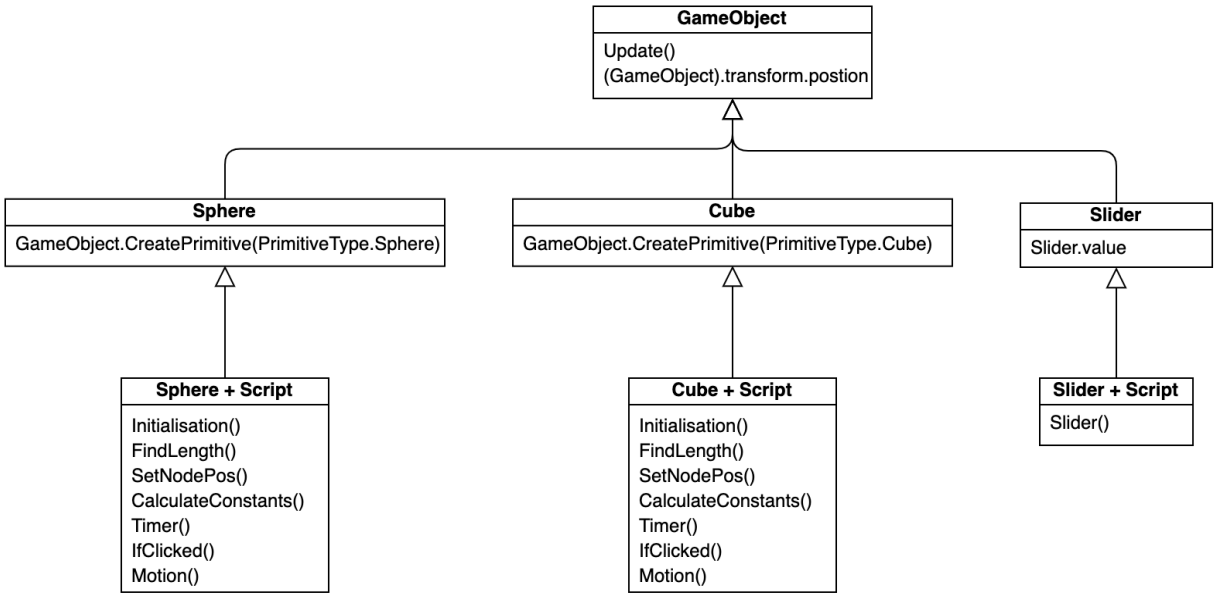
The input for this simulation is limited to only mouse input. There is therefore no requirement for a keyboard. This makes the simulation extremely accessible and easy to navigate. Furthermore, because the program can be operated using only a mouse, it follows that the program can be operated using one hand. This is a particularly useful design feature when considering the program as a tool, as it effectively frees one of the users hands allowing them to possibly record the results obtained from the simulation using this free hand.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Finally, there is no need to implement accessibility features for the hard of hearing. This is because, as part of the minimalist design, I will not include audio as part of the simulation. Therefore, the hard of hearing will not be disadvantaged.

Key variables, data structures, classes

Below is a revised version of the class diagram shown in the program structure section, this time including a list of attributes and methods.



Below is a table of all of the key variables to be used in the simulation.

Module/Routine	Identifier	Data type	Description	Justification
CalculateConstants()	maxDisplacement	float	The maximum displacement of the pendulum from equilibrium.	Used to calculate the current displacement of the pendulum from equilibrium at current time t.
Motion()	currentDisplacement	float	The current displacement of the pendulum	Used to calculate the angle of the pendulum from

			from equilibrium at current time t.	equilibrium at current time t.
CalculateConstants()	lengthOfRod	float	The length of the rod that is suspended by the pivot and from which the mass is suspended by.	Used in the calculation of the time period of the pendulum.
Timer()	currentTime	float	The current time (initialised at 0), which is incremented every frame.	Used to calculate the current displacement of the pendulum from equilibrium at current time t.
CalculateConstants()	timePeriod	float	The time taken for one left swing and one right swing.	Used to calculate the current displacement of the pendulum from equilibrium at current time t.
CalculateConstants()	gravitationalAcceleration	float	The acceleration of gravity. On Earth this is equal to 9.80665 m/s^2.	Used in the calculation of the time period of the pendulum.
IfClicked()	maxAngle	float	The maximum angle of the pendulum from equilibrium. This is equal to the angle at which the pendulum is released from.	Used to calculate the maximum displacement of the pendulum from equilibrium at current time t. Also used to set the rotation of the pivot when the mouse is clicked.

Motion()	currentAngle	float	The angle of the pendulum from equilibrium at current time t, due to its current displacement.	Used to set the rotation of the pivot at current time t.
Damping()	dampingFactor	float	The factor the maximum angle decreases by every frame.	Used to geometrically reduce the maximum angle of the pendulum from equilibrium, until it is equal to 0.

Validation

The Unity editor itself will run validation checks on the input data before the program is compiled. Since the simulation will not take a great amount of input from the user during runtime, the Unity editor will deal with most of the validation. However the editor will not be able to detect valid extreme data so validation checks for valid extreme data will have to be put in place, as follows:

Module/Routine	Input	Validation rule used	Explanation of validation rule	Justification of validation rule	Test data	Justification of chosen test data
IfClicked()	Mouse click	Format check	Checks that the input is in the specified format of a left click.	Taking only left clicks can reduce the number of accidental clicks made by the user.	Right and left click multiple times.	This includes both a valid format and an invalid format.
Slider()	Rod length	Range	Checks	The length	Drag the	By

		check	that the slider input is within the specified range of values.	cannot be negative and so has a minimum value of 0. The length cannot be infinitely long and must therefore have some maximum value. All values between 0 and this maximum value are considered valid.	slider to the minimum and maximum inputs.	gradually dragging the slider from the minimum to the maximum value all valid, as well as extreme valid, data are tested. The use of a slider prevents the user from being able to input invalid data. Hence, invalid data does not need to be tested.
Damping()	Damping factor	Range check	Checks that the slider input is within the specified range of values.	The damping factor must be between 0 and 1.	Drag the slider to the minimum and maximum inputs.	By gradually dragging the slider from the minimum to the maximum value all valid, as well as extreme valid, data are tested.

						The use of a slider prevents the user from being able to input invalid data. Hence, invalid data does not need to be tested.

Test Design

Inter-development testing

Module/Routine	Test	Test data	Justification	Expected outcome
Initialisation()	Run the simulation.	N/A	This will test that the simulation starts at equilibrium.	The simulation will start at equilibrium when run. That is, the entire pendulum is vertical and stationary.
FindLength()	Run the simulation and compare the calculated value with the real value.	Debug.Log()	This will test that the calculated value is accurate.	The calculated value will be equal to the real value.
SetNodePos()	Click the mouse on or drag the mouse to various positions on the	Click the mouse on 5 randomly generated positions on the	This will test that the node consistently moves to the	The node consistently follows the mouse to its

	screen.	screen. Drag the mouse to 5 randomly generated positions on the screen.	mouse position when the mouse is both clicked and dragged.	various positions.
IfClicked()	Click the mouse on a specified angle along the protractor. Let the pendulum swing for 3 cycles.	Set the damping factor to 0. Then click the mouse on 3 different values (as x = 360 - x) along the protractor in steps of 60°, starting from 0° and ending on 360°. Let the pendulum swing for 3 cycles for each angle.	This will test whether the pendulum continues to reach the same position it was released from (maximum angle) after 3 full cycles, when the damping factor is 0.	The pendulum will continue to reach its maximum angle consistently after 3 cycles for various values of θ_0.
Slider()	Drag the slider to the minimum and maximum inputs.	Keep the damping factor constant. Drag the slider through from the minimum value to the maximum value. This is preferable to clicking as it allows all values of the length of the rod to be tested easily.	This will test that the length of the rod can be changed appropriately using the slider.	The length of the rod will increase as the value of the slider increases.
Damping()	Drag the slider to the minimum and maximum inputs.	Keep the length constant throughout. Drag the slider through from the minimum value to the maximum	This will test that the pendulum decreases in amplitude with each swing, until it eventually	The amplitude of the pendulum will: not decrease when the damping factor is equal to 0; decrease

		value. This is preferable to clicking as it allows all values of the damping factor to be tested easily.	reaches 0.	proportionally to the damping factor as it tends to 1 and decrease to 0 after one cycle when the damping factor is equal to 1.
Motion()	Record the time taken for the pendulum to return to its maximum angle and compare this value with the expected value for varying starting positions and starting lengths.	Set the damping factor to 0. Then click the mouse on some position on the screen. Record time taken using external stopwatch. Repeat this for 5 randomly generated lengths of rod.	This will test that, when damping = 0, the time taken for one left swing and one right swing is as expected.	The values recorded should be equal to the values calculated using the equation: $T = 2\pi\sqrt{\frac{L}{G}}$

Post-development testing

Test	Test type	Test data	Justification	Requirements for successful result
Record the time taken for the pendulum to return to its maximum angle and compare this value with the expected value for varying starting positions and starting lengths. This will test	Functionality	Set the damping factor to 0. Then click the mouse on some position on the screen. Record time taken using external stopwatch. Repeat this for 5 randomly generated lengths of rod.	This will show if success criterion 1 and 3 have been met.	The values recorded will be equal to the values calculated using the equation: $T = 2\pi\sqrt{\frac{L}{G}}$

that, when damping = 0, the time taken for one left swing and one right swing is as expected.				
Beta testing, in which potential users are asked to use the simulation, in order to provide feedback on the usability of the program.	Usability	Usability is measured with use of feedback from users.	This will show if success criterion 2 has been met.	Positive feedback received.
Click the mouse on a specified angle along the protractor. Let the pendulum swing for 1 cycle. Repeat this 6 times. Drag the slider to the minimum and maximum inputs. This will test that the length of the rod can be changed appropriately using the slider.	Functionality	Set the damping factor to 0. Then click the mouse on 6 different values along the protractor in steps of 60°, starting from 0° and ending on 360°. Let the pendulum swing for 1 cycle for each angle. Keep the damping factor constant. Drag the slider through from the minimum value to the maximum value. This is preferable to clicking as it allows all values	This will show if success criterion 3 and 4 have been met.	The pendulum will continue to reach its maximum angle consistently after each cycle for the various values of θ_0. The length of the rod will increase as the value of the slider increases.

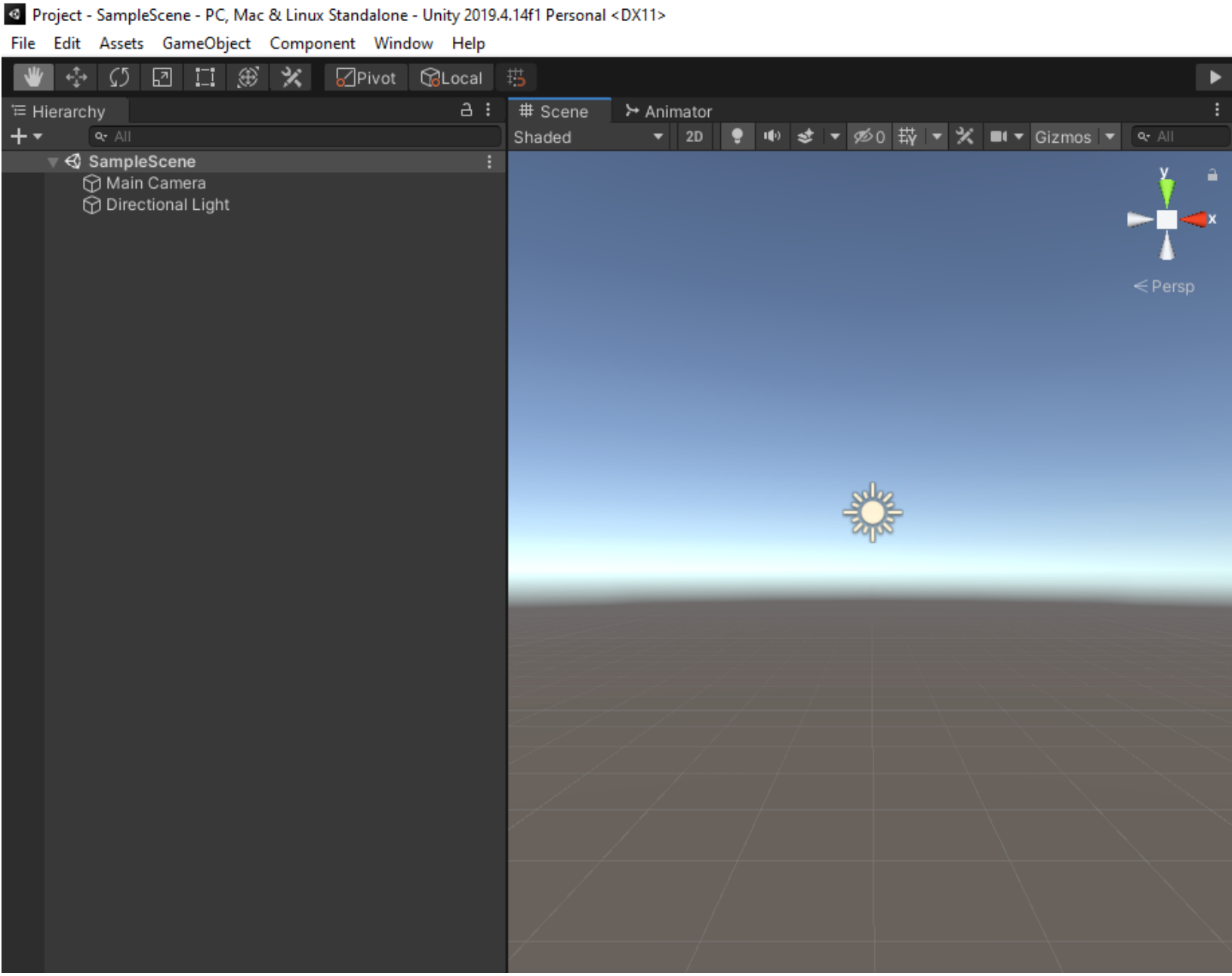
		of the length of the rod to be tested easily.		
See validation checks.	Robustness	See validation test data.	This will show if success criterion 5 has been met.	The program is able to deal with all valid, invalid and extreme inputs from the user.
Use the simulation from within a web browser.	Robustness	Open the link to the web page containing the program.	This will show if success criterion 6 has been met.	The program can be used from within a web browser and retains full functionality when used in this manner.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Development

Creating the graphical user interface

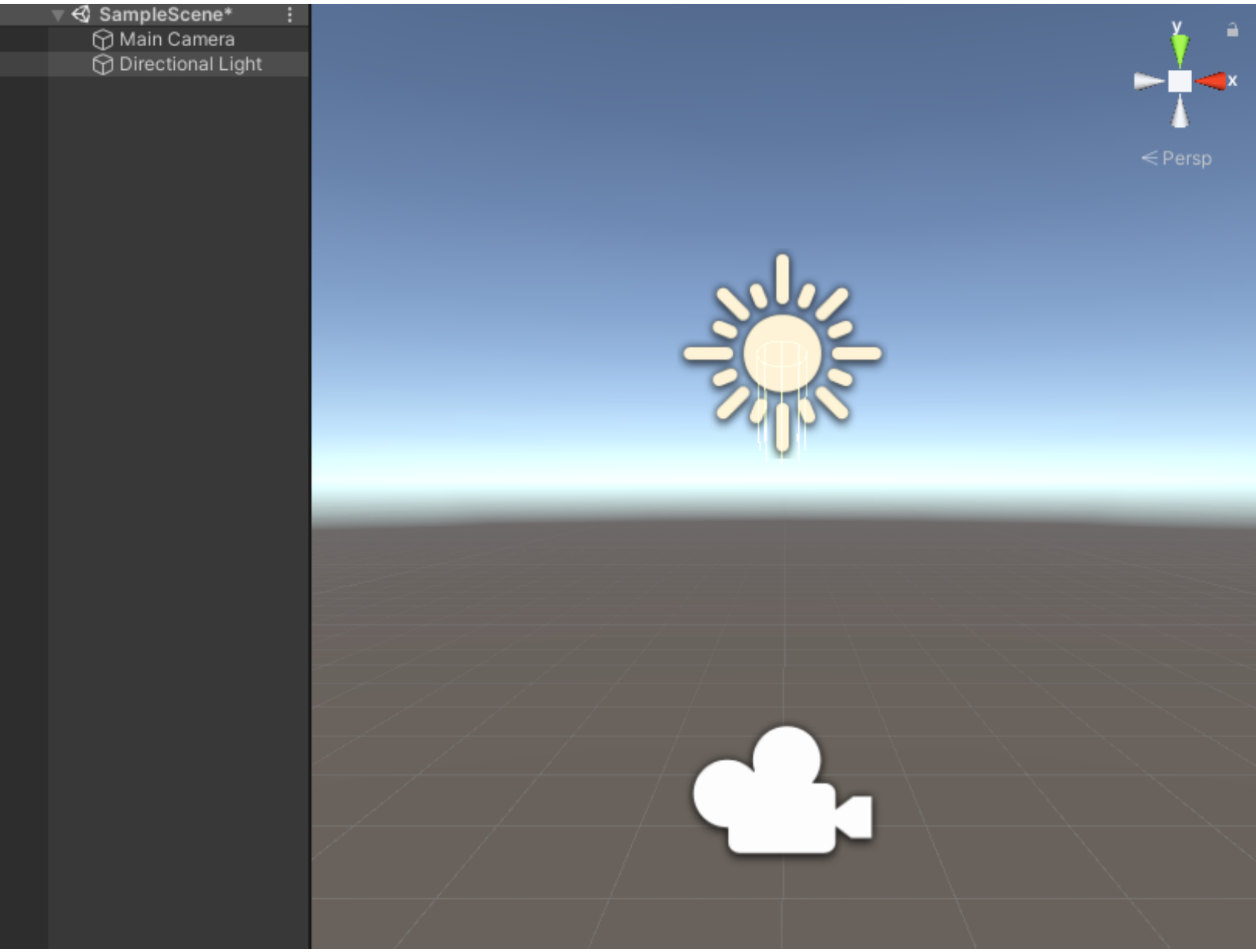
The first stage of development will be creating the graphical user interface for the simulation. This will consist of creating the classes, subclasses and objects as identified in the ‘program structure’ section of the design phase. It will also include generating the minimalist interface as identified in the ‘usability features’ section of the design phase.



Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Above is the initial interface provided by Unity 3D. The environment is empty except for a Camera (called Main Camera) and a Light (called Directional Light).

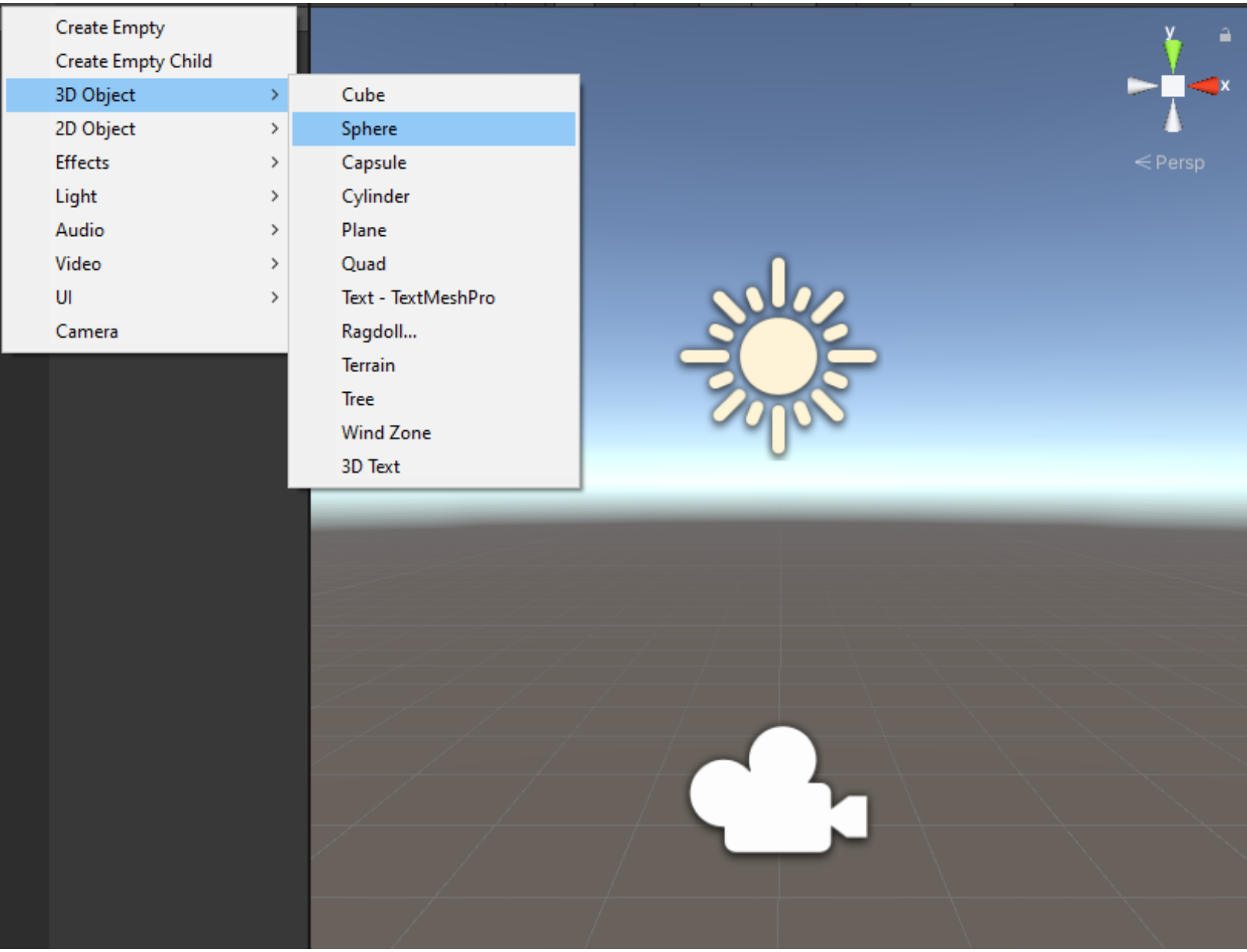
It should be noted that while I am working in Unity 3D, referring back to the diagram identified in the ‘usability features’ section of the design phase, the simulation will actually possess a 2D appearance. This is because, as explained in the ‘applicable computational methods’ section of the analysis, a 2D elevation enables us to see the full oscillation of the pendulum. Therefore, in order to create the desired 2D appearance, subsequent objects should all be created both: parallel to and a fixed distance from the Main Camera. Furthermore, ths Directional Light should also be both: parallel to and a fixed distance from the Main Camera in order to create a coherent source of light. I achieve this by positioning the Light a fixed distance behind the Main Camera and then adjusting the direction of the Light so that it is parallel to the direction of the Camera, as demonstrated below:



Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

A coherent source of light is preferred as it provides the simulation with a cleaner and a more professional appearance, as stated in the ‘usability features’ section of the design phase.

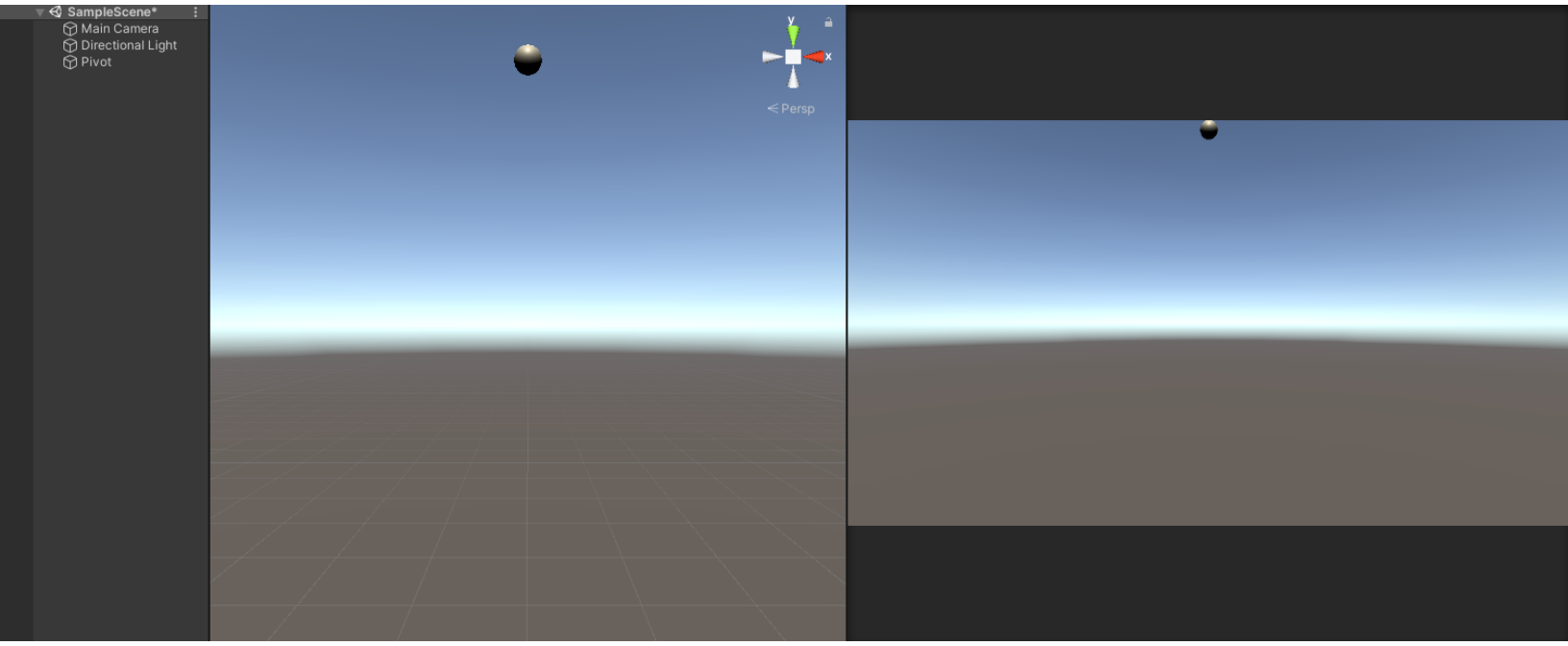
The next task is to instantiate the object ‘Pivot’ of the inbuilt Unity class, ‘Sphere’. This is achieved simply as follows:



Linking back to the diagram identified in the ‘usability features’ section of the design phase, it follows that this pivot should be both: at the centre and at the top of the users screen. This can be achieved by changing the position of the Pivot, as follows:

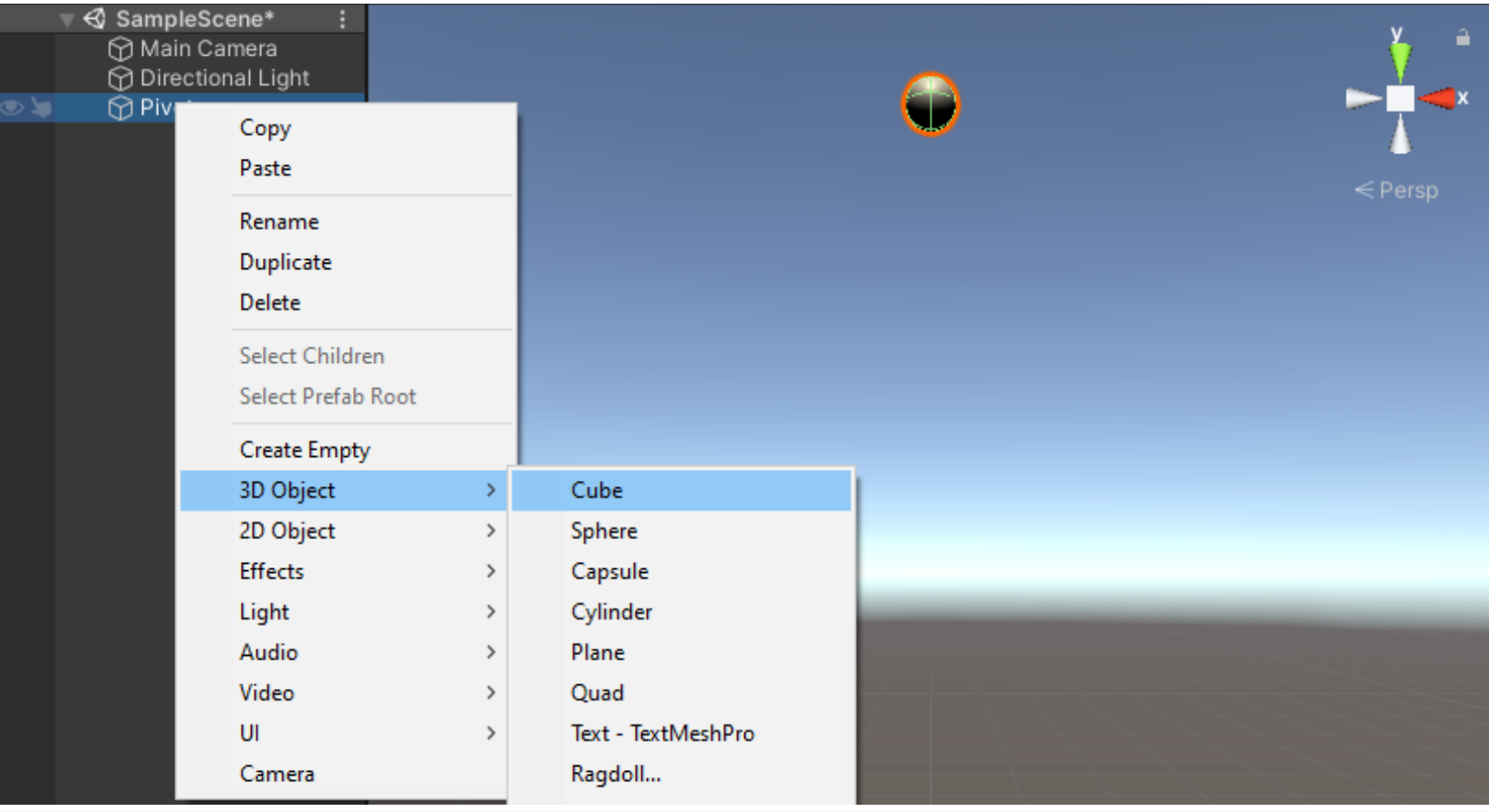
Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

This results in the following scene being displayed:



It should be noted that the panel on the right displays what is presented to the user when the program is run.

Next, the rod that is suspended from the pivot must be created. This can be achieved by instantiating the object ‘Rod’ of the inbuilt Unity class, ‘Cube’. The Rod may be created using a method similar to the method used to create the Pivot, as demonstrated below:

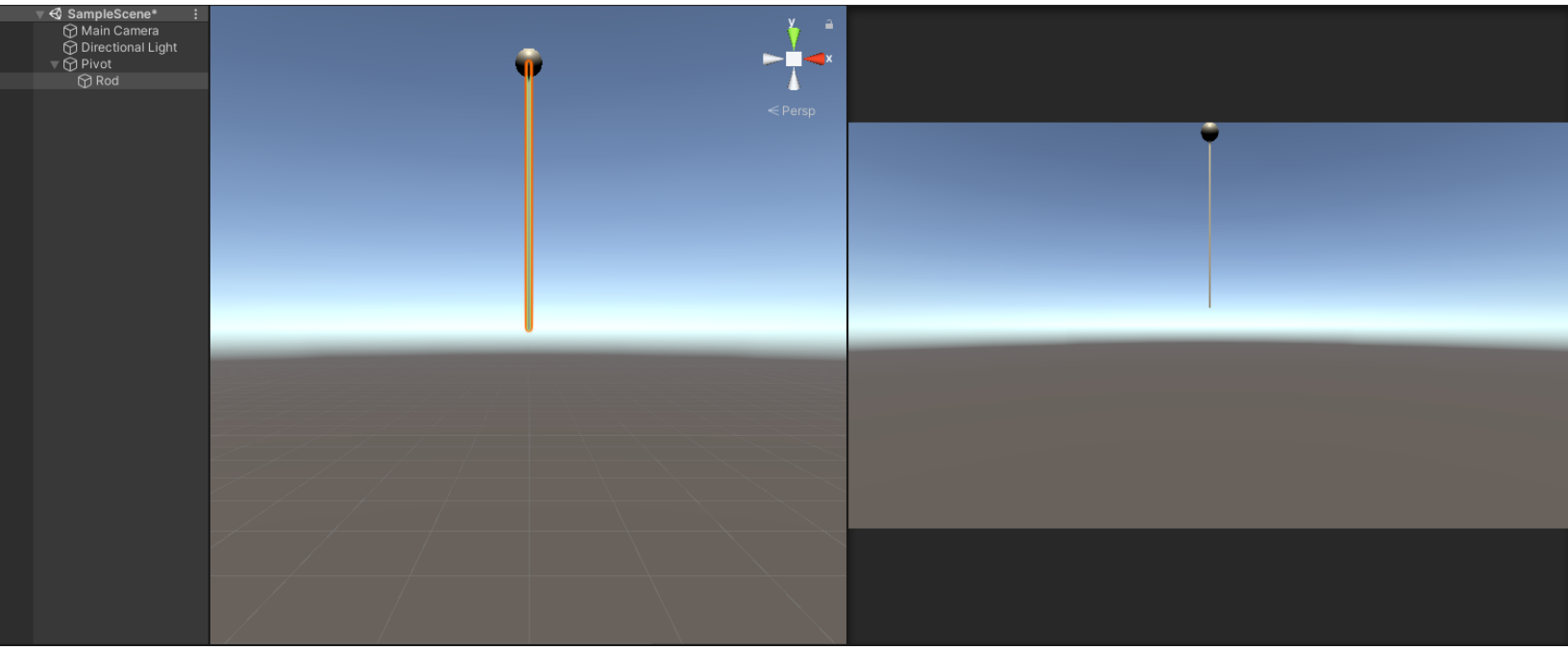


Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

It should be noted that the class ‘Cube’, from which the object Rod is instantiated from, is a subclass of the class ‘Sphere’. It follows that, as development progresses, the object Rod will inherit all attributes and methods of the object Pivot. I arranged the objects like this because it will mean that the subsequent scripts will only need to be assigned directly to object Pivot. This is because the script will be indirectly inherited by all child objects (such as Rod, Mass). This is an ideal solution as directly assigning the script to only one object makes the script easier to manage and therefore test.

The dimensions of the object Rod must be adjusted so that the cuboid is characteristic of a thin rod. I have adjusted the dimensions of the cuboid by changing its scale, as follows:

It should be noted that the position of the Rod is relative to the centre of the Pivot. Therefore, the above position translates to the Rod being 0.5 units (which is equal to the radius of the Pivot) below the centre of the Pivot. After altering the position and scale of the object Rod, the appearance of the simulation is now as follows:



Finally, to complete the pendulum, the mass that is suspended from the rod must be created. This is achieved by instantiating the object ‘Mass’ (again) of the inbuilt Unity class, ‘Sphere’.

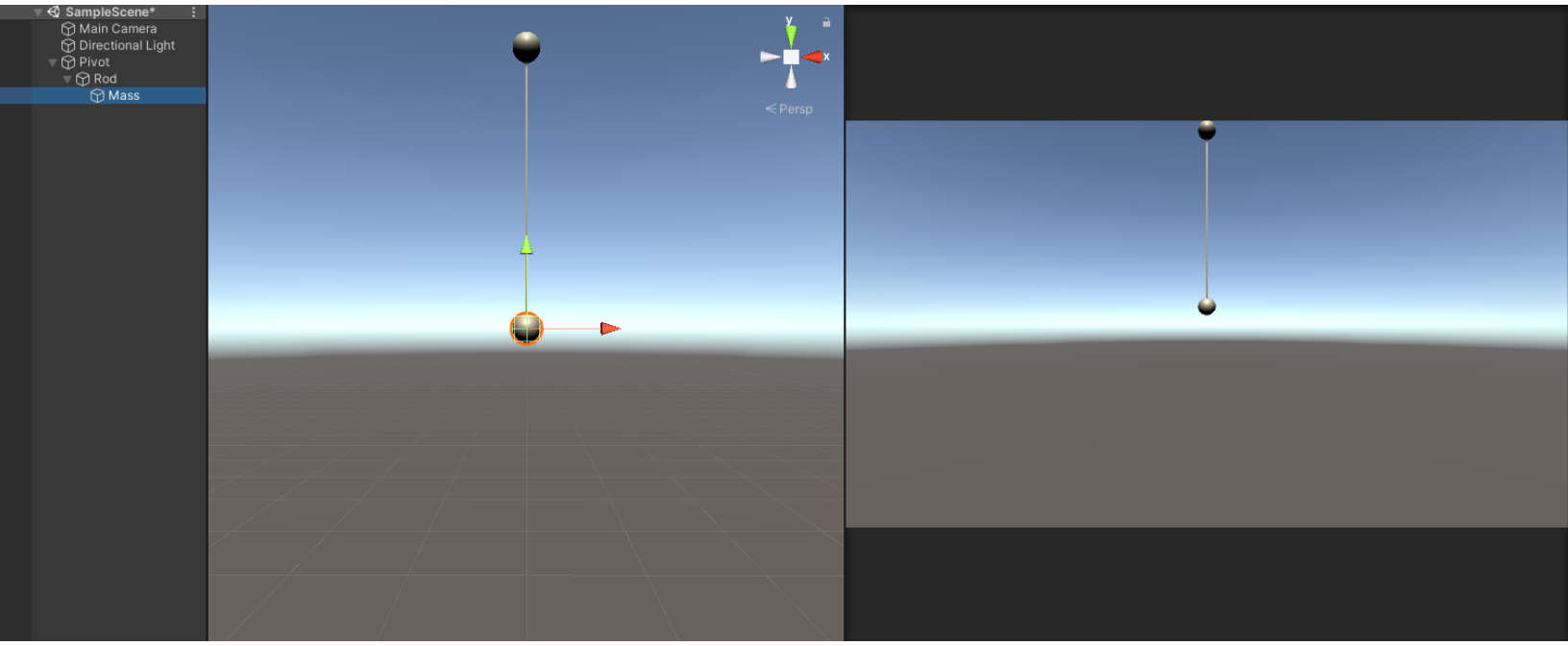
Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

This class will be a subclass of the class ‘Rod’ and will therefore also be a subclass of the class ‘Pivot’. The object is created using the same method used to create the object Pivot.

The position of the object Mass will have to be adjusted so that it is at the end of the Rod. As the Mass is a child object of the class Rod, the position of the Mass is relative to the centre of the Rod. Therefore, in order to position the Mass at the end of the Rod, the position must be adjusted as follows:

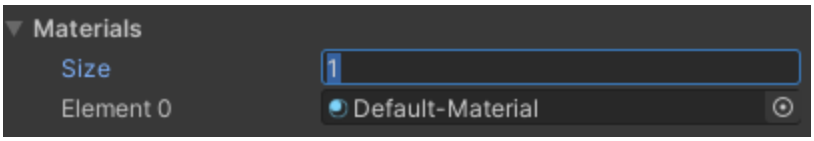
It should be noted that the scale of the Mass here (0.1, 10, 0.1) is used to reverse the scale (10, 0.1, 10) inherited by the Rod. Multiplying these two scales together gives the Mass a final scale of (1, 1, 1), in turn giving it the shape of a perfect sphere.

After altering the position and scale of the object Mass, the appearance of the simulation is now as follows:

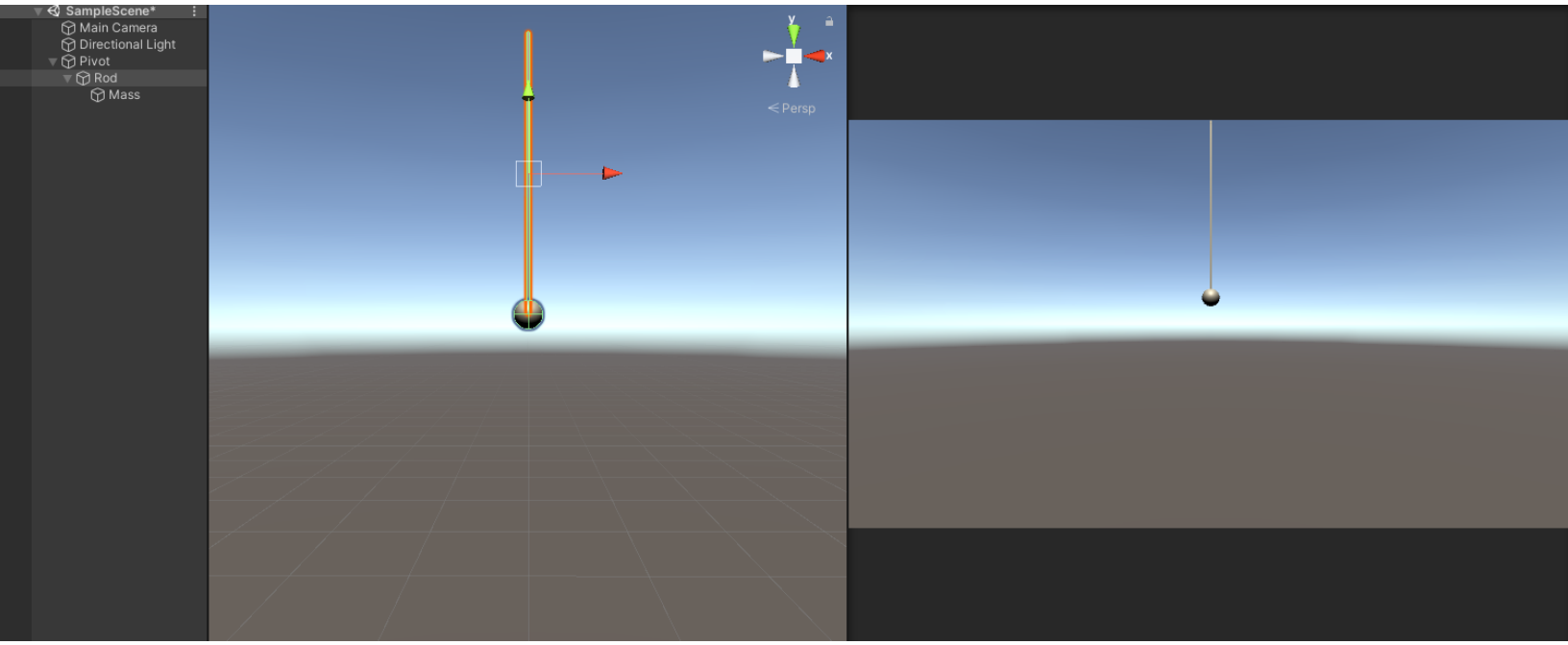


Upon review, I have decided that the pivot object here looks out of place and actually obtrudes the design. However, as mentioned in the ‘program structure’ section of the design phase, the pivot object does not necessarily have to be physically visible to the user in order for it to act as a ‘logical pivot’. This result can be achieved by reducing the size of the pivot to 0. This ensures that the pivot continues to function in the context of the program while being physically invisible

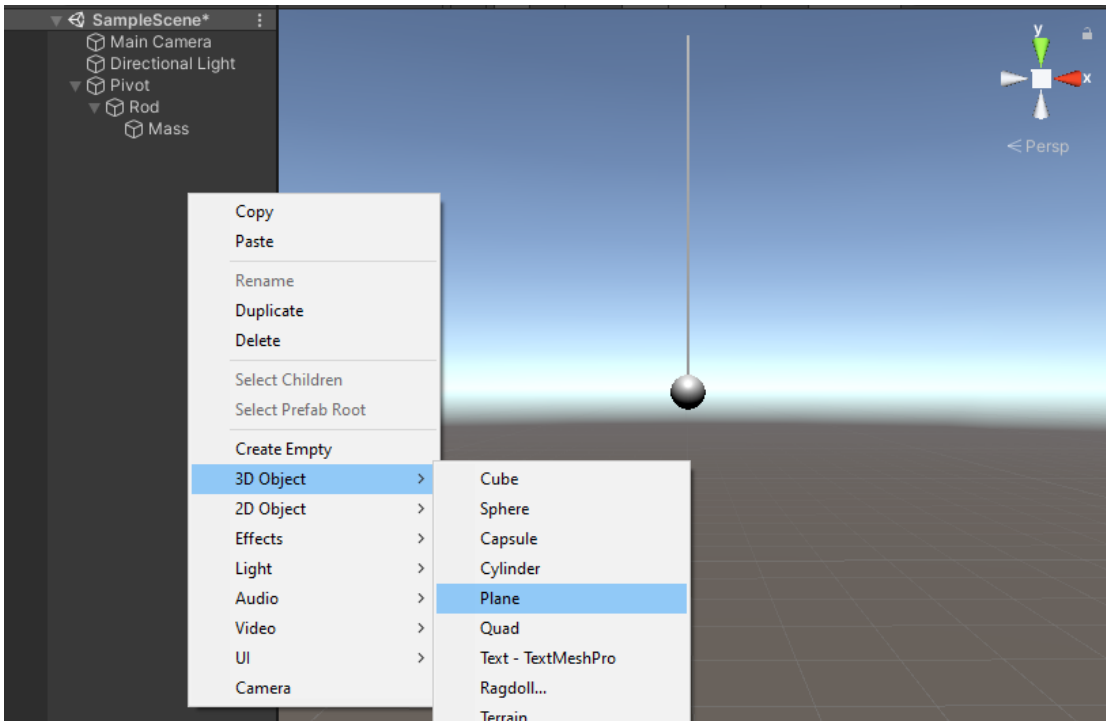
to the user. The size of the pivot can be set to 0 by simply changing the value of Size to 0, as illustrated below:



Upon changing the size of the pivot to 0 and shifting the object Rod up by 0.5 units, to account for the absence of the pivot, the simulation now appears as follows:

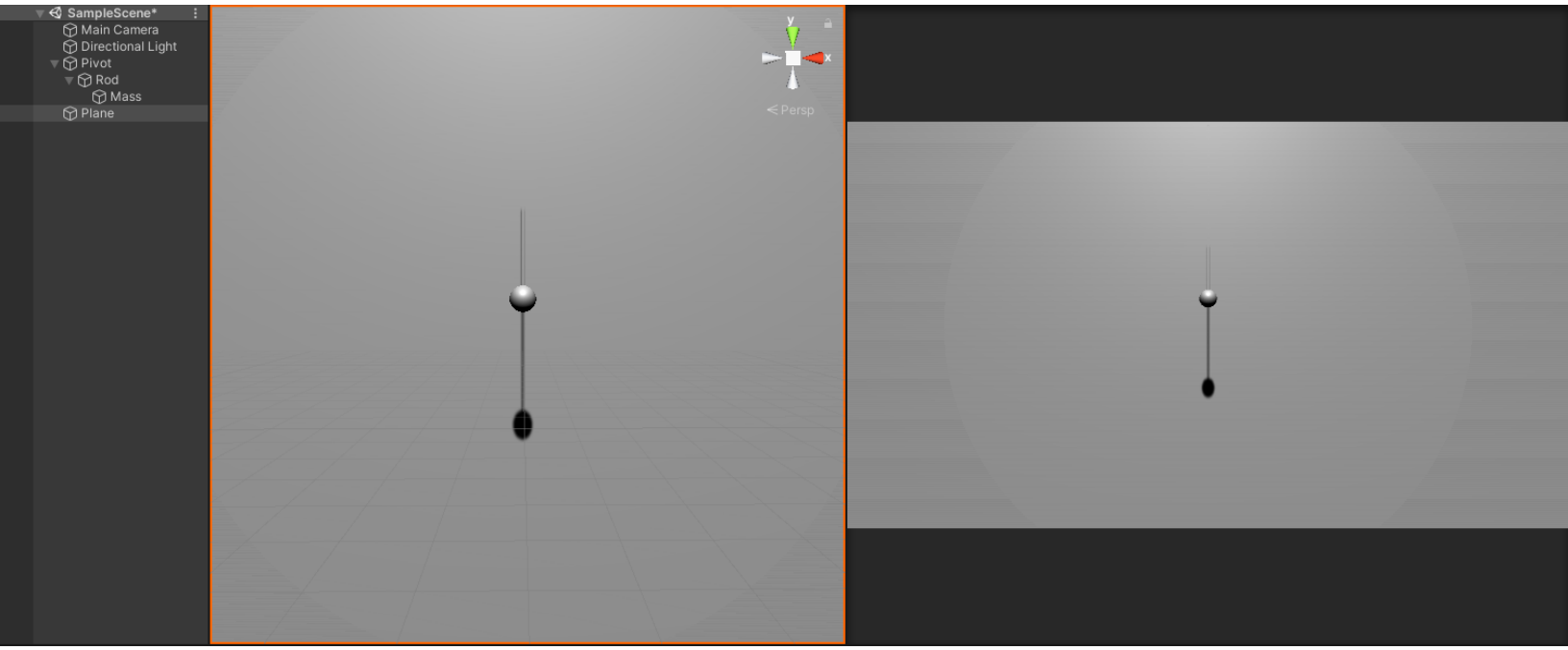


Next, the background for the simulation will need to be created. As the background is essentially a 2D object in the context of the simulation it can be modelled as a plane and may therefore be created as follows:

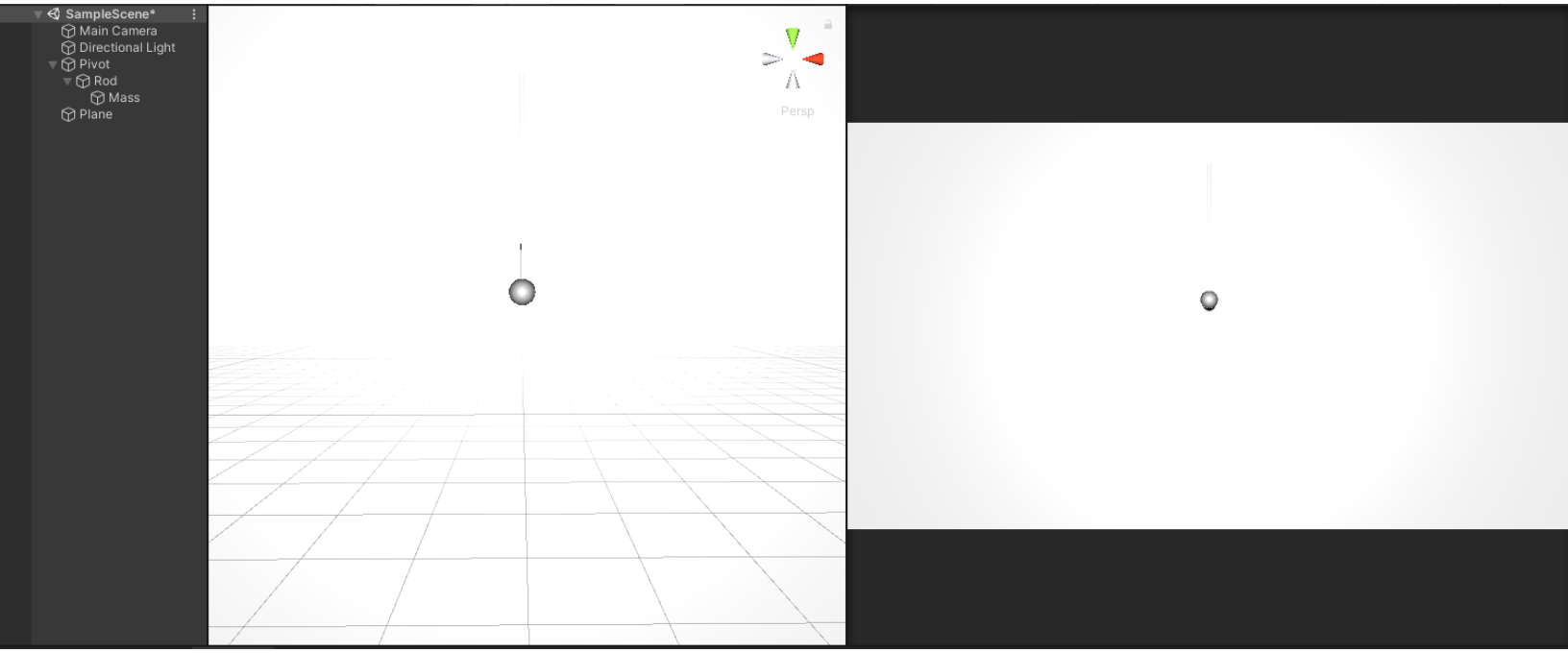


Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

As explained in the ‘usability features’ section of the design phase, I will make the background white so that the simulation is as pleasant to use as possible. Upon changing the colour of the plane to white and changing the size of the plane, such that it fills the users field of view, the appearance is now as follows:



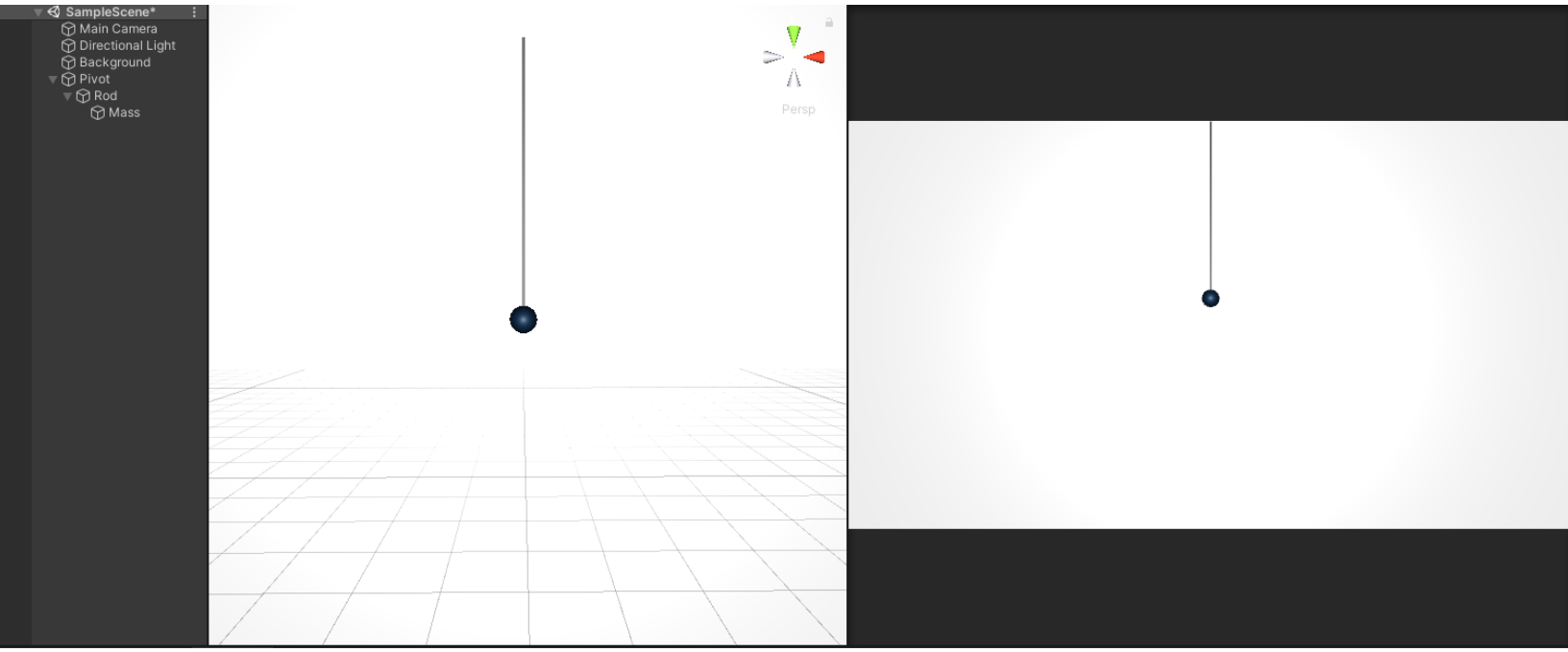
This is clearly not the desired result. Despite setting the colour of the plane to white, the outputted result is more of a greyish colour. To fix this issue, I decided to change the light source such that it is overall brighter than before, as doing so would allow the white colour to be more visible. To do this, I increased the intensity of the light source from 0.8 to 1. The result of doing this is as follows:



Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

The background now appears white, as required.

In order to further improve the usability of the simulation, the colours of the rod and the mass must be changed. This is because the colours of the rod and the mass do not currently create contrast with the background and are therefore overall difficult to distinguish. Therefore, in order to create contrast and hence increased visibility, I have set the colours of the rod and the mass to grey and Oxford Blue respectively. I have also removed shadows from the simulation, as I feel like these would be a distraction to the user. Upon changing these features, the simulation is now viewed as follows:



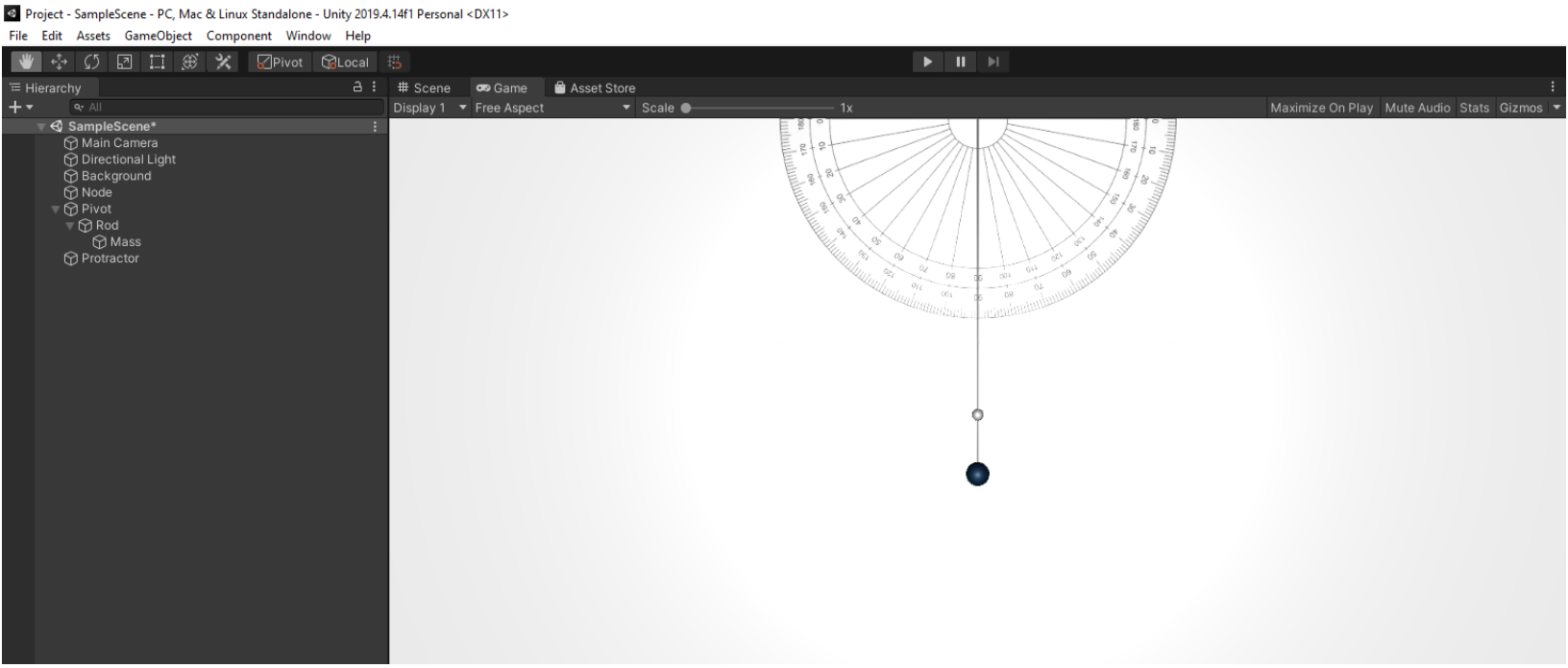
Next the protractor, used to measure the angle at which the pendulum is released from, will need to be created. As explained in the '[program structure]' section of the design phase, the protractor object will be of the class Plane and can therefore be created using the same method used to create the background. However, in order for the object to be of any use, it will need to have an image of a protractor embedded within its plane. This can be achieved by simply dragging an image of a protractor onto the Plane, as illustrated in the following video:

https://drive.google.com/file/d/1rLY_3dY0-R8R2X7gwwVoDv2cTHoT6zJF/view?usp=sharing

Finally the node, primarily used to make the subsequent calculations of the amplitude of the pendulum easier to develop, must be created. This is achieved very simply by instantiating the object Node of the class Sphere. I will make this sphere smaller than the sphere used for the pivot.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

The final appearance of the simulation is therefore as follows:



Implementation of Prototype 1

I am taking an iterative approach for this project, as per the specification of the exam board. This will involve producing successive prototypes of the simulation until a final version, that meets all of the success criteria, is produced and approved by the end user. In the context of my project this will mean that, following the initial design phase and the creation of the user interface, I will be making increasingly refined prototypes with reduced functionality. These prototypes will be designed, coded, tested and evaluated with the end user.

I have chosen an iterative approach because producing software using this approach allows for changing requirements to be dealt with. Furthermore, the continuous feedback from the client as a result of this methodology means the solution is likely to have excellent usability.

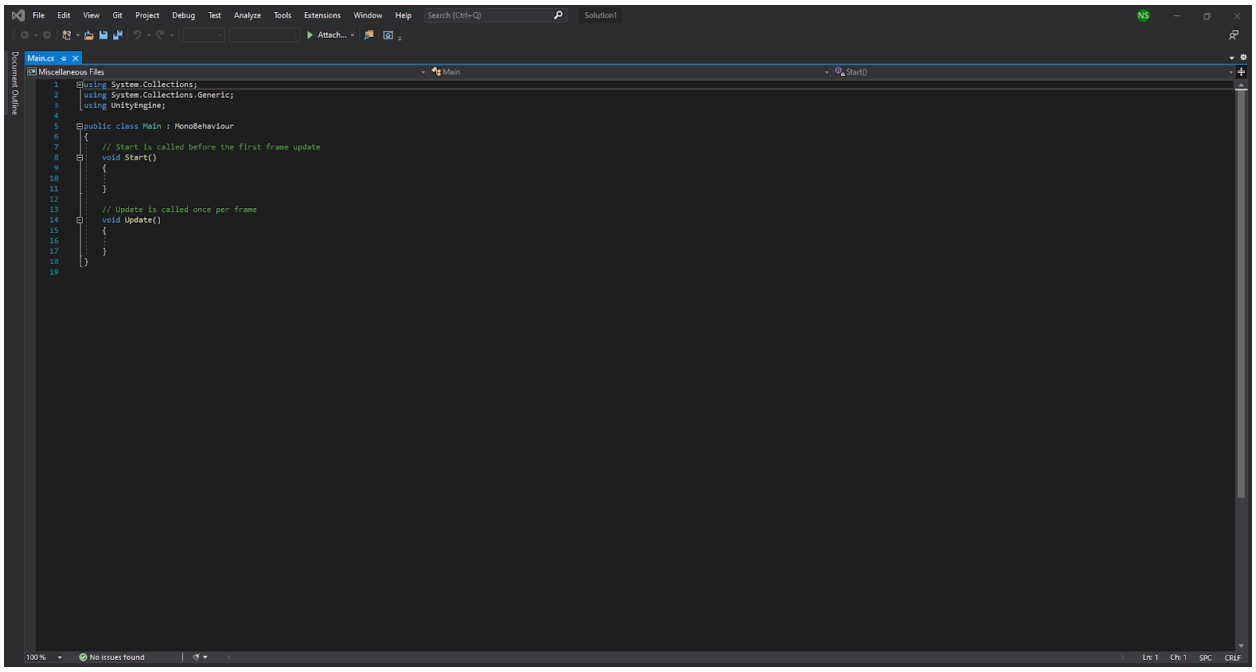
I have decided that the first prototype I will be implementing will aim to meet the following success criteria:

Criterion	Justification	Measurement

The program must demonstrate pendulum motion.	This will determine whether or not the pendulum is demonstrating the relevant equations correctly to the user.	The simulation will display characteristics of a motion that is widely accepted and defined as pendulum motion.
The controls must be intuitive and comfortable to use.	This will make the program efficient to use and easy to learn for new users.	The user should be able to use the simulation without any difficulty and without the need for any prompts outside of the program.
The program must have a practical use i.e. the ability to be used as a tool.	This will fulfil the primary purpose of the program that is to complement the A Level Physics course.	The user should be able to obtain correct calculations and answers using the results obtained from this simulation, linking back to the first success criteria.
The program must be able to deal with all valid, invalid and extreme inputs from the user.	This will make the program more robust and less prone to crashing. This provides the user with a more reliable experience.	The user should not be able to crash the program as a direct result of an invalid or extreme input.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

As Unity scripts are written in C#, the integrated development environment (IDE) I will be using for the remainder of this project will be Microsoft Visual Studio. The window for Visual Studio is below.



Like most IDEs, Visual Studio consists of a source code editor, build automation tools and a debugger and these features are shown in the above window. However, in order to save space on the page, subsequent screenshots of the source code will not include the entire window and will instead only include the relevant portion of code.

The first script I will be creating will be named Main.cs. This is because it will be the 'pendulum motion script' described in the program structure section of the design phase and will therefore be the main script of the program, allowing the objects created earlier to undergo pendulum motion.

Defining variables

The first step in creating this initial prototype is defining the variables highlighted in the key variables section of the design phase. The variables outlined here may be defined as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

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

public class Main : MonoBehaviour
{
    public Transform pivot;
    public Transform node;
    public Transform mass;

    public float maxDisplacement;
    public float currentDisplacement;
    public float lengthOfRod = 1f; //arbitrary value
    public float currentTime = 0f;
    public float timePeriod = 0f;
    public float gravitationalAcceleration = 9.80665f;
    public float maxAngle = 0f;
    public float currentAngle = 0f;
    public float dampingFactor = 0f;
```

As you can see, the values for pivot, node and mass are all of the class Transform. In Unity, every object in a Scene has a Transform. It's used to store and manipulate the position, rotation and scale of the object. Therefore, by assigning the values of pivot, mass and node to their corresponding game objects Pivot, Mass and Node (the procedure for which I will show later on in development, when required), the use of pivot, node and mass in any subsequent calculations will result in the appropriate transformation of its corresponding game object. In other words, the use of objects here will allow the abstract calculations that determine pendulum motion to be presented to the user in a visual way that can be observed and understood easily.

Note that the variable lengthOfRod is commented as arbitrary as this value will later be recalculated in the function FindLength(), according to the Rod's real length .

Algorithms for motion

I will now implement the algorithms in the design phase, testing each algorithm as it is implemented. I will start with the implementation of:

FindLength()

Referring to the pseudocode in the design phase, the algorithm may be implemented in Unity as follows:

```
float FindLength(Transform mass, Transform pivot)
{
    float length = Mathf.Sqrt(((mass.transform.position.x - pivot.transform.position.x) ^ 2)
        + ((mass.transform.position.y - pivot.transform.position.y) ^ 2)
        + ((mass.transform.position.z - pivot.transform.position.z) ^2));
    return length;
```

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

However, running the above code produced the following error message(s):

```
float FindLength(Transform mass, Transform pivot)
{
    float length = Mathf.Sqrt(((mass.transform.position.x - pivot.transform.position.x) * (mass.transform.position.x - pivot.transform.position.x))
        + ((mass.transform.position.y - pivot.transform.position.y) * (mass.transform.position.y - pivot.transform.position.y))
        + ((mass.transform.position.z - pivot.transform.position.z) * (mass.transform.position.z - pivot.transform.position.z)));
    return length;
```

From this error message, it is clear that the solution to the problem would be to replace the instances of ‘x^2’ with ‘x*x’. I apply this solution below:

```
Project   Console
Clear | Collapse | Clear on Play | Clear on Build | Error Pause | Editor
[20:35:10] Assets\Main.cs(22,35): error CS0019: Operator '^' cannot be applied to operands of type 'float' and 'int'
[20:35:10] Assets\Main.cs(23,16): error CS0019: Operator '^' cannot be applied to operands of type 'float' and 'int'
[20:35:10] Assets\Main.cs(24,16): error CS0019: Operator '^' cannot be applied to operands of type 'float' and 'int'
```

Now, no error messages are produced when the code is run. However, while the syntax of the code is correct, whether the code produces the correct/ expected output still needs to be determined/ tested.

Testing FindLength()

In order to derive the actual length of the rod, we must remind ourselves of the positions of the positions of each component of the pendulum. The position and scale of the pivot is given by:

Transform			
Position	X 0	Y 12.5	Z 20
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

The position and scale of the rod is relative to position and scale of the pivot. The position of the rod refers to the position of the centre of the rod. It is given by:

Transform			
Position	X 0	Y -7.5	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 0.1	Y 15	Z 0.1

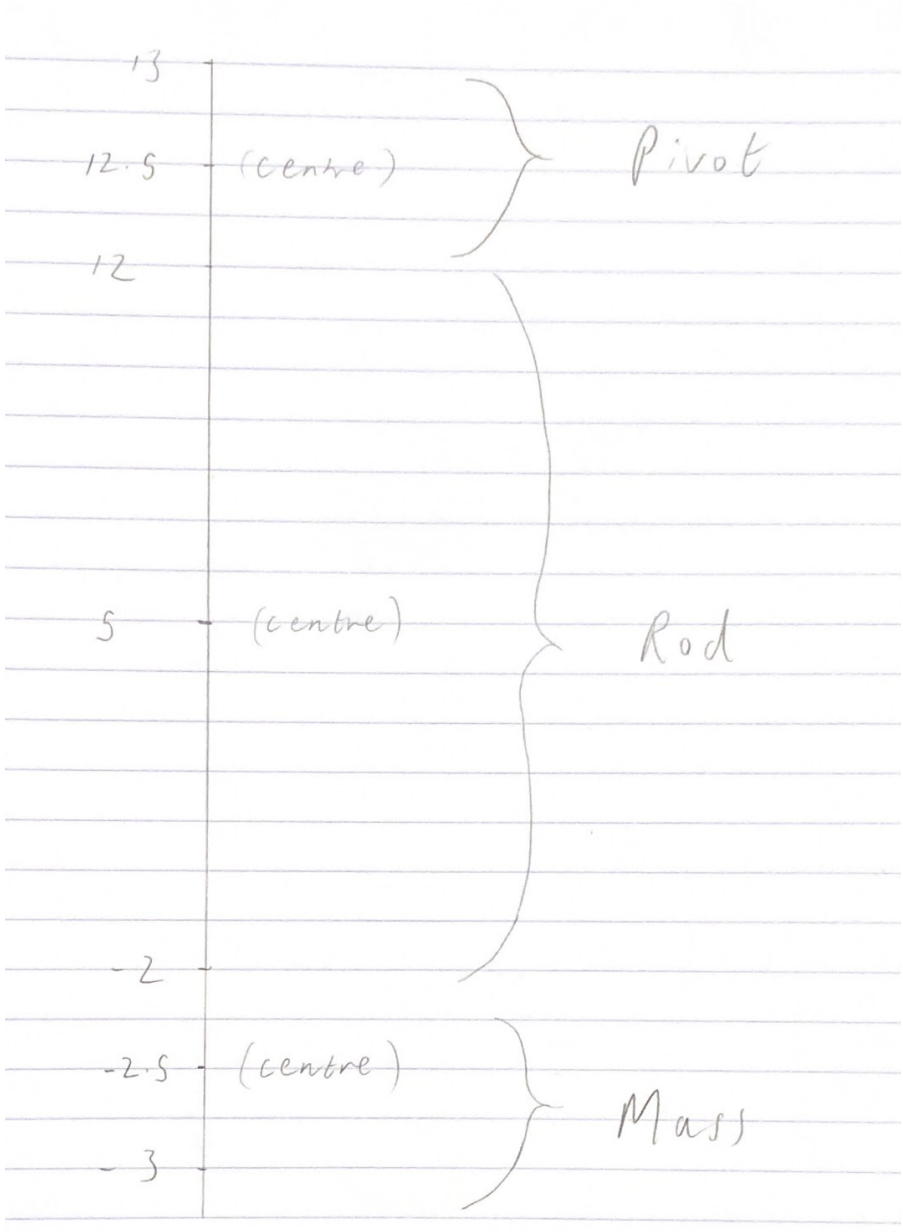
Finally, the position and scale of the mass is relative to the position and scale of the rod. This is given by:

Transform			
Position	X 0	Y -0.5	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 10	Y 0.066666	Z 10

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Using this information, an abstract diagram may be produced to provide a visual representation of the positions and scales of all the objects.

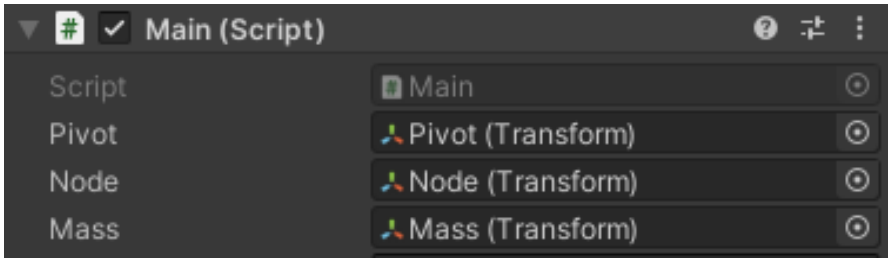
This is an example of how visualisation and abstraction can be used to simplify a problem, the problem in this case being to find the actual length of the rod. The final diagram is presented below:



From this diagram, it is clear that the expected length of the rod is 12.5 - - 2.5 = 12 + 2.5 = 15. The procedure must now be tested by checking whether the true length of the rod is equal to its expected length, or not.

However, before the procedure can be tested the values of pivot, mass and node must be assigned to their corresponding game objects Pivot, Mass and Node. This can be achieved by first attaching the current script, Main.cs, to the Pivot game object. As demonstrated earlier, the game objects Mass and Node are children of the Pivot game object. Therefore, by directly attaching the script to the Pivot game object, the children objects (Mass and Node) can indirectly have the script Main.cs attached to them, via inheritance. Therefore, the script is indirectly attached to all game objects related to the pendulum. This allows the effects of the script to act on the relevant game objects and consequently allows the effects of the script to be seen. This in turn allows me, as a developer, to obtain the relevant test results.

The script can be attached to the game object by simply dragging the script Main.cs to the game object Pivot, as demonstrated in this video. The values of pivot, mass and node can be assigned to their corresponding game objects Pivot, Mass and Node by simply selecting the relevant game object for each variable pivot, mass and node. The results of these actions are illustrated below:



By using the inbuilt Unity function Debug.Log(), the result of the function FindLength() can be returned. Comparing this value with the actual value of the length of the rod will help determine whether this function is accurate, or not. The function Debug.Log() can be implemented as follows:

```
// Start is called before the first frame update
void Start()
{
    lengthOfRod = FindLength(mass, pivot);
    Debug.Log(lengthOfRod);
}
```

The test can therefore be conducted as follows:

https://drive.google.com/file/d/1JmXeel9LIvM7og79duddtLW2kfuIoWdt/view?usp=sharing

As seen in the above video, the observed value is equal to the expected value. The function FindLength() can therefore be deemed as accurate/ functional. As the function has now been tested, the Debug.Log() function is now no longer needed, and can therefore be removed from the source code. The test and its results can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
FindLength()	Run the simulation and compare the calculated value with the real value.	Debug.Log()	This will test that the calculated value is accurate.	The calculated value will be equal to the real value.	Pass

This concludes the implementation of the FindLength() function. The next algorithm to implement is:

CalculateConstants()

Referring to the pseudocode in the design phase, the algorithm may be implemented in Unity as follows:

```
void CalculateConstants()
{
    lengthOfRod = FindLength(mass, pivot);
    maxDisplacement = maxAngle * lengthOfRod;
    timePeriod = 2 * Mathf.PI * Mathf.Sqrt(lengthOfRod / gravitationalAcceleration);
}
```

Running the program with this portion of code produces no errors, indicating that this procedure is syntactically correct. Unlike FindLength(), CalculateConstants() cannot be tested in isolation using the Debug.Log() function as there is no expected value to compare the result of the procedure with. Rather, there is simply an expected motion to compare the observed motion of the simulation with. Therefore, this procedure can only be implicitly tested once the motion of the simulation has been established i.e. after the Motion() procedure has been implemented. Thus, the implementation of this procedure comes to a conclusion.

Initialisation()

The next algorithm to implement is the procedure Initialisation(). This is a relatively simple algorithm with the purpose of initialising variables, most notably the rotation of the pivot. The algorithm can be written in Unity as follows:

```
// Start is called before the first frame update
void Start()
{
    maxAngle = 0; //initial angle
    pivot.transform.localEulerAngles = new Vector3(0, 0, 0);
    currentTime = 0;
    CalculateConstants();
}
```

In Unity, as described in the comments, the procedure Start() is called on the frame when the script is enabled just before any of the Update() methods are called the first time. Therefore, as the purpose of the procedure is to initialise certain variables at the beginning of the program, I have decided to place the code for procedure Initialisation() within the procedure Start(), as doing so will produce the required result.

Testing Initialisation()

As described in the design phase, the procedure can be tested by simply running the simulation and checking whether the simulation starts at equilibrium, or not. The test can therefore be conducted as follows:

https://drive.google.com/file/d/1r_l_LKb2kY97WWR3X0LRecF61M7GLccY/view?usp=sharing

As seen in the above video, the simulation starts at equilibrium. The algorithm can therefore be deemed as functional. The test and its results can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
Initialisation()	Run the simulation.	N/A	This will test that the simulation starts at equilibrium.	The simulation will start at equilibrium when run. That is, the entire pendulum is vertical and stationary.	Pass

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

This concludes the implementation of the procedure Initialisation().

SetNodePos()

The next algorithm I will implement is the procedure SetNodePos(), which sets the position of the node to the position of the mouse. As mentioned in the design phase, due to the need for mouse position, I will need to make use of inbuilt Unity functions in order to implement this procedure.
After some research I discovered that, in Unity, such a procedure is typically written in the following format:

```
Vector3 temp = Input.mousePosition;

temp.z = 10f; //set this to be the distance you want the object to be

placed in front of the camera.

this.transform.position = Camera.main.ScreenToWorldPoint(temp);
```

The inbuilt Unity function Camera.main.ScreenToWorld featured in this template [transforms a point from screen space](#) (mouse position) [into world space](#) (node position) and is therefore the main component of the procedure SetNodePos().

By adjusting the template in accordance to the context of the simulation, the procedure SetNode() can be written as follows:

```
Vector3 mouse = (Input.mousePosition.x, Input.mousePosition.y); //the

z-coordinate does not need to be defined, as it is defined separately

in the subsequent line.

mouse.z = 20f; //this is set to 20 as this is the distance the pivot

(and therefore the mass) is from the camera.

node.transform.position = Camera.main.ScreenToWorldPoint(mouse);
```

Therefore, in Visual Studio, the procedure may be written as follows:

```
void SetNodePos()
{
    Vector3 mouse = (Input.mousePosition.x, Input.mousePosition.y); //the z-coordinate does not need to be defined, as it is defined separately in the subsequent line.
    mouse.z = 20f; //this is set to 20 as this is the distance the pivot (and therefore the mass) is from the camera.
    node.transform.position = Camera.main.ScreenToWorldPoint(mouse);
}
```

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

However, compiling the program produces the following error message(s):

This error message is produced since type `(float x, float y)` is not necessarily type `Vector3`. Therefore, the compiler is not able to implicitly convert type `(float x, float y)` to type `Vector3`. Therefore, in order to resolve the error, I have to explicitly convert type

```
Project   Console
Clear  Collapse  Clear on Play  Clear on Build  Error Pause  Editor
[23:11:32] Assets\Main.cs(37,25): error CS0029: Cannot implicitly convert type '(float x, float y)' to 'UnityEngine.Vector3'
```

`(float x, float y)` to type `Vector3`. This can be achieved as follows:

```
void SetNodePos()
{
    Vector3 mouse = new Vector3(Input.mousePosition.x, Input.mousePosition.y); //the z-coordinate does not need to be defined, as it is defined separately in the subsequent line.
    mouse.z = 20f; //this is set to 20 as this is the distance the pivot (and therefore the mass) is from the camera.
    node.transform.position = Camera.main.ScreenToWorldPoint(mouse);
}
```

The program now compiles with no errors. Therefore, the procedure SetNodePos() must now be tested. Before it can be tested the procedure must first be called somewhere within the program. By referring to the design phase, the procedure should only be called when the mouse is clicked. However, this has highlighted a flaw in the logic of the procedure to me. I have come to the realisation that, similar to the way the procedure SetNodePos() should only be called when the mouse is clicked, the node should only be visible when the procedure is called and therefore when the mouse is clicked. In other words, the node should only be visible when the mouse is clicked and should be invisible for the rest of the program. However, in its current state, the procedure does not achieve this as the node is visible throughout the entire duration of the program, regardless of whether the mouse has been clicked or not.

In order to fix this the node should be set to ‘invisible’. This state should be kept constant, unless the mouse has been clicked in which case the node should be set to ‘visible’ for the duration of the mouse click. After this duration has expired, the node should revert back to its ‘invisible’ state.

After some research, I have discovered that this result can be achieved using the inbuilt Unity function GameObject.SetActive(). This function activates/deactivates a game object, depending on the given true or false value. The game object node can therefore be activated/deactivated as follows:

```
node.gameObject.SetActive(true/false)
```

As the node should only be visible when the procedure is called, the procedure SetNodePos() can be rewritten as follows:

```
void SetNodePos()
{
    node.gameObject.SetActive(true);
    Vector3 mouse = new Vector3(Input.mousePosition.x, Input.mousePosition.y); //the z-coordinate does not need to be defined, as it is defined separately in the subsequent line.
    mouse.z = 20f; //this is set to 20 as this is the distance the pivot (and therefore the mass) is from the camera.
    node.transform.position = Camera.main.ScreenToWorldPoint(mouse);
}
```

The node should be invisible when the procedure is not being called/ for the rest of the program. As shown earlier, Update() is called once per frame, every frame. Therefore, by deactivating the game object node in Update(), the node will be invisible for every frame of the program when the procedure is not being called.

As mentioned earlier, the procedure should only be called when the mouse is clicked. Therefore, the procedure SetNodePos() should only be called within the procedure IfClicked(). However, IfClicked() has not been implemented yet. For this reason, until IfClicked() has been implemented, I will temporarily call SetNodePos() from within Update(), in order to allow testing to proceed. Update() may therefore be written as follows:

```
// Update is called once per frame
void Update()
{
    node.gameObject.SetActive(false);
    SetNodePos();
}
```

Testing SetNodePos()

As described in the design phase, the procedure can be tested by moving the mouse to various positions on the screen and checking whether the node consistently follows the mouse to its various positions, or not. The test can therefore be conducted as follows:

https://drive.google.com/file/d/1NHrOuTCuHLLXFZVy9_-Mrup-G6JtYil4/view?usp=sharing

As seen in the above video, the node consistently follows the mouse to its various positions. The procedure SetNodePos() can therefore be deemed as functional. As the procedure has now been tested, the call in Update() is now no longer needed, and can therefore be removed from the source code. The test and its results can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
SetNodePos()	Move the mouse to various positions on the screen.	Move the mouse to 5 randomly generated positions on the screen.	This will test that the node consistently moves to the mouse position when the mouse is both clicked and	The node consistently follows the mouse to its various positions.	Pass

			dragged.		

This concludes the implementation of the procedure SetNodePos(). The next algorithm to implement is:

Timer()

This is a relatively simple algorithm designed to act as a timer by incrementing the current time (initialised at 0) every frame. This can be achieved by using the Unity feature Time.deltaTime, which is the interval in seconds from the last frame to the current one. The procedure can therefore be written in Unity as follows:

```
void Timer()
{
    currentTime += Time.deltaTime;
}
```

By calling Timer() within Update(), the interval in seconds from the last frame can continuously be added to the current time, every frame. However, this has highlighted a flaw in the program to me. The current time (t) is needed for the future calculation of current displacement (x), which is given by $x = A\cos(\omega t) = A\cos((2\pi / T) * t)$. From this equation we can see that, in order for the calculation to work, the current time (t) must be less than the time period (T). However, in its current state, the procedure Timer() continuously adds the interval in seconds from the last frame to the current time. This will eventually lead the current time (t) to be greater than the time period (T), hence invalidating the calculation of current displacement. Therefore, in order to fix this, (as the current time (t) cannot be greater than the time period (T)) if the current time (t) is greater than the time period (T) then the current time (t) needs to be reset to 0. This could be achieved in many ways. The first solution I designed was as follows:

```
if (currentTime > timePeriod)
    currentTime = currentTime % timePeriod; //would cause error if 0
```

However, I quickly realised that the calculation currentTime MOD timePeriod would cause an error is timePeriod = 0. This is because dividing by 0 is undefined. This solution is therefore invalid. The next solution I designed was as follows:

```
if (currentTime > timePeriod)
    currentTime = 0;
```

Unlike the previous solution, this solution works for all cases. However, this is not the most efficient solution. This is because, after some research, I discovered that: ‘subtracting two is more accurate over time than resetting to zero’. I can therefore rewrite the algorithm as follows:

```
if (currentTime > timePeriod)
    currentTime -= timePeriod;
```

This is now the most efficient version of the algorithm. The procedure Timer() can therefore be rewritten as follows:

```
void Timer()
{
    currentTime += Time.deltaTime;
    if (currentTime > timePeriod)
    {
        currentTime -= timePeriod;
    }
}
```

In order to call this procedure once per frame, as mentioned earlier, the procedure should be called within Update(). Update() can therefore be written as follows:

```
// Update is called once per frame
void Update()
{
    node.gameObject.SetActive(false);
    Timer();
}
```

The procedure Timer() cannot be tested effectively in isolation. Rather, it can only be tested implicitly once the motion of the simulation has been established i.e. after the Motion() procedure has been implemented. Thus, the implementation of the procedure Timer() comes to a conclusion.

IfClicked()

The next algorithm to implement is the procedure IfClicked(). This is one of the most important algorithms in the program, as it is responsible for determining the motion of the pendulum when the mouse is clicked. By referring to the design section, the algorithm can be implemented in Unity as follows:

```
void IfClicked()
{
    SetNodePos();
    maxAngle = Mathf.Atan2((node.transform.position.y - pivot.transform.position.y), (node.transform.position.x - pivot.transform.position.x)); //resultant angle is in radians.
    pivot.transform.localEulerAngles = new Vector3(0, 0, maxAngle * Mathf.Rad2Deg); //Euler angles are in degrees. Therefore, maxAngle must be converted from radians to degrees.
    currentTime = 0;
}
```

However, running the program produces the following result:

https://drive.google.com/file/d/1n-VPwwpDeHhO2LXE81xOPBTkqrTCJghK/view?usp=sharing

As seen in the above video the rotation of the pivot, and hence the rotation of the pendulum, is consistently 90 degrees out of phase with the maximum angle of the pendulum from equilibrium (the position of the node). The pivot therefore needs to be rotated a further 90 degrees

(clockwise) in order for the rotation of the pivot to match the maximum angle of the pendulum from equilibrium. This can be achieved as follows:

```
void IfClicked()
{
    SetNodePos();
    maxAngle = Mathf.Atan2((node.transform.position.y - pivot.transform.position.y), (node.transform.position.x - pivot.transform.position.x)) + (0.5f * Mathf.PI); //resultant angle is in radians.
    pivot.transform.localEulerAngles = new Vector3(0, 0, maxAngle * Mathf.Rad2Deg); //Euler angles are in degrees. Therefore, maxAngle must be converted from radians to degrees.
    currentTime = 0;
}
```

The program now runs as follows:

https://drive.google.com/file/d/1fCu2vxYRljt6WLQxWWdibygSKgJWAcUq/view?usp=sharing

As seen in the above video, the rotation of the pivot is now equal to the maximum angle of the pendulum from equilibrium, as required. As its name suggests, the procedure IfClicked() must only be called when the mouse is clicked. As the mouse could be clicked at any time during the program's execution, the procedure should be called within Update() alongside an if statement determining whether the mouse has been clicked, or not. Update() may therefore be rewritten as follows:

```
// Update is called once per frame
void Update()
{
    node.gameObject.SetActive(false);
    Timer();
    if (Input.GetKey(KeyCode.Mouse0)) //KeyCode.Mouse0 refers to the left/ primary mouse button.
    {
        IfClicked();
    }
```

The program now runs as follows:

https://drive.google.com/file/d/1SwtPJmFiUz-0B3PjppoPRSUwsqLQ6ye8/view?usp=sharing

As seen in the above video the rotation of the pivot is equal to the maximum angle of the pendulum from equilibrium, only when the mouse is clicked. While the procedure appears functional now, it cannot be formally tested yet as the test outlined in the design phase requires the Motion() procedure to have been implemented. Therefore, I will formally test the procedure IfClicked() once the procedure Motion() has been implemented. However, while I cannot currently test it, I can perform the validation check (outlined in the design phase) for this procedure. The validation check consists of repeatedly clicking the left and right mouse buttons in succession. I will perform a left click first followed by a right click second and repeat this process until I am satisfied that only the left clicks are taken as input. The validation check can therefore be carried out as follows:

https://drive.google.com/file/d/1avR8-yQ8ZfKO_iZBJViaPjny5oc8P0v1/view?usp=sharing

From the above video, we can see that the procedure IfClicked() only takes inputs in the specified format of a left click, as required. The procedure has therefore passed the validation check.

Motion()

The next algorithm to implement is the procedure Motion(). This is a key algorithm, as it is responsible for determining the motion of the pendulum when the mouse is not clicked. By referring to the design section, the procedure can be implemented in Unity as follows:

```
void Motion()
{
    CalculateConstants();
    currentDisplacement = maxDisplacement * Mathf.Cos((2 * Mathf.PI / timePeriod) * currentTime);
    currentAngle = currentDisplacement / lengthOfRod; //this angle is in radians.
    pivot.transform.localEulerAngles = new Vector3(0, 0, currentAngle * Mathf.Rad2Deg); //Euler angles are in degrees. Therefore, currentAngle must be converted from radians to degrees.
}
```

As Motion() is responsible for determining the motion of the pendulum when the mouse is not clicked, it follows that the procedure should only be called when the mouse is not clicked.
As seen earlier, the procedure IfClicked() is called within an if statement, that is, if the mouse is clicked then the procedure IfClicked() is called. Therefore, by turning this if statement into an if...else statement, the procedure Motion() can be called within the else part of this if...else statement, and can thus be called when the mouse is not clicked, as required. Update() can therefore be rewritten as follows:

```
// Update is called once per frame
void Update()
{
    node.gameObject.SetActive(false);
    Timer();
    if (Input.GetKey(KeyCode.Mouse0)) //KeyCode.Mouse0 refers to the left/ primary mouse button.
    {
        IfClicked();
    }
    else
    {
        Motion();
    }
}
```

The program compiles with no errors. As Motion() has now been implemented, I am finally able to formally test the procedure IfClicked().

Testing IfClicked()

As outlined in the design phase the procedure can be tested by clicking the mouse at specified angles along the protractor, letting the pendulum swing for 3 cycles for each angle, and checking whether the pendulum continues to reach the same position it was released from, or not. I will change the range of angles from which the pendulum is released at steps of 60°, starting from 0° and ending on 360°, as originally proposed in the design phase, to steps of 30°, starting from 0°

and ending on 180°. This is because I cannot currently measure angles in the range 180° to 360° using my simulation.

However, in order to test valid extreme data, I will attempt to test IfClicked for a further two values in the range 180° to 360°, if this is possible. The test can therefore be conducted as follows:

https://drive.google.com/file/d/1k96epHCfAEXjv1aEJHw-Ycr6cN6YOnzQ/view?usp=sharing

As seen in the above video, the procedure is functional for the range 0° to 180°. However, as seen at 01:50 of the video, the procedure fails to produce the expected result when the maximum angle exceeds 180°. This means that the procedure IfClicked() is erroneous when maxAngle is greater than 180°, and remedial action must be taken to resolve this error. As a result of this error, the above test must be considered as a failed test. The test can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
IfClicked()	Click the mouse on a specified angle along the protractor. Let the pendulum swing for 3 cycles.	Click the mouse on 3 different values (as x = 360 - x) along the protractor in steps of 30°, starting from 0° and ending on 180°. Click the mouse on a further two values in the range 180° to 360°. Let the pendulum swing for 3 cycles for each angle.	This will test whether the pendulum continues to reach the same position it was released from (maximum angle) after 3 full cycles, when the damping factor is 0. The values in the range 180° to 360° are included in order to test valid extreme data.	The pendulum will continue to reach its maximum angle consistently after 3 cycles for various values of θ_0.	Fail

I will now take remedial action to fix the aforementioned error. By analysing the video, it is clear that when the maximum angle is greater than 180°, the pendulum rotates in the clockwise

direction as opposed to the expected anticlockwise direction. In order to convert a rotation from clockwise to anti-clockwise, you must subtract 360 from the angle of rotation. In other words, if the angle of rotation is clockwise then (angle of rotation - 360) is anti-clockwise. In the case of IfClicked(), the angle of rotation is equal to maxAngle. IfClicked() may therefore be rewritten as follows:

```
void IfClicked()
{
    SetNodePos();
    maxAngle = Mathf.Atan2((node.transform.position.y - pivot.transform.position.y), (node.transform.position.x - pivot.transform.position.x)) + (0.5f * Mathf.PI); //resultant angle is in radians.
    if (maxAngle > Mathf.PI)
    {
        maxAngle = maxAngle - (2 * Mathf.PI);
    }
    pivot.transform.localEulerAngles = new Vector3(0, 0, maxAngle * Mathf.Rad2Deg); //Euler angles are in degrees. Therefore, maxAngle must be converted from radians to degrees.
    currentTime = 0;
}
```

The remedial action can be summarised as follows:

Module/Routine	Explanation of remedial action	Justification of remedial action taken
IfClicked()	When the maximum angle is greater than 180°, convert the rotation from clockwise to anti-clockwise by subtracting 360 from the angle of rotation.	If the angle of rotation is clockwise then the (angle of rotation - 360) is anti-clockwise.

Now that remedial action has been taken I will conduct the test again, as follows:

https://drive.google.com/file/d/1fQ9so_OtR3_2K3aH4JT0gBD_S81Y-_JV/view?usp=sharing

As seen in the above video, the procedure is now functional for the range 180° to 360°. It has also retained its functionality for the range 0° to 180°. It follows that the procedure is now functional for all values of maxAngle. The (re)test and its results can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
IfClicked()	Click the mouse on a specified angle along the protractor. Let the pendulum	Click the mouse on 3 different values (as x = 360 - x) along the protractor in	This will test whether the pendulum continues to reach the same position it was	The pendulum will continue to reach its maximum angle consistently	Pass

	swing for 3 cycles.	steps of 30°, starting from 0° and ending on 180°. Click the mouse on a further two values in the range 180° to 360°. Let the pendulum swing for 3 cycles for each angle.	released from (maximum angle) after 3 full cycles, when the damping factor is 0. The values in the range 180° to 360° are included in order to test valid extreme data.	after 3 cycles for various values of θ_0.	

The procedure IfClicked() can now be deemed as functional. This concludes the implementation of IfClicked().

Testing Motion()

The procedure Motion() must now be tested. As outlined in the design phase the procedure can be tested by recording the time taken for the pendulum to return to its maximum angle and comparing this value with the expected value for time period, for varying starting positions and starting lengths. However, as I have not yet implemented the slider responsible for changing the length of the rod, I cannot test the procedure for varying lengths of rod. Therefore, I cannot fully test the procedure Motion() until the slider has been implemented. However, I can partially test the procedure, by testing it for varying (3) starting positions only.

As calculated from FindLength(), the length of the rod is 15. Therefore, by using the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g)`, the expected value for time period is approximately 7.77 seconds. As the expected value has now been calculated, the test may be conducted as follows:

Partial Motion Test

The 3 observed values I obtained from this test are presented on the right. These observed values can be verified by the above video. The mean of these 3 observed values is 23.3/3 = 7.76. This is approximately equal to the expected value of 7.77. Therefore, as the observed value is approximately equal to the expected value, the procedure Motion() can be deemed accurate, for varying starting positions. Whether the procedure is similarly accurate for

varying starting lengths, or not, will need to be determined later, following the implementation of the slider. The test and its results can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
Motion()	Record the time taken for the pendulum to return to its maximum angle and compare this value with the expected value for varying starting positions.	Set the damping factor to 0. Then click the mouse on some position on the screen. Record time taken using external stopwatch. Repeat this for 3 random starting positions.	This will test that, when damping = 0, the time taken for one left swing and one right swing is as expected.	The values recorded should be equal to the values calculated using the equation: $T = 2\pi\sqrt{\frac{L}{G}}$	Pass

This concludes the implementation of Motion(), for now.

The program now meets all of the success criteria that I outlined at the start of the 'prototype'. It therefore follows that the implementation of 'Prototype 1', has come to a conclusion.

Feedback for Prototype 1

As mentioned earlier, I am taking an iterative approach for this project, as per the specification of the exam board. This will involve producing successive prototypes, where the purpose of each prototype is to elicit feedback. This feedback will in turn guide the requirements for the next logical prototype. As the first prototype has now been implemented, I must therefore obtain feedback for this prototype, in order to guide the requirements for the next (second) prototype.

My simulation will primarily be aimed at students studying the Physics A Level. Therefore, I will obtain feedback for Prototype 1 from students studying the Physics A Level. This is because obtaining feedback directly from potential users/ stakeholders ensures that the feedback is as accurate and as relevant as possible. The beta testers I have chosen are Glenn Aranha and James Gordon of the Royal Latin School, both of whom currently study A Level Physics. In order to

obtain feedback, I asked both beta testers to use the simulation and provide feedback in the form of a 'what went well...' and an 'even better if...'. The feedback obtained is as follows:

James said: "I liked the choice of the colour scheme because it's very minimalistic and easy on the eyes. I found the ball that appears when you click the mouse a bit unnecessary though, since there's already a cursor and a ball on top of the cursor is a bit too much in my opinion.'

Glenn said: "The pendulum motion is very smooth and appears to be very accurate as well. However I feel like, in the context of Required Practical 7, the time period of the pendulum is a tad too large. In my opinion the time period should be a bit shorter as this would reduce time taken to use the simulation, in turn allowing the simulation to be used more efficiently. Also maybe add a feature that allows you to change the length of the pendulum, although I'm sure you're planning to add that anyway."

The key points from this set of feedback may be summarised as follows:

- Remove the node from the appearance of the simulation, as it is unnecessary and compromises the overall aesthetic of the simulation.
- Reduce the time period of the pendulum, as the current time period is too large for the simulation to be used practically.
- Add a slider that allows the user to change the length of the pendulum.

This feedback will be used to guide the requirements for the next logical prototype, Prototype 2.

Implementation of Prototype 2

Referring to the feedback obtained from Prototype 1, the second prototype I will be implementing will aim to meet the following requirements:

- The simulation will function without the appearance of a node.
- The time period will be less than the time period observed in Prototype 1.
- The user will be able to change the length of the pendulum by using a slider.
- An additional slider will be implemented, which will in turn allow the user to vary the force the pendulum experiences due to friction.

As well as meeting the above requirements, the second prototype will more formally aim to meet the following success criteria:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Criterion	Justification	Measurement
The program must demonstrate pendulum motion.	This will determine whether or not the pendulum is demonstrating the relevant equations correctly to the user.	The simulation will display characteristics of a motion that is widely accepted and defined as pendulum motion.
The controls must be intuitive and comfortable to use.	This will make the program efficient to use and easy to learn for new users.	The user should be able to use the simulation without any difficulty and without the need for any prompts outside of the program.
The program must have a practical use i.e. the ability to be used as a tool.	This will fulfil the primary purpose of the program that is to complement the A Level Physics course.	The user should be able to obtain correct calculations and answers using the results obtained from this simulation, linking back to the first success criteria.
The program must allow a range of different lengths and angles to be inputted.	As stated in the essential features section, this is so that the user can experiment with different scenarios and thus gain a deeper	The user should be able to run the simulation using any combination of values, provided the values are valid.

	understanding of the topic.	
The program must be able to deal with all valid, invalid and extreme inputs from the user.	This will make the program more robust and less prone to crashing. This provides the user with a more reliable experience.	The user should not be able to crash the program as a direct result of an invalid or extreme input.

Some of the success criteria are the same as the success criteria outlined in Prototype 1. This is because the simulation must both: satisfy the new set of requirements and still meet the success criteria from the previous prototype. In other words, it must not lose functionality. The new success criterion introduced for this prototype is success criterion 4.

I will begin by first addressing the feedback obtained from Prototype 1.

Removing Node

The first piece of feedback I will address is the removal of the node from the appearance of the simulation. It is important to note the distinction between removing the node from the appearance of the simulation and removing the node from the simulation. At first glance, it would appear that both would produce the same result to the user. However, fundamentally, the node object is more than simply an aesthetic feature as it is required in the procedures SetNodePos() and IfClicked. It is therefore, ultimately, an essential object in the context of the program and so removing it from the simulation would result in the simulation failing. It is therefore essential that this object remains in the program. However, it is not essential for the object to be visible to the user.

The visibility of objects in Unity is controlled by the Mesh Renderer. Therefore, by disabling the Mesh Renderer for the node object, the node can be removed from the appearance of the simulation without actually being removed from the program.The Mesh Renderer can be disabled by very simply unchecking a box, as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Upon running the simulation, the node is no longer visible, as required. Therefore, this piece of feedback has now been addressed.

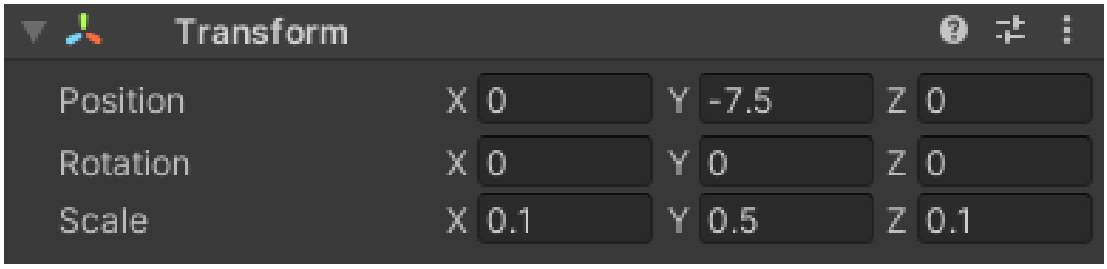
Reducing time period

The next, and final, piece of feedback to address is the reduction of the time period. By referring to the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g)` it is clear that the time period can be reduced by reducing the value of lengthOfRod only. This is because all of the other values are constant and cannot be changed. Therefore, in order to satisfy the feedback, the current length of the rod must be reduced.

As calculated from FindLength(), the current length of the rod is 15. The equation `timePeriod = 2 * π * sqrt(lengthOfRod / g)` requires all variables to be in SI units. In other words, the equation requires timePeriod to be in seconds and lengthOfRod to be in metres. It therefore follows that the current length of the rod is 15 metres. This makes the elicited feedback very justified, as 15 metres is an extremely large and unrealistic value for the length of the rod!

I must now find a more sensible value for the length of the rod. To assist me with this, I will refer to the primary reason this simulation was conceived, Required Practical 7. As outlined in the attached document the length of the rod varies from 1.500m to 0.500m, with the length of the rod decreasing from 1.500m at the start of the practical to 0.500m at the end of the practical, in 0.100m intervals. It therefore follows that I should set the length of the rod at 1.500m at the upper bound and 0.500m at the lower bound. I can then vary the length of the rod between these two bounds through the use of a slider, which I will implement later on in development.

The length of the rod can be set to 0.500 as follows:

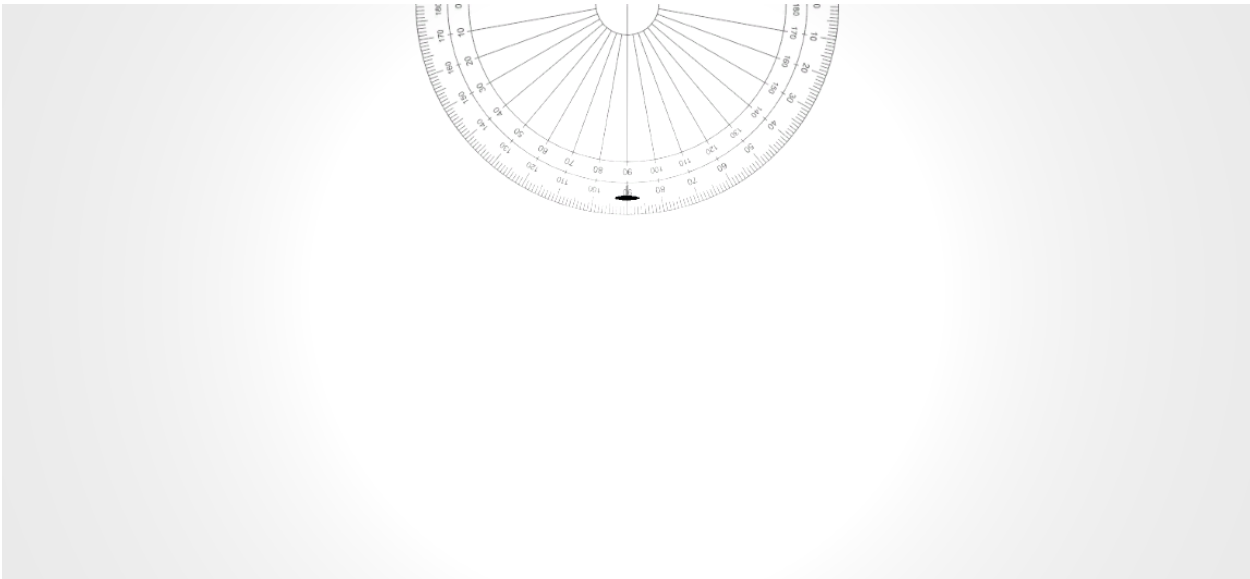


Similarly, the length of the rod can be set to 1.500 as follows:

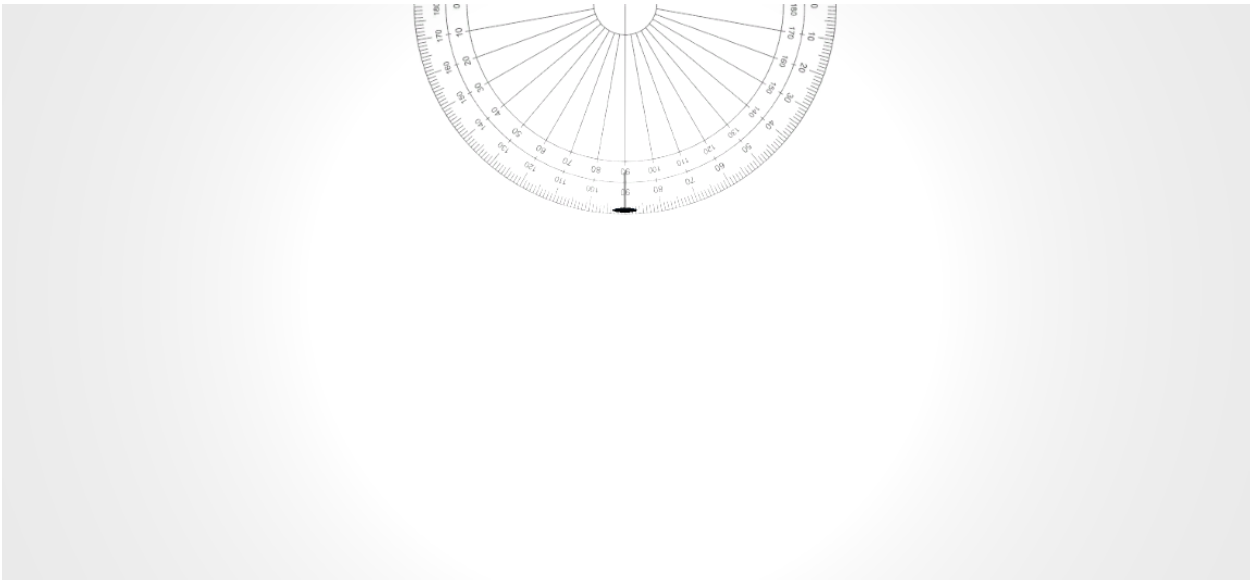
Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

However the result of these actions, for 0.500 and 1.500 respectively, are as follows:

0.500



1.500



There are exactly three problems that can be identified from both of the above results.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

The first problem is the shape of the mass. By nature, the mass should be spherical. However it appears that upon changing the length of the rod, the mass loses its spherical property. The solution to this problem lies in the scale of the mass object. As outlined early in development, the scale of the mass should be the exact reciprocal of the scale of the rod . This results in the product of the two scales giving a final scale of (1, 1, 1), which in turn allows the mass to retain its shape of a perfect sphere. It therefore follows that as the scale (length) of the rod changes, the scale of the mass should also change, such that scale of mass = 1 / scale of rod. This problem has therefore been explained.

The second problem is the position of the rod. As seen in the above images, the position of the top of the rod is no longer level with the straight edge of the protractor as required. In order to solve this problem, it is important to note that the scale of the rod is relative to the centre of the rod. Therefore, for example, if you increase the scale (length) of the rod by 10 units in the y-direction, you will in fact add 5 units to each side of the rod. It follows that the rod must be moved down 5 units, to account for the 5 units added to the top of the rod. This ensures that the top of the rod will remain level with the straight edge of the protractor. As the mass is a child of the rod, it will automatically move 5 units down. Therefore, the 5 units added to the bottom of the rod will not need to be accounted for. The previous scale of the rod was as follows:

By applying the above example and increasing the scale of the rod by 10 units in the y direction, the position of the rod can be altered as follows:

We can apply this same logic for when the scale of the rod is decreased. If you decrease the scale (length) of the rod by x units in the y-direction, you will subtract x / 2 units from each side of the rod. It follows that the rod must be moved up x / 2 units, to account for the x / 2 units subtracted from the top of the rod. Therefore, by setting the length of the rod to 0.500, as before, the position of the rod can be altered as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

By observation, in order to ensure that the top of the rod remains level with the straight edge of the protractor, the y-position of the rod must be exactly -0.5 times that of the y-scale of the rod. In other words, y-position = -(y-scale / 2). This problem has therefore been solved.

The third, and perhaps most glaring, problem is the length of the rod. As seen in the images earlier, despite being the required lengths 0.500 and 1.500, the length of the rod in either case is far too small for the simulation to be used practically by the user.

In order to overcome this problem, I will: 1) keep the actual length of the rod at value(s) such that the simulation can be used comfortably by the user and 2) manipulate the calculated length of the rod such that it is in the range 0.500-1.500. This ensures that both the length of the rod is large enough and the time period is short enough for the simulation to be used practically.

I must now choose two bounds for the actual length of the rod such that that the simulation can consistently be used comfortably by the user. After a bit of experimenting, the two values I have chosen are 10 and 20.

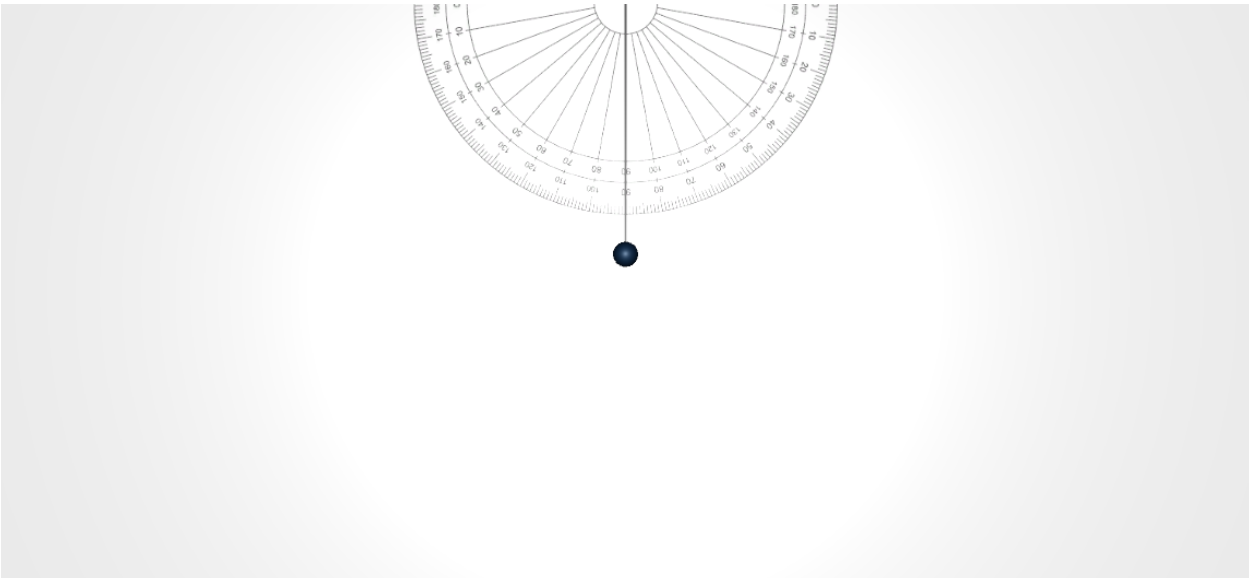
The length of the rod can be set to 10 as follows:

Similarly, the length of the rod can be set to 20 as follows:

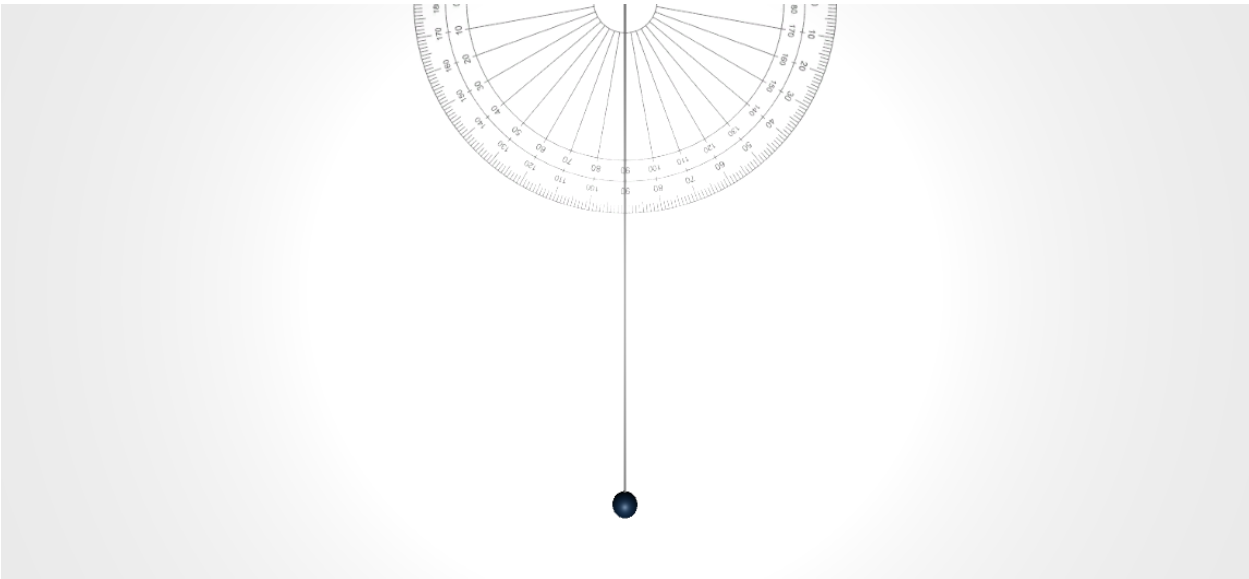
For each length, the rod now appears as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

10:



20:



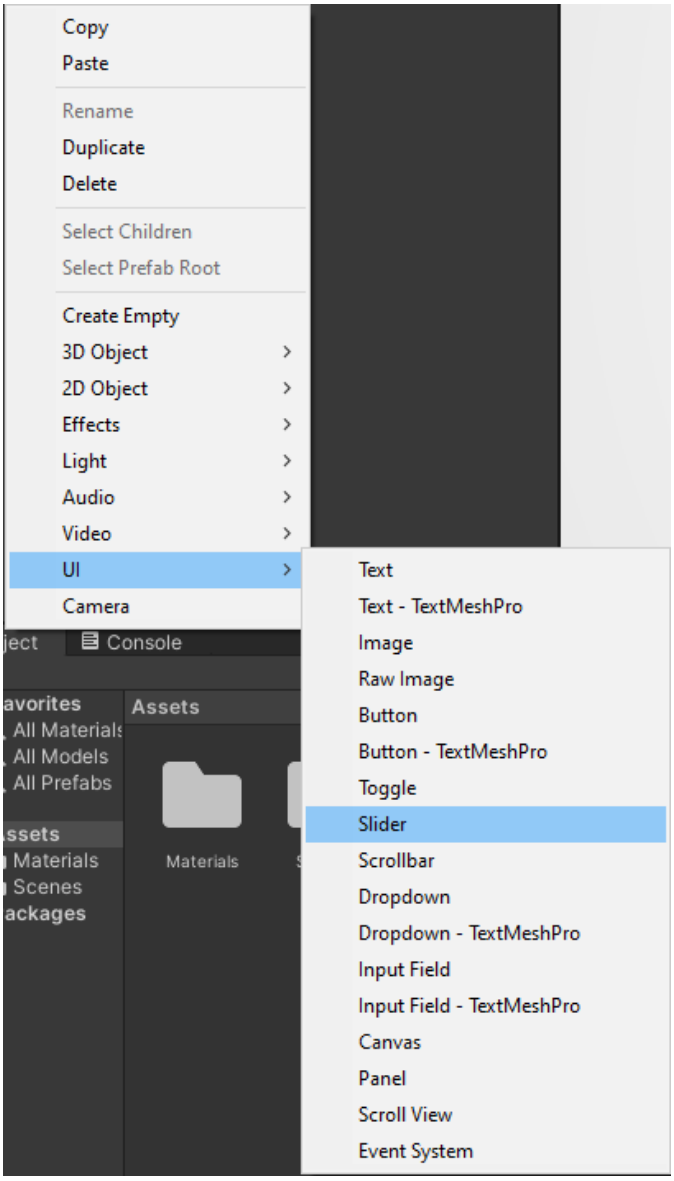
As mentioned earlier, I must now manipulate the calculated length of the rod such that the calculated value for the rod at 10 is 0.500, and the calculated value for 20 is 1.500. This can be achieved by first subtracting 5 from the length of the rod, then dividing the result by 10. It therefore follows that the procedure CalculateConstants() must be rewritten as follows:

```
void CalculateConstants()
{
    lengthOfRod = (FindLength(mass, pivot) - 5) / 10;
    maxDisplacement = maxAngle * lengthOfRod;
    timePeriod = (2 * Mathf.PI) * Mathf.Sqrt(lengthOfRod / gravitationalAcceleration);
}
```

The time period is now short enough for the simulation to be used practically. Therefore, this piece of feedback has now been addressed. As all feedback for Prototype 1 has been addressed, I can now begin the implementation of Prototype 2.

Slider

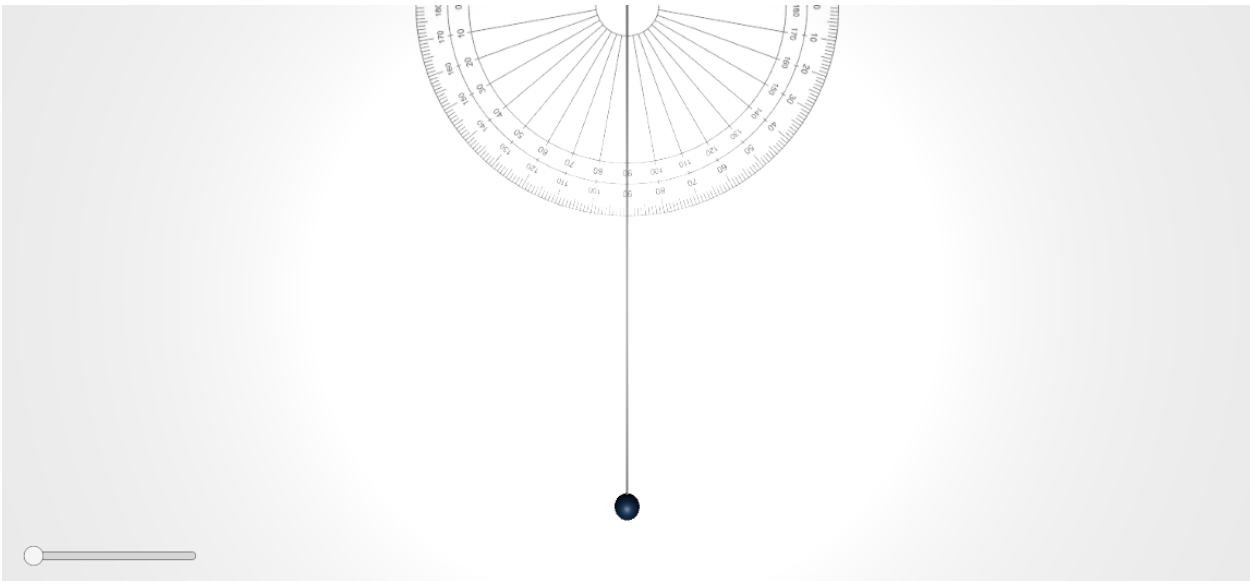
The first feature I will be implementing is the main slider for the simulation, that is, the slider responsible for changing the length of the rod. A slider object can be made in Unity relatively simply, as follows:



Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

I can then adjust the position of the slider as follows:

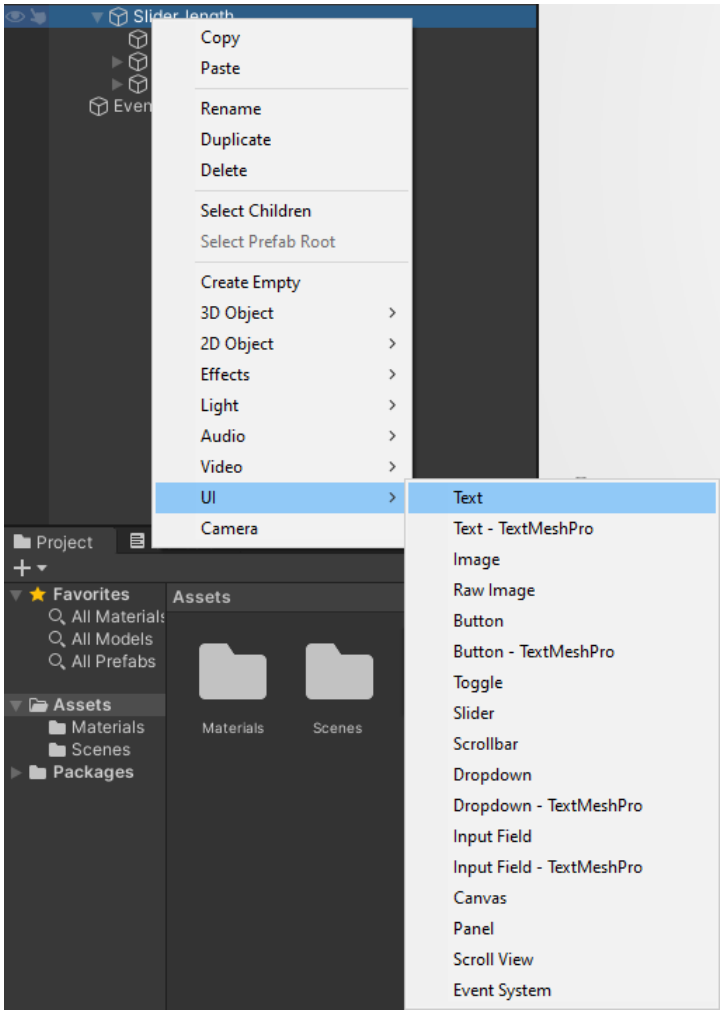
This resulted in the following appearance for the slider:



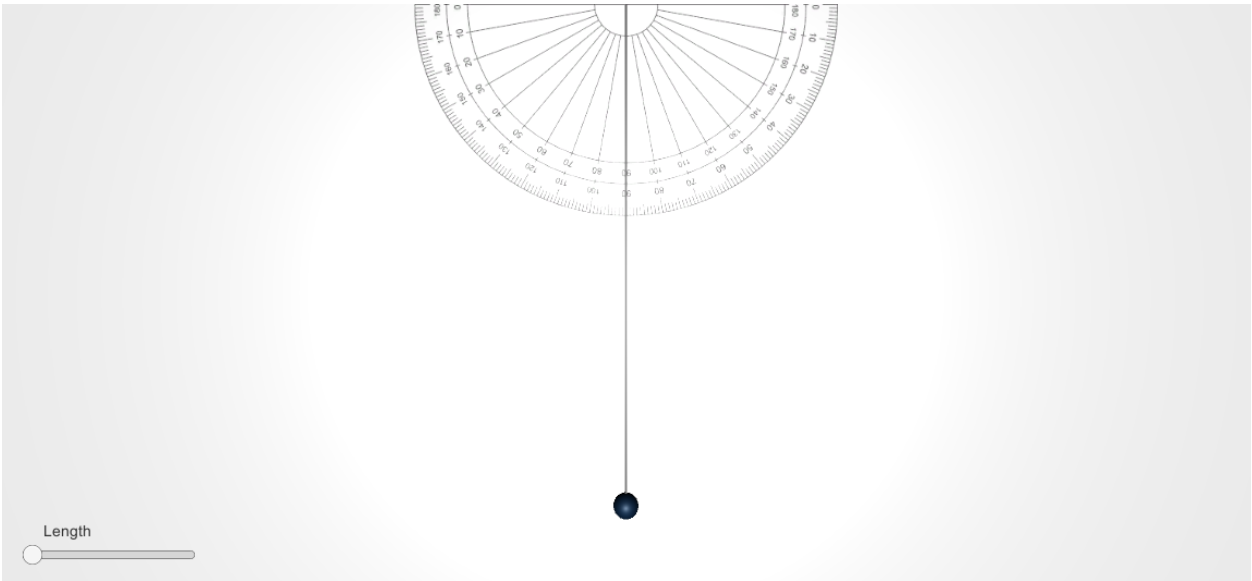
Next I must add text for the slider, in order to indicate to the user what the slider is used for. This is particularly important as I am planning to implement a second slider, used to vary the force the pendulum experiences due to friction, so text is also required to distinguish between these two sliders.

Text may be added as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

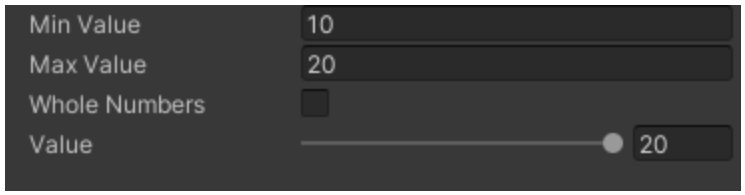


I can then change the text to read ‘Length’ and adjust the position of the text so that it is directly above its corresponding slider. The result is as follows:

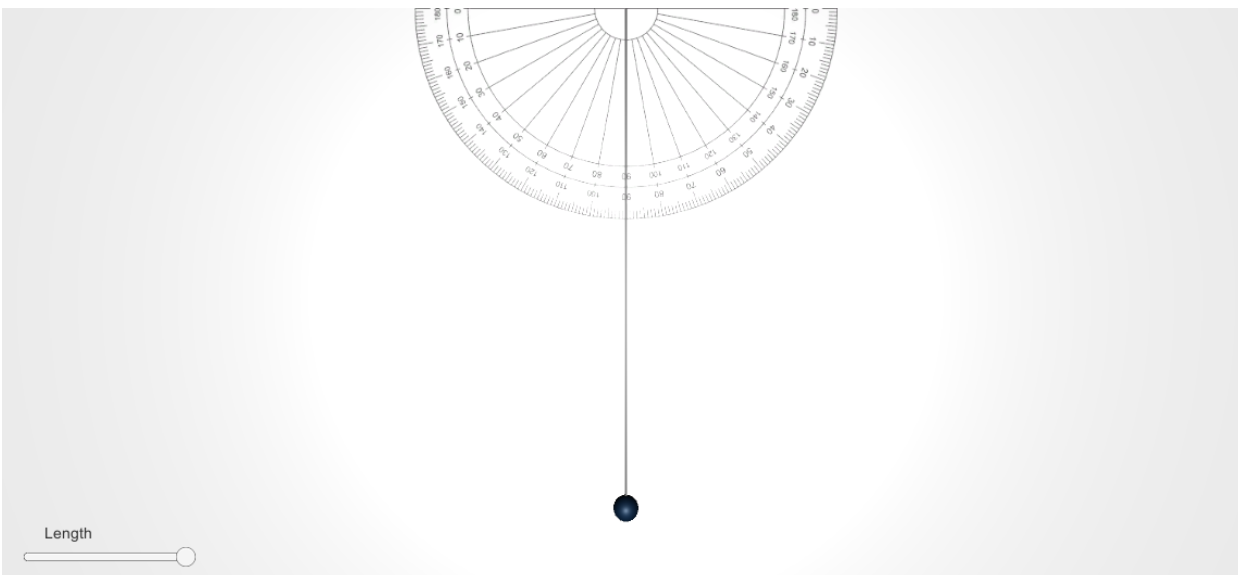


Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Finally, I must set the minimum and maximum values for the slider. As mentioned earlier, the actual minimum and maximum values for the length of the rod are 10 and 20 respectively. I can therefore set the minimum and maximum values for the slider as follows:



Note that I have set the starting value of the slider at its maximum, 20. This is because, as outlined in the document for Required Practical 7, the length of the rod must start at its maximum value, and then decrease to its minimum value. Upon making these changes, the simulation now appears as follows:



It must now write the script responsible for changing the length of the rod, based on the value of the slider. I wrote a very rough algorithm for this procedure in the design section. However, I do not plan to refer to this past pseudocode very much as, given what I now know about the behaviour of the rod object, I can write a more practical procedure. The script can be written as follows:

```
public class Slider_length : MonoBehaviour
{
    public Transform rod;
    public Transform mass1; //already a variable named mass in Main.cs. Therefore, named mass1 to avoid confusion.
    public Slider slider;
    public float length;

    // Update is called once per frame
    void Update() //Slider()
    {
        length = slider.value;
        rod.localScale = new Vector3(rod.localScale.x, length, rod.localScale.z);
        rod.localPosition = new Vector3(rod.localPosition.x, -(length / 2), rod.localPosition.z); //as y-position = -(y-scale / 2)
        mass1.localScale = new Vector3(mass1.localScale.x, 1 / length, mass1.localScale.z); //as scale of mass = (1 / scale of rod)
    }
}
```

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

The script compiles with no errors. Before I can formally test the script I must first ensure that the script roughly works as expected. I can therefore attach the script Slider_length.cs to the Pivot game object, thus attaching it to all game objects related to the pendulum, and then assign the values of rod, mass1 and slider to their corresponding game objects Rod, Mass and Slider1. Upon doing this, running the program produces the following results:

https://drive.google.com/file/d/14nwRLvP2vyyEAMzKdxGHmlhBId5AYgNV/view?usp=sharing

As seen from the above video, the script appears to produce the expected results. In order to formally test this script, however, I must know what length the rod is for each position on the slider. I must therefore write another script that alters the text placed directly above the slider, so that it displays the value of the slider. It is important to note here that it should not display the true value of the slider (e.g. 20), rather it should display the manipulated value of the slider (e.g. 1.5) as it is this value that will be used in the calculation of the time period. The script should therefore be written to reflect this, as follows:

```
public class LengthViewer : MonoBehaviour
{
    public Text sliderValue;
    public Slider slider;

    // Update is called once per frame
    void Update()
    {
        sliderValue.text = "Length: " + ((slider.value - 5)/ 10).ToString("0.000");
    }
}
```

Upon attaching this script to the Slider1 object and assigning the values of sliderValue and slider to Text1 and Slider1 respectively, the program now runs as follows:

https://drive.google.com/file/d/1vZULSTOSzWVVlQJRILTik4wQDghRHsyJ/view?usp=sharing

As seen in the above video, the script LengthViewer.cs functions as expected. Therefore, as the length of the rod for each position is now known, I can formally test the script Slider_length.cs.

Testing Slider_length.cs

As outlined in the design phase the procedure can be tested by dragging the slider to the minimum and maximum inputs and checking whether the length of the rod decreases as the value of the slider decreases, or not. The test can therefore be conducted as follows:

https://drive.google.com/file/d/1scel9ewC3-uPzF688mYgGRjKICYUErk4/view?usp=sharing

As seen in the above video, the length of the rod decreases as the value of the slider decreases. The procedure LengthViewer.cs can therefore be deemed as functional. The test and its results can be summarised as follows:

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
Slider()	Drag the slider to the minimum and maximum inputs.	Keep the damping factor constant. Drag the slider through from the minimum value to the maximum value. This is preferable to clicking as it allows all values of the length of the rod to be tested easily.	This will test that the length of the rod can be changed appropriately using the slider.	The length of the rod will decrease as the value of the slider decreases.	Pass

This concludes the implementation of the main slider for the simulation, responsible for changing the length of the rod.

Testing Motion()

Earlier, I could not fully test the procedure Motion(). This is because the full test required me to test the procedure for varying lengths of rod, which I could not do at the time as I had not yet implemented the slider responsible for changing the length of the rod. However, I have now implemented this slider. Therefore, I can now fully test the procedure Motion().

The procedure can be tested by recording the time taken for the pendulum to return to its maximum angle and comparing this value with the expected value for time period, for varying starting lengths. In order to conduct this test, I will need to create a table of starting lengths and expected values. I will then fill out an additional column containing the recorded value for each length of rod. Comparing this recorded value with the corresponding expected value from the

table will therefore help determine whether the procedure is accurate, or not. The table may be created as follows:

Length of rod (m)	Expected value (s)	Observed value (s)
1.500	2.46	
1.300	2.29	
1.100	2.10	
0.900	1.90	
0.700	1.68	

The test can be conducted as follows:

https://drive.google.com/file/d/16LvQlbcKt9dgzxyqWUSAirSLYic_4rUM/view?usp=sharing

From the above test, the observed values I obtained are as follows:

Length of rod (m)	Expected value (s)	Observed value (s)
1.500	2.46	2.42
1.300	2.29	2.33
1.100	2.10	2.15
0.900	1.90	1.94
0.700	1.68	1.70

As you can see, the observed values are all approximately equal to their corresponding expected values. Therefore, as the observed values are approximately equal to the expected values, the procedure Motion() can be deemed fully accurate, for both varying starting positions and varying starting lengths. The test and its results can be summarised as follows:

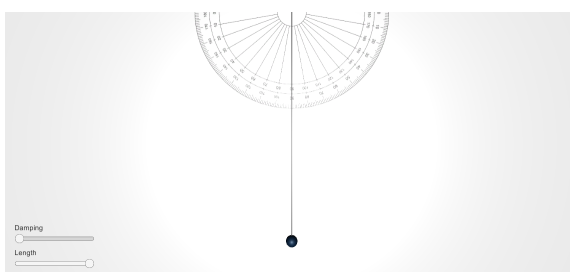
Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
Motion()	Record the	Set the	This will test	The values	Pass

	time taken for the pendulum to return to its maximum angle and compare this value with the expected value for varying starting positions and starting lengths.	damping factor to 0. Then click the mouse on some position on the screen. Record time taken using external stopwatch. Repeat this for 5 randomly generated lengths of rod.	that, when damping = 0, the time taken for one left swing and one right swing is as expected.	recorded should be equal to the values calculated using the equation: $T = 2\pi\sqrt{\frac{L}{G}}$	

This concludes the implementation of the procedure Motion().

Damping()

The next, and last, feature I must implement is the second slider responsible used to vary the force the pendulum experiences due to friction. The slider may be created using the same method used to create the previous slider. Therefore, to save time, I will not reiterate this method in detail and will simply present the final appearance of the simulation:



It is important to note that I have chosen the minimum and maximum values for damping to be 0 and 1 respectively.

Damping is the factor the maximum angle of the pendulum from equilibrium decreases by, until this maximum angle is equal to 0. I must therefore now write the procedure Damping(), which is responsible for reducing the maximum angle based on the value of the slider. As outlined in the design phase, I wish to take a geometric approach when implementing this procedure. However, I wish to use a different method than the one featured in the pseudocode in the design phase.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

As I am taking a geometric approach, I will be using the general formula $A = Pe^{rt}$ to model how the maximum angle changes over time. In order to find r, I must first define what is meant by damping. After some consideration, I have decided to define damping as the factor the maximum angle of the pendulum from equilibrium decreases by after one time period. In other words, after one time period, the maximum angle of the pendulum will be equal to (maximum angle * damping factor). I now have enough information to create a formula to model how the maximum angle changes over time. The derivation of this formula is presented below:

$$\theta = \theta_0 e^{rt}$$

let k = damping factor

let T = time period

when $t = T$,

$$k\theta_0 = \theta_0 e^{rT}$$

$$k = e^{rT}$$

$$\ln k = rT$$

$$r = \frac{\ln k}{T}$$

$$\therefore \theta = \theta_0 e^{\frac{\ln k}{T} t}$$

The procedure Damping() may now be implemented within Main.cs. As the procedure requires the value of the damping factor, determined by the value of the second slider, a slider variable will need to be created, as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

```
public Slider slider1; //already a variable named slider in Slider_length.cs. Therefore, named slider1 to avoid confusion.
```

This variable can then be assigned to the second slider, created earlier. Using the formula derived earlier, the procedure Damping() may be written as follows:

```
void Damping()
{
    dampingFactor = slider1.value;
    maxAngle = maxAngle * Mathf.Exp(((Mathf.Log(dampingFactor)) / timePeriod) * Time.deltaTime);
}
```

The program compiles with no errors. As Damping() makes use of the Time.deltaTime feature, it must be called every frame, and must therefore be called within Update(). Update() may therefore be rewritten as follows:

```
// Update is called once per frame
void Update()
{
    node.gameObject.SetActive(false);
    Timer();
    Damping();
    if (Input.GetKey(KeyCode.Mouse0)) //KeyCode.Mouse0 refers to the left/ primary mouse button.
    {
        IfClicked();
    }
    else
    {
        Motion();
    }
}
```

However, running the program produces the following result:

https://drive.google.com/file/d/1NUw1XD2GsU9hhxwqn3XKj5Z5ZVXaLuRf/view?usp=sharing

There are exactly two problems that can be identified from the above result.

The first, and most obvious, error is that small values of damping (close to 0) seem to greatly decrease the value of maximum angle over time whereas large values of damping (close to 1) seem to only slightly decrease the value of maximum angle over time. Therefore, it appears that the values for damping seem to produce results that are the exact opposite of the expected results.

However, upon observation, this would make sense as in my working I had defined the maximum angle at time T as $k\theta_0$, where k is the damping factor. It therefore follows that when k = 1 the maximum angle will remain unchanged, hence resulting in and explaining the above error. In order to fix this error I must simply let the damping factor = 1 - damping factor, as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

```
void Damping()
{
    dampingFactor = slider1.value;
    dampingFactor = 1 - dampingFactor;
    maxAngle = maxAngle * Mathf.Exp(((Mathf.Log(dampingFactor)) / timePeriod) * Time.deltaTime);
}
```

The second error is the effect damping has when the damping factor is at its maximum. As seen in the above video, when the damping factor is equal to 0, the pendulum immediately stops oscillating. While this behaviour is in line with the formula I created, it is not very realistic in the context of a simulation. Therefore in order to solve this problem, when the damping factor is equal to 1 (as damping factor = 1 - damping factor, from earlier) I will set the value of damping factor to a value extremely close to but not equal to 1, such as 0.9999. This will ensure that the maximum angle will still reduce at an extremely high rate, but not so high that the motion becomes unrealistic. This solution may therefore be implemented as follows:

```
void Damping()
{
    dampingFactor = slider1.value;
    if (dampingFactor == 1)
    {
        dampingFactor = 0.9999f;
    }
    dampingFactor = 1 - dampingFactor;
    maxAngle = maxAngle * Mathf.Exp(((Mathf.Log(dampingFactor)) / timePeriod) * Time.deltaTime);
}
```

The program now runs as follows:

https://drive.google.com/file/d/1Q1SqsI41mKj17NUNmjJtuUSc7ExzB2Eb/view?usp=sharing

As seen in the above video, the errors have now been resolved and the procedure performs as expected. However, there is one final problem I would like to address regarding the maximum angle. As we know. the maximum angle will continue to decrease over time. However, due to the nature of the formula, the maximum angle will never reach 0. Rather the maximum angle will simply tend towards 0, becoming infinitely small in the process, without ever becoming equal to 0. This is a problem, as continuing to calculate an infinitely small angle uses up unnecessary CPU time, hence making the procedure less efficient than it could be. Therefore, in order to fix this problem, I will rewrite the procedure such that if the maximum angle is below some minimum angle then the maximum angle is set to 0, as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

```
void Damping()
{
    dampingFactor = slider1.value;
    if (dampingFactor == 1)
    {
        dampingFactor = 0.9999f;
    }
    dampingFactor = 1 - dampingFactor;
    maxAngle = maxAngle * Mathf.Exp(((Mathf.Log(dampingFactor)) / timePeriod) * Time.deltaTime);
    if (Mathf.Abs(maxAngle) < 0.0001f) //absolute used to account for angles in anticlockwise direction
    {
        maxAngle = 0;
    }
}
```

The procedure Damping() must now be tested. However, before the procedure can be tested, I must know what the value of the damping factor is for each position on the slider. I must therefore write another script that alters the text placed directly above the slider, so that it displays the value of the slider. This script can be written in the same way as the script written for the length slider, as follows:

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

public class DampingViewer : MonoBehaviour
{
    public Text sliderValue;
    public Slider slider;

    // Update is called once per frame
    void Update()
    {
        sliderValue.text = "Damping: " + slider.value.ToString("0.000");
    }
}
```

The procedure Damping() may now be tested. As outlined in the design section, the procedure may be tested by dragging the slider to the minimum and maximum inputs and checking whether the pendulum decreases in amplitude with each swing, until it eventually reaches 0, or not. The test may be conducted as follows:

https://drive.google.com/file/d/1nbEvqubSghfyfLKnYm2unbKjKTcazD2J/view?usp=sharing

As seen in the above video, the amplitude of the pendulum: decreases when the damping factor is equal to 0; decreases proportionally to the damping factor as it tends to 1 and decreases to 0 after one cycle when the damping factor is equal to 1, as required. The procedure Damping() may therefore be deemed as functional. The test and its results may be summarised as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Module/Routine	Test	Test data	Justification	Expected outcome	Pass/Fail
Damping()	Drag the slider to the minimum and maximum inputs.	Keep the length constant throughout. Drag the slider through from the minimum value to the maximum value. This is preferable to clicking as it allows all values of the damping factor to be tested easily.	This will test that the pendulum decreases in amplitude with each swing, until it eventually reaches 0.	The amplitude of the pendulum will: not decrease when the damping factor is equal to 0; decrease proportionally to the damping factor as it tends to 1 and decrease to 0 after one cycle when the damping factor is equal to 1.	Pass

This concludes the implementation of Damping(). The program now meets all of the success criteria that I outlined at the start of the 'prototype'. It therefore follows that the implementation of 'Prototype 2', has come to a conclusion.

Feedback for Prototype 2

I must now obtain feedback for this prototype. Using the same beta testers as before, the following feedback may be obtained:

Glenn said: "This is much better. However one thing I would say is that you should maybe make it so that the pendulum is disabled when using the slider, so that using the slider does not interfere with the pendulum motion. Apart from that, all good."

James said: “This is really good. In the context of RP7 however, I would try and make it so that the length of the rod can only go up in 0.100 intervals, as its quite hard to set the length of the rod at these intervals at the moment”

The above feedback may be summarised as follows:

- Disable pendulum motion when using the slider.
- Change the slider used for varying the length of the pendulum, such that it only goes up in 0.100 intervals.

As before, this feedback will form the requirements for the next prototype, Prototype 3.

Implementing Prototype 3

Referring to the feedback obtained from Prototype 1, the second prototype I will be implementing will aim to meet the following requirements:

- The simulation will allow the user to use the slider, without this interfering with the motion of the pendulum.
- The slider the slider used for varying the length of the pendulum can only be used in 0.100 intervals.
- The simulation will be accessible from within a web browser.

As well as meeting the above requirements, the second prototype will more formally aim to meet the following success criteria:

Criterion	Justification	Measurement
The program must demonstrate pendulum motion.	This will determine whether or not the pendulum is demonstrating the relevant equations correctly to the user.	The simulation will display characteristics of a motion that is widely accepted and defined as pendulum motion.

The controls must be intuitive and comfortable to use.	This will make the program efficient to use and easy to learn for new users.	The user should be able to use the simulation without any difficulty and without the need for any prompts outside of the program.
The program must have a practical use i.e. the ability to be used as a tool.	This will fulfil the primary purpose of the program that is to complement the A Level Physics course.	The user should be able to obtain correct calculations and answers using the results obtained from this simulation, linking back to the first success criteria.
The program must allow a range of different lengths and angles to be inputted.	As stated in the essential features section, this is so that the user can experiment with different scenarios and thus gain a deeper understanding of the topic.	The user should be able to run the simulation using any combination of values, provided the values are valid.
The program must be able to deal with all valid, invalid and extreme inputs from the user.	This will make the program more robust and less prone to crashing. This provides the user with a more reliable experience.	The user should not be able to crash the program as a direct result of an invalid or extreme input.

The program can be used from within a web browser.	This will make the program as accessible as possible making it possible for students and teachers to use the program both in lessons and at home.	The user should be able to open the URL and run the program smoothly on any web browser and therefore any operating system.

This is the full set of success criteria, as outlined in the analysis section. Some of the success criteria are the same as the success criteria outlined in Prototype 2. This is because the simulation must both: satisfy the new set of requirements and still meet the success criteria from the previous prototype. In other words, it must not lose functionality. The new success criterion introduced for this prototype is success criterion 6.

Disabling motion when using slider

The first piece of feedback to address is the disabling of motion when using the slider. This may be achieved by first identifying the region in which the slider is typically used. After some experimentation, I discovered that this region is approximately x< 200 and y < 120. Therefore, by allowing IfClicked() to be called only if the mouse position is outside of this 'slider region' (i.e. the region x > 200 and y > 120), the slider may be used independently of IfClicked(). This in turn allows the slider to be used without altering the maximum angle of the pendulum and hence prevents the slider from interfering with the motion of the pendulum. This may be achieved as follows:

```
// Update is called once per frame
void Update()
{
    node.gameObject.SetActive(false);
    Timer();
    Damping();
    if (Input.GetKey(KeyCode.Mouse0) && (Input.mousePosition.x > 200 || Input.mousePosition.y > 120)) //KeyCode.Mouse0 refers to the left/ primary mouse button.
    {
        IfClicked();
    }
    else
    {
        Motion();
    }
}
```

Upon running the program, the use of the slider no longer interferes with the motion of the pendulum, as required. Therefore, the feedback has now been addressed and the problem has been resolved.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Changing the slider to 0.100 intervals only

This piece of feedback can be achieved by very simply checking the 'Whole Numbers' box for the slider, as follows:

This is because the length of the rod is calculated by dividing the actual length of the rod by 10. Therefore, restricting the slider values to whole numbers only will in turn remove all values of length that are not to 1 decimal place, hence allowing the length of the rod to only increase in intervals of 1 decimal place, i.e. 0.100, as required. Therefore, this feedback has now been addressed and resolved.

Web browser implementation

Now that all feedback has been addressed, I can now implement the final feature for this prototype: implementation within a web browser. I will achieve this using GitHub pages, as follows:

I must first build the program in Unity using the WebGL build option. The WebGL build option allows Unity to publish content as JavaScript programs which use HTML5 technologies and the WebGL rendering. The program can therefore be built in WebGL by selecting the following options: File ->Build Settings -> HTML5/WebGL -> Build. This will produce an index.html file, a Build/ directory and a TemplateData/ directory.

The next step is to move, commit and push all of these files to the root of my GitHub repository. The final step is to then set up GitHub pages. such that the source is set to the root of my GitHub repository (and hence the built WebGL program). GitHub Pages is a static site hosting service designed to host your personal, organization, or project pages directly from a GitHub repository. Therefore, by setting the source of GitHub pages to the root of my GitHub repository, the program may be hosted, and hence accessed, directly from my GitHub repository.

The final web page is as follows:

https://nishaddeokar1.github.io/rp7/

The web page, and hence the simulation, may therefore be accessed and run as follows:

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

https://drive.google.com/file/d/1gMZOR9LIe7qW1YwkTDRjE-pKsjf13K2P/view?usp=sharing

As seen from the above video, the program runs smoothly from within a web browser. It follows that the program is now accessible from within a web browser, as required. The implementation of this feature may therefore be considered a success.

The program now meets all of the success criteria that I outlined at the start of the 'prototype'. It follows that the implementation of 'Prototype 3', has now come to a conclusion. This prototype is the last prototype I will make, as I do not have time to make anymore. I will therefore proceed with my final evaluation of the project.

Evaluation

Success, partial success or failure, in relation to test results - from each success criteria

The first part of evaluation will consist of assessing my program against the success criteria from the analysis, and determine whether this success criteria has been fully met, partially met or not met at all. As this must be in relation to test results, I will conduct this part of my evaluation referring to the following post development testing:

Post-development testing

Test	Test type	Test data	Justification	Requirements for successful result	Pas/Fail
Record the time taken for the pendulum to return to its maximum angle and compare this value with the expected value for varying starting	Functionality	Set the damping factor to 0. Then click the mouse on some position on the screen. Record time taken using external stopwatch. Repeat this	This will show if success criterion 1 and 3 have been met.	The values recorded will be equal to the values calculated using the equation: $T = 2\pi\sqrt{\frac{L}{G}}$	Pass

positions and starting lengths. This will test that, when damping = 0, the time taken for one left swing and one right swing is as expected.		for 5 randomly generated lengths of rod.			
Beta testing, in which potential users are asked to use the simulation, in order to provide feedback on the usability of the program.	Usability	Usability is measured with use of feedback from users.	This will show if success criterion 2 has been met.	Positive feedback received.	Pass. Refer to usability comments.
Click the mouse on a specified angle along the protractor. Let the pendulum swing for 1 cycle. Repeat this 6 times. Drag the slider to the minimum and maximum inputs. This	Functionality	Set the damping factor to 0. Then click the mouse on 6 different values along the protractor in steps of 60°, starting from 0° and ending on 360°. Let the pendulum swing for 1 cycle for each	This will show if success criterion 3 and 4 have been met.	The pendulum will continue to reach its maximum angle consistently after each cycle for the various values of θ_0. The length of the rod will increase as the value of	Pass

will test that the length of the rod can be changed appropriately using the slider.		angle. Keep the damping factor constant. Drag the slider through from the minimum value to the maximum value. This is preferable to clicking as it allows all values of the length of the rod to be tested easily.		the slider increases.	
See validation checks.	Robustness	See validation test data.	This will show if success criterion 5 has been met.	The program is able to deal with all valid, invalid and extreme inputs from the user.	Pass Pass Pass
Use the simulation from within a web browser.	Robustness	Open the link to the web page containing the program.	This will show if success criterion 6 has been met.	The program can be used from within a web browser and retains full functionality when used in this manner.	Pass. Also tested on multiple operating systems (macOS, Windows) and multiple browsers (Chrome, Safari, Firefox,

					Microsoft Edge and Opera). The program can be accessed and run smoothly in all cases.

Therefore, referring to this post development testing, each individual success criteria from the analysis section can be evaluated as follows:

Criterion [1]	Justification	Measurement
The program must demonstrate pendulum motion.	This will determine whether or not the pendulum is demonstrating the relevant equations correctly to the user.	The simulation will display characteristics of a motion that is widely accepted and defined as pendulum motion.

Referring to post-development test 1 and its corresponding result, it is clear that the motion of the pendulum is consistently in accordance with the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g)`.

However, whether this confirms that the program demonstrates pendulum motion or not is still uncertain. The reason for this is that, as explained in the limitations section later, the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g)` is only valid for small angles. More specifically, it is only valid for angles that are less than approximately 10 degrees.
In my simulation, however, the user can input values far greater than this 10 degrees, with the highest value reaching up to 180 degrees. Furthermore, the motion of the pendulum for these large angles is still determined by the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g`, despite these angles being far outside the range of valid values for this equation.

The reason I allowed this during development is because the simulation is primarily aimed for students taking A Level Physics, using the simulation to conduct Required Practical 7. In this practical, students are not required to use the simulation above 10 degrees. It therefore follows

that my simulation is accurate, in the context of this practical where the angle does not exceed 10 degrees.

However, outside of this context, where the angle may very well exceed 10 degrees, the simulation no longer becomes accurate at all.

It therefore follows that the success criterion 'The program must demonstrate pendulum motion' has only been partially met, as the program demonstrates accurate pendulum motion only for a range of angles, as opposed to all possible angles.

Criterion [2]	Justification	Measurement
The controls must be intuitive and comfortable to use.	This will make the program efficient to use and easy to learn for new users.	The user should be able to use the simulation without any difficulty and without the need for any prompts outside of the program.

As evidenced in post-development test 2 and its corresponding results, the usability of the simulation has widely been considered excellent. The controls have been described as 'intuitive to use' and 'super easy to manipulate' while the colours have been described as 'very complementary and soft'. This directly matches the measurement outlined for this success criterion from the analysis section and hence there is strong evidence to suggest that this success criterion has been met.

One argument against this, however, is the fact that the simulation has thus far only been used and tested by students taking the A Level Physics course. It could be argued that these users, due to past experience of dealing with pendulums and observing pendulum motion, may have had an advantage when using the simulation. This raises the question as to whether a user with no background in Physics would have been able to use the simulation as comfortably as the current set of beta testers, and whether this selection of beta testers has potentially skewed the results obtained from usability testing.

While this is a valid point that could be made, it is clear that this simulation is targeted towards people with either an existing background in Physics or an interest in Physics (which would imply that they have had some exposure to Physics concepts). Therefore, people that have had no

exposure to the concept of a pendulum should not be particularly considered when evaluating the usability of the pendulum simulation as they are highly unlikely to be using the simulation anyway, especially for its intended purpose. Furthermore, in the unlikely event that such a person were to use the simulation, given the feedback I have received and given the way I have designed the simulation, I am confident that due to the intuitiveness of its controls, such a user would be able to use my simulation without much assistance.

I therefore believe that success criterion 2 has been fully met and that the controls are consistently intuitive and comfortable to use, as required.

Criterion [3]	Justification	Measurement
The program must have a practical use i.e. the ability to be used as a tool.	This will fulfil the primary purpose of the program that is to complement the A Level Physics course.	The user should be able to obtain correct calculations and answers using the results obtained from this simulation, linking back to the first success criteria.

As evidenced in post-development tests 1 and 3 and their results, it is clear that, for the range of values defined in the documentation for Required Practical 7, the program performs the required calculations correctly. It therefore follows that the simulation may be used to obtain results for Required Practical 7 accurately, as required. This directly matches the measurement outlined for this success criterion from the analysis section and hence there is strong evidence to suggest that this success criterion has been met.

However, the program may have arguably been designed to complement the A Level Physics even further. This could have been achieved by implementing an inbuilt stopwatch within the program, so that the user may have recorded the time period internally within the simulation, as opposed to externally using an external stopwatch.

While this implementation of a stopwatch could have certainly increased the simulation's practicality as a tool, I believe that the simulation can still be used as an effective tool for conducting required Practical 7 in its current, and therefore do not consider this potential improvement of a stopwatch as a major point against the simulation meeting this success criteria.

I therefore believe that the simulation I have created completely fulfils its primary purpose of complementing the A Level Physics course and hence believe that success criterion 3 has been fully met.

Criterion [4]	Justification	Measurement
The program must allow a range of different lengths and angles to be inputted.	As stated in the essential features section, this is so that the user can experiment with different scenarios and thus gain a deeper understanding of the topic.	The user should be able to run the simulation using any combination of values, provided the values are valid.

Referring to post-development test 3 and its corresponding result, the program allows the user to input a range of different lengths and angles.

However, as explained in the limitations section later, while the program allows for a range of different lengths and angles to be inputted, this range is rather limited. This is because the range was specifically chosen in the context of Required Practical 7. It therefore follows that, outside of the context of Required Practical 7, the range of values permitted by the simulation may very well be considered limited/ restricted.

It should be noted however that this restriction allows success criterion 3, from earlier, to be more easily met. Increasing the range of values, while on one hand allowing the user to experiment with a wider range of scenarios, would also make the simulation significantly harder to use as a practical tool, used to conduct Required Practical 7.

Therefore, while the range of values provided by the program is limited, this restriction was imposed as a result of a tradeoff, in order for the simulation to have greater practicality as a tool. Furthermore, as demonstrated by the test results, the simulation is perfectly functional for any combination of values within these ranges.

I therefore believe that success criterion 4 has been fully met, as the range of values are both justfied and operative.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Criterion [5]	Justification	Measurement
The program must be able to deal with all valid, invalid and extreme inputs from the user.	This will make the program more robust and less prone to crashing. This provides the user with a more reliable experience.	The user should not be able to crash the program as a direct result of an invalid or extreme input.

Referring to post-development test 4 and its results, it can be seen the program is able to deal with all valid, invalid and extreme inputs from the user.

Due to the nature of the program, there is not a significant amount of validation required. This is because, in order to increase the usability of the program, I minimised the amount of input required by the user. As a result, the amount of validation required is also minimised.

Nevertheless all inputs, that is, mouse input and slider inputs have all been validated. It therefore follows that success criterion 5, 'The program must be able to deal with all valid, invalid and extreme inputs from the user', has been fully met.

Criterion [6]	Justification	Measurement
The program can be used from within a web browser.	This will make the program as accessible as possible making it possible for students and teachers to use the program both in lessons and at home.	The user should be able to open the URL and run the program smoothly on any web browser and therefore any operating system.

As highlighted in post-development test 5 and its results, it can be seen that the program can be used from multiple different platforms, within multiple different web browsers.

While the program can be used from any web browser, on any operating system, it may be tempting to conclude that the program may be accessible from anywhere in the world. However, as outlined in the limitation section later, there are two flaws in this logic.

In order for the simulation to run within the web browser the device running the simulation must have an Internet connection. Furthermore, in order for the simulation to run smoothly, as required, the device running the simulation must exceed the hardware requirements outlined in the analysis section.

While these are limitations to the program's accessibility, and hence robustness, these limitations were outlined in the analysis section as requirements. Therefore, by specifying these limitations as requirements upfront, I believe that the simulations accessibility/ robustness is overall maximised as much as possible.

I therefore believe that success criterion 6 has been fully met as, provided the specified hardware requirements are met and an Internet connection is established, the program can be used from within any web browser, on any operating system, on any device.

Verdict

In conclusion, while there are limitations to my solution, my final solution has, as seen above, fully met 5 out of 6 of the success criteria outlined in the analysis section. Therefore, to this extent, I believe that the project can be deemed successful. As the solution has only partially met success criterion 1, improvements will need to be taken in order to fully meet success criterion 1, the details for which will be outlined later, in the 'potential improvements to limitations' section.

Usability features

The usability features outlined in the design section simply described usability in terms of general aesthetic/ colour scheme and accessibility for the most part.

However, I feel that I have managed to go beyond this and have managed to provide usability in terms of using the simulation specifically as a tool, more specifically as a tool for Required Practical 7. An example of this specific usability is by allowing the slider used to vary the length of the rod to only go up in 0.100 intervals. This makes the simulation more usable in a practical sense, which I think is extremely valuable. Another example of where the simulation was made more usable was the implementation in the web browser, allowing the simulation to be accessed from anywhere.

To back up my claims of excellent usability, I decided to obtain some final feedback from potential stakeholders, asking them what they thought of the program in terms of usability. The people I chose for my feedback were Akshita Purohit and Rhys Cooke, both of whom study A Level Physics at the Royal Latin School.

Akshita said: “I really like this simulation. The pendulum motion looks really smooth and the colours are very complementary and soft. The controls are very intuitive to use and the sliders are super easy to manipulate.”

Rhys said “This is actually a really sweet simulation. Being able to run it from within a web browser is pretty impressive and I love the way the damping works as it seems super realistic, and the colours look super clean as well.”

Therefore, as you can see from the above feedback, the usability can be considered a success, and I am very happy with how the usability of the pendulum turned out.

Limitations of the solution

There are, however, still clear limitations to my solution. They can be described in terms of usability, functionality and robustness, as follows:

Usability

The largest limitation in terms of usability is the limitation in the range of lengths that can be chosen. The range of lengths for the simulation is 0.500-1.500. I chose these lengths in accordance with the A level Physics course. While this increases usability for those using the simulation for A level Physics, it actually decreases usability for users that do not require simulation for Required Practical 7, that may be using the simulation for a different purpose. Hence, the range of lengths is a limitation.

Functionality

As outlined in the analysis section, one limitation of the solution in terms of functionality is the implementation of damping. This is not an accurate model of damping, which takes into account air resistance, rather it is a very simplified model that simply provides an idea of what damping may look like. As mentioned earlier, this is satisfactory as most students using the simulation will not need to know the ins and outs of proper air resistance. Furthermore, avoiding true air resistance greatly simplifies calculation, which in turn can allow the simulation to run smoother. However, this is still a limitation as the damping is not 100% accurate.

Another, and perhaps the largest, limitation of the solution is the use of the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g`. This equation is only valid for small angles. More specifically, it is only valid for angles that are less than approximately 10 degrees. In my simulation, however, the user can input values far greater than this 10 degrees, with the highest value reaching up to 180 degrees. As the motion of the pendulum for these large angles is still determined by the equation `timePeriod = 2 * π * sqrt(lengthOfRod / g`,

despite these angles being far outside the range of valid values for this equation, it follows that the simulation is largely inaccurate for large angles.

This limitation therefore prevents the program from fully meeting success criterion 1, as the program can only demonstrate accurate pendulum motion for only a range of angles, as opposed to all possible angles.

Robustness

The largest limitation in terms of robustness is the requirement for an Internet connection in order to access the simulation. This prevents people without an Internet connection from using the simulation, making it less of a robust solution that it could be. While this is not a major issue, as most people with a computer will have an Internet connection, it is not ideal as it means the simulation cannot be used from anywhere i.e. places where there is no Internet connection.

Potential improvements to these limitations

I will now propose potential improvements to each limitation, as follows:

Usability

This limitation could be addressed by allowing the user to input a large range of lengths. Furthermore, to appeal to all audiences, there can be a Required Practical 7 version of the simulation, specifically for A Level students, and a general version for more broad/ general use. This would not require a substantial amount of reprogramming and can therefore be considered a feasible solution to this limitation.

Functionality

The implementation of damping could be worked around by rewriting the procedure for damping. While this would require a substantial amount of reprogramming, the procedure Dampin() itself is relatively isolated from the rest of the program. Therefore, this limitation could potentially be worked around in a short time frame. However, whether it would be worth doing so would need to be reviewed and determined.

In order to overcome the limitation imposed through the use of the formula, `timePeriod = 2 * π * sqrt(lengthOfRod / g`, an alternative formula would need to be used. This formula would need to be valid for all angles, as opposed to just small angles. After some research, I discovered that such a formula exists, and is as follows:

$$T = 2\pi\sqrt{\frac{\ell}{g}}\left(1 + \frac{1}{16}\theta_0^2 + \frac{11}{3\,072}\theta_0^4 + \frac{173}{737\,280}\theta_0^6 + \frac{22\,931}{1\,321\,205\,760}\theta_0^8 + \frac{1\,319\,183}{951\,268\,147\,200}\theta_0^{10} + \frac{233\,526\,463}{2\,009\,078\,326\,886\,400}\theta_0^{12} + \cdots\right),$$

While this formula is significantly longer than the original formula, it should be noted that once implemented, it will be relatively easy to compute as well as completely abstracted away from both me as a developer and the user. It therefore follows that this formula acts as a very feasible solution, to overcome the aforementioned limitation.

Robustness

In order to overcome this final limitation, I could produce the program in two versions, an offline version (executable file) and an online version (web page). Each version clearly has its individual benefits and tradeoffs. However, by giving users a choice, the best version according to the individual set of priorities for the user can be chosen and hence the limitation of forcing users who would prefer to use an offline version to use an online version can be overcome. This solution does not require any additional reprogramming, so can be considered ideal.

Maintenance

The final piece of my evaluation will be to assess the maintainability of my program. This can be done in terms of: corrective maintenance, adaptive and perfective maintenance, as follows:

Corrective

As seen in all of the screenshots taken throughout this project, all code is appropriately commented. Furthermore, through the use of several isolated procedures, tested in isolation, the code is very modular. I believe this all contributes to the code being very well structured and as a result easy to maintain.

Adaptive

As the program has been written in Unity, it can be run on any device regardless of operating system. As long as the device has Unity, the unbuilt file can be opened and modified, making the code easy to adapt and hence maintain.

Perfective

Finally, I have taken measures to make the code run more efficiently where possible, for example in the procedure Damping() where I reduced the number of calculations made by setting the maximum angle to 0 after a certain value had been reached. However, certain procedures could potentially be written more efficiently throughout the program.

As each procedure has been written and tested in isolation, perfective maintenance and improvements to these procedures is made easy, as procedures can be improved and tested continuously with confidence that all other procedures remain unaffected by this process.

This concludes my project. The built and unbuilt files are included in submission.

Nishad Deokar
Candidate Number: 6296
Centre Number: 52131

Source code

https://drive.google.com/drive/folders/1FKqzivAHQ9bRmz_zrgmb5T06rQRwvVJU?usp=sharing