**GMAPS**

[A picture containing text, table, vector graphics

Description automatically generated](https://soundcloud.com/user-879892202)

**Dynamics**

**& Kinematics**

**Worksheet**

**To see the additional comments and resources, make sure you select All Markup in the Review/Tracking pane**

A picture containing graphical user interface

Description automatically generated

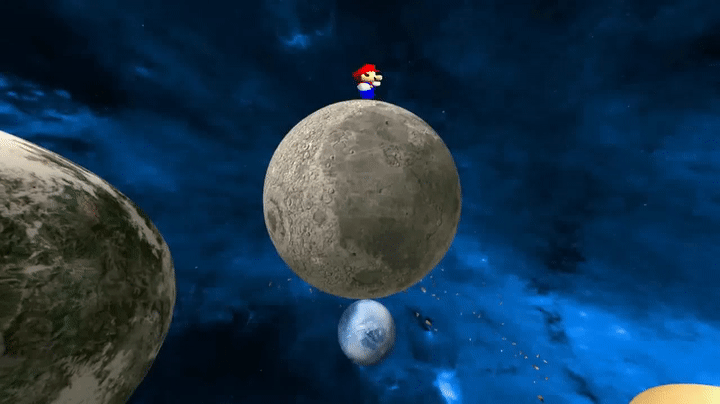
It makes sense to use Unity’s physics engine for much of your game development. So why bother to learn how to code physics yourself?

Well, education isn’t just about doing – it’s also about **understanding**.

If you understand how something works, you become much better at using it. When things go wrong, you can solve problems more quickly.

There’s actually a high chance that you might have to write your own physics for some projects.

You’ve already implemented this!



Or maybe you need to calculate the path of a projectile?

[](https://mathematica.stackexchange.com/questions/86255/animating-a-3d-shape-along-path-a-2d-and-3d-parametric-function-curve-update)

As you can guess, it’s all math!

Diagram, schematic

Description automatically generated with medium confidence

**Some History - Aristotle**

Aristotle was a philosopher in Ancient Greece, born in 384 BC. He is one of the greatest thinkers in human history, influencing Politics, Science and [Philosophy](https://plato.stanford.edu/contents.html) for many centuries.

Graphical user interface

Description automatically generated with medium confidence

Aristotle took a very common-sense view of motion, but which turns out to be wrong (like many “common-sense” things in life).

Aristotle thought that heavenly objects (e.g. planets) move forever in circles without any external force acting on them.

He thought that an earthly object will only move when a force is applied to it – and if the force is removed, the object will come to a stop.

Aristotle also believed that when objects were moved by people, this was due to “unnatural force” (e.g. an artificial force). And when the unnatural force ran out (like running out of energy), then “natural” motion would take over again.

[This](https://science.howstuffworks.com/innovation/scientific-experiments/10-things-we-thought-were-true-before-scientific-method9.htm) is how the trajectory of a cannonball was shown in 1561.



Aristotle supposed that it would move in a straight line (due to the unnatural force), and then would fall straight down (due to a different, natural force).

Then along came Isaac Newton, who totally demolished Aristotle.



Aristotle’s views were eventually overthrown by one of the most influential scientists of all time, Isaac Newton, whose Three Laws of Motion are considered to be a correct model of how the universe works.

Isaac Newton was a complete 17th Century rock star scientist – though he had periods of madness when he accused everyone of plotting against him!

We’ll look at Newton’s three laws, and see how these relate to Unity Physics, and how we can implement them ourselves.

**Part 1 : Basic Dynamics & Kinematics**

**Newton’s 1st Law of Motion**

**An object at rest stays at rest and an object in motion stays in motion with the same speed and in the same direction unless acted upon by an unbalanced force.**

This is also called the **Law of Inertia**.

Okay, let’s see how this works in Unity.

A picture containing text, clipart

Description automatically generated

Download **03\_KINEMATICS\_Worksheet.zip** from LMS. Unzip, and add the folder to the **Assets** folder in your worksheets project.

Go to the **Part 1** folder.

The **FirstLaw.cs** script is provided.

Add a **Vector3** public property called **force**, so that the force value applied to the sphere can be set in the Inspector.

In the Start() function, apply force **just once**. Apply different values along X, Y, Z each time you run it to see what happens.

Print out the position of the sphere after every frame.

Graphical user interface, application

Description automatically generated

You should see something like this:

Graphical user interface, text

Description automatically generated

The force was only applied once, in Start().

Does this follow Newton’s First Law? Explain.

In reality, we can’t get away from forces. There’s gravity, air resistance, the wind blowing, and so on. To get away from all those forces we’d have to fly into space.

But in a game, we control what happens. So, the example above simulates a world with no forces acting on an object except for the forces we program.

[](https://phys207-project.physics.wisc.edu/ann-seliger/newtons-1st-law-of-motion/)

A picture containing clipart

Description automatically generated

Now, create a second sphere called **Ball2** that does *not* have a rigidbody.

Add a script called **Motion.cs** that moves the ball in the same way as **Imp1**, but using just your code and not Unity physics.

Add this code to **FixedUpdate()**.

Chart

Description automatically generated with medium confidence

Text

Description automatically generated

So far, you’ve just been moving an object by changing its position based on applying a force once, or simply setting the object’s velocity and moving the object based on that. But what about other more complex problems that you might have when developing a game?

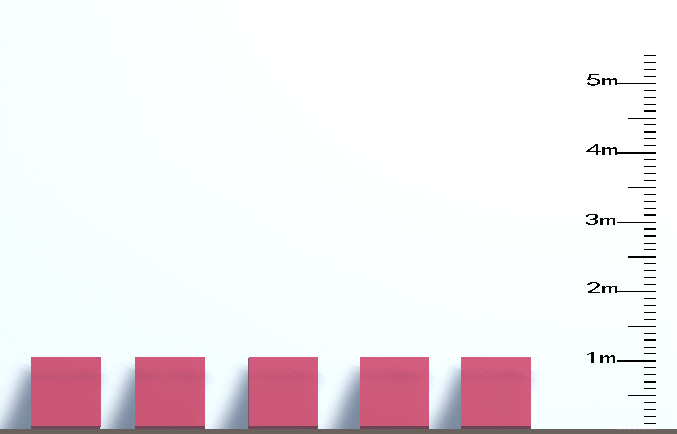
A picture containing company name

Description automatically generated

One problem you might face is how to make a character jump to a certain height.

We can also solve this using one of the SUVAT equations!

A picture containing table

Description automatically generated

Here’s the basic math for how it’s done:

v\*v = u\*u + 2as

u\*u = v\*v - 2as

u = sqrt(v\*v - 2as)

So, open the **JumpToHeight** folder, and create a new file called **JumpToHeight.cs**.

Complete the code below to make each cube jump to a different height (each cube’s individual jump height is set by **Height** in the Inspector.

Add this script to each cube in the scene, and set its jump height.

Press the space bar to make the cubes jump.

Timeline

Description automatically generated

Text

Description automatically generated

**Part 2 : Shooting Pool**

Now we’ll figure out how to shoot a pool ball with a pool cue! We’ll do this without using any Unity Physics APIs.

A cartoon character with horns and pink ears

Description automatically generated

Open **Part 2 Pool** in your **GMAPS\_Oct\_2023\_Worksheets STUDENT** project.

First, we need a class to represent a 2D ball. For now, we’ll create a simple class that responds to the change in velocity caused by interacting with a pool cue.

The pool cue is really just the mouse. We click on the ball, then drag and release the mouse to set the ball’s velocity. The black line below is the “pool cue”, which we’ll draw as we drag the mouse.

Text, timeline

Description automatically generated with medium confidence

So, to create the effect of a pool cue hitting the ball, we must:

* Check if the mouse position is inside the ball
* Draw a line from the mouse to the ball, to represent the ball’s velocity after releasing the mouse.
* Set the velocity of the ball
* Move the ball according to its velocity

1. First, we’ll create a ball prefab.

* Open the **Pool** scene.
* From the **Sprites** folder, drag **ball** into the scene. Make sure its transform is reset to all zeros.

Graphical user interface

Description automatically generated

The ball should appear at the bottom left of the table.

Shape, rectangle

Description automatically generated

* Drag the ball object from the Hierarchy into the Prefabs folder.
* Drag the ball so that it sits somewhere near the centre of the table. Rotate the camera around a bit. You might find that the ball isn’t visible from some angles.
* This is due to the order of the ball sprite in the rendering layer.

See if you can figure out how to fix this so that the ball is always visible above the table (hint, check the ball’s **SpriteRenderer** component).

Ok, the ball game object is in place. Now to create the script.

* Create a C# script called **Ball2D.cs**. Add this script to the ball prefab. If you’ve done it correctly, you should see the script as a component on the ball object in the scene.

We need to keep track of the ball’s *position* and *velocity*. Both of these will be represented as 2D vectors, using **HVector2D**.

* In Ball2D, declare two public variables called **Position** and **Velocity**. Set them to be new HVector2D objects with x=0 and y=0.



Set the actual Position.x and Position.y values in Start.

Text

Description automatically generated

* Declare a public float variable called **Radius**.

To set the value of Radius, we need to know the sprite size. Add this code to the Start function:

Text, Word

Description automatically generated

This should be fairly obvious, apart from the **pixelsPerUnit** property.

In the last line, since the sprite is square shaped, we can just divide the x value (which is the sprite’s width) by 2 to get the radius of the ball.

We now have a ball object in the scene, with a Ball2D script attached to it.

**Text

Description automatically generated**

1. In the diagram below, we see that if the distance from the ball’s centre position to the mouse is *less than or equal to the ball’s radius*, then the mouse position is *inside the ball*.

Diagram

Description automatically generated

How to calculate the distance from the ball’s centre position to the mouse position?

One way would be to create a vector from the ball to the mouse, and then calculate the vector’s magnitude. But that would require creating a new object, when all we need is a simple mathematical calculation.

The diagram below should help you figure it out using basic trigonometry.

Shape

Description automatically generated with low confidence

Text

Description automatically generated with medium confidenceWe’ll write a function to do this. But finding the distance between two objects isn’t really part of any specific object’s functionality, so we’ll create a general *utility* script for this.

Create a class called **Util.cs**. Add a static function called **FindDistance**, as shown below. Complete the missing code.

A picture containing graphical user interface

Description automatically generated

A static function applies at the class level. So, to calculate the distance between two HVector2D objects, we write:

Text

Description automatically generated with medium confidence

Test your code to make sure it works properly!

1. A picture containing text

   Description automatically generatedNow to check if the mouse position is inside the ball. Another way of saying “inside the ball” is ***colliding*** with the ball. So, we’ll write a function in Ball2D called **IsCollidingWith**, as shown below.

Complete the missing code.

Graphical user interface

Description automatically generated with low confidence

1. To check if the mouse position is inside the ball, we make use of our answers to (a) and (b).

Where should we put this code, though?

Inside Ball2D?

Text

Description automatically generated with medium confidenceBut checking if the mouse is inside the ball is really part of the pool cue’s functionality.

So we should create a new class called **PoolCue.cs** and put the code in there.

Timeline

Description automatically generated with medium confidence

Add this script to the **PoolCue** object in the scene.

The **LineFactory** and **Ball** objects in the scene should be dragged into the relevant slots in the Inspector.

**🡪 IMPORTANT!** Run the game. What happens? The line keeps getting drawn even after the mouse is released! How to fix this? Basically, you also need to check if the mouse button is up. If it is, disable line drawing and set the **drawnLine** object to null.

Do that now.

Okay, we’ve solved the first two problems:



* Check if the mouse position is inside the ball
* Draw a line from the mouse to the ball, to represent the ball’s velocity after releasing the mouse.

Now for the other two:

* Set the velocity of the ball
* Move the ball according to its velocity

**Text

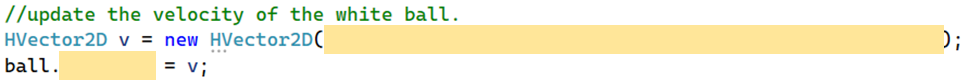
Description automatically generated**

1. The **Ball2D** script has a **Velocity** property. We need to set this to the correct value to make the ball object move.

A picture containing text

Description automatically generatedThis velocity value is set when we release the mouse while drawing the cue. The velocity is actually the vector from the mouseup position to the ball position. So, we need to set this velocity value when the mouse is released. Where in your code is this done?

Add this line of code somewhere in PoolCue.cs.



Run the game. Print out the ball’s velocity. Does the ball move? Nope! We still have to add that code to the Ball2D class.

1. Add this code to **Ball2D**.

Graphical user interface, text, application, email

Description automatically generated

It’s always good practice to separate your code into specific functions. In this case, we define a function for updating the ball’s physics.

* Text

  Description automatically generated with medium confidenceComplete the physics code below.

Chart, bar chart

Description automatically generated

You can now move the ball by using only the mouse as a pool cue and your knowledge of Newtonian dynamics and kinematics!

**Text

Description automatically generated**

**PART 3: Reflection**

1. What was the most difficult part of this worksheet. Explain why you found it difficult, and how you managed to overcome this (if at all).

*You might like to reference specific lecture slides and lines of code, give some URLs and diagrams as references, etc.*

The most difficult part of this worksheet for me is coding the JumpToHeight.cs. I have chosen not to follow the formula given by the comments and instead try to understand how the calculation works instead. This is because I did not like following the given hint and completing the question with ease, without understanding what I have been typing. Hence, I took my time to try and resolve my lack of understand on SUVAT. This is because I was struggling to understand what each equation of SUVAT is for and how each of them is derived. All this difficulty was overcome once I understood the 5 equations of SUVAT, know where, when and how to use them depending on the context to achieve the effect I wanted. After understanding the use of the different equations, it allowed me to complete the formula and calculations in the worksheet to create the intended effect for JumpToHeight.cs without blindly following the hints given in the worksheet.

1. Choose a kinematics and dynamics topic that are you still unsure about. Explain that topic, and the steps you will take to clarify your doubts. You will be graded on how well you analyse your problem with understanding that topic.

If you are confident about all the topics, select what you consider to be the most difficult topic and explain ONE more advanced concept related to the topic. You will be graded on the complexity of the concept, and how well you explain it.

*You might like to reference specific lecture slides and lines of code, give some URLs and diagrams as references, etc.*

A topic that I am still unsure about for the kinematics and dynamics topic is the part on SUVAT. Although I understand when and where to use each of the 5 different equations. I still do not understand how the equations are derived, as googling and inquiring ChatGPT about the 5 equations does not really help me to understand. Google and ChatGPT just tells me that the formulars are derived from using integration, which is a topic that that I am unfamiliar with. This is because I was not taught on this topic for secondary school normal academic level, thus leading me to learn differentiation from scratch to understand what integration is. However, all of these are self-study, and I do not have a concrete way of proving that I clearly understand what differentiation and integration is. In the end, all I could do is try and guess how does the 5 SUVAT equations is derived through drawing diagrams for myself to visualize the different equations and how they can be derived from there.

1. Write a reflection about this worksheet. You may like to include how effective it is to motivate and help you learn, how well it relates to games development, etc.

This worksheet got me to do my own self-study and research to understand the concepts, instead of just sitting in the class and thinking that I understand the concept. This is because when there is a time to apply the concepts I see and hear in class, I realized that I do not really understand the concept until I did some self-studying. This worksheet is effective at teaching how to make my own gravity and movement system so that I can more flexibly adjust the game physics to fit my need. This is very useful, as I would otherwise always just blindly take Unity’s Rigidbody or CharacterController physics then following an online tutorial on how to achieve the effect I want. Not fully utilizing the possibility that I can create a unique game physics without making use of the mathematics taught in this worksheet. This worksheet is also very effective in teaching how to code in a more systematic manner, by testing out each methods functionality before going onto create the next method that the current method is supposed to be implemented in. Before this worksheet, I would frequently hop from 1 script to another, just to meet with the fact that I did not properly implement the method in the previous script.