[1. Analysis 3](#_Toc161739550)

[1.1 Problem Identification 3](#_Toc161739551)

[1.2 Research 3](#_Toc161739552)

[1.2.1 Stakeholder Research 3](#_Toc161739553)

[1.2.1.1 Stakeholder Identification 3](#_Toc161739554)

[1.2.1.2 Stakeholder Interview 4](#_Toc161739555)

[1.2.1.3 Survey Research 5](#_Toc161739556)

[1.2.2 Existing Simulation 6](#_Toc161739557)

[1.2.3 Existing Code Research 7](#_Toc161739558)

[1.2.4 Programming Research 8](#_Toc161739559)

[1.2.4.1 Pygame Basics 8](#_Toc161739560)

[1.2.4.2 User Inputs with Pygame 9](#_Toc161739561)

[1.2.4.2.1 Buttons 9](#_Toc161739562)

[1.2.4.2.2 Sliders: 10](#_Toc161739563)

[1.3 Objectives 12](#_Toc161739564)

[1.4 Initial Modelling 12](#_Toc161739565)

[1.4.1 Flowcharts 12](#_Toc161739566)

[1.4.2 Screen Layout 13](#_Toc161739567)

[1.4.3 Calculations 14](#_Toc161739568)

[2. Design 15](#_Toc161739569)

[2.1 Diagrams 15](#_Toc161739570)

[2.1.1 Hierarchy Charts 15](#_Toc161739571)

[2.1.1.1 Files 16](#_Toc161739572)

[2.1.1.2 Functions 16](#_Toc161739573)

[2.1.2 Class Diagrams 16](#_Toc161739574)

[2.1.3 HCI Diagrams 17](#_Toc161739575)

[2.2 Algorithm description 18](#_Toc161739576)

[2.2.1 Pseudocode 18](#_Toc161739577)

[2.2.1.1 Particle Path Function 18](#_Toc161739578)

[2.2.1.2 Particle find Maximum Bounces function 19](#_Toc161739579)

[2.2.1.3 Run Function 19](#_Toc161739580)

[2.2.2 Flowcharts 21](#_Toc161739581)

[2.2.2.1 Overall Flowchart 21](#_Toc161739582)

[2.2.2.2 Individual Flowcharts for User Inputs 21](#_Toc161739583)

[2.2.2.2.1 Trail Button 21](#_Toc161739584)

[2.2.2.2.2 Velocity Button 21](#_Toc161739585)

[2.2.2.2.3 Obstacle Button 22](#_Toc161739586)

[2.2.2.2.4 Restitution Slider 22](#_Toc161739587)

[2.2.2.2.5 Ball Size Slider 22](#_Toc161739588)

[2.2.2.2.6 Gravity Slider 22](#_Toc161739589)

[2.2.2.3 Individual Flowchart for Play Button 22](#_Toc161739590)

[2.2.2.4 Individual Flowchart for Reset Button 22](#_Toc161739591)

[3. Technical Solution 22](#_Toc161739592)

[3.1 Code 23](#_Toc161739593)

[3.1.1 User Inputs File 23](#_Toc161739594)

[3.1.2 Game Objects File 27](#_Toc161739595)

[3.1.3 Main Game File 35](#_Toc161739596)

[3.2 Objective Completion 42](#_Toc161739597)

[3.3 Notable Algorithms/Features 44](#_Toc161739598)

[4. Testing 45](#_Toc161739599)

[4.1 Test Video 45](#_Toc161739600)

[4.2 Test Table 45](#_Toc161739601)

[5. Evaluation 46](#_Toc161739602)

[5.1 Checking back with Objectives 47](#_Toc161739603)

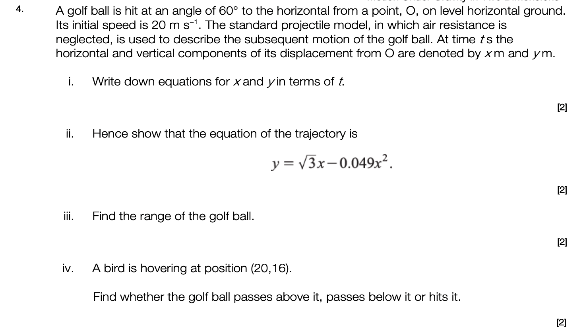
[5.2 Final Check-In with Stakeholder 47](#_Toc161739604)

[5.3 Future Improvements 48](#_Toc161739605)

# Analysis

## 1.1 Problem Identification

In both physics and maths, students do countless calculations concerning kinematic scenarios like projectile motion. Although equations and mathematical working may be enough for some people, others, including myself, struggle to visualise what the calculations we work with would look like in real-life, and therefore struggle with the concept.

On the left is an example of a projectile motion question in which a ball is launched at an angle with a given velocity. This is one of the simpler questions, but for a new student it is still hard to wrap your mind around the calculations you need to do when you have no perception of what the golf ball’s trajectory would look like in reality.

The goal of my project is to provide a bridge between the theoretical side of mechanics and its application in the real world by creating a visual representation of projectile motion. It will consist of a ball being projected at a given angle/velocity so that the user can clearly see the path the ball would travel on. This will ideally help new students study projectile motion by providing them with a simulation showing how the equations would translate to real life.

## 1.2 Research

### 1.2.1 Stakeholder Research

#### 1.2.1.1 Stakeholder Identification

My project is going to be aimed mainly at people who study mechanics, ideally those who have just started learning because it will help the user with visualising problems. My main stakeholder that I will check in with at every stage of my project will be Theo Bahns, an A-Level student who does both maths and physics, as well as being a competent programmer, and is therefore the ideal end-user for my project.

#### 1.2.1.2 Stakeholder Interview

I conducted a preliminary interview with Theo to get an idea of initial requirements for my project. I asked about two different aspects: the appearance and functionality of the project, as well as the programming behind it.

***What mathematical details would you like to see incorporated into the simulation?***

*I would like to see the effects of restitution as we do a lot of that in lessons, so collisions with different coefficients of restitution should be included. Also maybe include calculations with friction or maybe conservation of energy.*

***What about physics details?***

*Things like variable gravities would be cool, and maybe projections with air resistance. Also having obstacles would be useful, and you could have different materials or shapes to project.*

***What user inputs do you think the simulation should consider?***

*Definitely the normal things like the angle of projection and the initial velocity, but maybe also the size of the ball or the acceleration too.*

***What information about the projection do you think should be displayed?***

*It would be nice to see the values of SUVAT (displacement, initial and final velocities, acceleration, and projection time) alongside the projection as maths calculations usually rely on those values.*

***Do you have any suggestions for what the simulation should look like?***

*Definitely use bright colours so everything is very clear because you’ll have a lot of stuff going on, so make sure the ball and everything is noticeable. Maybe also add a game aspect to make it more entertaining.*

***Do you have any suggestions for which Python modules to use for the program?***

*I think the best options are either Tkinter or Pygame but I think you should use Pygame because it has a lot of inbuilt libraries that will be useful for displaying the projection easily.*

***What do you want the user interaction to be like? As in would you prefer a GUI or a web interface, etc?***

*Personally I prefer GUIs because they’re a lot easier for users to work with, but a web interface would also be fine. Definitely not just a command line interface though because you want to create a visual simulation. I’d say GUI.*

***Are there any specific programming tools you would like the project to make use of?***

*If you end up using Pygame there are lots of existing objects that you could use to draw different aspects of the simulation’s display, so I’d definitely look into some of those features. Then I guess you will need to import other generic libraries like math for example because your project involves lots of calculations.*

Initial objectives from preliminary interview:

* include collisions with restitution, friction, and energy
* vary gravity
* include obstacles
* different materials for the particle
* let user change initial velocity, angle, ball size, acceleration
* display suvat values
* use bright colours
* use Pygame
* have a GUI

#### 1.2.1.3 Survey Research

Building on the conclusions from Theo’s interview, I decided to conduct a survey for people in my year to get data from a larger sample and therefore a better idea of the objectives I should focus on including in the simulation, and to see first and foremost if my plan is good.



Thankfully, the survey results from 40 people show that 90% of students have trouble visualising mechanics scenarios, and 68% of them would find a projectile simulation useful, with a further 30% thinking a simulation would be interesting regardless, meaning my project idea would complete the goal I intend. In addition, the results show that the most interesting features I could include in my project would be a game aspect of some sort, exploration into interactions between the projectile and walls/obstacles with restitution, and the effect of different atmospheres (i.e., varying the acceleration under gravity).

Updated initial objectives after the survey as well as the stakeholder interview, based on what both agree on as well as some ideas I personally liked from the interview:

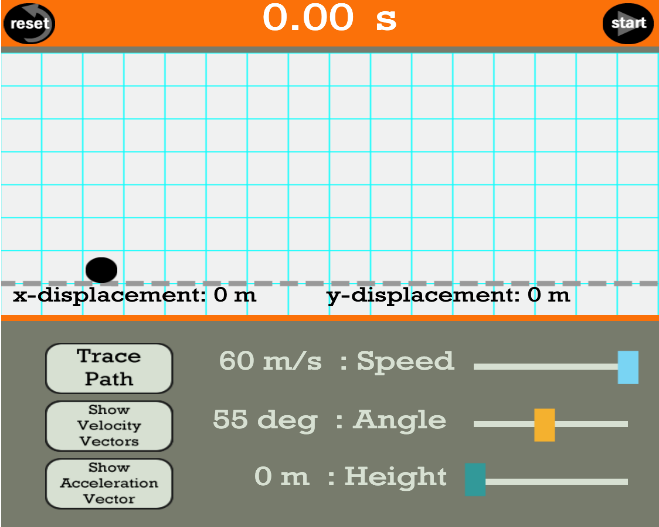
* include a game aspect
* include collisions with restitution
* include variable acceleration due to gravity
* include obstacles
* allow user to edit initial conditions
* display suvat values
* use bright colours
* use Pygame
* have a GUI

After the survey, I checked back with Theo and showed him my conclusions from stakeholder research and therefore what my initial list of objectives (above) was, to make sure he agreed with the plan so far. I then decided to do some research on existing simulations

### 1.2.2 Existing Simulation

As preliminary research, I decided to find some existing projectile simulations to see what features I liked and which I could improve on.

This is an interactive projectile simulator on The Physics Classroom:



Here to see it working: <https://www.physicsclassroom.com/Physics-Interactives/Vectors-and-Projectiles/Projectile-Simulator/Projectile-Simulator-Interactive>

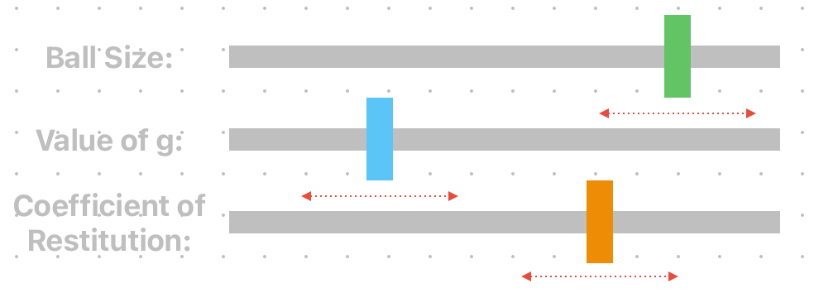
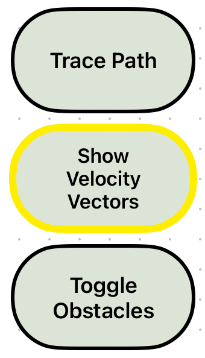
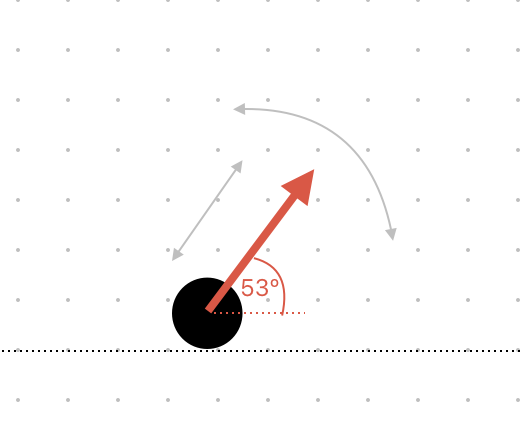
These are the things that the simulation includes:

* A button for ‘Trace Path’, that toggles whether the path of the ball is shown or not
* A button for ‘Show Velocity Vectors’ that toggles arrows showing the separate components of the ball’s velocity
* A button for ‘Show Acceleration Vector’ that toggles an arrow showing vertical acceleration
* Three sliders, one for speed, angle and height, allowing the user to vary the initial conditions
* A timer at the top of the screen that times how long the ball takes to land
* Displayed x-displacement and y-displacement that shows how far the ball moves

Things I will integrate into my project:

* I will allow the user to change the initial angle and velocity by having an arrow coming out from the ball that can be rotated to change the initial angle or extended to change the initial velocity. This is because in the existing simulation it would be nicer if you could see the direction the ball would travel in before it is launched.
* I like the idea of the sliders, but instead of changing speed, angle and height, I will change the size of the ball, the magnitude of acceleration due to gravity, and the magnitude of the coefficient of restitution.
* I will have buttons for path tracing and velocity vectors, but I will also add one for toggling obstacles.

Initial drawings I made for what my user inputs/display will look like:



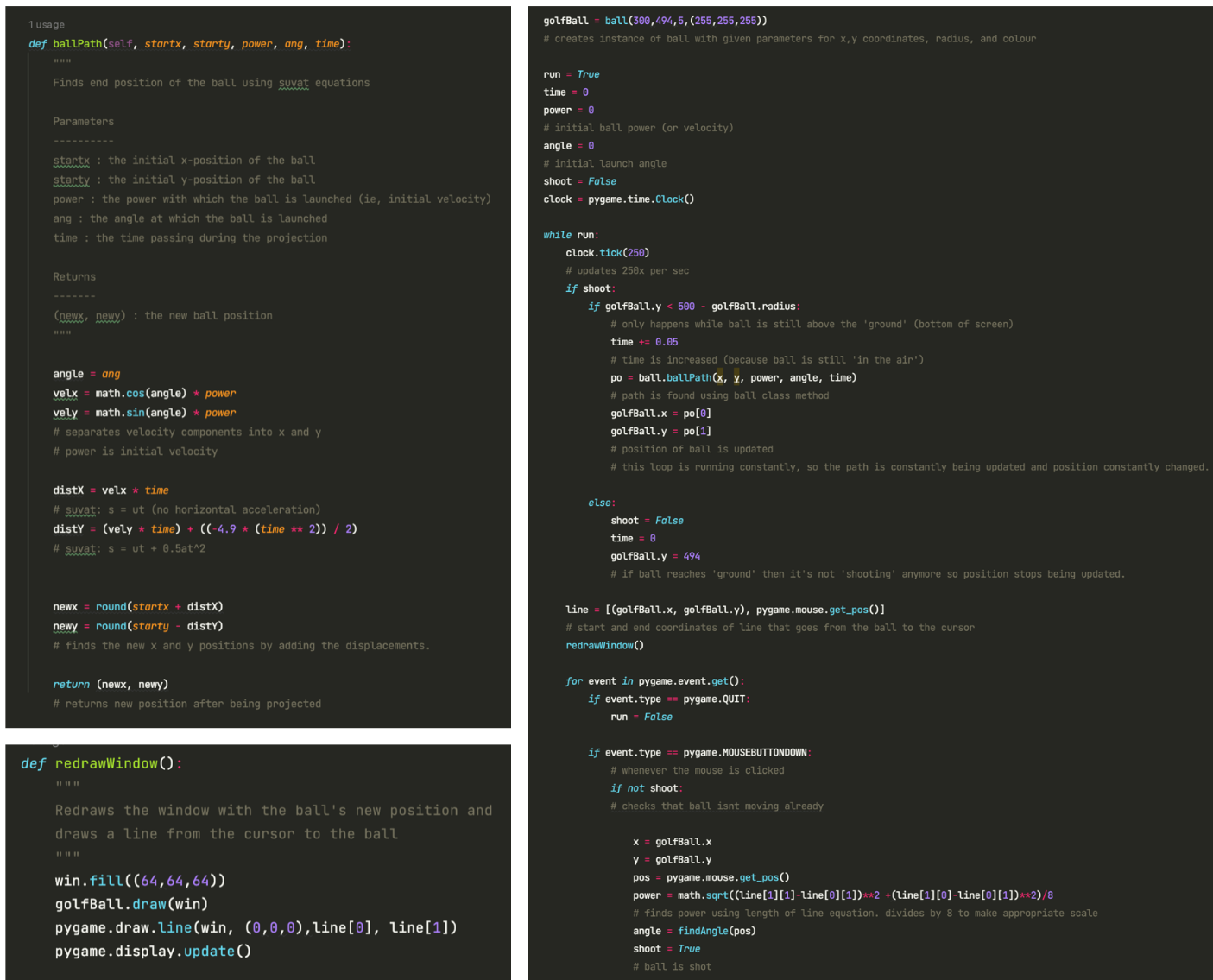
### 1.2.3 Existing Code Research

I then decided to do further research into the code behind projectile simulation by watching some tutorials on YouTube. I found someone who wrote a program that creates a ball that bounces around.

Tutorial Link: <https://www.youtube.com/watch?v=_gDOz7E6HVM>

Link to code being run: <https://screenpal.com/watch/cZnIDZVdk1R>

Attached to the video was a link to his code on GitHub, so I decided to go through the code and comment it so I could clearly see what was going on. This was very useful as it included lots of the same logic that I need in my own program, and it uses Pygame, so it helped teach me some techniques I can use later. Here are some screenshots of the main parts of the program which I commented: the method that calculates the projection path, the main running loop, and the function that displays the simulation.



Things included in this program:

* A Ball class with attributes: x, y, radius, and colour and methods: draw() and ballPath().
* A line that connects the cursor to the ball which allows the user to aim the launch
* A loop that constantly updates the position of the ball if it has been launched
* The length of the line from the cursor to the ball determines the initial power or velocity, so the longer the line, the more forcefully the ball is launched.

Things I will keep or change in my own program:

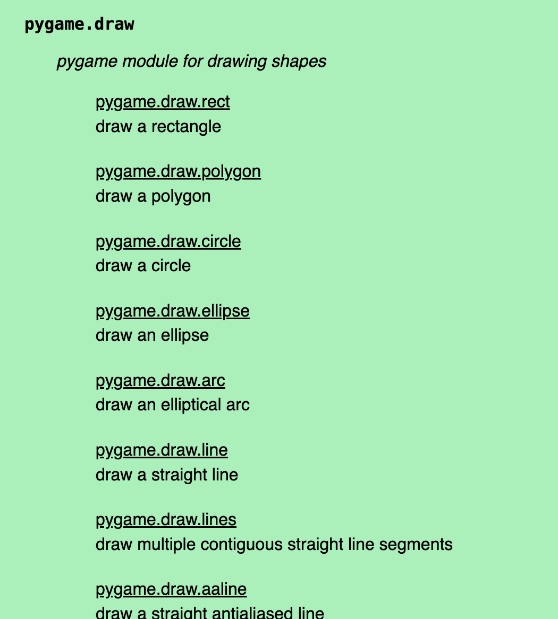
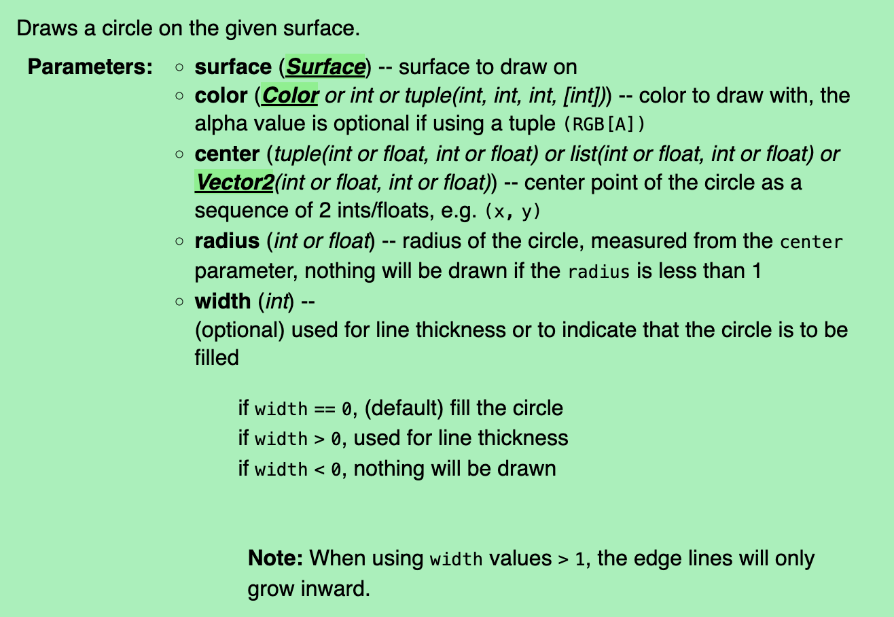
* Will use the Object-Oriented approach but will have more than one class, eg, one for obstacles and buttons, etc.
* Will keep line connecting cursor to the ball as it makes it easier to see where you are aiming
* Will use similar equations to find the path of the ball but will incorporate restitution
* Will make it so that the ball cannot leave the window – ie, make it bounce against the walls.
* Will add user inputs

### 1.2.4 Programming Research

#### 1.2.4.1 Pygame Basics

Seeing as my project is going to use pygame, which I haven’t used before, I decided to do some research and to work with some tutorials so I could get acquainted with the library.

I found this website: <https://www.pygame.org/docs/ref/draw.html>, which has summaries of all the functions in the pygame library, detailing their parameters and what they will do.

I looked into functions that I would probably use frequently, for example, pygame.draw(), especially for drawing a circle to represent the projectile. I made a list of some of the pygame functions I might need, writing what I could use them for:

pygame.display.set\_mode - initialises a window to display

pygame.display.update or pygame.display.flip - updates display surface

pygame.draw() - draws objects: for creating ball, obstacles, etc

pygame.display.set\_allow\_screensaver - sets when the screensaver runs: could be set when user inputs

pygame.image() - handles images: could be used for background

pygame.key() - handles keyboard: could be used for user input

pygame.time() - handles realtime: could be used for motion calculations

#### 1.2.4.2 User Inputs with Pygame

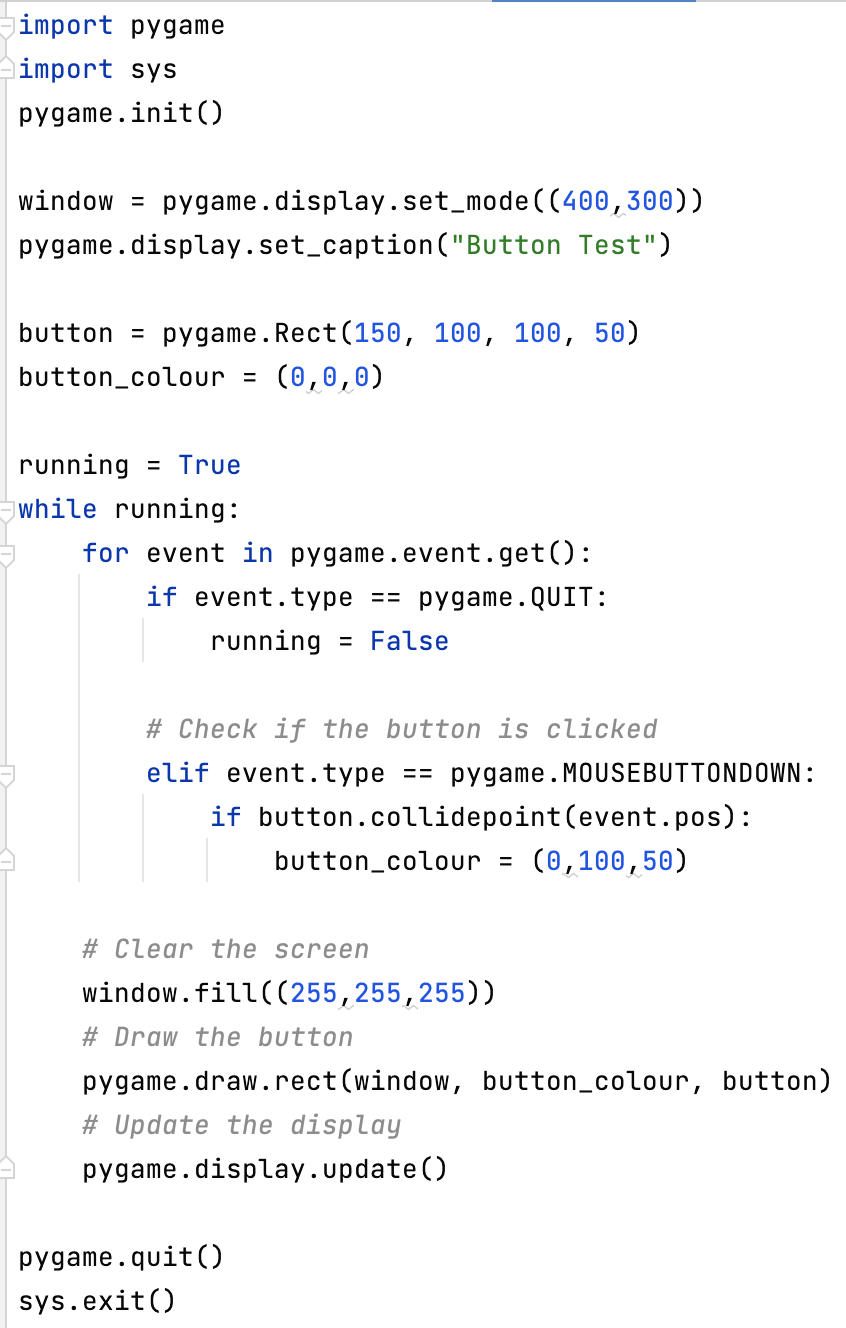
I also decided to do some research into handling user inputs with Pygame. Because of the existing simulation I looked at, I decided to look into buttons and sliders. Aside from helping me choose my preferred methods for obtaining user inputs for my project, this also helped me get more used to using Pygame.

I created prototype code for buttons and sliders and used the Rect() pygame object for both. Using Rect() was so helpful because it has inbuilt methods that check whether the cursor intersects with the object (for example, Rect.collidepoint(), which returns whether the cursor is inside the shape), which makes it easy to check if the user clicks on a button as all I have to add is an extra check that sees if the mouse is pressed down. I think this will come in very useful when I add obstacles to the simulation as I can make them Rect objects and therefore easily check whether the particle collides with them.

##### 1.2.4.2.1 Buttons

Here is the prototype code that creates a button that changes colour when clicked by checking whether the mouse intersects with the button, which is a Rect instance. As said above, using Rect’s inbuilt methods, creating a button was pretty simple, so I will definitely have buttons in my actual project code.

Here is the commented code and a link to a video of it working:



Link to code working: <https://screenpal.com/watch/cZnIDrVdkhd>

In my project I will probably have a Button Class with a ‘button\_is\_pressed’ attribute so I can regularly use buttons for user inputs. Now that I have this code, it will be much easier to implement into my project when I start writing it. As mentioned above, I will probably use buttons to allow the user to toggle things like tracing the path of the ball, the presence of obstacles, and showing velocity vectors for the ball.

##### 1.2.4.2.2 Sliders:

I then created code for a slider instead, also using the pygame.Rect object to represent the slider. This was a lot harder to code than the button, because not only did I have to check if the slider was clicked, but I had to check if it was clicked and dragged to a new value. I created a sliderTrack as part of the slider that creates boundaries for the slider, so it has maximum and minimum values. I also printed the current value of the slider on the screen.

Here is the commented code and a link to a video of it working:



Link to code working: <https://screenpal.com/watch/cZnIbKVdkOX>

Even though the code for a slider is more complicated than for a button, I think it is essential for my project as it allows the user to input specific values rather than only binary ones (as a button allows), which will be helpful for inputting things such as values for acceleration due to gravity and coefficient of restitution. As with the button, I will also create a Slider Class from this prototype code so I can easily create multiple instances of it in my project.

## 1.3 Objectives

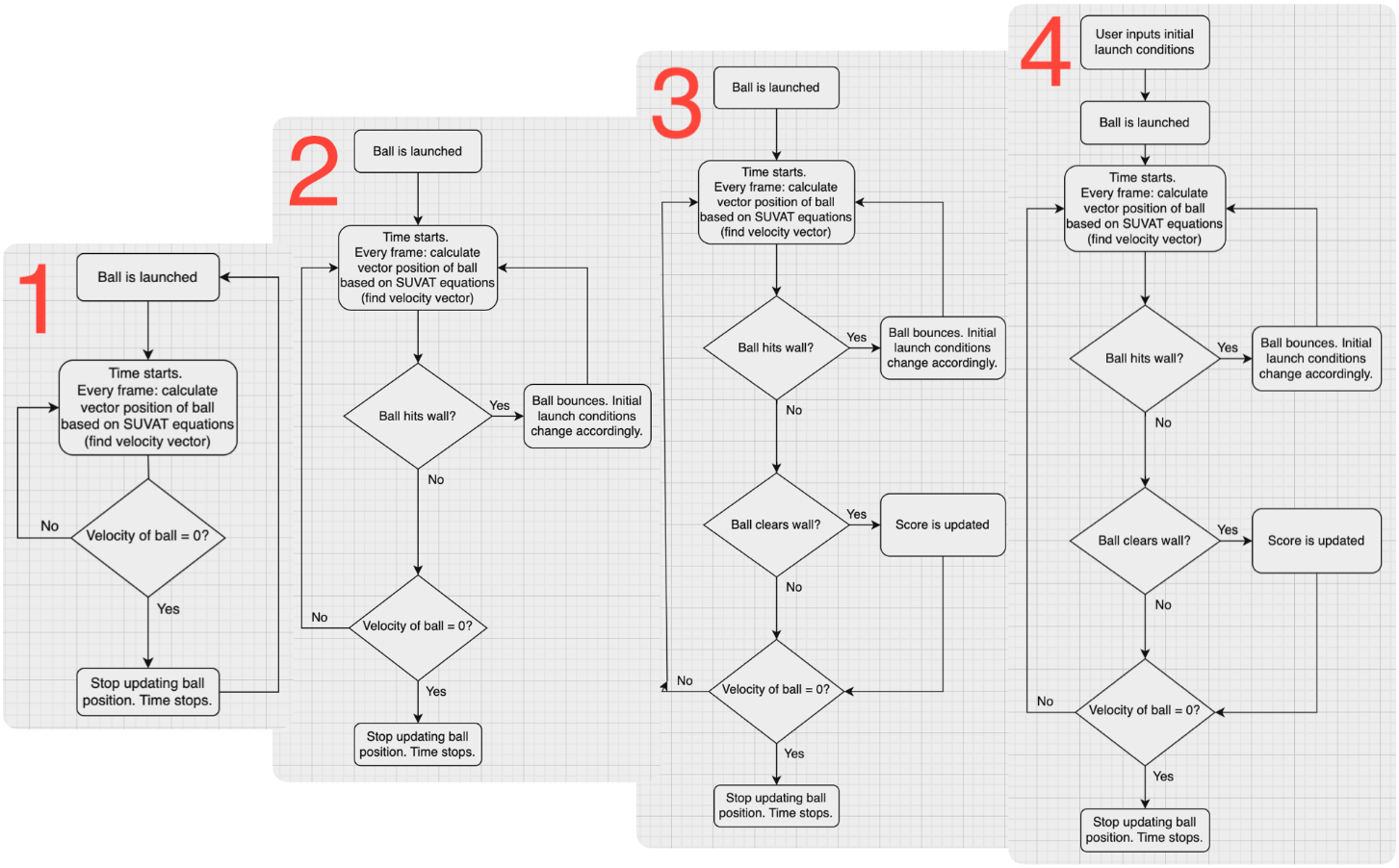
After plenty of research, I had enough ideas to construct my list of primary objectives for my project, centring on what it will do. For each objective I then explained further the separate things I can do to improve its level. In red are the programming techniques I could use for each aim.

1. Create a ball that is projected across the screen
   1. Create a particle class using OOP
   2. Use equations of motion with constant acceleration to find the ball’s path
   3. Show velocity vector as it moves
2. Have a GUI to display projection
   1. Use Pygame to create a window when code runs
   2. Close the window when program stops running
3. Allow user to edit initial values
   1. Create buttons and sliders for user inputs
   2. Edit initial velocity
   3. Edit projection angle
   4. Toggle velocity arrows
4. Add a game aspect
   1. Create target class for ball to hit
   2. Add points system
   3. Add timer/countdown using Time import
5. Add interactions with walls
   1. Add bouncing using Rect objects
   2. Use restitution to find velocity after bouncing off a wall
6. Add obstacles the user must project the ball over:
   1. Create obstacle class
   2. Make the obstacles move
   3. Allow user to decide whether or not obstacles are present
7. Add different atmospheres
   1. Allow user to input atmosphere, eg, the Moon, Jupiter, and use the according values of acceleration due to gravity.
   2. Change background of screen depending on atmosphere
8. Display important values of the projection on screen
   1. Display all current SUVAT values
   2. Use Pygame to display text on screen
9. Make all aspects of the simulation distinguishable and clear
   1. Use bright colours for different objects, eg, particle, walls
   2. Use clear font to display text and values

## 1.4 Initial Modelling

### 1.4.1 Flowcharts

Here are some flowcharts I made based on my initial objectives to get a better idea of what my program will look like.



1: Basic initial loop showing what will happen when the ball is launched.

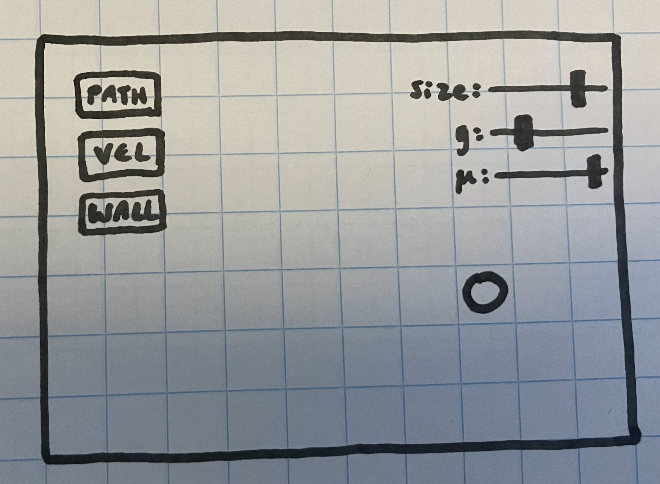
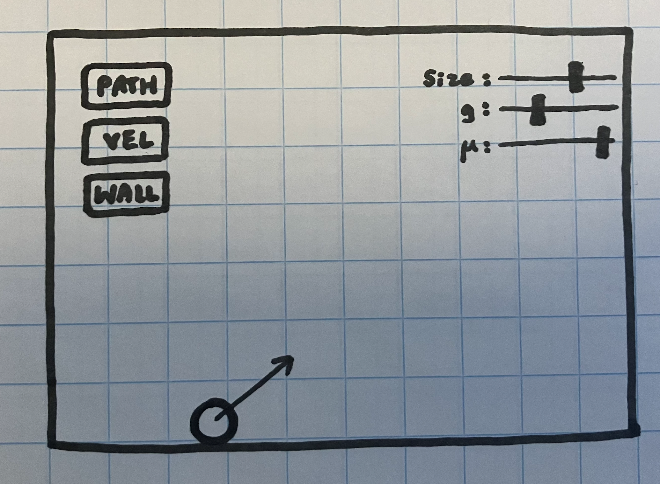
2: Added a loop that checks whether the ball hits a wall and bounces it off if it does.

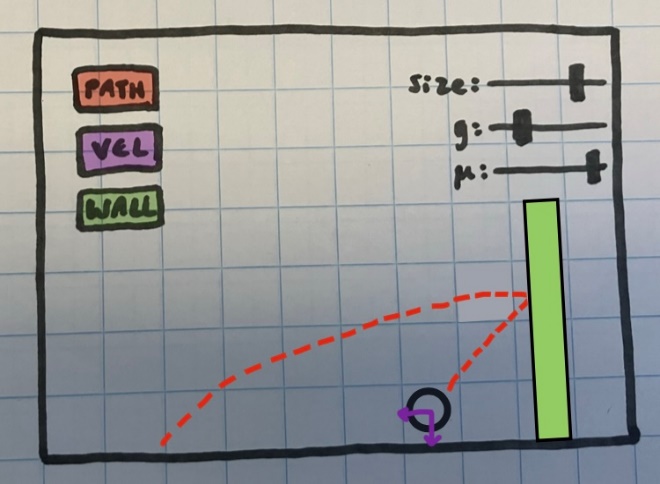
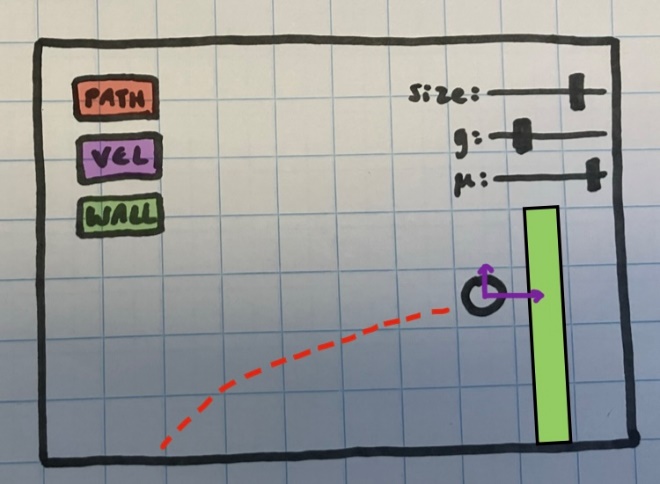
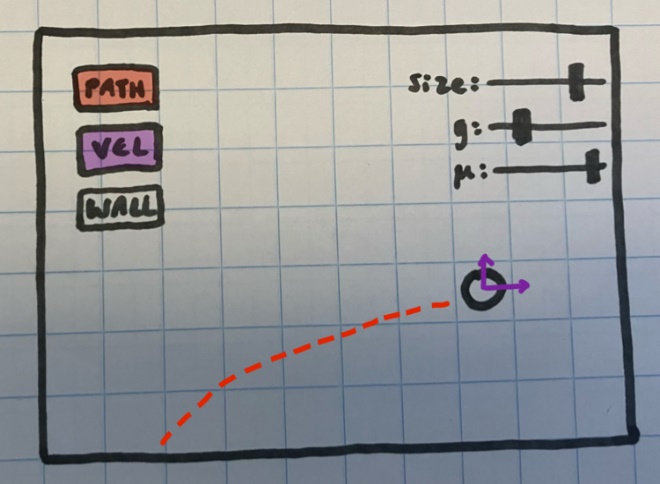
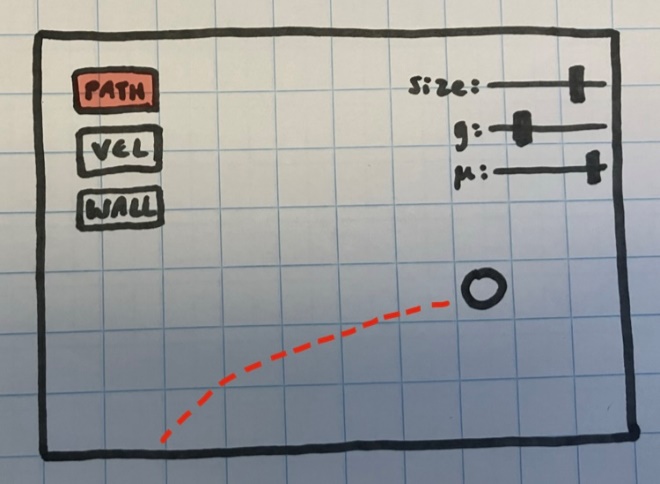
3: Added another loop that checks if the ball clears the obstacle/wall and updates the score if it does. This would be for the potential game aspect if I choose to incorporate it.

4: Added an extra start step that considers user inputs before launching the ball.

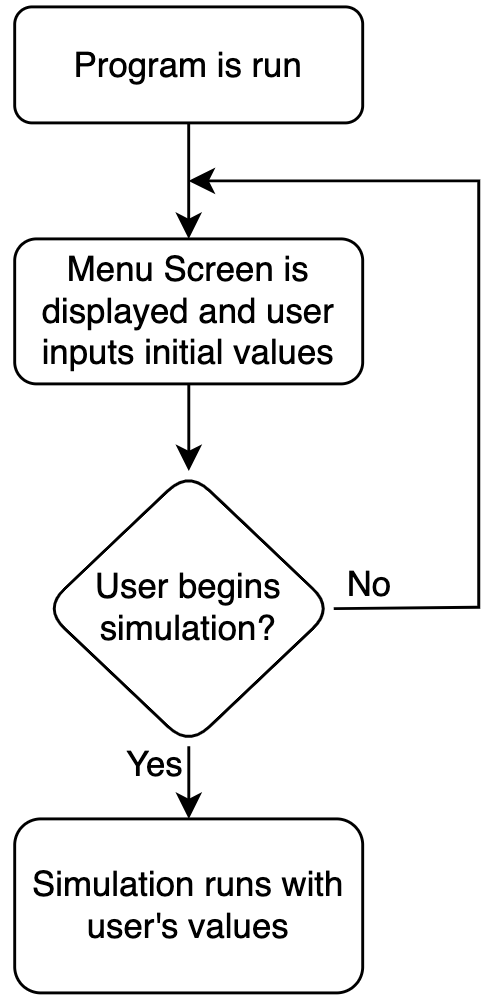
### 1.4.2 Screen Layout

Here are initial diagrams of what I want the screen to look like based on the objectives I have set.





This shows what the projection will look like based on which buttons have been pressed. I have also used bright colours in the drawings to have everything displayed clearly like the stakeholder requested.



I showed Theo these drawings to ask him what he thought and on the whole his response was positive, however, one suggestion he made was to create a menu screen that appears when you initialise the program so that the user can input all the initial conditions and then run the simulation. This would make the simulation screen a lot clearer as it wouldn’t be crammed with all the user inputs as well as the projection. On the right is a flowchart of how this menu screen would be incorporated:

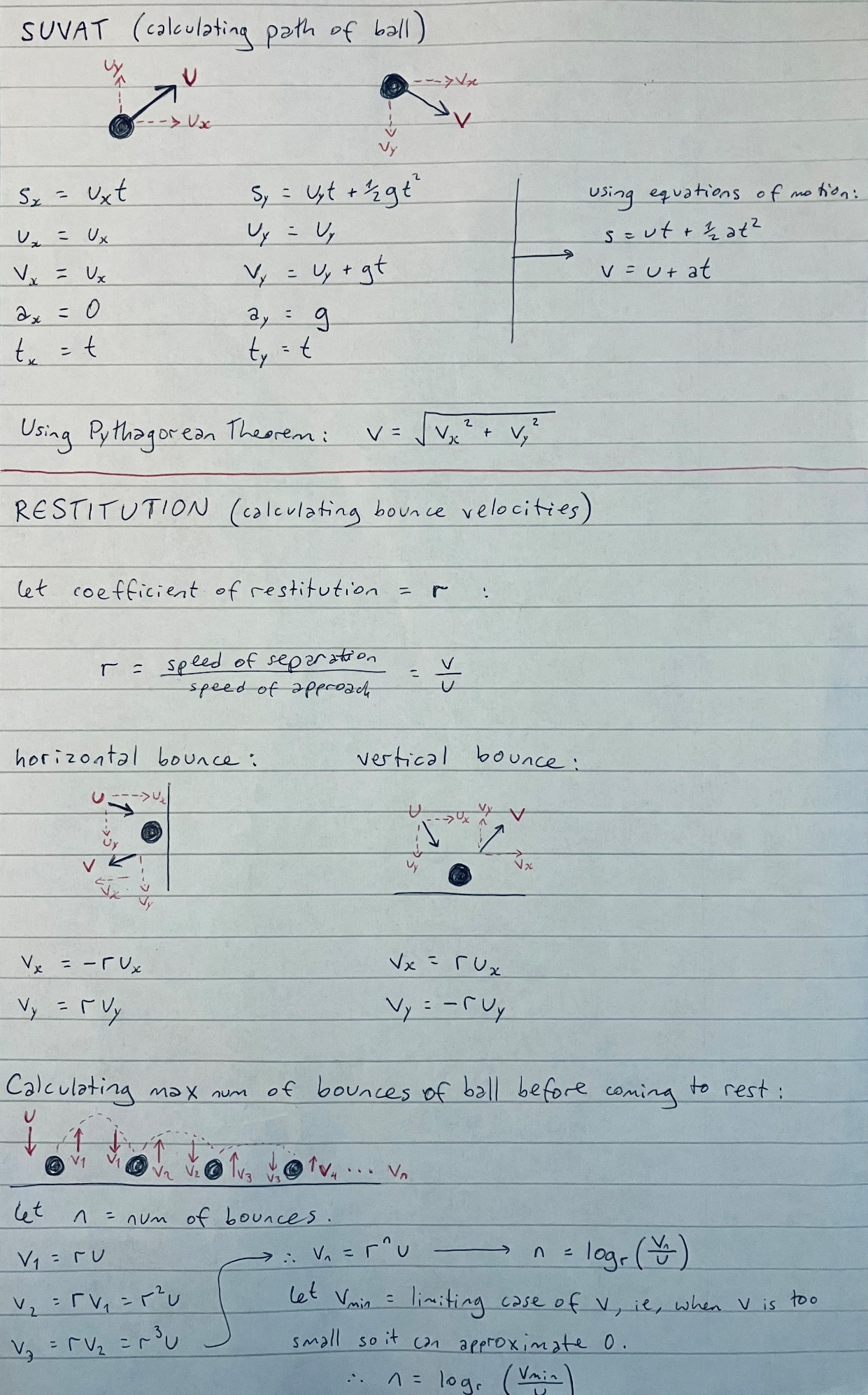
### 1.4.3 Calculations

Next, I decided to plan out the calculations I would have to use in my project. These are the calculations I performed and what they will be used for:

SUVAT: the equations of motion. These will be used to calculate the path of the projectile by finding its displacement in a given time, which will then determine where it will be displayed on screen.

RESTITUTION: this will be used to find the velocity of the particle after it bounces against a wall or obstacle based on its initial velocity.

Here is the actual maths I did:



From these, I found equations for the velocity and displacement of the particle at any point in a projection, which I will implement into my running functions.

# Design

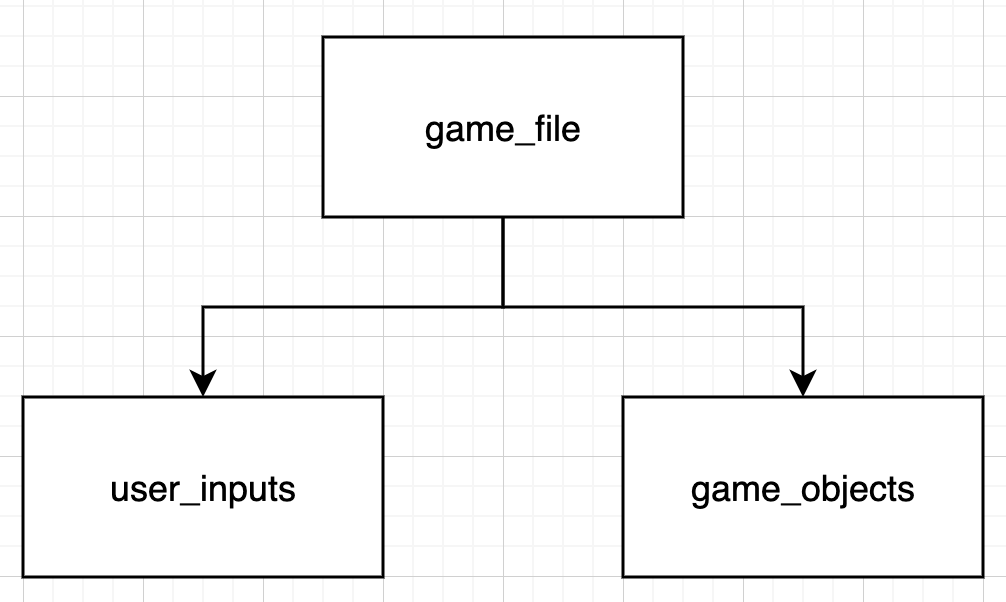
## 2.1 Diagrams

### 2.1.1 Hierarchy Charts

I created some hierarchy charts to show visually the different relationships between files and functions in my project.

#### 2.1.1.1 Files

This is a diagram showing the files that are ran in my project:



game\_file: creates and runs the game using the imported inputs and objects.

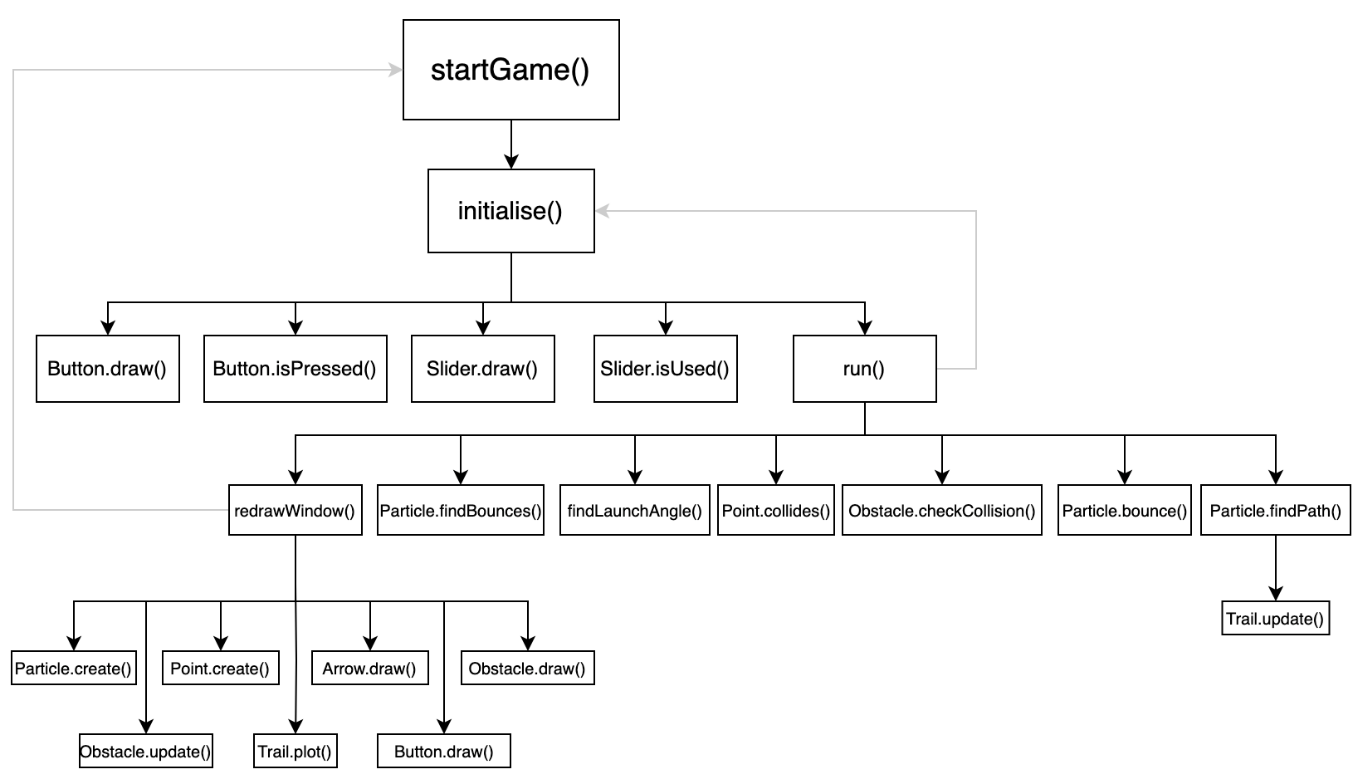
user\_inputs: creates buttons/sliders to take in user inputs

game\_objects: creates objects for the game, eg, the particle.

Both the user\_inputs and the game\_objects files are imported into the main game\_file.

#### 2.1.1.2 Functions

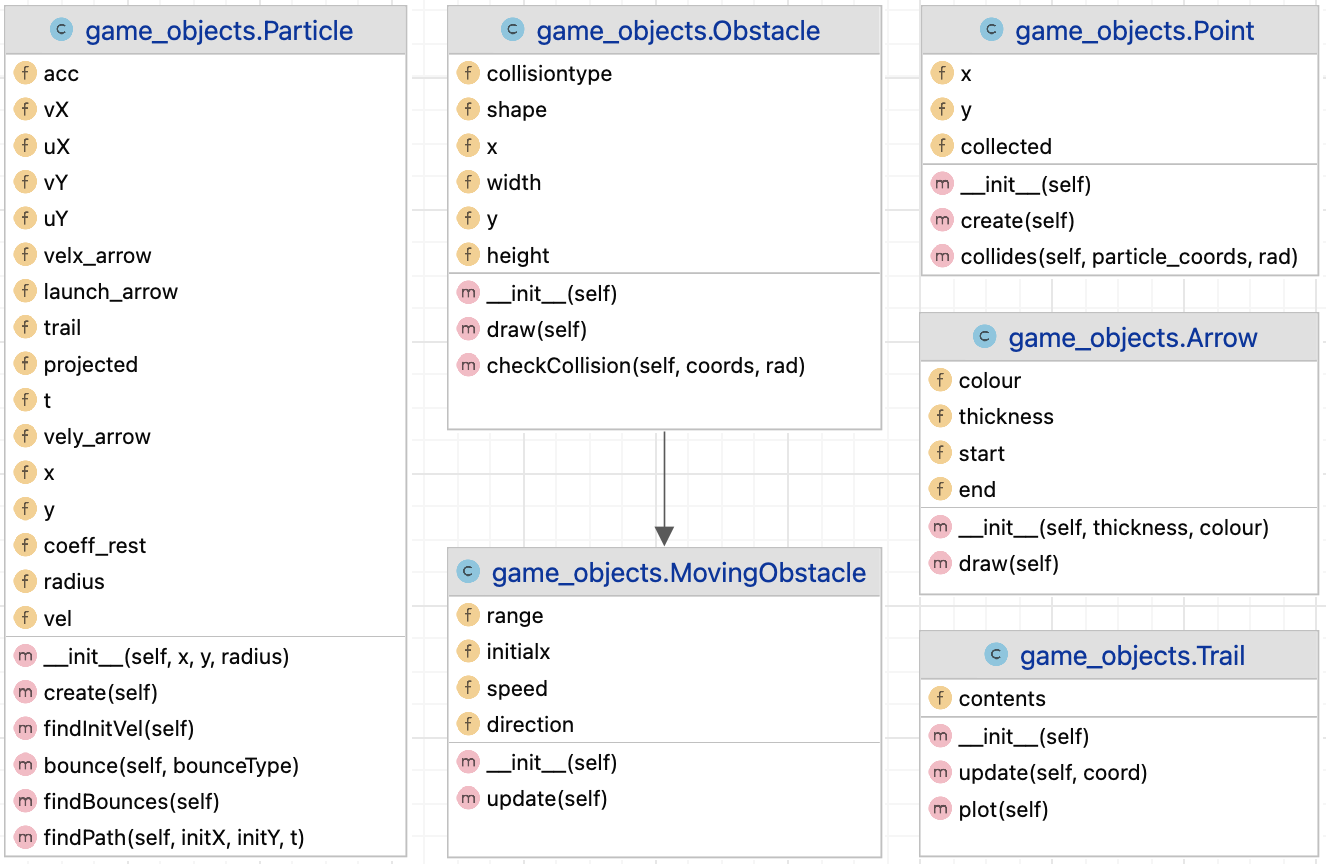
This diagram shows all the functions in my project and when they are ran:

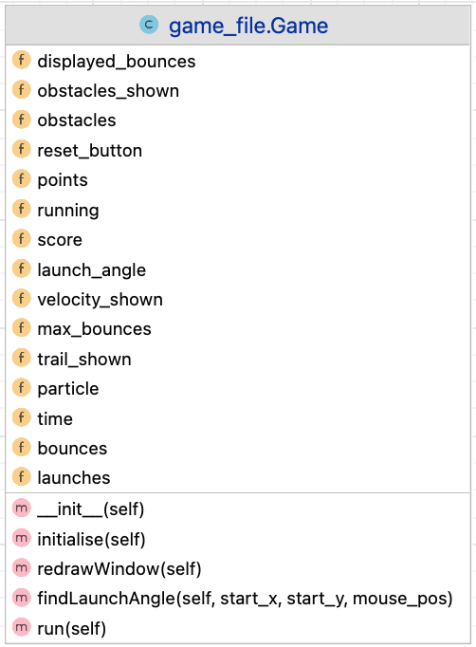
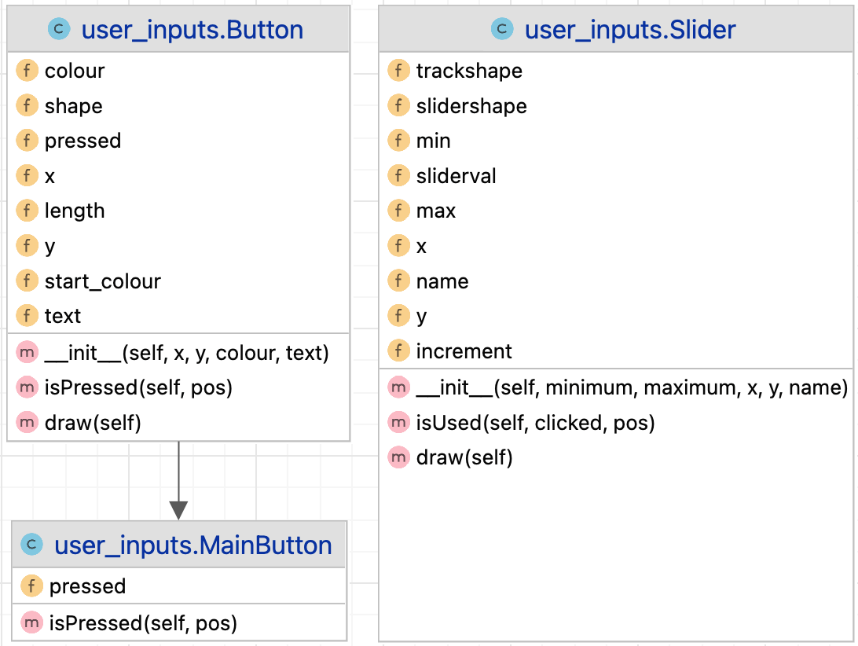


(Descriptions and pseudocode for some of these functions are in the Algorithm Description section)

### 2.1.2 Class Diagrams

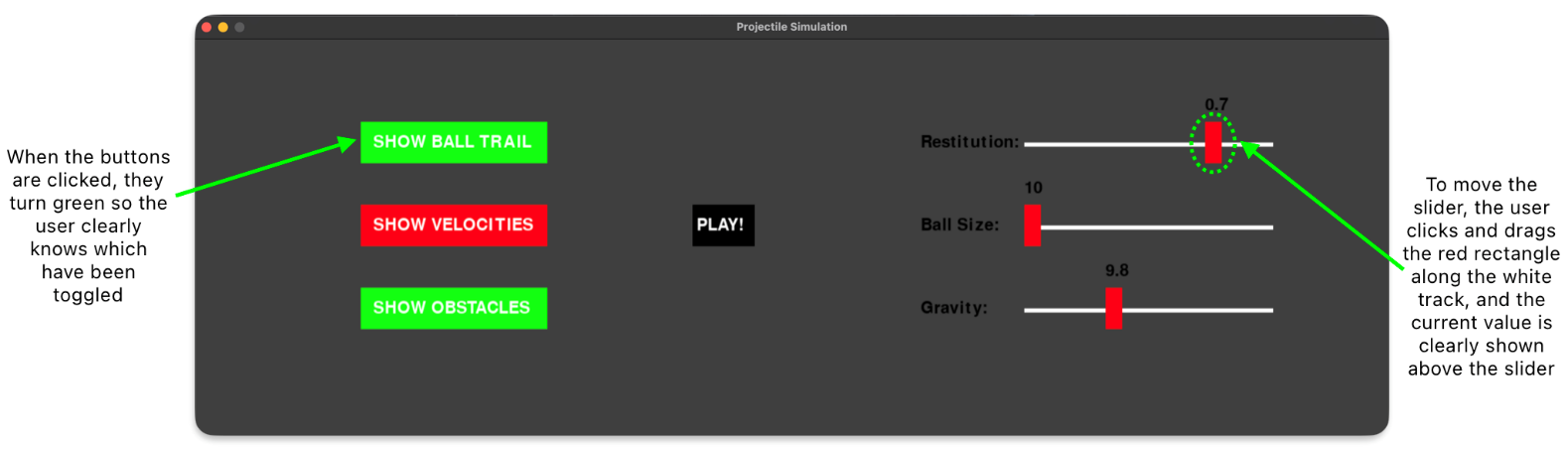
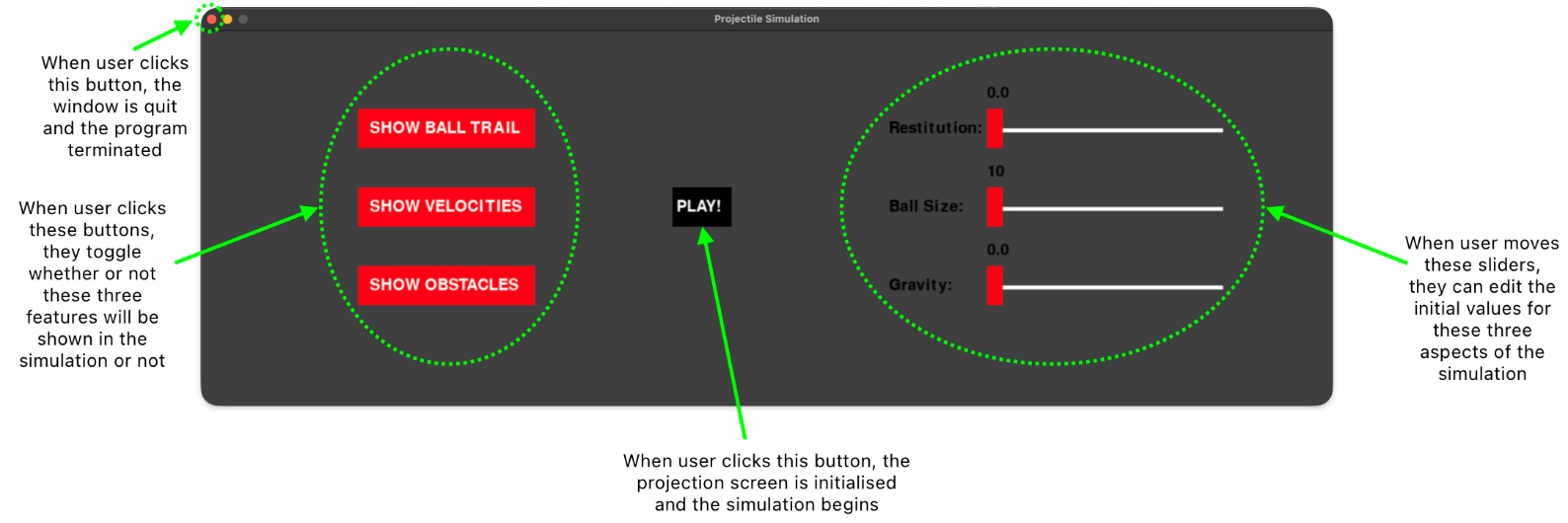
My project uses a lot of object-oriented programming, so it makes use of a lot of different classes. Here are class diagrams showing the attributes and methods of each created class:





### 2.1.3 HCI Diagrams

These are user interface diagrams that show the screen where the user inputs the initial conditions for the simulation. The first diagram is how the screen looks when the program is initially run, and the second is what it looks like after the user has edited the conditions.



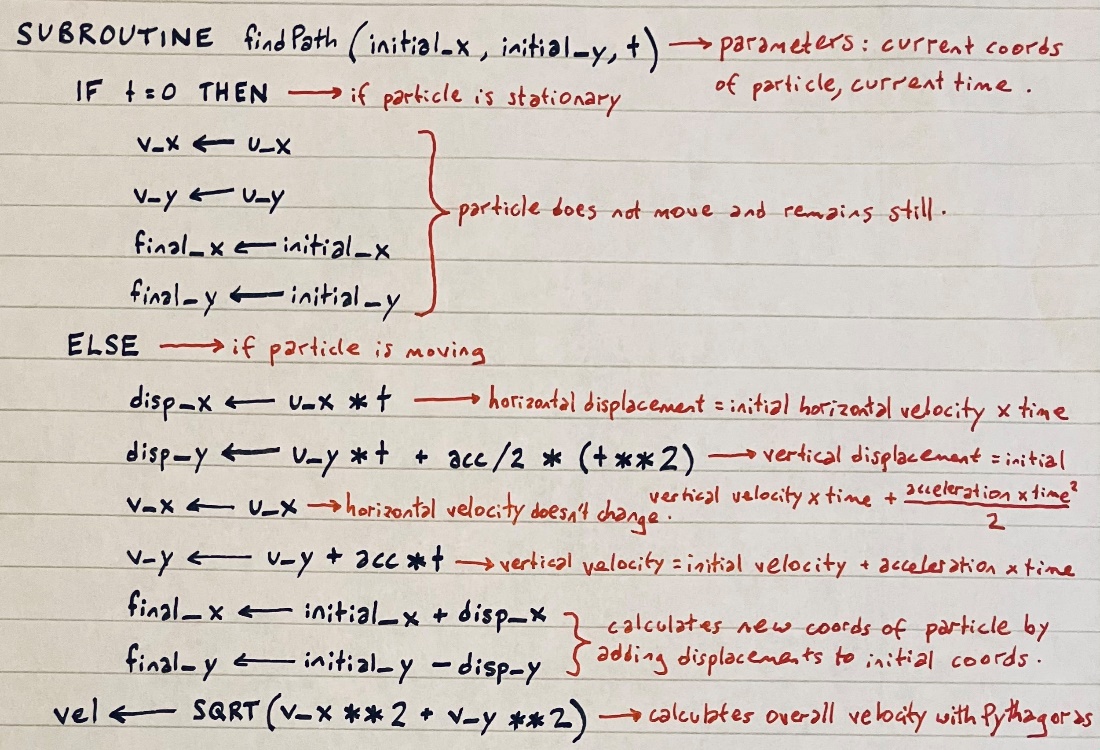
## 2.2 Algorithm description

Here are some of the main functions in my project in a simplified form.

### 2.2.1 Pseudocode

#### 2.2.1.1 Particle Path Function

The function 'findPath’ calculates the path of the projectile using the SUVAT equations. It is a method in the Particle class and is called in the Game class’s ‘run’ method.



The equations used are:

***s = ut + ½ at2***

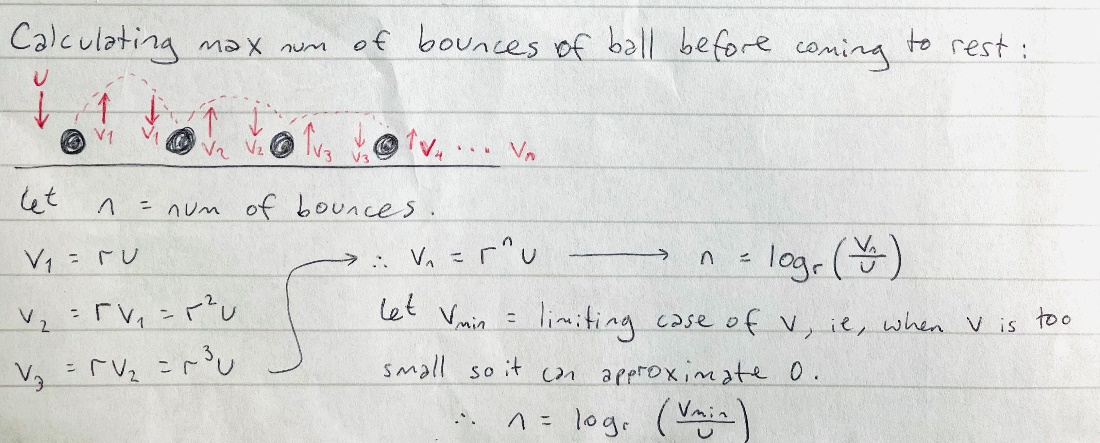
***v = u + at***

***a2 = b2 + c2***

#### 2.2.1.2 Particle find Maximum Bounces function

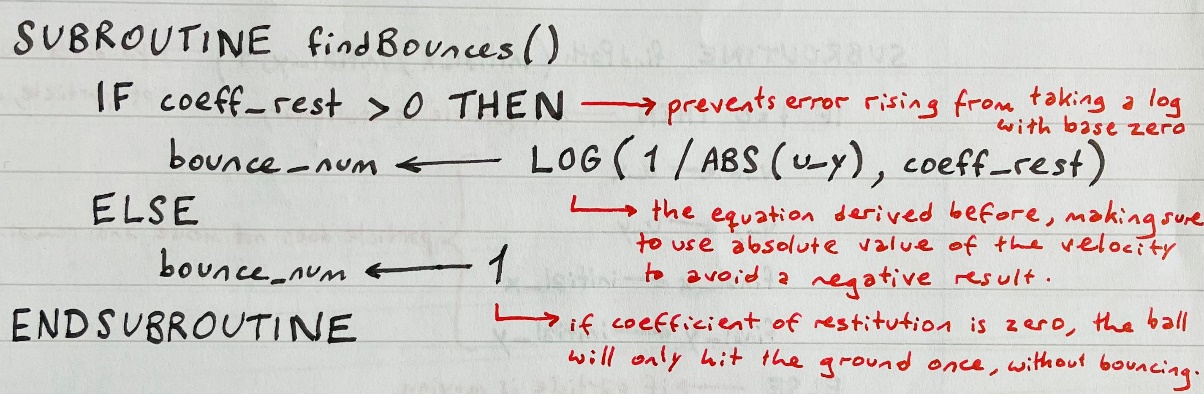
When trying to incorporate restitution into the particle’s collisions with walls and obstacles, I found that the velocity of the particle never actually reached zero because Pygame couldn’t handle calculations when the values for the velocity became very small, therefore, once the velocity reached a minimum value, it would just keep bouncing forever. To fix this, I decided to do the maths to find out how many times the particle would bounce in a real-life scenario based on its initial velocity and the coefficient of restitution.

Here is how I found a logarithmic equation for the maximum number of bounces the particle would complete before its velocity was so small that it could approximate zero and therefore be assumed to be still:



To use this equation, I had to decide on a limiting case of v, ie, at what velocity I could assume the particle is moving slow enough to be still. I decided to keep it simple and to use 1 as my limiting velocity.

Here is some pseudocode for the function I wrote to find the maximum bounces of the particle:

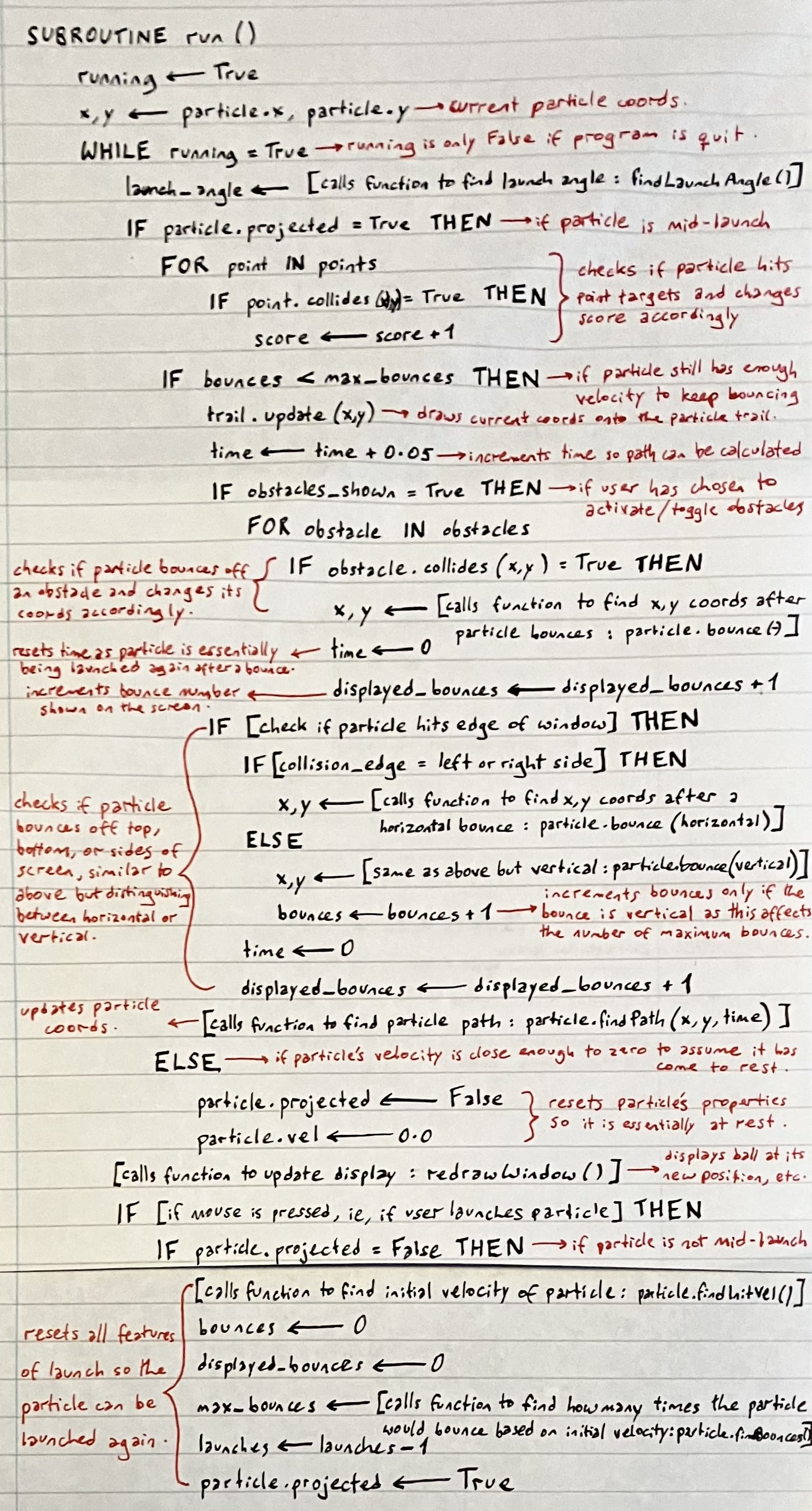


The velocity used is only the vertical component of velocity because that is the only one that affects whether the particle comes to rest or not, ie, the particle always has to land on the ground.

This function is a method in the Particle class and is called at the beginning of every projection.

#### 2.2.1.3 Run Function

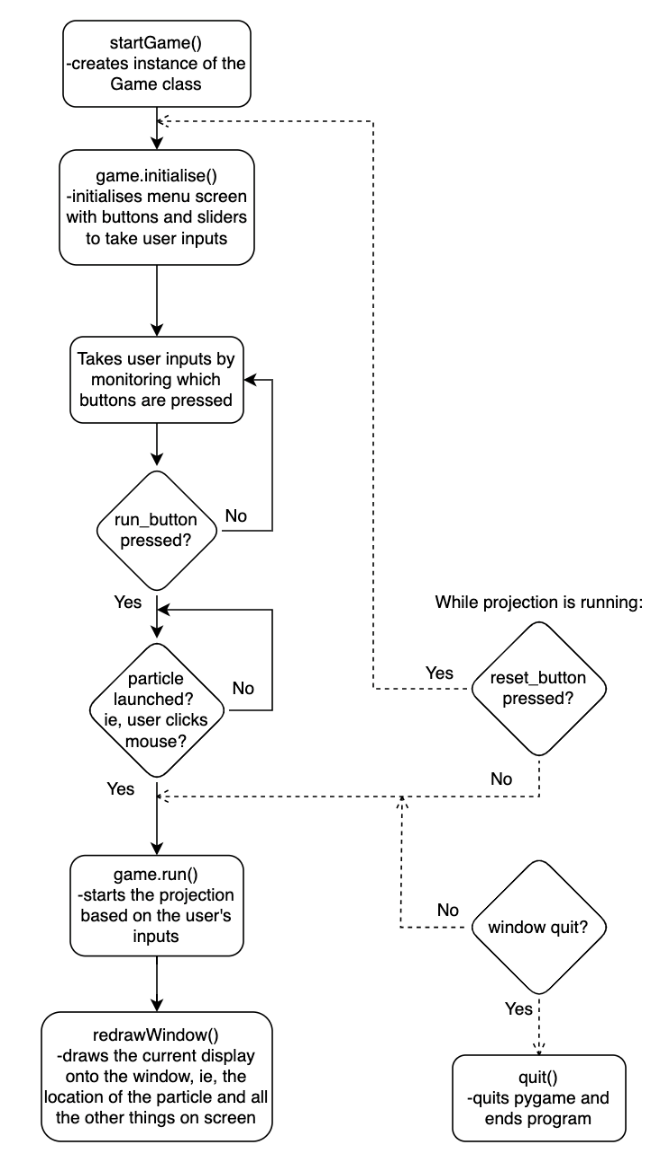
The ‘run’ function is the main function in my project and it runs the whole simulation. It is a method in the Game class.



### 2.2.2 Flowcharts

#### 2.2.2.1 Overall Flowchart

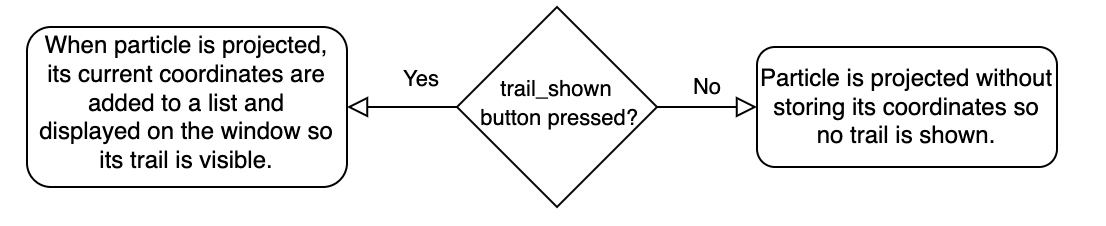
Here is a flowchart of all the functions that run when my code is executed, showing what they do and when they are called:



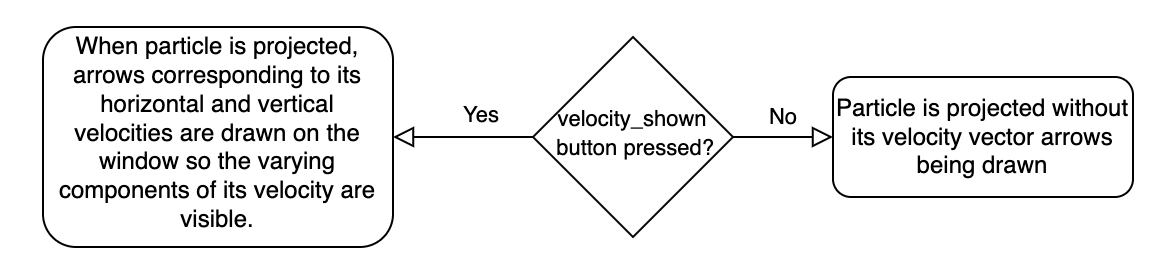
#### 2.2.2.2 Individual Flowcharts for User Inputs

These are individual flowcharts showing what happens to the projection when the user inputs their projection conditions in the initial menu screen:

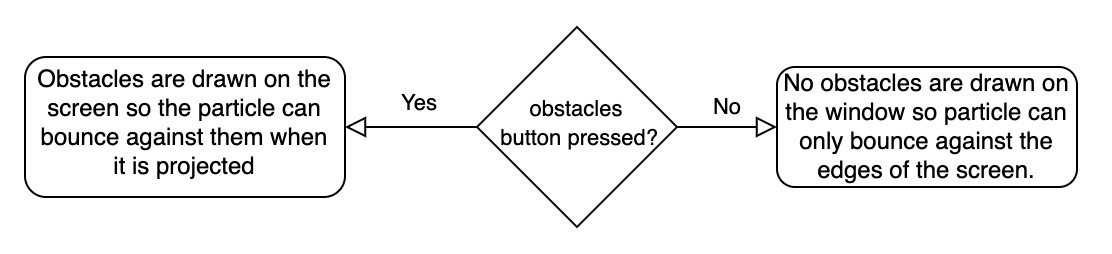
##### 2.2.2.2.1 Trail Button



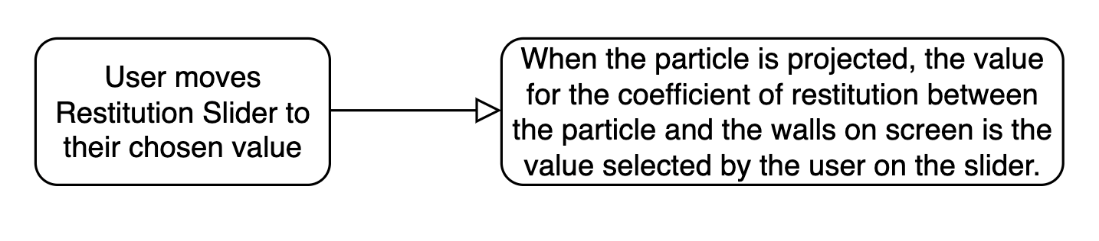
##### 2.2.2.2.2 Velocity Button



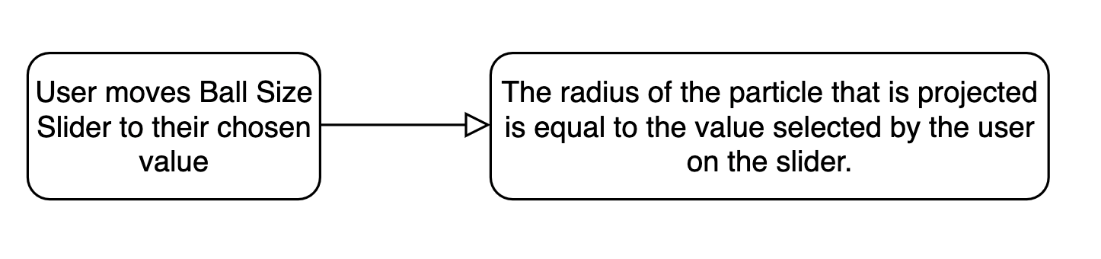
##### 2.2.2.2.3 Obstacle Button



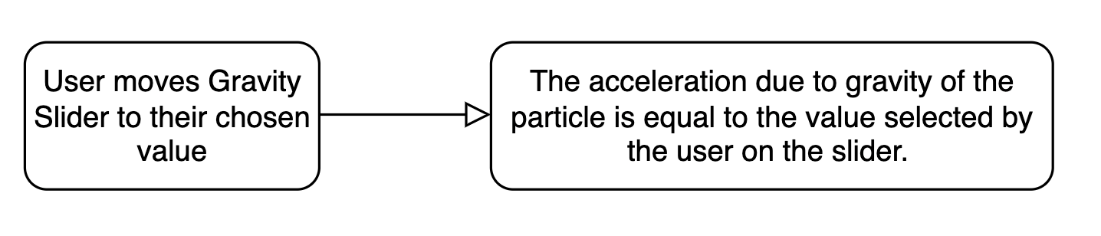
##### 2.2.2.2.4 Restitution Slider



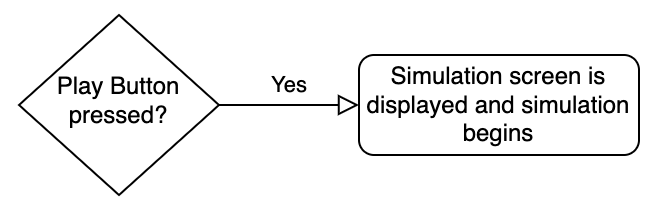
##### 2.2.2.2.5 Ball Size Slider



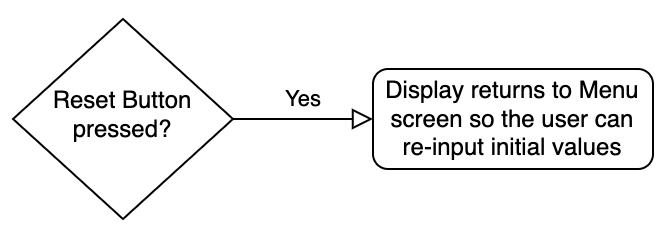
##### 2.2.2.2.6 Gravity Slider



#### 2.2.2.3 Individual Flowchart for Play Button



#### 2.2.2.4 Individual Flowchart for Reset Button



# Technical Solution

## 3.1 Code

### 3.1.1 User Inputs File

Here is the code for the user\_inputs file that includes all the classes defined for receiving user inputs.

import pygame  
pygame.init()  
  
BLACK = 0, 0, 0  
WHITE = 255, 255, 255  
GREY = 64, 64, 64  
RED = 255, 0, 0  **i1**  
GREEN = 0, 255, 0  
BLUE = 0, 0, 255  
SKYBLUE = 120, 190, 255  
YELLOW = 255, 255, 0  
  
wScreen = 1440  
hScreen = 800  
window = pygame.display.set\_mode((wScreen, hScreen))  **b1**  
font = pygame.font.SysFont('Ariel', 30)  **i2**  
  
  
class Button(object):  **c1**  
 *"""*  
 *A class to represent a button.*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *x,y : int*  
 *coordinates of the top left corner of the button*  
 *pressed : Bool*  
 *a boolean value for whether the button is toggled*  
 *start\_colour : tuple*  
 *RGB value for the initial colour of the button, ie, when it hasn't been pressed*  
 *colour : tuple*  
 *RGB value for the current colour of the button*  
 *text : str*  
 *the text that will be displayed on the button*  
 *length : int*  
 *the length of the text, which will determine the length of the button itself*  
 *shape : Rect*  
 *the Rect representation of the button's shape*  
  
  
 *Methods*  
 *-------*  
 *isPressed(pos):*  
 *Determines whether the button has been pressed*  
 *draw():*  
 *Draws the button on the window*  
 *"""*  
  
def \_\_init\_\_(self, x, y, colour, text):  
 *"""*  
 *Initialises all the attributes of the Button class*  
  
 *Parameters*  
 *------*  
 *x : int*  
 *x coordinate of top left corner of button*  
 *y : int*  
 *y coordinate of top left corner of button*  
 *colour : tuple*  
 *initial button colour*  
 *text : str*  
 *text displayed on the button*  
 *"""*  
  
self.x = x  
 self.y = y  
 self.pressed = False  
 self.start\_colour = colour if colour else RED  
 self.colour = self.start\_colour  
 self.text = font.render(f"{text.upper()}", True, (255, 255, 255))  
 self.length = len(text)  
 self.shape = pygame.Rect(x, y, self.length\*15, 50)  
  
  
 def isPressed(self, pos):  
 *"""*  
 *Determines whether the button has been pressed*  
  
 *Parameters*  
 *------*  
 *pos : tuple*  
 *current coordinates of the cursor*  
 *"""*  
  
if self.shape.collidepoint(pos):  
 self.pressed = not self.pressed  
 if self.pressed:  
 self.colour = GREEN  
 else:  
 self.colour = self.start\_colour  
  
 def draw(self):  
 *"""*  
 *Draws the button on the window*  
 *"""*  
  
pygame.draw.rect(window, self.colour, self.shape)  
 window.blit(self.text, (self.x + self.length, self.y + 15))  
  
  
class MainButton(Button): **E**  
 *"""*  
 *Inherits from the Button class to create a button that returns a value when pressed*  
  
 *"""*  
  
def isPressed(self, pos):  
 *"""*  
 *Determines whether button has been pressed and returns a Boolean value accordingly*  
  
 *Parameters*  
 *------*  
 *pos : tuple*  
 *current coordinates of the cursor*  
  
 *Returns*  
 *-------*  
 *True if button is pressed*  
 *"""*  
  
if self.shape.collidepoint(pos):  
 self.pressed = True  
 return True  
  
  
class Slider(object):  **c1**  
 *"""*  
 *A class to represent a slider.*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *x,y : int*  
 *coordinates of the top left corner of the slider*  
 *min, max : int*  
 *the minimum and maximum values the slider can display*  
 *name : str*  
 *the name of the simulation feature controlled by the slider*  
 *increment : int*  
 *the amount by which the slider value changes when it is moved by one pixel*  
 *slider\_val : int*  
 *the current value displayed by the slider*  
 *slider\_shape : Rect*  
 *the Rect representation of the slider's shape*  
 *track\_shape : Rect*  
 *the Rect representation of the slider track's shape*  
  
  
 *Methods*  
 *-------*  
 *isUsed(pos):*  
 *Determines whether the slider has been dragged/used*  
 *draw():*  
 *Draws the slider and track on the window, as well as its current value*  
 *"""*  
  
def \_\_init\_\_(self, minimum, maximum, x, y, name):  
 *"""*  
 *Initialises all the attributes of the Slider class*  
  
 *Parameters*  
 *------*  
 *minimum : int*  
 *the minimum value of the slider*  
 *maximum : int*  
 *the maximum value of the slider*  
 *x : int*  
 *x coordinate of the top left corner of the slider*  
 *y : int*  
 *y coordinate of the top left corner of the slider*  
 *name : str*  
 *simulation feature represented by the slider*  
 *"""*  
  
self.x = x  
 self.y = y  
 self.min = minimum  
 self.max = maximum  
 self.name = name + ':'  
 self.increment = (self.max-self.min)/300  
 self.slider\_val = self.min  
 self.slider\_shape = pygame.Rect(x, y, 20, 50)  
 self.track\_shape = pygame.Rect(x, y + 25, 300, 5)  
  
 def isUsed(self, clicked, pos):  
 *"""*  
 *Determines whether the slider has been dragged/used*  
  
 *Parameters*  
 *------*  
 *clicked : int*  
 *integer value showing which mouse button has been pressed (1 = primary button, ie, normal left click)*  
 *pos : tuple*  
 *current coordinates of the cursor*  
 *"""*  
  
if clicked == 1 and self.slider\_shape.collidepoint(pos):  
 if pos[0] - 10 < self.x:  
 self.slider\_shape.x = self.x  
 elif pos[0] - 10 > self.x + 300:  
 self.slider\_shape.x = self.x + 300  
 else:  
 self.slider\_shape.x = pos[0] - 10  
  
 self.slider\_val = round((self.min + (self.slider\_shape.x - self.x) \* self.increment), 1)  
  
 def draw(self):  
 *"""*  
 *Draws the slider and track on the window, as well as its current value*  
 *"""*  
  
pygame.draw.rect(window, WHITE, self.track\_shape)  
 pygame.draw.rect(window, RED, self.slider\_shape)  
 window.blit((font.render(f"{self.name}", True, BLACK)), (self.x-125, self.y+15))  
 window.blit((font.render(f"{self.slider\_val}", True, BLACK)), (self.slider\_shape.x, self.y - 30))

### 3.1.2 Game Objects File

Here is the code for the game\_objects file that includes the classes created to represent all the simulation objects, including the particle, obstacles, etc.

import user\_inputs **B**  
import pygame  
import math  
from random import randint  
pygame.init()

class Particle(object):  **a1**  
 *"""*  
 *A class to represent a particle.*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *x,y : int*  
 *x and y coordinates of the particle*  
 *radius : float*  
 *radius of the particle*  
 *uX,uY : float*  
 *initial x and y components of the particle's velocity*  
 *vX,vY : float*  
 *final x and y components of the particle's velocity*  
 *acc : float*  
 *value for the acceleration of the particle*  
 *t : float*  
 *the time elapsed in the current projection*  
 *vel : float*  
 *the overall velocity of the particle*  
 *coeff\_rest : float*  
 *the value for the coefficient of restitution between the particle and walls*  
 *trail : Trail*  
 *an instance of the Trail class that stores the past coordinates of the particle*  
 *launch\_arrow : Arrow*  
 *an instance of the Arrow class that creates an arrow from the particle to the cursor*  
 *velx\_arrow, vely\_arrow : Arrow*  
 *instances of Arrow class that create arrows for the x and y components of the particle's velocity*  
 *projected : Bool*  
 *a boolean value for whether the particle is mid-projection or not*  
  
  
 *Methods*  
 *-------*  
 *create():*  
 *draws the particle as a circle on the window*  
 *findInitVel():*  
 *finds the initial x and y components of the velocity of the particle*  
 *findBounces():*  
 *calculates the number of times the particle will bounce before its velocity approximates zero*  
 *findPath(initX, initY, t):*  
 *calculates the path of the particle by finding its new coordinates using SUVAT equations*  
 *bounce(bounce\_type):*  
 *finds the velocity and coordinates of the particle after it bounces*  
 *"""*  
  
def \_\_init\_\_(self, x, y, radius):  
 *"""*  
 *Initialises all the attributes of the Particle class*  
  
 *Parameters*  
 *------*  
 *x : int*  
 *x coordinate of the particle*  
 *y : int*  
 *y coordinate of the particle*  
 *radius : int*  
 *radius of the particle*  
 *"""*  
  
self.x = x  
 self.y = y  
 self.radius = radius  
 self.uX = 0  
 self.uY = 0  
 self.vX = 0  
 self.vY = 0  
 self.acc = -9.81  
 self.t = 0  
 self.vel = 0.0  
 self.coeff\_rest = None  
 self.trail = Trail()  
 self.launch\_arrow = Arrow(1, user\_inputs.WHITE)  
 self.velx\_arrow, self.vely\_arrow = Arrow(3, user\_inputs.RED), Arrow(3, user\_inputs.RED)  
 self.projected = False  
  
 def create(self):  
 *"""*  
 *Draws the particle as a circle on the window*  
 *"""*  
  
pygame.draw.circle(user\_inputs.window, user\_inputs.BLACK, (self.x, self.y), self.radius)  
  
 def findInitVel(self):  **c2**  
 *"""*  
 *Finds the initial x and y components of the velocity of the particle*  
 *"""*  
  
line\_length = pygame.mouse.get\_pos()[0] - self.x  
 line\_height = self.y - pygame.mouse.get\_pos()[1]  
  
 self.uX = line\_length / 5  
 self.uY = line\_height / 5  
  
 def findBounces(self): **C**  
 *"""*  
 *Calculates the number of times the particle will bounce before its velocity approximates zero*  
  
 *Returns*  
 *-------*  
 *bounce\_num : int*  
 *the maximum number of times the particle will bounce*  
 *"""*  
  
bounce\_num = math.log(1 / abs(self.uY), self.coeff\_rest) if self.coeff\_rest > 0 else 1  
 return bounce\_num  
  
 def findPath(self, initX, initY, t):  **a2 C**  
 *"""*  
 *Calculates the path of the particle by finding its new coordinates using SUVAT equations*  
  
 *Parameters*  
 *------*  
 *initX : int*  
 *the initial x coordinate of the particle*  
 *initY : int*  
 *the initial y coordinate of the particle*  
 *t : float*  
 *the current time elapsed in the projection*  
 *"""*  
  
self.trail.update((self.x, self.y))  
  
 if t == 0:  
 self.vX, self.vY = self.uX, self.vY  
 self.x, self.y = initX, initY  
  
 else:  
 sX = self.uX \* t  
 sY = (self.uY \* t) + (self.acc / 2 \* (t \*\* 2))  
 self.vX = self.uX  
 self.vY = self.uY + self.acc \* t  
 self.x, self.y = initX + sX, initY - sY  
  
 self.vel = math.sqrt(self.vX \*\* 2 + self.vY \*\* 2)  
  
 def bounce(self, bounce\_type):  **e2 C**  
 *"""*  
 *Finds the velocity and coordinates of the particle after it bounces*  
  
 *Parameters*

*------*  
 *bounce\_type : str*  
 *value that specifies if the bounce is horizontal or vertical so the right velocity components are changed*  
  
 *Returns*  
 *-------*  
 *x,y : int*  
 *new x and y coordinates of the particle*  
 *"""*  
  
x\_sign = -1 if self.vX > 0 else 1  
 y\_sign = -1 if self.vY > 0 else 1  
  
 if bounce\_type == 'horizontal':  
 self.uY = self.coeff\_rest \* self.vY  
 self.uX = -self.coeff\_rest \* self.vX  
 x, y = self.x + x\_sign \* self.radius, self.y  
 return x, y  
  
 elif bounce\_type == 'vertical':  
 self.uY = -self.coeff\_rest \* self.vY  
 self.uX = self.coeff\_rest \* self.vX  
 x, y = self.x, self.y - y\_sign \* self.radius  
 return x, y  
  
  
class Trail(object):  
 *"""*  
 *A class to represent a trail for a particle.*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *contents : list*  
 *a list of (past) coordinates*  
  
 *Methods*  
 *-------*  
 *update(coord):*  
 *adds the given coordinate to the contents of the trail and makes sure trail isn't too long*  
 *plot():*  
 *draws the coordinates in the trail onto the window as dots*  
 *"""*  
  
def \_\_init\_\_(self):  
 *"""*  
 *Initialises all the attributes of the Trail class*  
 *"""*  
  
self.contents = [] **A**  
  
 def update(self, coord):  
 *"""*  
 *Adds the given coordinate to the contents of the trail and makes sure trail isn't too long*  
  
 *Parameters*  
 *------*  
 *coord : tuple*  
 *the coordinate to be added to the trail*  
 *"""*  
  
if len(self.contents) > 1000:  
 self.contents.pop(0) **A**  
 self.update(coord) **D**  
 else:  
 self.contents.append(coord) **A**  
  
 def plot(self):  
 *"""*  
 *Draws the coordinates in the trail onto the window as dots*  
 *"""*  
  
if self.contents:  
 for coord in self.contents: **A**  
 pygame.draw.circle(user\_inputs.window, user\_inputs.WHITE, coord, 1)  
 self.contents.pop(0) **A**  
  
  
class Point(object):  **d1**  
 *"""*  
 *A class to represent a point in the game aspect of the simulation*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *x,y : int*  
 *the x and y coordinates of the point*  
 *collected : Bool*  
 *a boolean value representing whether the point has been collected or not*  
  
 *Methods*  
 *-------*  
 *create():*  
 *draws the point on the window as a small yellow circle*  
 *collides(coords, rad):*  
 *returns whether the particle hits the point*  
 *"""*  
  
def \_\_init\_\_(self):  
 *"""*  
 *Initialises all the attributes of the Point class*  
 *"""*  
self.x = randint(10, user\_inputs.wScreen - 10)  
 self.y = randint(10, user\_inputs.hScreen - 10)  
 self.collected = False  
  
 def create(self):  
 *"""*  
 *Draws the point on the window as a small yellow circle*  
 *"""*  
pygame.draw.circle(user\_inputs.window, user\_inputs.YELLOW, (self.x, self.y), 5)  
  
 def collides(self, coords, rad):  
 *"""*  
 *Returns whether the particle hits the point*  
  
 *Parameters*  
 *------*  
 *coords : tuple*  
 *the coordinates of the particle*  
 *rad : float*  
 *the radius of the particle*  
  
 *Returns*  
 *-------*  
 *True if the point is hit*  
 *"""*  
  
if coords[0]-rad <= self.x <= coords[0]+rad and coords[1]-rad <= self.y <= coords[1]+rad:  
 self.collected = True  
 return True  
  
  
class Obstacle(object):  **f1**  
 *"""*  
 *A class to represent an obstacle*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *x,y : int*  
 *the x and y coordinates of the top left corner of the obstacle*  
 *width : int*  
 *the width of the obstacle*  
 *height : int*  
 *the height of the obstacle*  
 *shape : Rect*  
 *the Rect representation of the obstacle's shape*  
  
 *Methods*  
 *-------*  
 *draw():*  
 *draws the obstacle on the window*  
 *checkCollision():*  
 *checks whether the particle hits the obstacle*  
 *"""*  
  
def \_\_init\_\_(self, r):  
 *"""*  
 *Initialises all the attributes of the Obstacle class*  
  
 *Parameters*  
 *------*  
 *r : radius of the particle*  
 *"""*  
  
self.x = randint(0, user\_inputs.wScreen/2 - r - 50) or randint(user\_inputs.wScreen/2 + r, user\_inputs.wScreen - 50)  
 self.y = randint(0, user\_inputs.hScreen - 50)  
 self.width = 50  
 self.height = user\_inputs.hScreen - self.y  
 self.shape = pygame.Rect(self.x, self.y, self.width, self.height)  
  
 def draw(self):  
 *"""*  
 *Draws the obstacle on the window*  
 *"""*  
pygame.draw.rect(user\_inputs.window, user\_inputs.BLUE, self.shape)  
  
 def checkCollision(self, coords, rad):  
 *"""*  
 *Checks whether the particle hits the obstacle*  
  
 *Parameters*  
 *------*  
 *coords : the current coordinates of the particle*  
 *rad : the radius of the particle*  
  
 *Returns*  
 *-------*  
 *True if the particle hits the obstacle*  
 *"""*  
  
x, y = coords[0], coords[1]  
  
 if self.shape.left-rad <= x <= self.shape.right+rad:  
 if y > self.shape.top:  
 return True  
  
  
class MovingObstacle(Obstacle):  **f2E**  
 *"""*  
 *Inherits from the Obstacle class to create an obstacle that moves*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *initial\_x : int*  
 *the obstacle's starting x coordinate*  
 *speed : float*  
 *the speed the obstacle moves at*  
 *range : int*  
 *the range of the obstacle's movement*  
 *direction : int*  
 *+1 or -1 depending on the direction of the obstacle's movement*  
  
  
 *Methods*  
 *-------*  
 *update():*  
 *moves the obstacle to its new position*  
 *"""*  
  
def \_\_init\_\_(self, r):  
 *"""*  
 *Initialises all the attributes of the Moving Obstacle class*  
  
 *Parameters*  
 *------*  
 *r : the radius of the particle*  
 *"""*  
  
super().\_\_init\_\_(r)  
 self.initial\_x = self.x  
 self.speed = randint(1,5)/10  
 self.range = randint(100,400)  
 self.direction = 1  
  
 def update(self):  
 *"""*  
 *Moves the obstacle to its new position*  
 *"""*  
  
self.x += self.speed\*self.direction  
 self.shape.x = self.x  
  
 if self.x >= self.initial\_x+self.range or self.x <= self.initial\_x-self.range:  
 self.direction \*= -1  
  
  
class Arrow(object):  
 *"""*  
 *A class to represent an Arrow.*  
  
 *...*  
  
 *Attributes*  
 *------*  
 *thickness : int*  
 *a value for the thickness of the arrow*  
 *colour : tuple*  
 *an RGB value for the colour of the arrow*  
 *start, end : tuple*  
 *the coordinates of the arrow's points/ends*  
  
  
 *Methods*  
 *-------*  
 *draw():*  
 *draws the arrow on the window*  
 *"""*  
  
def \_\_init\_\_(self, thickness, colour):  
 *"""*  
 *Initialises all the attributes of the Arrow class*  
  
 *Parameters*  
 *------*  
 *thickness : int*  
 *a value for the thickness of the arrow*  
 *colour : tuple*  
 *an RGB value for the colour of the arrow*  
 *"""*  
  
self.thickness = thickness  
 self.colour = colour  
 self.start = None  
 self.end = None  
  
 def draw(self):  
 *"""*  
 *Draws the arrow on the window*  
 *"""*  
  
pygame.draw.line(user\_inputs.window, self.colour, self.start, self.end, self.thickness)  
  
 horizontal = self.end[0] - self.start[0]  
 vertical = self.end[1] - self.start[1]  
 angle = math.atan2(vertical, horizontal)  
  
 arrowhead\_points = [  
 self.end,  
 (self.end[0] + 15 \* math.cos(angle + math.pi \* 5 / 6),  
 self.end[1] + 15 \* math.sin(angle + math.pi \* 5 / 6)),  
  
 (self.end[0] + 15 \* math.cos(angle - math.pi \* 5 / 6),  
 self.end[1] + 15 \* math.sin(angle - math.pi \* 5 / 6))  
 ]  
  
 pygame.draw.line(user\_inputs.window, self.colour, self.end, arrowhead\_points[1], self.thickness)  
 pygame.draw.line(user\_inputs.window, self.colour, self.end, arrowhead\_points[2], self.thickness)

### 3.1.3 Main Game File

Finally, here is the code for the main game\_file file, which ties everything together and creates the simulation.

import user\_inputs **B**  
import game\_objects **B**  
  
import pygame  
import sys  
import math  
pygame.init()  
pygame.display.set\_caption('Projectile Simulation')  
  
  
class Game(object):  
 *"""*  
 *A class to represent the whole Game*  
  
 *...*  
  
 *Attributes*  
*------*  
 *running : Bool*  
 *a boolean value for whether the simulation is running or not*  
 *particle : Particle*  
 *the particle that will be projected*  
 *launch\_angle : float*  
 *the initial launch angle of the particle*  
 *time : float*  
 *the current time elapsed in the projection*  
 *max\_bounces : int*  
 *the number of times the particle will bounce before its velocity approximates zero*  
 *bounces : int*  
 *the current number of times the particle has bounced vertically*  
 *displayed\_bounces : int*  
 *the particle's total number of bounces*  
 *launches : int*  
 *the number of times the particle has been launched*  
 *score : int*  
 *the current score based on how many points have been collected*  
 *points : list*  
 *a list of Point instances to represent the total collectable points on the window*  
 *obstacles : list*  
 *a list of Obstacle instances to represent the obstacles on the window*  
 *trail\_shown : Bool*  
 *a boolean value for whether the particle trail is shown in the simulation*  
 *velocity\_shown : Bool*  
 *a boolean value for whether the particle's velocity arrows are shown in the simulation*  
 *obstacles\_shown : Bool*  
 *a boolean value for whether there are obstacles in the simulation*  
 *reset\_button : MainButton*  
 *an instance of the MainButton class to represent a button that resets the simulation*  
  
  
 *Methods*  
 *-------*  
 *initialise():*  
 *sets up the initial menu window where the user can input initial projection values*  
 *redrawWindow():*  
 *creates and displays the current projection frame on the window*  
 *findLaunchAngle(start\_x, start\_y, mouse\_pos):*  
 *static method that finds the particle's initial launch angle*  
 *run():*  
 *runs the projection*  
 *"""*  
  
def \_\_init\_\_(self):  
 *"""*  
 *Initialises all the attributes of the Game class*  
 *"""*  
  
self.running = None  
 self.particle = game\_objects.Particle(user\_inputs.wScreen / 2, user\_inputs.hScreen / 2, 10)  **- A**  
  
 self.launch\_angle = 0  
 self.time = 0  
 self.max\_bounces = None  
 self.bounces = 0  
 self.displayed\_bounces = 0  
 self.launches = None  
 self.score = 0  **d2**  
 self.points = [] **A**  
 self.obstacles = []  **F A**  
  
 self.trail\_shown = False  
 self.velocity\_shown = False  
 self.obstacles\_shown = False  
  
 self.reset\_button = user\_inputs.MainButton(1300, 30, user\_inputs.BLACK, 'RESET')  
  
 def initialise(self):  **C**   
 *"""*  
 *Sets up the initial menu window where the user can input initial projection values*  
 *"""*  
  
self.launches = 10  
 self.particle.x, self.particle.y = user\_inputs.wScreen / 2 , user\_inputs.hScreen / 2  
 self.points = [game\_objects.Point() for i in range(10)] **A**  
 self.obstacles = [game\_objects.Obstacle(self.particle.radius), game\_objects.Obstacle(self.particle.radius), game\_objects.MovingObstacle(self.particle.radius)] **A**  
  
 run\_button = user\_inputs.MainButton(600, 200, user\_inputs.BLACK, 'Play!')  
 trail\_button = user\_inputs.Button(200, 100, None, 'Show Ball Trail')  
 velocity\_button = user\_inputs.Button(200, 200, None, 'Show Velocities')  
 obstacle\_button = user\_inputs.Button(200, 300, None, 'Show Obstacles ')  
  
 input\_buttons = [run\_button, trail\_button,velocity\_button, obstacle\_button] **A**  
  
 restitution\_slider = user\_inputs.Slider(0.0, 0.9, 1000, 100, 'Restitution')  
 size\_slider = user\_inputs.Slider(10, 50, 1000, 200, 'Ball Size')  
 grav\_slider = user\_inputs.Slider(0.0, 30, 1000, 300, 'Gravity')  **G**  
  
 input\_sliders = [restitution\_slider, size\_slider, grav\_slider] **A**  
  
 while not run\_button.pressed:  
 user\_inputs.window.fill(user\_inputs.GREY)  
 for button in input\_buttons:  
 button.draw()  
 for slider in input\_sliders:  
 slider.draw()  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()  
 elif event.type == pygame.MOUSEBUTTONDOWN:  
 for button in input\_buttons:  
 button.isPressed(event.pos)  
 elif event.type == pygame.MOUSEMOTION:  
 for slider in input\_sliders:  
 slider.isUsed(event.buttons[0], event.pos)  
  
 pygame.display.update()  
  
 self.trail\_shown = trail\_button.pressed  
 self.velocity\_shown = velocity\_button.pressed  
 self.obstacles\_shown = obstacle\_button.pressed  
  
 self.particle.radius = size\_slider.slider\_val  
 self.particle.coeff\_rest = restitution\_slider.slider\_val  
 self.particle.acc = -grav\_slider.slider\_val  **g1**  
  
 self.run()  
  
 def redrawWindow(self):  **B**  
 *"""*  
 *Creates and displays the current projection frame on the window*  
 *"""*  
  
user\_inputs.window.fill(user\_inputs.SKYBLUE)  
  
 self.reset\_button.draw()  
  
 if self.launches < 0:  
 self.initialise()  
  
 self.particle.create()  
 self.particle.launch\_arrow.start, self.particle.launch\_arrow.end = (self.particle.x, self.particle.y), pygame.mouse.get\_pos()  
 self.particle.launch\_arrow.draw()  
  
 for point in self.points:  
 if not point.collected:  
 point.create()  
 else:  
 self.points.remove(point) **A**  
 self.points.append(game\_objects.Point())  **DA**  
  
 if self.trail\_shown:  
 self.particle.trail.plot()  
  
 if self.velocity\_shown:  **c4**  
 self.particle.velx\_arrow.start, self.particle.velx\_arrow.end = ((self.particle.x, self.particle.y), (self.particle.x+self.particle.vX, self.particle.y))  
 self.particle.vely\_arrow.start, self.particle.vely\_arrow.end = ((self.particle.x, self.particle.y), (self.particle.x, self.particle.y-self.particle.vY))  
 self.particle.velx\_arrow.draw()  
 self.particle.vely\_arrow.draw()  **a3**  
  
 if self.obstacles\_shown:  **f3**  
 for obstacle in self.obstacles:  
 obstacle.draw()  
 if type(obstacle) == game\_objects.MovingObstacle:  
 obstacle.update()  
  
 score\_surface = user\_inputs.font.render(f'Score = {self.score}', False, user\_inputs.WHITE)  
 velocity\_surface = user\_inputs.font.render(f'Current Velocity = {self.particle.vel:.3f}', False, user\_inputs.WHITE)  
 angle\_surface = user\_inputs.font.render(f'Launch Angle = {self.launch\_angle:.2f}', False, user\_inputs.WHITE)  
 bounce\_surface = user\_inputs.font.render(f'Num of Bounces = {self.displayed\_bounces}', False, user\_inputs.WHITE)  
 launch\_surface = user\_inputs.font.render(f'Launches remaining = {self.launches}', False, user\_inputs.WHITE)  
  
 user\_inputs.window.blit(velocity\_surface, (100, 100))  **h1**  
 user\_inputs.window.blit(angle\_surface, (100, 120))  
 user\_inputs.window.blit(bounce\_surface, (100, 140))  **H**  
 user\_inputs.window.blit(score\_surface, (100, 180))  
 user\_inputs.window.blit(launch\_surface, (100, 160))   
  
 pygame.display.update()  **h2**  
  
 @staticmethod  
 def findLaunchAngle(start\_x, start\_y, mouse\_pos):  **c3**  
 *"""*  
 *Static method that finds the particle's initial launch angle*

*Parameters*  
 *------*  
 *start\_x : int*  
 *the particle's initial x coordinate*  
 *start\_y : int*  
 *the particle's initial y coordinate*  
 *mouse\_pos : tuple*  
 *the current coordinates of the cursor*  
  
 *Returns*  
 *-------*  
 *An integer value representing the launch angle*  
 *"""*  
  
mouse\_x = mouse\_pos[0]  
 mouse\_y = mouse\_pos[1]  
  
 if mouse\_x == start\_x:  
 if mouse\_y < start\_y:  
 return 90  
 else:  
 return 270  
  
 elif mouse\_y == start\_y:  
 if mouse\_x < start\_x:  
 return 180  
 else:  
 return 0  
  
 else:  
 angle = math.atan((start\_y - mouse\_y) / (start\_x - mouse\_x))  
  
 if start\_x > mouse\_x and mouse\_y > start\_y:  
 *# bottom left quad*  
angle = math.pi + abs(angle)  
  
 elif start\_x > mouse\_x and start\_y > mouse\_y:  
 *# top left quad*  
angle = abs(math.pi - angle)  
  
 elif mouse\_x > start\_x and mouse\_y > start\_y:  
 *# bottom right quad*  
angle = abs((math.pi \* 2) - angle)  
  
 elif mouse\_x > start\_x and start\_y > mouse\_y:  
 *# top right quad*  
angle = abs(angle)  
  
 return abs(angle \* (180 / math.pi))  
  
 def run(self):  
 *"""*  
 *Runs the projection*  
 *"""*  
  
running = True  
 x, y = self.particle.x,self.particle.y  
  
 while running:  
  
 self.launch\_angle = self.findLaunchAngle(x, y, pygame.mouse.get\_pos()) if not self.particle.projected else self.launch\_angle  
  
 if self.particle.projected:  
  
 for point in self.points:  
 if point.collides((self.particle.x, self.particle.y), self.particle.radius):  
 self.score += 1  
  
 if self.bounces < self.max\_bounces:  
 self.particle.trail.update((self.particle.x, self.particle.y))  
 self.time += 0.05  
  
 if self.obstacles\_shown:  
 for obstacle in self.obstacles:  
 if obstacle.checkCollision((self.particle.x, self.particle.y), self.particle.radius):  
 x, y = self.particle.bounce('horizontal')  
  
 self.time = 0  
 self.displayed\_bounces += 1  
  
 if self.particle.x <= self.particle.radius or self.particle.x >= user\_inputs.wScreen - self.particle.radius:  **E**  
  
 x, y = self.particle.bounce('horizontal')  
  
 self.time = 0  
 self.displayed\_bounces += 1  
  
 if self.particle.y <= self.particle.radius or self.particle.y >= user\_inputs.hScreen - self.particle.radius:  
 x, y = self.particle.bounce('vertical')  
 self.time = 0  
 self.displayed\_bounces += 1  
 self.bounces += 1  
  
 self.particle.findPath(x, y, self.time)  
  
 else:  
 self.particle.projected = False  
 self.particle.vel = 0.0  
 self.particle.y = y + self.particle.radius - 1  
 self.particle.x = x  
  
 self.redrawWindow()  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 self.running = False  
 pygame.quit()  
 sys.exit()  **b2**  
  
 if event.type == pygame.MOUSEBUTTONDOWN:  
  
 if self.reset\_button.isPressed(event.pos):  
 self.running = False  
  
 self.initialise()  
  
 else:  
  
 if not self.particle.projected:  
 self.particle.findInitVel()  
 self.bounces = 0  
 self.displayed\_bounces = 0  
 self.max\_bounces = self.particle.findBounces()  
 self.launches -= 1  
 self.particle.projected = True  
  
  
def startGame():  
 *"""*  
 *Creates and initialises an instance of the Game class*  
 *"""*  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 game = Game()  
 game.initialise()  
  
startGame()

## 3.2 Objective Completion

Here is a table of my initial objectives and if/where they appear in my solution. The markers show the tag and page number of where the objective evidence is.

|  |  |  |
| --- | --- | --- |
| OBJECTIVE | MARKER | DETAILS |
| 1. Create a ball that is projected across the screen | **A, pg** | The ball is created when the projection starts and the user can launch it across the screen |
| 1.1 Create a Particle class using OOP | **a1, pg** | I defined a Particle class with various methods and attributes so the particle can be created easily |
| 1.2 Use equations of motion with constant acceleration to find the ball’s path | **a2, pg** | I used the SUVAT equations to find the ball’s displacement at each point in the projection so its position is updated correctly |
| 1.3 Show velocity vector as it moves | **a3, pg** | The velocity vectors in the x and y directions are shown as arrows from the centre of the particle |
|  | | |
| 1. Have a GUI to display projection | **B, pg** | All the features of the projection are displayed on a separate window |
| 2.1 Use Pygame to create a window when the code runs | **b1, pg** | The pygame window opens when the user runs the program |
| 2.2 Close the window when the program stops running | **b2, pg** | The window closes when the user presses quit |
|  | | |
| 1. Allow user to edit initial values | **C, pg** | An initial menu is created so the user can input initial conditions and values for the simulation |
| 3.1 Create buttons and sliders for user inputs | **c1, pg** | I created different classes for both Buttons and Sliders that take user inputs and translate them to the simulation |
| 3.2 Edit initial velocity | **c2, pg** | The length of the arrow connecting the cursor to the ball determines its initial velocity so user can move the cursor further or closer according depending on the required velocity |
| 3.3 Edit projection angle | **c3, pg** | The user can edit the launch angle by moving the cursor so the arrow connecting it to the particle points to the required direction |
| 3.4 Toggle velocity arrows | **c4, pg** | In the initial menu, the user can choose to show or hide velocity arrows by clicking the Show Velocity button |
|  | | |
| 1. Add a game aspect | **D, pg** | There are targets on the window that the user can launch the ball into in order to gain points |
| 4.1 Create target class for the ball to hit | **d1, pg** | I created a Point class to represent the target for the ball to hit |
| 4.2 Add a points system | **d2, pg** | If the ball hits a target, the user’s score increases by one |
| 4.3 Add a countdown using Time import | **d3, pg** | I did not do this, but I instead set a maximum number of times the user can launch the ball, so if they run out of launches the simulation ends. Collecting points adds an extra launch to the launches remaining |
|  | | |
| 1. Add interactions with walls | **E, pg** | If the ball hits any of the four edges of the screen, it bounces back |
| 5.1 Add bouncing using Rect objects | **e1, pg** | I did not use Rect objects to detect collisions with the walls, however, I did use them for collisions with obstacles. |
| 5.2 Use restitution to find velocity after bouncing | **e2, pg** | I created a method in the Particle class that finds the velocity of the ball after a bounce given the value for the coefficient of restitution |
|  | | |
| 1. Add obstacles the user must project the ball over | **F, pg** | Obstacles are created on the window in random positions so the user can bounce the ball off them |
| 6.1 Create obstacle class | **f1, pg** | I defined an Obstacle class so they could be created quickly |
| 6.2 Make moving obstacles | **f2, pg** | I defined a class that inherits from the Obstacle class that creates moving obstacles |
| 6.3 Allow the user to decide whether obstacles are present | **f3, pg** | In the initial menu, the user can choose if obstacles are present by pressing the Show Obstacles button |
|  | | |
| 1. Add different atmospheres | **G, pg** | Instead of adding different atmospheres, I made it so the user can vary the strength of gravity and therefore the acceleration of the ball in the simulation |
| 7.1 Allow user to input atmospheres and use according values for acceleration | **g1, pg** | In the initial menu, there is a slider the user can use to vary the ball’s acceleration due to gravity |
| 7.2 Change background of screen depending on atmosphere | **g2, pg** | I did not do this because when I put an image as the background of the window it made the projection less clear so I decided I should prioritize the clarity of the simulation instead by having a solid colour background |
|  | | |
| 1. Display important values of the projection on screen | **H, pg** | When the simulation begins, the current values of the particle velocity, launch angle, number of bounces, score, and launches remaining are displayed |
| 8.1 Display all current SUVAT values | **h1, pg** | I didn’t display all the SUVAT values because I didn’t think they were all important, so I just displayed the velocity |
| 8.2 Use Pygame to display text on screen | **h2, pg** | I used pygame to show the values on the top left corner of the screen |
|  | | |
| 1. Make all aspects of the simulation distinguishable and clear | **I, pg** | The simulation is very clear overall. I used simple shapes for everything, ie, either rectangles or circles, so the projection is easy to see |
| 9.1 Use bright colours for different objects | **i1, pg** | I defined bright, clear colours as constants and used them for all the displayed features, making sure they were all different colours and therefore easily distinguishable, ie, the particle is black, the points are yellow, the velocity arrows are red, and the obstacles are dark blue. |
| 9.2 Use clear font to display text and values | **i2, pg** | The font I used for all the text is Arial, size 30, as it’s easy on the eyes and the clearest font to read. |

## 3.3 Notable Algorithms/Features

Here is a table of notable algorithms and features that I used in my solution, as well as markers to where they can be found in the code.

|  |  |
| --- | --- |
| FEATURE | MARKER |
| Lists and list operations | **A, pg** |
| Organised Files | **B, pg** |
| Complex Maths algorithms | **C, pg** |
| Recursion | **D, pg** |
| Complex Object-Oriented Programming | **Shown throughout the program with lots of different class definitions** |
| Inheritance | **E, pg** |

# Testing

## 4.1 Test Video

Here is a video of me testing out all the different features of the simulation.

\*\*\*PUT VIDEO\*\*\*

## 4.2 Test Table

Here is a table of the things tested in the video and any potential issues that arose. Green, amber and red results represent high, medium and low levels respectively. The time column shows at what timestamp this test appears in the video:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | Time | Result | How test was carried out | Details |
| Menu Screen appears properly |  |  | Program was run |  |
| Buttons display properly |  |  | [observation] |  |
| Sliders display properly |  |  | [observation] |  |
| Buttons operate properly |  |  | Buttons were pressed to see if they toggle colour |  |
| Sliders operate properly |  |  | Sliders were dragged to different values to see if correct value shows | have to move slow |
| Button and Slider values translate properly to simulation |  |  | Simulation was played with different input values to make sure each one works | trail |
| Launch arrow shows properly and moves with mouse |  |  | Mouse was moved around to make sure it remains connected by the arrow |  |
| Particle launches properly |  |  | Mouse was clicked to make sure particle launches |  |
| Particle comes to rest after an appropriate amount of time |  |  | [observation] |  |
| Particle relaunches properly |  |  | Particle was launched several times consecutively in different directions to make sure it moves properly |  |
| Particle trail shows properly |  |  | Show Trail button was toggled and simulation ran to see if trail appears properly when particle is launched |  |
| Particle velocity shows properly |  |  | Show Velocity button was toggled and simulation ran to see if the velocity arrows show up properly when the ball is launched |  |
| Obstacles show properly |  |  | Show Obstacles button was toggled and simulation ran to see if obstacles appear properly |  |
| Obstacles move properly |  |  | [observation] | can overlap |
| Button inputs work in different combinations |  |  | Ran simulation with different combos of buttons activated |  |
| Values display properly |  |  | Simulation was played and values observed to see if they display and update properly |  |
| Particle bounces against walls properly |  |  | Particle was launched at all 4 walls to see if it bounces properly |  |
| Particle bounces against obstacles properly |  |  | Obstacles were toggled and particle was launched against them to make sure it bounces properly | top dont work |
| Points are displayed properly |  |  | Simulation was run and points observed to see if they show properly | can overlap text |
| Points are collected properly and score and launches are updated |  |  | Particle was launched at points to make sure both the score and launches remaining increment |  |
| Points are regenerated once they are collected |  |  | Particle was launched at points to see if they regenerate |  |
| Simulation ends once all the launches have been used up |  |  | Particle was launched enough times so that the launches ran out to make sure the simulation ends |  |
| Reset button works properly |  |  | Reset button was clicked during the simulation to make sure it goes back to the menu screen |  |
| Quit button terminates the program |  |  | The quit button was pressed on both the menu screen and the simulation screen to ensure the program terminates |  |

# Evaluation

## 5.1 Checking back with Objectives

After completing my project, I decided to look back at my initial objectives to see to what extent I managed to complete them.



Overall, I managed to complete all of my main objectives, with some slight changes here and there. I missed out on a couple of the more specific objectives, like 4.3 and 7.2, but I’m not too worried about those as their absence doesn’t take away from the main goal of the project.

I also added some things that weren’t explicitly included in the initial objectives but I’m glad I incorporated anyway, such as the Particle trail and the menu screen, which I think work really well.

## 5.2 Final Check-In with Stakeholder

After completing my project, I did a final check-in with Theo so he could see the completed solution. Overall, his reaction was positive, but he had a few small pointers as well. Here is a table showing some of his main thoughts:

|  |  |
| --- | --- |
| Positive Features | Things to improve for next time |
| * The user inputs are good, it’s useful to be able to see the effects of changing things like restitution and acceleration * The moving obstacles add another level of complexity that is nice to experiment with * I like that the launches remaining increase as you collect points * The whole thing is really clear which is very important as it makes the projection really easy to observe * I like that you can change the size of the ball without losing any functionality * I like that the obstacles and points appear randomly as it means it’s not the same simulation each time | * Make the game aspect optional in case the user wants to focus more on the projectile simulation without being distracted by the game features * Include information or instructions about the game aspect because it’s not very self-explanatory on its own * Maybe include general instructions on how to use the simulation, for example, tell the user that the arrow length determines the initial velocity of the particle |

## 5.3 Future Improvements

If I had more time, these are some improvements I would add to my project, based not only on Theo’s final thoughts but also on my personal opinions about what I thought the project was missing:

To do:

* Vid and timestamp/editing
* Future imprs

\*\*\*\*FINAL STEPS FOR WEDS\*\*\*\*

* Put page numbers in objectives and features
* Put all vids and stuff in folder github
* Fill in checklist w pg nums