projectile motion simulator

Jack Sanders

Contents

[Analysis 3](#_Toc63247107)

[Problem Analysis 3](#_Toc63247108)

[The Idea 3](#_Toc63247109)

[Stakeholders 3](#_Toc63247110)

[Why Use a Computational Solution? 4](#_Toc63247111)

[Research 5](#_Toc63247112)

[Existing Solutions 5](#_Toc63247113)

[Features 11](#_Toc63247114)

[Essential Features: 11](#_Toc63247115)

[Limitations: 11](#_Toc63247116)

[Requirements 12](#_Toc63247117)

[Hardware: 12](#_Toc63247118)

[Software: 13](#_Toc63247119)

[Success Criteria 13](#_Toc63247120)

[Design 14](#_Toc63247121)

[Version Breakdown 14](#_Toc63247122)

[Key Usability Features 15](#_Toc63247123)

[User Interface 15](#_Toc63247124)

[Decomposition 15](#_Toc63247125)

[Version 1 – Gravity and collisions 15](#_Toc63247126)

[Version 2 – Interface and Velocities – Classes 19](#_Toc63247127)

[Version 2 – Interface and Velocities – Flow Charts 24](#_Toc63247128)

[Version 2 – Interface and Velocities – UI Design 28](#_Toc63247129)

[Version 3 – Better User Experience – Overview 28](#_Toc63247130)

[Version 3 – Better User Experience – Classes 29](#_Toc63247131)

[Version 3 – Better User Experience – Flow Charts 36](#_Toc63247132)

[Version 3 – Better User Experience – UI Design 38](#_Toc63247133)

[Key Functions and variables 42](#_Toc63247134)

[Test Plan 45](#_Toc63247135)

[Version 1 45](#_Toc63247136)

[Version 2 46](#_Toc63247137)

[Version 3 47](#_Toc63247138)

[Development 48](#_Toc63247139)

[Version One 48](#_Toc63247140)

[Validation 52](#_Toc63247141)

[Testing 52](#_Toc63247142)

[Development of Version 1: Review 54](#_Toc63247143)

[Development of Version 1: Full Code Listing 55](#_Toc63247144)

[Version Two 58](#_Toc63247145)

[Development of Version Two: Interface Module (Pgkinter) 58](#_Toc63247146)

[Development of Version Two: Main Program 64](#_Toc63247147)

[Testing 67](#_Toc63247148)

[Development of Version 2: Review 71](#_Toc63247149)

[Development of Version 2: Stakeholder Feedback 71](#_Toc63247150)

[Version Three 71](#_Toc63247151)

[Development of Version Three: Interface Module (Pgkinter) 71](#_Toc63247152)

[Development of Version Three: Main Code 84](#_Toc63247153)

[Testing 114](#_Toc63247154)

[Development of Version 3: Review 122](#_Toc63247155)

[Evaluation 122](#_Toc63247156)

[Post Development Testing 122](#_Toc63247157)

[Regression Testing 122](#_Toc63247158)

[Usability/Robustness Testing 122](#_Toc63247159)

[Success Criteria 123](#_Toc63247160)

[Criteria Met 123](#_Toc63247161)

[Evidence 123](#_Toc63247162)

[Usability Features 127](#_Toc63247163)

[Limitations 128](#_Toc63247164)

[Maintenance 128](#_Toc63247165)

[Final Code 129](#_Toc63247166)

# Analysis

## Problem Analysis

### The Idea

An understanding of how physics works (gravity, momentum, etc.) is one of the first things that humans learn when they grow and develop. However, the exact calculations that relate to physics are something that generally isn’t taught until secondary school.

My idea is to make a physics simulator that will be able to help to educate students on the basics of mechanics and motion, as well as allowing them to see the changes that different factors bring about with regards to motion.

I plan to make this in Python, using the Pygame library to do animations and visualisation. I also intend to create my own library that will be functionally similar to Tkinter, but written in Pygame, allowing me to place widgets such as checkboxes, buttons, and input boxes into a Pygame window.

### Stakeholders

My stakeholders would primarily be physics teachers, teaching at GCSE level. As GCSEs are compulsory in-school education, that every child in the UK must do, teachers will have to cater for a very wide range of pupils. For example, while some may be willing and able to learn from textbooks, others may be visual learners, needing a greater emphasis placed on demonstrations of physics. My project will be exactly that – a visual demonstration of physics, intending to aid in teaching those students who require more visual teaching methods. It will achieve this by providing helpful diagrams, accurate physics, and the ability to set up scenarios and then track individual particles as the simulation progresses, making for a polished, high-quality tool that GCSE teachers will be able to use in the classroom.

My stakeholders’ requirements would be accurate, true-to-life physics with a simplified display, but one that is still engaging enough to keep students interested. Also, as many schools have somewhat dated hardware, my stakeholders would have the additional requirement of the program being well-optimised and not very demanding to run. My solution also has the benefit of being developed in python, which many schools have installed on their systems, rather than an exe file, which many schools will block from running, unless it is pre-approved by the system administrator.

### Why Use a Computational Solution?

While it is indeed possible to see the effects of gravity and velocity outside using tennis balls, some conditions (such as wind speed or air resistance) are out of your control, and will affect the flight of the tennis ball, making it much harder to perform accurate calculations. A computational solution would bring all these conditions and more under your control, and therefore it would be a much easier way to study mechanics than throwing tennis balls.

#### Decomposition

This project can be broken down into three simple steps, which will make the program much simpler to create. My ideas for the steps are:

1. Simulating spheres of a fixed size and material falling under the effects of gravity and colliding with one another.
2. Allowing the user to add speed and direction to the spheres, therefore allowing them to set up scenarios such as a cannon firing.
3. Adding the ability to change the radius/material of the spheres and other additional features.

Following this outline, I will be able to concentrate my effort towards one single part of the project, and will ensure that it is fully developed and bug-free before moving on to the next part.

#### Abstraction

An exact simulator of real life will require massive amounts of computational power and programming time due to the sheer number of factors that will effect a ball’s flight. Abstraction will allow me to create a simulator that is simple to use, predictable, and appealing to look at all while remaining accurate enough for its intended purpose.

Some examples of abstraction that I could apply to my solution go as follows:

* 2D basic graphics and motion. While in reality, objects can and will move along 3 main axes, GCSE and A-Level physics education only deals with two. 3D modelling and simulation would also require a lot more processing power and programming time.
* Ignoring the effects of air resistance. Air resistance is usually assumed to be negligible when performing projectile motion equations throughout GCSE and A-Level, and therefore there is no point including it in my simulator, as it will only massively complicate the calculations for little to no benefit.
* Only visualising circular projectiles. Adding multiple shapes of projectiles would be good in terms of added realism, however each extra shape would mean extra collision masks, and extra formulae needed to calculate the mass of the object. Also, as I am making air resistance negligible, all shapes will behave identically and therefore do not necessarily need to be included.

Overall, abstraction allows me to simplify unnecessary tasks that would otherwise take a lot of time and effort to program, and instead focus on developing the features that will have a noticeable benefit on the outcome of my project, i.e. accurate physics.

#### Formulae

Naturally, a projectile motion simulator will require many formulae to accurately simulate reality. These calculations could be performed by hand, but they would be extremely laborious. Especially so when considering that in order to get the most accurate simulation possible, they may have to be calculated and recalculated hundreds of times per second, which is impossible for any human to do.

This will be used to determine how quickly a projectile will be moving in each direction based off the time that has elapsed and the acceleration (from gravity or collisions)

The equation for momentum will be used to calculate the amount that an object will speed up/slow down when it collides with another object

These equations will be used to calculate the new momentum of both particles involved in a collision, using the total momentum of the collision, p, and the angle between the two particles, theta.

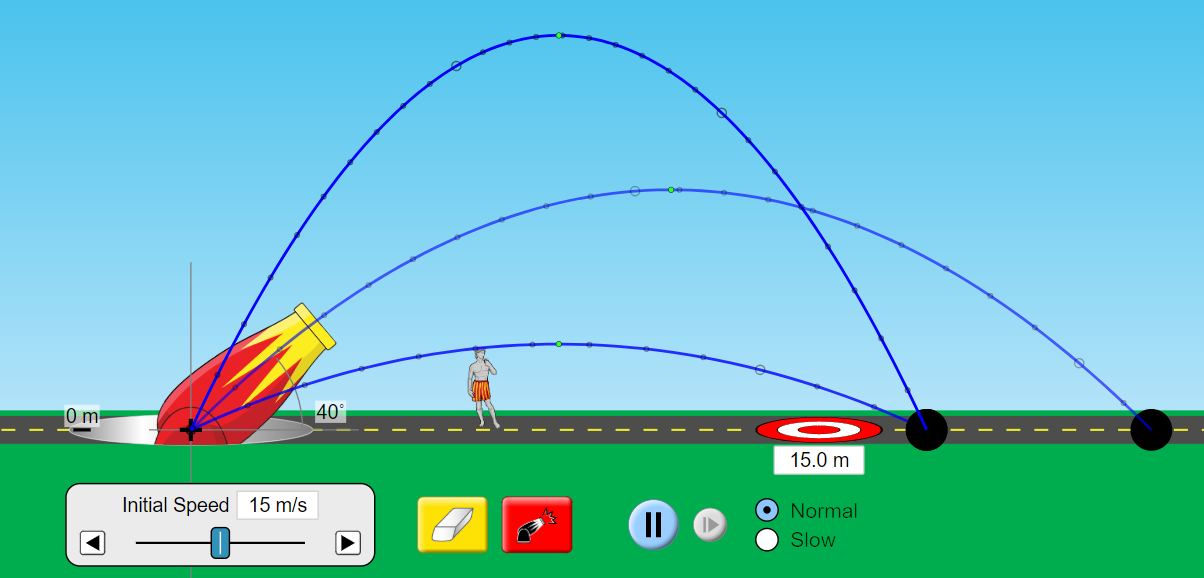
Equations used to calculate how far to move an object in each direction based on the magnitude (v) and angle (θ) of its velocity. Returns the amount of pixels on screen to move the object. Angle is measured in radians anti-clockwise from the horizontal.

## Research

### Existing Solutions

#### PhET Projectile Motion

<https://phet.colorado.edu/sims/html/projectile-motion/latest/projectile-motion_en.html>



7

6

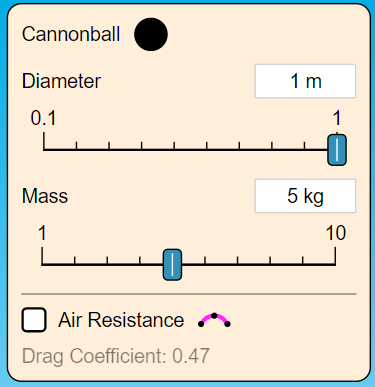
5

4

3

2

1



8

##### Adjustable launch angle

This feature allows the user to change the angle at which the sphere is fired, allowing them to see the difference that launch angle will make on the flight of the object. It also allows for the setup of more complex scenarios. This is an essential feature and therefore **it will be included in my project.**

##### Adjustable initial speed

This gives the user the ability to see what happens to the flight of the object when it has more/less speed. This is essential and **will also be included.**

##### Highlighted flight path

This makes it easier for the user to find specific points during the object’s flight (highest point etc.), which is very useful for checking calculations. **I will be including this.**

##### Time controls

These allow the user to slow down the simulation, which will make it easier to see what is happening, which will be especially helpful when momentum is involved. **I will try to include this, if it can be done in pygame.**

##### Target distance

This is a feature that I initially hadn’t thought of adding. However, on looking at PhET’s simulator, it adds a slight ‘game’ aspect to it, and will make it easier for students to stay interested while learning about projectiles. Also, it adds incentive to perform calculations before firing. **I will include this**.

##### Graphics

The simulator’s graphics include things such as, a road, sky, grass, a cannon, and a human for scale. While these may be nice to have, drawing the sprites would take up a lot of time for marginal gain, as extremely detailed graphics will not be helpful to my stakeholders as my project is intended as an educational tool, and over-complicated graphics will distract from the main idea that they are trying to teach. Therefore **I will only be using very barebones graphics.**

##### Adjustable launch height

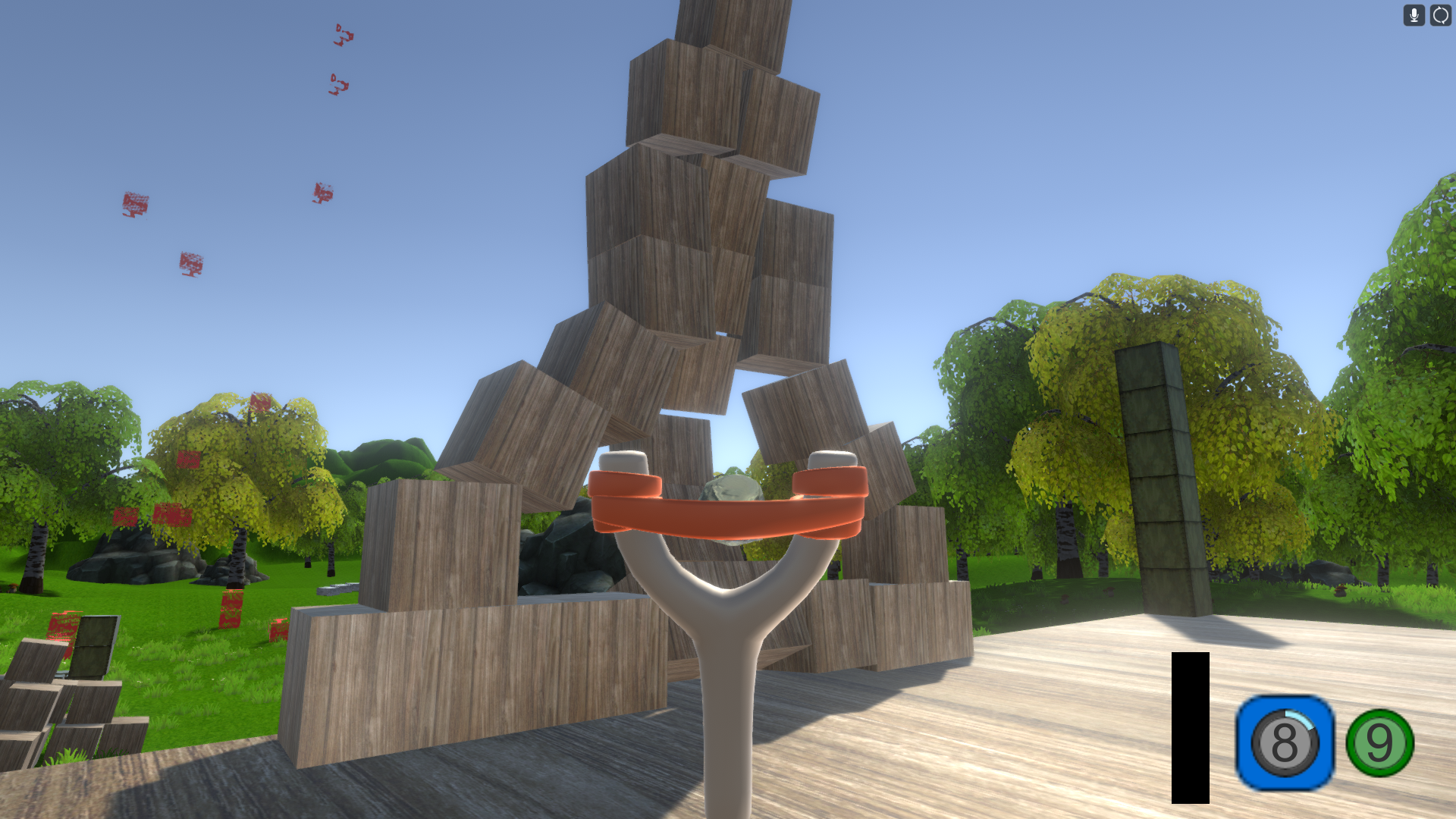
This feature allows the user to change the height from which the sphere is fired, allowing them to see the difference that height above the ground will make on the flight of the object. It also allows for the setup of more complex scenarios. This is an essential feature and therefore **it will be included**.

##### Adjustable properties

This allows the user to change the diameter and mass of the ball, in order to observe how those properties impact the results of the simulation. **I will be including this in my project** as mass is fundamental to all momentum calculations.

#### Physics Playground

<https://cammin.itch.io/physics-playground>



5

4

3

2

1

This physics simulator is one that has a different focus to what mine will have, being more of a sandbox type game than a learning tool. However, the physics in it are still seemingly accurate, and there are some features from it that I will be attempting to implement within my project.

##### Detailed 3D environment

This game was created in Unity, and therefore has a very detailed and appealing environment, which consists of foliage, a skybox, and 3D models for objects that are effected by physics. While the 3D graphics work well in this situation, my implementation will not be a game, but a learning tool. 3D graphics will just overcomplicate things, and make it much harder to understand what is going on. **I will not be including graphics to this detail**. Instead I intend to use simple but easy to understand 2D graphics.

##### Accurate collisions

The collisions in this game are largely accurate in that everything acts as you would expect it to when struck by another object, for example, when one box is hit by another, they both move in the correct direction. **I will be including accurate collisions**, although they will not need to be as accurate as they are in this game, as objects will only move in two dimensions.

##### Playable character

In this game, you play as a character with a slingshot, which you can use to knock boxes over. This is the primary method of interacting with the environment and experimenting with physics. **My project will not have a playable character**, but instead the user will be able to edit the parameters of ‘particles’ that they place (Velocity in both axes, mass, radius, material etc.) in order to interact with the simulation.

##### Boxes with varying materials

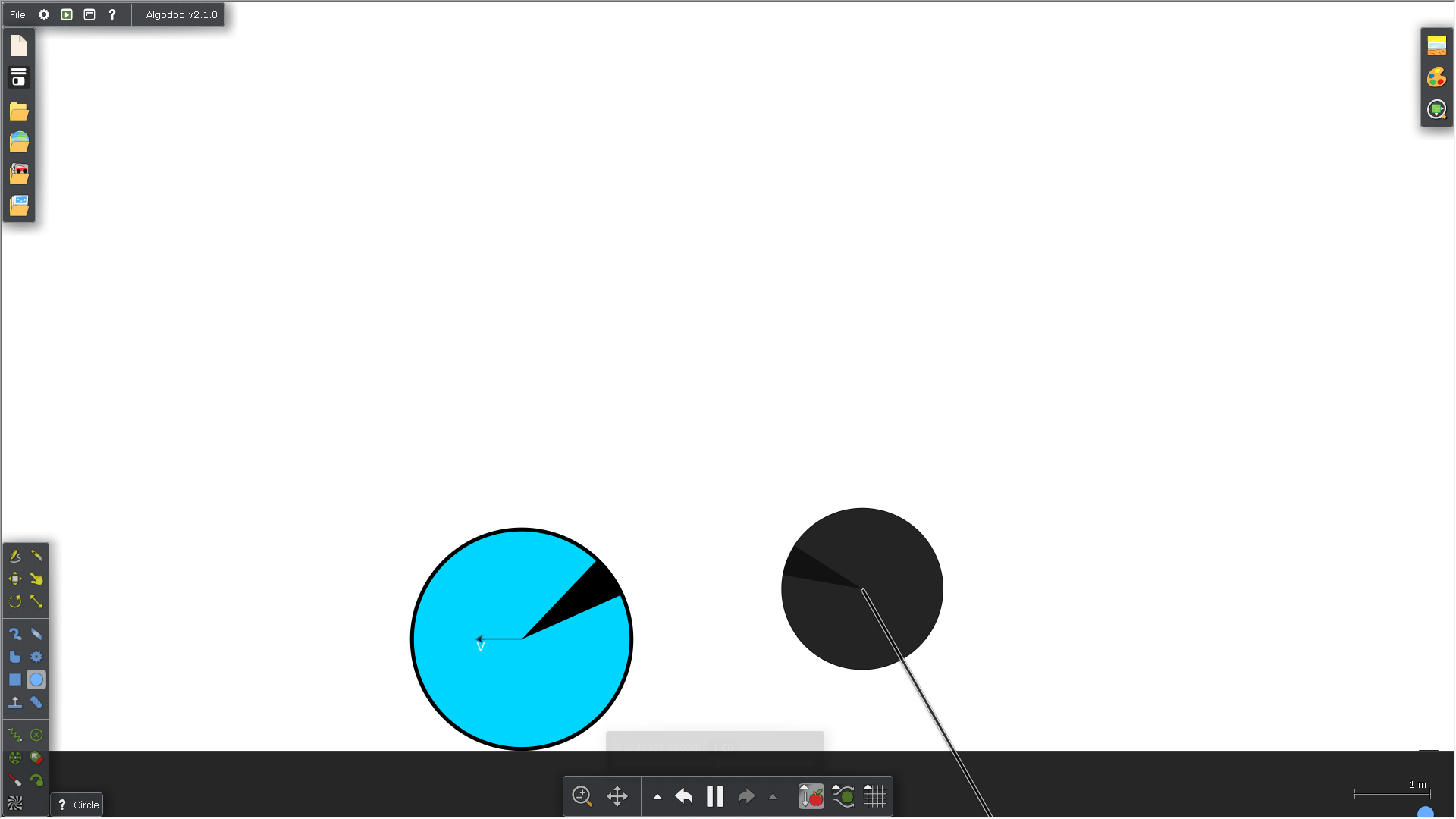
In this sandbox, there are two types of physics-effected boxes – wooden ones (centre), and metal ones (right) – each of which have different properties. For example, the wooden ones are much lighter than the metal ones, and so require much less force to move. **This will be included in my implementation**, although they will be spheres – represented as circles.

##### Ammo counter & timer

In the game, you have a limited amount of ammo that you can fire from your slingshot, as well as a timer between shots (the ring around the outside of the counter). This is presumably to prevent the user from firing hundreds of shots, therefore creating hundreds of objects and slowing down the game massively. **My project will have a similar restriction** in that there will be a 0.25 second timer after placing a particle before you can place another one.

#### Algodoo

<http://www.algodoo.com/>



63

5

2

4

3

1

Algodoo is a physics simulator that is more in the vein of what my project will be, being purely a simulator rather than a game. It has very accurate physics, as well as a host of features that I will be taking inspiration from for my project.

##### Save/load scenarios

The menu in the top left allows the user to save a scenario that they have created, or load one from various categories. This could potentially prove to be extremely useful for teachers - if they want to teach the same lesson to multiple classes, they can simply save the setup of a scenario, play it for one class, then reload it for the next. **I may include this** if I have time at the end of version 3 of my project.

##### Various drawing tools

These allow for greater variety in the simulation – the user can use any shape that they can think of. However, more shapes add more complexity to the simulation algorithms. Therefore, **I will not be including the majority of these tools**. One tool that I will include, however, is the erase tool, so that the user can delete particles that they no longer want on screen.

##### Simple 2D graphics

The graphics in this simulator are easy to understand, simple, and not at all resource intensive. This is great for a simulator, as the primary focus will be on the accuracy of the physics, as opposed to how pretty it looks. The simulator also has an optional velocity arrow, which I will be including as well. **These are the type of graphics that I will be using**.

##### Simulation controls

These controls offer the user the ability to change the speed of the simulation, zoom in or out, or even toggle gravity/air resistance. These are all very useful features, as it allows the user to change parameters depending on the scenario they want to simulate. **I will be including some of these features**, such as zooming in and out or pausing the simulation. I will also be including a gravity setting, except instead of a toggle, the user will be able to input any value.

##### Particle settings

These settings allow the user to change the properties of particles that they have placed (material, density, restitution). **I will be including a menu like this**, except the density and restitution will be tied to the material, and not editable separately. Also, I will include an option that allows the user to set a starting velocity and acceleration for the particle.

##### Scale

This is a simple line on-screen that shows the user how big one metre is in the simulation. **I will include something similar to this**, although it will be a grid background, with the distance between each line being 1m.

## Features

### Essential Features:

1. **Simple 2D graphics with all the necessary information** – 2D graphics are more than detailed enough to convey the key ideas of my project, while at the same time not being as resource-intensive as 3D graphics. This was the graphical style of choice implemented by 2/3 of the pre-existing solutions that I researched, and by far made understanding what was happening in the simulation easiest.
2. **Adjustable particle settings –** The user will be able to change many of the properties of the particles – such as radius, mass, material, starting velocity, starting acceleration, launch height, and launch angle. This will allow them to fully customise the simulation in order to create virtually any scenario in order to assist with teaching.
3. **Simulation settings** – The user will also be able to change settings that are not to do with the particles themselves, but rather the simulation. These settings will allow the user to pause/play, change the strength of gravity, as well as other visual settings, such as zooming in or out, and highlighted flight paths or velocity arrows. These things, while not strictly essential for a physics simulator, are always nice to have, as they will massively improve user experience and ease of use.
4. **Momentum** – Each particle will have its own momentum values (one for each direction) that will be updated every time its velocity changes. I will be using the equations that I have stated earlier in order to calculate particles’ new velocities after collisions.
5. **Gravity** – The user will be able to change the value of gravity so that they can, for example, simulate a ball falling on the moon. The default value will be 9.81Nkg^-1, but there will also be set presets for different planets (for example, 3.71Nkg^-1 for Mars, or 24.79Nkg^-1 for Jupiter), as well as an option to enter a custom value. These different values will assist my stakeholders in showing students that objects act differently with varying forces of gravity
6. **Universal hardware support –** being a physics simulator, accurate physics is imperative. I will ensure that the physics in my project are as accurate as possible by conducting multiple tests and trying out various settings (different FPS, screen size, even using different monitors.) in order to see how they affect the accuracy of the physics, and I will implement universal support for all possible environments.

### Limitations:

1. **Graphics –** 3D graphics would be extremely time-consuming to develop, as well as being more taxing on the computer, as it means that there is another dimension to consider when simulating collisions and velocity. Another reason I am not using 3D graphics is that they are harder to understand and so would not make for a great teaching aide. Therefore, I will be sticking with flat 2D graphics in order to make it easier to develop, as well as more effective for the user to learn from.
2. **Sound –** My simulator will not have sound. While sourcing sounds for various materials colliding may not prove to be too challenging, it will take a lot of time to implement, while providing zero gain for the user, as there would be zero difference between using the simulator with sound on, or with sound off.
3. **Air resistance –** Air resistance tends to be considered negligible in all physics education until university, and therefore I will not be including it as it will cause the user’s calculations to be inaccurate if they are assuming air resistance to be negligible (which my target audience would)
4. **Effects of wind** – While wind plays a very important role in real-life physics, it is often unpredictable, and will render equations incorrect. I have chosen not to include wind in my solution as it would be impossible to accurately model it. Also, my project is intended to show students how accurately physics can be calculated, and if wind were to be included, the program would have different results to the equations, which would show students the opposite of my program’s intention.
5. **Particle shape** – My program will be built to treat all particles as circular objects. I have made this choice as circles are easy to accurately model – other shapes, such as squares, act differently when colliding with their flat sides or their corners, whereas circles act the same no matter where they collide with another object. This simplifies the program massively, as I will not have to code multiple shapes, and then multiple behaviours for those different shapes.

## Requirements

|  |  |  |
| --- | --- | --- |
|  | Requirement | Explanation |
| 1 | Ability to place particles | So that the user can set up customised scenarios in order to see what happens when they are subject to accurate physics. |
| 2 | Ability to customise properties of particles | So that scenarios can be further customised in order to show how mass and velocity effect behaviour of particles. Also allows the program to be used within GCSE & A-Level classroom settings. |
| 3 | Ability to save and load scenarios | Allows for more extensive use within classroom settings. The teacher will be able to reuse scenarios in multiple lessons, knowing that every time they run them, they will behave identically. |
| 4 | Ability to change simulation settings | Lets the user adjust the simulation settings (e.g. window size, scale, camera position, and simulation speed) to further suit their needs. |
| 5 | Ability to monitor particles | If the user clicks on a particle while the simulation is running, they will be greeted with a small window that contains stats for the particle, such as velocity, acceleration, height off of ground, and distance travelled. |
| 6 | Distance markers | These will make it much easier for the user to see how far a particle has travelled, which would help teachers massively, as a scenario that this program would be perfect for would be a cannon firing, in which distance travelled is an important aspect. |
| 7 | Menus | Menus will be a necessary requirement in my program, as they will provide an easy way for the user to navigate the program, and find the settings that they want. |
| 8 | Compatibility | The program needs to be able to run on any computer (with powerful enough hardware) regardless of the monitor’s resolution or aspect ratio. |

### Hardware:

A computer that is powerful enough to run the program. As my program will not be extremely resource-intensive, this should not be too much of a limiting factor.

A mouse, keyboard and monitor are required for the user to be able to see and control what is happening in the simulation.

### Software:

Windows 10 or higher will be required, as this program will be written and tested entirely on windows 10.

A python interpreter that supports version 3.7.1 will be required to successfully run the code on your system.

The pygame external library will also be required, as without it, it would not be possible to do any form of animation or graphics using Python.

The pgplot library will be required to run the final version of the project as it will be used to draw graphs of velocity against time for certain particles.

## Success Criteria

|  |  |  |
| --- | --- | --- |
|  | *Criteria* | *Explanation* |
| *1* | 2D graphics | Are the simulator’s graphics 2-dimensional and easy to use? |
| *2* | Placeable particles | Can the user place particles anywhere on-screen? |
| *3* | Changeable properties of particles | Can the user change radius, mass, starting velocities, material, and starting acceleration of particles? |
| *4* | Instructions | Does the simulator provide instructions on how to use it when it is first opened, and can the user access those instructions again? |
| *5* | Simulation settings | Can the user pause/play the simulation?  Can they change the scale?  Can they change the screen size? |
| *6* | Momentum | Is the transfer of momentum accurate when two or more particles collide? Do the particles move in the correct direction? |
| *7* | Gravity | Can the player change the strength of gravity? Does it accelerate the particles at the correct rate? |
| *8* | Universal support | Does the simulator act in a similar manner no matter what framerate or screen size it is running at? |
| *9* | UI | Is all information accessible? Can the user interact with the program entirely within one window? |

# Design

## Version Breakdown

## Key Usability Features

There are many features that I will be including in my project in order to improve the user experience. Below is a list of the most important ones – the ones that will improve UX the most.

* **Clearly labelled menu items**. The project will have a sidebar style menu, similar to the one in the third pre-existing solution that I researched (algodoo). To make this as user-friendly as possible, I will make sure to clearly label every option, making the function of each button obvious.
* **Clear and concise instructions.** When the user first loads the program, they will be greeted with an instructions page. It is very important that the information shown on this page is conveyed clearly, as without it, some features may not be immediately obvious.

## User Interface

As my project is a simulator, rather than a game, most of the effort will be focused on the accuracy of the physics rather than the graphics. Therefore, I do not plan to create any custom sprites for it, and will instead use simple shapes that are available within the pygame library. For example, all particles will be represented as circular objects of varying size and colour depending on material and radius. There will also be no texture for the ground, as that will instead be represented by the bottom of the window.

The selection screen will also be designed using shapes available in pygame. I intend it to look similar to the properties screen in the first pre-existing solution that I researched (PhET Projectile Motion), although I will replace the sliders with boxes that the user can type their desired value into.

A screenshot of a cell phone

Description automatically generatedI have designed a mockup of how I would like the particle menu to look in photoshop. This menu will be constantly in the upper-right hand corner of the window until the simulation has started, and every particle that you place will have the properties set in this menu (Y acceleration will be added on to the global force of gravity.)

## Decomposition

### Version 1 – Gravity and collisions

As of version 1, there is only one class in the program, the Particle.

#### Particle Object

The particle object will be the primary object in the program, and it will inherit attributes and methods from the pygame Sprite class, which will allow me to add it to a sprite group, therefore giving me the ability to call the group .update() method, which will call the .update() method of every sprite in the group.

#### A screenshot of a cell phone Description automatically generatedParticle Object - attributes

**Radius** is a simple float that stores the size of the circle’s radius in pixels.

**Velocity** is a 2D Vector (a class added in the Pygame module) that stores the value of the velocities in the x and y direction separately. The use of a vector is useful as you can easily add it to other vectors (adding velocity to position, for example) or calculate its magnitude and direction. Vectors are also necessary for the equation that I will be using to calculate velocities after collisions.

**Mass** simply stores the mass of the sphere (currently a constant but in later versions it will be calculated based off of the volume of the sphere and density of the chosen material)

**Momentum** is, like velocity, a Pygame 2D Vector

**Colour** is purely cosmetic. It is a tuple that is used by the pygame.draw function to set the colour of the circle

**Rect and pos** control the location of the circle after it has been created. Rect is a Pygame Rect object that is used by the draw method to place the particle on the screen, and pos is a pygame 2D vector that will contain the exact position of the particle.

**Acceleration** is another 2D vector, this time storing the acceleration of each particle object. It will divided by the frame rate and added to the velocity vector every loop.

**Direction** will be updated every frame using the inverse tan of velocity.y over velocity.x to get the angle from the horizontal to the direction in which the particle is moving.

##### Particle Object – methods

**\_\_init\_\_** is called only once: when the object is instantiated. Its purpose is to assign all of the attributes with their starting values and draw the circle in its initial position.

**angleTo** will be called in the collide procedure and will return the angle between the direction of a particle’s momentum and the centre of another.

**Collide** will be called by the hasCollided method every time the particle collides with another. It will simply use the equation that I mentioned in the analysis section to update the velocity vectors of each particle involved in the collision.

**draw** will be called every update cycle to redraw the circle in its new position.

**hasCollided**, again, will be called every update cycle and will run through every sprite in the ‘particles’ sprite group’ and check if the distance between their respective centres is less than the sum of their radii. If it is, then they have collided.

**updateDirection** is called every update cycle and will use the inverse tan of yVelocity over xVelocity to update the direction attribute.

**update** will be called every frame. It will check the position of the particle to ensure that it is still onscreen, and it will also update attributes relating to position/motion, as well as calling the hasCollided and updateDirection methods. Finally, it will call the draw function.

#### A close up of a map Description automatically generatedFlow Charts

##### Main loop

Since I am making this primarily using pygame, I will need a clock-controlled loop. This flow chart outlines how it will run, at 60 times per second (or whatever the frame rate is set to)

##### Pseudocode for detecting mouse press and instantiating a particle object

if event.type == MOUSEBUTTONUP:

if event.button == 1:

if timer > 0.25:

particles.add(Particle(50, (0, 0), (144,202,249), getMouseCoords(), 9.81))

            timer = 0

This code will run at the start of every event loop in pygame. It will check whether the user has lifted a mouse button, then it will check if that button is the left mouse button. If both of these conditions are true, and it has been more than 0.25 seconds since the last particle was placed, the program will instantiate another particle object at the mouse pointer’s location.

The timer helps to somewhat reduce lag - by adding a limit to the amount of times per second that the user can place a particle, there will be less particles on-screen for the computer to perform calculations about. It will also help to prevent accidentally placing multiple particles.

##### Particle update methodA screenshot of a cell phone Description automatically generated

##### Pseudocode for collisions with screen edges

if yPos + radius >= SCREEN\_HEIGHT and yVelocity > 0:

yVelocity = round(yVelocity \* COEFFICIENT\_RESTITUTION \* -1, 4)

yPos = SCREEN\_HEIGHT - radius

elif yPos - radius <= 0 and yVelocity < 0:

yVelocity = round(yVelocity \* COEFFICIENT\_RESTITUTION \* -1, 4)

yPos = 0 + radius

if xPos + radius >= SCREEN\_WIDTH and xVelocity > 0:

xVelocity = round(xVelocity \* COEFFICIENT\_RESTITUTION \* -1, 4)

xPos = SCREEN\_WIDTH - radius

elif xPos - radius <= 0 and xVelocity < 0:

xVelocity = round(xVelocity \* COEFFICIENT\_RESTITUTION \* -1, 4)

xPos = 0 + radius

This section of code will need position ± radius to detect the edge of the circle crossing the screen boundaries, rather than the centre. Additionally, it will ned to check the directional velocity to make sure that the particle is in fact moving off of the screen, as opposed to moving back on screen after having just collided with the edge.

##### Pseudocode for updating velocities in a collision – uses formula from analysis section

p1NewVelocity = v1-((2\*m2)/(m1+m2))\*(((v1-v2).dot(pos1-pos2))/((pos1-pos2).length())\*\*2)\*(pos1-pos2)

p2NewVelocity = v2-((2\*m1)/(m2+m1))\*(((v2-v1).dot(pos2-pos1))/((pos2-pos1).length())\*\*2)\*(pos2-pos1)

##### Pseudocode for updating position vector

pos +=  velocity \* SCALE / FRAME\_RATE

### Version 2 – Interface and Velocities – Classes

In version 2, I will add many different classes that will be used to create an appealing interface that is well-integrated with the Pygame window – unlike Tkinter, which requires a separate window. Currently, the planned interface objects are a button, an input box, and a checkbox, all of which will be nested inside of a control object, as I will be writing the code for the interface in a separate file and so it will behave like an external library.

These new interface objects also mean that I will have to alter the flow of the program. Instead of having one main loop, in which you can place particles while the simulation runs, I will create a setup loop that will run before the main one, allowing the user to customise and place particles for as long as they want, before pressing a button to start the simulation. The only part of the program that will stay the same will be the particle object.

Below is the overall class diagram for version 2

A close up of text on a white background

Description automatically generated

#### Control Object

The control object will be used to ‘communicate’ between the main program and the interface module. I made the choice of having a single control object for the interface so that all interface objects I create (buttons, inputs etc.) can easily access the required functions and constants that they need to work.

A screenshot of a cell phone

Description automatically generated

##### Control Object – Attributes

**pgkGroup** will be a Pygame sprite group that will contain all of the interface objects, allowing me to update the entire interface with one statement.



**pgkTypingCursor** (above) will show in place of the default mouse pointer when you hover over an input box, to indicate that you can type there. It is a black image with a white outline, so it will always be visible no matter what colour the input box is.



**pgkPointerCursor** (above) will show when you hover over a checkbox or a button, as an indication that you can type there.

**pgkWhiteCrossImage** will simply be an image of a white cross that will show on a dark-coloured checkbox if it is active.

**pgkBlackCrossImage** will be an image of a black cross that will show on a light-coloured checkbox if it is active.

**pgkLetterKeys** will be used for input validation in the InputBox class – it will contain the keycodes that Pygame uses for letters, allowing the InputBox to check if the user has entered a letter.

**pgkNumberKeys** will serve the same purpose as pgkLetterKeys, but it will contain keycodes for number keys instead of letters.

##### Control Object – Methods

**\_\_init\_\_** defines the attributes of the Control Object during instantiation

**hoverEffect** will be used to provide a colour that a button will change to when it is hovered over, to indicate that the user can click the button. It will take an rgb tuple and returns another one that is either lighter or darker than the one given, depending on whether that one is light or dark. If the rgb code is light, it will return a darker colour by halving the rgb values of the colour. If it is dark, it will return a lighter colour by adding half of (255 minus the rgb value) for each rgb value in the tuple. This is to ensure that the method will always return a value that is both less than 255, and considerably different to the original (simply multiplying the values would not do, as black (0, 0, 0) would always end up the same)

**isLight** will take an rgb tuple and return True if it is light, or False if it is dark. It will calculate this using Pythagoras’ Theorem to compute the ‘length’ of the rgb code if it were a vector – the shorter the length, the darker the colour. I have decided to use a length of 220 as the dark/light boundary. If the vector representation of an rgb code is shorter than this, it is a dark colour. If it is longer than or equal to 220, it is a light colour.

**update** willsimply run the update method of the pgkGroup, to update all of the interface objects.

#### Button Object

A screenshot of a cell phone

Description automatically generatedThis will allow me to place buttons inside of the main Pygame window, making the UI look much more appealing and intuitive. It will be defined within the control object.

##### Button Object – Attributes

**parent** will be the control object that I will create at the start of the program. Having this object as an attribute of the button makes it easy for the button to access its attributes and methods

**screen** will be the surface that the program is running on, and what the image of the button will blit onto.

**timer** will act as an internal clock for the button, as I am designing it such that the button can only be pressed twice every second. This will be achieved by a simple if statement when handling events that will check that the timer is greater than or equal to 0.5

**previousFrame** will be a float that stores the current unix time (the number of seconds since the epoch, 00:00:00 on 01/01/1970). This will be done using python’s builtin time.time() method, as that returns the number of seconds up to 7 decimal places. I will use this to keep the timer accurate, by adding to it the difference between the current unix time and the previousFrame attribute.

**hovered** will be a simple Boolean that will be True when the user is hovering their mouse over the button, and False when they are not.

**font** will be a Pygame font object that the button will use to generate its text

**bgColour** and **textColour** will be tuples containing the rgb codes for the background of the button and its text, respectively.

**text** will be a string containing what is written on the button.

**height** and **width** will be integers that store the dimension of the button

**action** will be a reference to a function, that will be run when the button is pressed

**bgColourHover** will be another rgb colour code tuple that will be automatically generated using the Control object’s hoverEffect method

**rect**  will be a Pygame Rect object that will store the coordinates of the button, and thus make drawing it on-screen much easier

**displayText** will be a Pygame Surface object containing the image of the button’s text attribute, created using the font.Render() method.

**textRect** will be the Pygame Rect object that corresponds to displayText

##### Button Object – Methods

**\_\_init\_\_** defines the attributes of the Button object during instantiation, and adds it to its parent object’s pgkGroup attribute.

**config** will be a method that can be used to modify the Button Object’s attributes after instantiation.

**defaultAction** will be what the Button’s action attribute gets set to if a function is not passed during instantiation. It will simply do nothing.

**delete** will remove the button object from the parent object’s pgkGroup, at which point there will be no references to it and it will be collected by Python’s garbage collection.

**update** will detect whether the mouse pointer is hovering over the button, and update the hovered attribute accordingly, as well as changing the colour of the button and also changing the mouse pointer. It will also handle mouse click events, and draw the button on the screen.

#### Checkbox Class

This class will be used to allow the user to interact with Boolean variables, changing their state while the program is running.

A screenshot of a cell phone

Description automatically generated

##### Checkbox Class – Attributes

**parent, screen, hovered, font, bgColour, textColour, height, width, and rect** will all serve the same purpose in this class as they do in the Button class.

**output** will be a Boolean that will be True if the checkbox has been checked, and False if it has not.

**inlineText** will be similar to the Button class’s text attribute, although this time, the text will be written and displayed next to the checkbox. I have made this choice as simply generating the text in the main part of my program would require a lot of trial and error to find the correct coordinates for it to look how I want it to, whereas this class will automatically place the text next to it, taking into account the length of the text and the font size, creating a consistent look easily.

**inlineDisplayText** will be the rendered surface of the inlineText attribute that will be displayed on-screen

**inlineDisplayRect** will be the Pygame Rect object that stores the location data for the checkbox’s inlineDisplayText attribute

**outputDisplay** will be a Surface object that represents the status of the output attribute. If output is True, then outputDisplay will be set to the control object’s pgkWhiteCrossImage, or pgkBlackCrossImage, depending on whether bgColour is dark or light. If output is True, then outputDisplay will simply be blank.

##### Checkbox Class – Methods

**\_\_init\_\_, config, and delete** will all serve the same purpose in this class as they do in the Button class, .

**click** will be called when a mouse press is detected, if the mouse is hovering over the checkbox. Its purpose will be to invert the output attribute (True becomes False, False becomes True)

**get** will simply return the value of the output attribute

**update** will handle mouse click events and call the click method whenever a mouse click is detected over the checkbox. It will also detect whether the mouse pointer is hovering over the checkbox, and change the pointer image accordingly.

#### InputBox Class

This class will allow the user to directly enter data into the main program window, and create a more integrated and seamless feel to the program.

##### A screenshot of a cell phone Description automatically generatedInputBox Class – Attributes

**parent, screen, previousFrame, hovered, font, bgColour, textColour, inlineText, width, height, inlineDisplayText, inlineTextRect, and rect** will all serve the same purpose as the attributes of the same name within the Checkbox and/or Button classes.

**timer** will operate using the same method as it does in Button, however, it serves a different purpose in InputBox. For this class, I have taken inspiration from text editors such as word, and will implement a flashing cursor that will show the user that the input box is ‘active’. I will achieve this using the timer attribute to alternate between the cursor showing and not showing every second.

**active** will be a Boolean that represents the state of the InputBox. If the user clicks on the InputBox, it will become active, start the flashing cursor, and start recording keypresses and appending them to the output attribute. If the user clicks anywhere that isn’t the InputBox, active will be set to False, the flashing cursor will stop, and the object will stop recording keypresses.

**caps** will be a Boolean that stores whether or not the user is typing with caps lock on. If they are, then all keypresses detected will be converted to upper case. If not, then they will remain in lower case.

**allowLetters, allowNumbers, allowDecimal, and allownNegative** will for a built-in input validation system. They will use the Controller object’s pgLetterKeys and pgNumberKeys attributes, as well as the decimal (.) and the negative sign (-) to check any keypresses, and only allow the ones whose respective validation attribute is True. For example, if allowLetters is set to False, then only numbers, decimal, and negative signs will be accepted and appended onto the output attribute.

**charLimit** will be an integer that will act as another form of input validation. It will determine the maximum amount of characters that can be typed into the InputBox before no more are accepted.

**outputText** will be the string containing the data that the user has typed into the input box. I will also add the functionality to set a value that will appear in the input box by default, most likely 0. This will mean that the user only has to enter values into the relevant input boxes, and ignore the rest.

**outputTextDisplay** will be the rendered image of the outputText attribute, and it will be updated every time the outputText gets updated, meaning that every time the user types or deletes a character, it will be instantly reflected in the output display.

##### InputBox Class – Methods

**\_\_init\_\_, config, and delete** will all serve the same purpose in this class as they do in the Button and Checkbox classes.

**get** will return the outputText attribute as a string

**update** will handle mouse click events and set the active attribute to True whenever a mouse click is detected over the input box, and set it to False when a mouse click is detected not over the input box. It will also detect whether the mouse pointer is hovering over the input box, and change the pointer image to the control object’s pgkTypingCursor attribute.This method will also handle allkeypress events, and append the respective characters to the outputText attribute.

### Version 2 – Interface and Velocities – Flow Charts

##### Checkbox update method

##### Pseudocode for hover detection

if self.rect.containspoint(mouseCoords):

self.hovered = True

else:

self.hovered = False

This is how I intend to program the hover detection for all my interface objects – if the object’s rect attribute contains the coordinates of the mouse, that means the mouse must be hovering over the object.

A close up of text on a black background

Description automatically generated

##### Pseudocode for changing mouse pointer image

if self.hovered:

mouse.hide()

self.screen.blit(self.parent.pgkPointerCursor, mouseCoords)

##### Button update method

**A close up of a map

Description automatically generated**

In Pygame, there is no method to directly change the image of the pointer, so I have come up with a workaround. If the mouse is hovering over the button, it will be hidden, and the pointer image will be placed on its location, giving the impression that the mouse pointer has changed.

##### InputBox update method

A close up of a map

Description automatically generated

##### Pseudocode for keypress handling

if event.type == KEYPRESS:

if len(self.outputText) <= self.charLimit:

if self.allowLetters and event.key in parent.pgkLetterKeys:

self.outputText.append(keyName(event.key))

elif self.allowNumbers and event.key in parent.pgkNumberKeys:

self.outputText.append(keyName(event.key))

elif self.allowDecimal and keyName(event.key) == “.”:

self.outputText.append(“.”)

elif self.allowNegative and keyName(event.key) == “-“:

self.outputText.append(“-“)

This will be the main input validation for my input box objects. Firstly, the event has to be a keypress in order to be picked up and processed by the program. Next, the program looks at the length of the text that has been typed into the input box, and will only continue if it has not yet hit the character limit. If there is still room available for more characters, the key that has been pressed will be checked against the validation attributes that were set during instantiation, and if it is a valid key that the user has decided to allow to be typed into the input, it will be appended onto the outputText attribute. I will need to use keyName as Pygame’s event objects’ key attribute is only the keycode (such as K\_a, K\_b, and so on). Using keyName will convert that into a string that will be the letter the key represents (K\_a will become “a”).

##### Setup screen

A close up of text on a black background

Description automatically generated

##### Pseudocode for particle placement

if event.type == MOUSEBUTTONUP:  
 if event.button == 1:   
 if pRef.xPos + pRef.radius >= SCREEN\_WIDTH \  
 or pRef.xPos - pRef.radius <= 0 \  
 or pRef.yPos + pRef.radius >= SCREEN\_HEIGHT \  
 or pRef.yPos - pRef.radius <= 0 \  
 or len(pRef.hasCollided(particles)) != 0:  
 pass  
 else:  
 particles.add(Particle(50, (0, 0), (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])),  
 0, xA=0))  
 pRef = particles.sprites()[-1]

To improve the UI, I will add a ‘dummy’ particle underneath the mouse at all times, showing the user where the particle would end up if they were to place it at any given point in time. This particle will be referred to as ‘pRef’ – particle reference. When a mouse click is detected, the program will compare the edges of pRef (centre coordinates ± radius) to the edges of the screen – using similar code to the particle update method detecting wall collisions. If pRef is colliding with a wall, the ceiling, the floor, or another particle, nothing will happen. However, if pRef is not colliding with anything, it will be placed. I intend to do this by adding a new particle (with some default attributes, it will receive user-customised ones from the updateParticle function) to the particles sprite group, and making pRef refer to this new particle. The old dummy particle, with the user’s desired attributes, will be left at the position it was when the user clicked.

##### updateParticle function

A close up of a sign

Description automatically generated

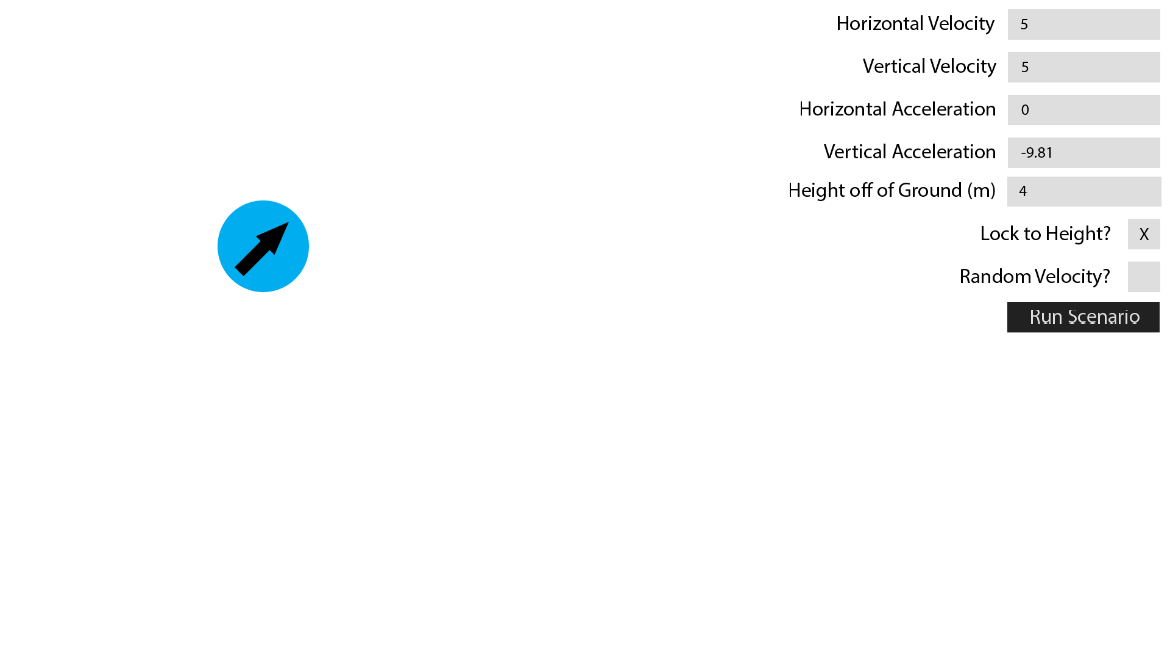
##### Updated main loop function

A close up of a logo

Description automatically generated

The pause function will simply be a loop with nothing in, that gets broken when the user presses esc again.

### Version 2 – Interface and Velocities – UI Design



Inputs for particle properties. User will be able to click on any of these and enter their own values, and the particle that they are editing will be updated as they are typing.

This is my plan for what the GUI in version 2 of my project will look like.

Circular representation of a spherical particle that will follow the mouse cursor until the left mouse button is pressed, at which point the particle will be placed.

### Version 3 – Better User Experience – Overview

In Version 3, I plan to add many features that will create a much more user-friendly experience. As my program is designed for use in classrooms, it needs to be even for those who don’t usually use a computer. This means it will need instructions. Another useful feature would be the ability to save and load different scenarios. This would be useful as it would allow teachers to create scenarios before their class starts in order to save time, and they would then be able to share that saved scenario with other teachers who want to teach the same lesson.

To implement these in the way that I want, I will need to create a new interface class – a dropdown menu – that will be used for the user to select a saved scenario. I also want to make the UI animated, so I will add a container object that will contain interface widgets, and handle animation.

#### New Features to Be Added:

* Animated UI
* Dropdown menus
* Instructions
* Save/load scenarios
* Ability to change volume/mass of particles
* Ability to change material of particles
* Ability to create custom materials
* Time controls during simulation

### Version 3 – Better User Experience – Classes

Below is the full class diagram for Version 3 (Split into two)

Diagram

Description automatically generated

A close up of text on a white background

Description automatically generated

#### Line Class

A picture containing table

Description automatically generatedThe line class will be a class in the main program that will be used by particle objects to draw a line following their motion.

##### Line Class – Attributes

**screen** will be the surface that the program is running on, and what the image of the button will blit onto.

**colour** will be a tuple containing the RGB colour code of the line – the particle will pass its own colour during instantiation.

**plotCoords** will be a list of the coordinates of all plots on the line, in the form (x, y)

##### Line Class – Methods

**\_\_init\_\_** will assign the line’s attributes from the values passed, as well as adding the line to the graph’s lines attribute (graph will be passed on instantiation)

**draw** will make use of pygame’s draw.lines method, which takes a list of lines, as well as the surface to draw them on (screen), the colour of the lines (colour), and list of plots (plotCoords) as arguments and draws lines connecting them.

#### Graph Class

Text, letter

Description automatically generatedThis class will provide a background for the lines to be drawn onto, as well as handling the drawing of the lines. There will be background lines drawn on at 1m intervals, allowing the user to see where specific particles were at any moment in time.

##### Graph Class – Attributes

**screen** will serve the same purpose as in the Line class

**width** and **height** will be integers that store the dimensions of the graph. In this case they will simply be the width and height of the window.

**bgColour** will serve the same purpose as the Line class’s colour attribute – except it will be the colour of the background, rather than the lines – which will be black.

**rect** will be the Pygame Rect object that will be drawn onto the screen every frame. It will be a rectangle with the dimensions specified by width and height, and will have the colour of bgColour

**xLabels and yLabels** will be lists containing the locations (on the x and y axes, respectively) at which lines will be drawn to mark 1-metre gaps. The gap between each label will be equal to the scale (1m).

**lines** will be a list containing Line objects. The draw method for each of these will be called every time the graph’s draw method is called.

##### Graph Class – Methods

**\_\_init\_\_** has the same purpose as in the Line class

**draw** will draw the rect, as well as black lines for every x and y coordinate in the xLabels and yLabels attributes. It will also call the draw methods for every line inside the graph’s lines list. This saves me from having to call all of the draw methods inside of the main code

**changeLabelGap** will adjust the values inside of the x and yLabels lists such that the gaps between labels is equal to the new label gap, which will be passed to this method as an argument. It will also make sure that the labels don’t go further than the dimensions of the graph (no x labels greater than the width, or y labels greater than the height). This method will be useful as it will allow the user to zoom in and out of the program while maintaining the 1m gaps in the graph.

**clearLines** will delete all of the lines in the lines list, and then set the lines attribute back to a blank list, clearing the graph. This function will be called whenever the user returns back to the main menu, as it will reset the screen, effectively giving a blank slate.

#### Changes to the Particle Class

The only change to the particle class is the addition of the line attribute, which will store the particle’s line. The particle will add its own coordinates to the line object as a plot on every update method.

#### Container Class

A picture containing table

Description automatically generated

This class will be used like a ‘screen within a screen’, in that widgets will be placed onto it, with their positions relative to it, and then it will be placed on the main screen. This allows me to animate my UI by simply moving the container every frame, rather than updating the position of every widget. It also allows me to use ‘masks’ (rectangles drawn over the container) as I will be able to specify the order in which things are drawn.

##### Container Class – Attributes

**parent, screen, bgColour, height, width, rect, and previousFrame** will all serve the same function as they do in the button class.

**widgets** will be a list that will always start out as empty. When a new widget is created that I want to be added to the container, the container’s addWidget method will be called from the widget’s \_\_init\_\_ method. This method will simply append the widget to the widgets attribute.

**animation** will be a list containing data about the animation that is currently being executed. Currently I plan to have it store the type of animation, the time that I want the animation to be executing for, and the coordinates at which the animation will finish (for animations that involve moving the entire container)

**outlineColour** and **outlineThickness** will store the colour of the outline as an rgb tuple, and the thickness as an integer.

**bg** will be a Boolean that will be checked in the update method. If it is true, then the container’s rect and outlineRect will be drawn. If it is false, they will not and only the widgets will be drawn, so that the user can see what is on-screen underneath the container.

**outlineRect** will be needed as Pygame does not support adding a border to a filled rect object. Itwill be a rect similar to the container’s regular rect attribute, albeit with a slightly larger size. The new size will be calculated by adding 2 \* outlineThickness to the width and height of the original rect. In order to create the appearance of a border, this rect will be drawn first, with the regular rect drawn on top, so that just the edges of this rect are showing. This will give the appearance of an outline.

**maskColour** will be a tuple pssed on instantiation that contains the rgb code for the colour of the masks. In my case, it will be the same as the colour of the background, so that containers appear invisible until their animation starts playing and they are revealed.

**maskLeftRect, maskRightRect, maskTopRect, and maskBottomRect** will be the rect objects that correspond to each mask. The left mask will cover the left half of the container, the right mask will cover the right half, the top mask will cover the top half, and the bottom mask will cover the bottom half.

**deleteAfter** will be a Boolean that tells the container whether it should delete itself (and all widgets it contains) after it has finished its animation.

##### Container Class – Methods

**\_\_init\_\_ and config** will perform the same function as in all other widget classes

**addWidget** will be called from a widget’s \_\_init\_\_ method, and will take the widget as an argument. It will then append that widget to the container’s widgets attribute.

**delete** will call the delete method of all widgets within the container, before finally removing itself from its parent’s pgkGroup attribute, at which point there will be no references to it and it will be collected by Python’s garbage collection.

**getCorrectedCoords** will be called by the container’s child widgets in their draw methods, and will take their coordinates (relative to the container) as an argument, and will add together the x and y values of the widget and the container to get the absolute coordinates (from the upper left corner of the window) before returning these inside a tuple. This will allow the widgets to ‘follow’ the container as it moves.

**getRect** will simply return the container’s rect attribute

**animationDone** will return true if the container’s animation attribute (list) is empty, as that shows there is no animation currently occurring. It will return false if the animation attribute is not empty, as that shows there is an animation occurring.

**isEmpty** will return true if there are no widgets inside the container’s widgets attribute, or false if there are

**isMasked** will return true if the container is completely covered by its masks (meaning the contents are completely hidden). It will return false if this is not the case

**mouseMasked** will return true if the location of the mouse pointer is over any one of the container’s masks, or false if not. This method will be used in the update method of the other widgets so that they cannot be clicked on while hidden.

**removeWidget** will take a widget as an argument, and then remove it from the container’s widgets attribute.

**startAnimation** will take the following as arguments: animation type (string), whether the container is appearing or disappearing (string), animation time in seconds (float), and optionally (for animations that involve the whole container moving) the x/y coordinate at which the animation will finish (integer). If the animation type is either horizontalSlide or verticalSlide, then the width of the left and right masks will be set to 0, and the height of the top and bottom masks will be set to 0, making all masks invisible.

**centreAnimation** will be called in the update method, and will adjust the widths of the left and right masks, and the heights of the top and bottom ones, by ((rectWidthOrHeight/2) \* animationTime \* timeSinceLastFrame). This adjustment means that the masks will decrease in size every frame, thus slowly revealing the contents of the container from the centre outwards, until the width/height of the masks are zero (after the specified amount of time) at which point they will be invisible and the container underneath will be fully visible. This process will also be performed in reverse if a container is disappearing.

**horizontalSlideAnimation** will be called in the update method, and will adjust the x-coordinate of the container by (distance \* animationTime \* timeSinceLastFrame), where distance is the difference between the x coordinate that the container is travelling towards and its original x-coordinate. I don’t need to worry about the widgets inside of the container, as they will automatically follow by calling the getCorrectedCoords method from their draw method.

**verticalSlideAnimation** will perform the same task as horizontalSlideAnimation, except it will deal with y coordinates rather than x coordinates.

**update** will first check the animation type that is stored inside of the animations attribute, and call the corresponding method, in order to get the correct state of the container for each frame.

check if the bg attribute is true. If it is then it will draw the outlineRect, followed by rect on top of it. After this, regardless of the state of the bg attribute, it will call the draw method of the widgets attribute (this will be a sprite group, meaning it will then call the draw methods of everything within it), followed by drawing the mask rects.

#### Changes to the Button, Checkbox, and Inputbox Classes

Due to the addition of the container object, I will need to add a couple of attributes and methods to all other interface classes.

##### Attributes

**coords -** As the planned animations will involve the container and everything inside of it moving in sync, I will use the cords attribute to store each widget’s coordinates relative to the upper-left corner of the container. These will then be updated in the particle’s draw method to be absolute coordinates,

**container**  - This attribute will store a pointer to the container that contains this widget (if applicable). This widget will then use its container’s getCorrectedCoords method to change its coordinates from relative to absolute in the draw method.

###### Methods

**draw -** Due to the use of masks in the container for animations, widgets inside of a container using masks will need to be drawn before the masks but after the container’s background, so that the masks are drawn on top. I will add a draw method and move the code for drawing the widget out of the update method and into this new method. This means that the class’s update method will be able to draw the widget or not draw it, depending on whether the widget is inside of a container or not. If the widget is inside of a container, the container will be able to call the widget’s draw method after it has drawn its own background, but before it has drawn its masks.

#### Dropdown Class

A picture containing table

Description automatically generatedThis class will be used to allow the user to select from various options, for example when they are selecting a scenario to load, or a material for their particles. It will always display the currently selected option, then when pressed, it will display 4 more underneath, and the user will be able to scroll to show other options.

##### Dropdown Class – Attributes

**parent, screen, font, bgColour, inlineText, container, textColour, hoverColour, inlineDisplayText, inlineDisplayRect, coords, and hovered** will all perform the same function as they will in other widget classes.

**options** will be a list that stores the available options in their current order, with the currently selected item at the start of the list.

**originalOptions** will be a list that stores the available options in their original order. It will be created as a copy of options in the \_\_init\_\_ method and will not be changed during the running of the program. It will be used when a new item is selected, so that items are placed back into the correct order when deselected (options will be set to a copy of this list, and then the item that has been selected will be moved to the front of the list).

**currentOption** will be the currently selected option (the first string in the options list)

**width and height** will be the dimensions of the dropdown before it is expanded, and also the dimensions of each option display

**rects** will be a list of rect objects. Rather than just one rect object, like the rest of the widget classes, the dropdown needs to work like multiple buttons, and therefore needs multiple rects, one for each option. There will be 5 rects in the list, the first of which will be positioned at the coordinates that were passed during instantiation. This one will be reserved for the currently selected option, and will always be drawn. The other 4 will be positioned at regular intervals below the first. They will display the other visible options, and will only be drawn if the dropdown menu is expanded.

**expanded** will be a Boolean that shows the state of the dropdown menu. If the user clicks on the dropdown, then it will be set to true, and therefore all five rects will be drawn in the update method. If they then click again, it will be set to false, and only the first rect will be drawn in the update method.

**hoverRect** will be set when the user hovers over any one of the rects from the rects list. If the dropdown is not expanded, then it will only check the first one, but if it is expanded, then the program will get the coordinates of the mouse and then check through every rect to see if those coordinates are contained within it. If they are, then hoverRect will be set to this rect. hoverRect will then be drawn in a different colour to the others (hoverColour rather than bgColour).

**lower** and **upper** will be integers that represent the indices of the options list between which the options that will be displayed underneath the currently selected option are stored. For example, options[lower] will be displayed on rects[1], options[lower + 1] will be displayed on rects[2], and so on, until options[upper] is displayed on rects[4]. lower will start off as 1 (as position 0 is always the currently displayed option), and upper will start off as 4. When the user scrolls the mouse wheel down, while the menu is expanded, lower and upper will both decrease by 1 (provided lower is greater than 1). When the user scrolls up, lower and upper will both increase by 1 (provided there are available indices after upper)

##### Dropdown Class – Methods

**\_\_\_init\_\_, config, delete,** and **draw** will all serve the same purpose as in the other widget classes.

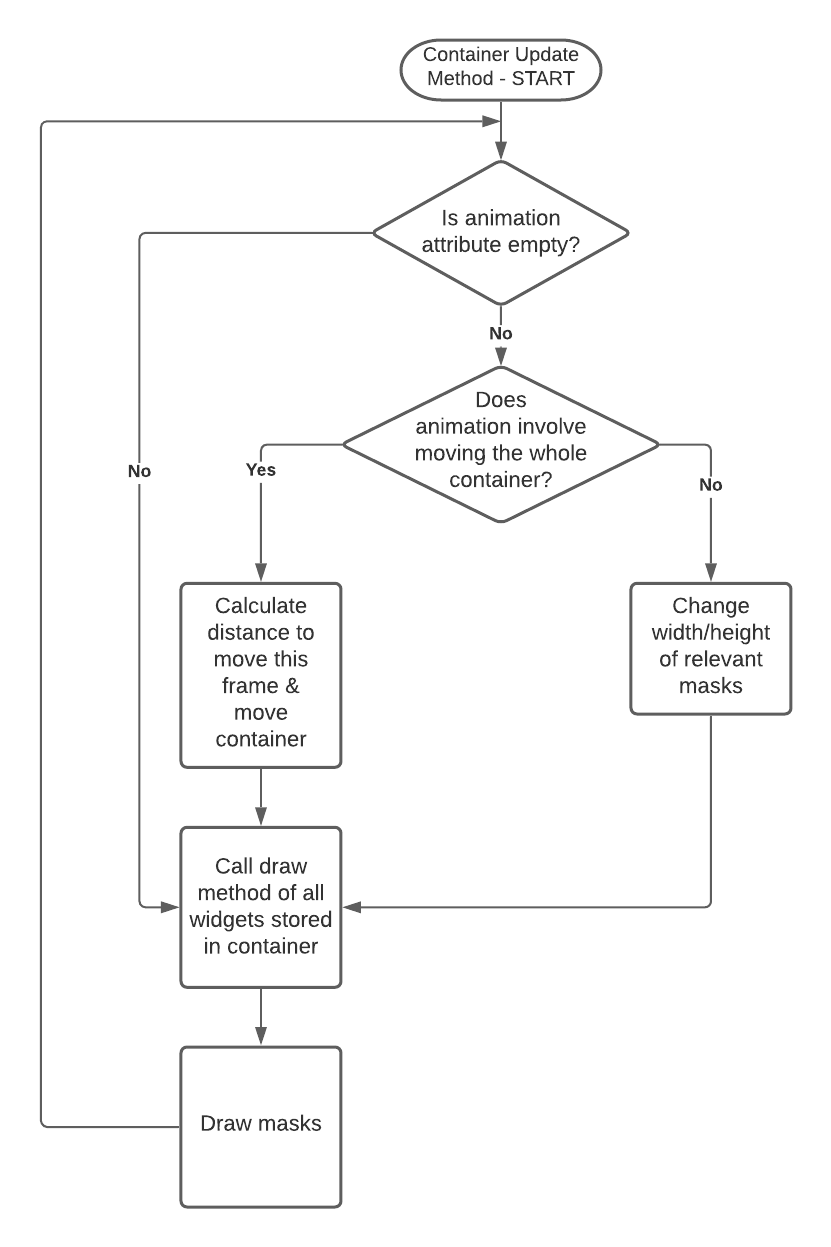
**get** will return the currentOption attribute of the dropdown.

**setSelected** will be used from the main section of the program in case I need to change the currentOption attribute in a dropdown menu without needing the user to click on it.

**update** will handle mouse click and scroll events, as well as updating the state of the hovered and expanded attributes. It will first check if the mouse pointer is over the dropdown, and that the mouse is not being obstructed by the container’s masks (if applicable). If this is true, then hovered will be set to true. Else hovered will be false. It will then designate hoverRect to whichever rect the mouse is hovering over, and finally call the draw method if the dropdown is not in a container, otherwise the draw method will be called by the container.

### Version 3 – Better User Experience – Flow Charts

##### Container update method



**Pseudocode:**

moveThisFrame = (totalDistanceToMove / framerate) / animationTime

This pseudocode calculates the distance to move every frame as a fraction of the total distance that the container needs to move over the course of the animation (in pixels).

##### New setup screen flow chart

Diagram

Description automatically generated

In version 3, I am adding the ability to edit particles after they have been placed. If the user makes a mistake when placing a particle, this means that they can easily change the values of the particle to the correct ones.

##### New updateParticle method

Diagram

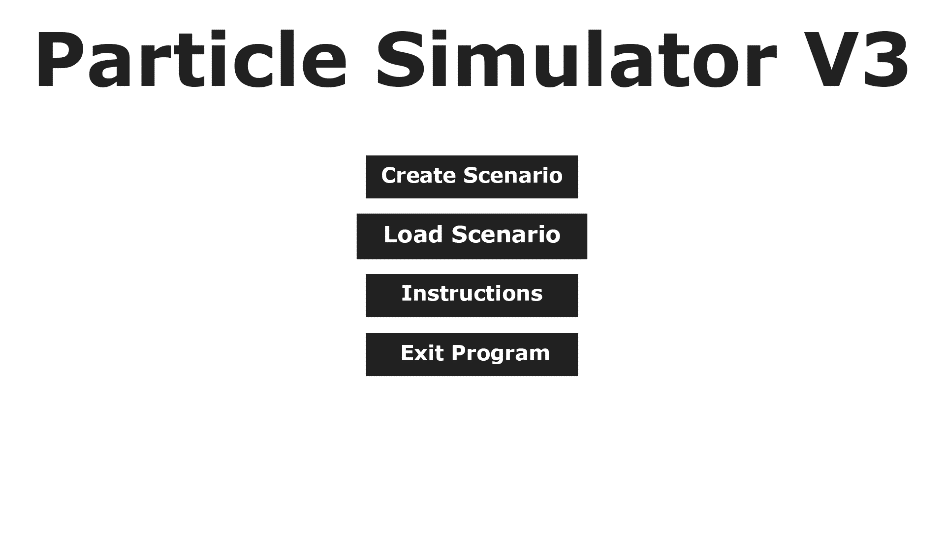
Description automatically generated

The updateParticle function will be updated to take into account the ability to edit particles. This will be achieved by setting a ‘currentParticle’ variable, and all of the properties will be applied to the currentParticle.

### Version 3 – Better User Experience – UI Design

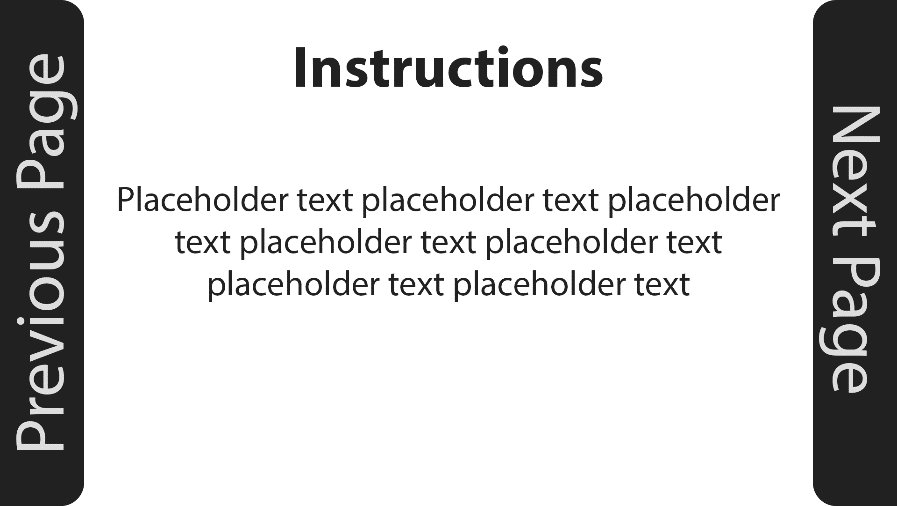
#### Main Menu Design

This is my initial design for the main menu of my program. It does not need to be overly complex, so I have gone with a simple design that fits the purpose of a main menu



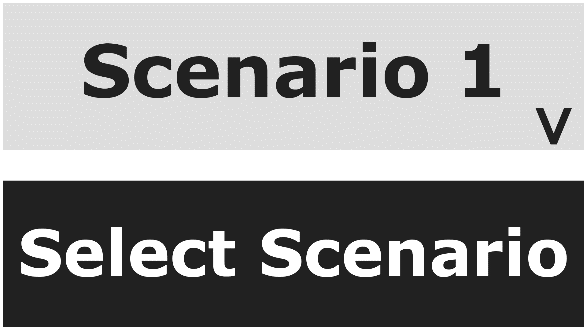
The menu will be responsive - when the mouse hovers over a button, that button will slowly swell (over about 0.25 seconds) to indicate that it is the currently selected option.

#### Instructions Page



Large side buttons make it obvious how to navigate between pages. Also, when one is clicked, the pages will make use of the animation method of the Container class to slide off the page. For example, when ‘Next Page’ is clicked, the current page will slide off to the left, and the next page will slide in from the right, as if the two pages were attached

#### Loading a Scenario

This is a closeup of the dropdown menu that will appear if the user selects ‘Load Scenario’ on the main menu

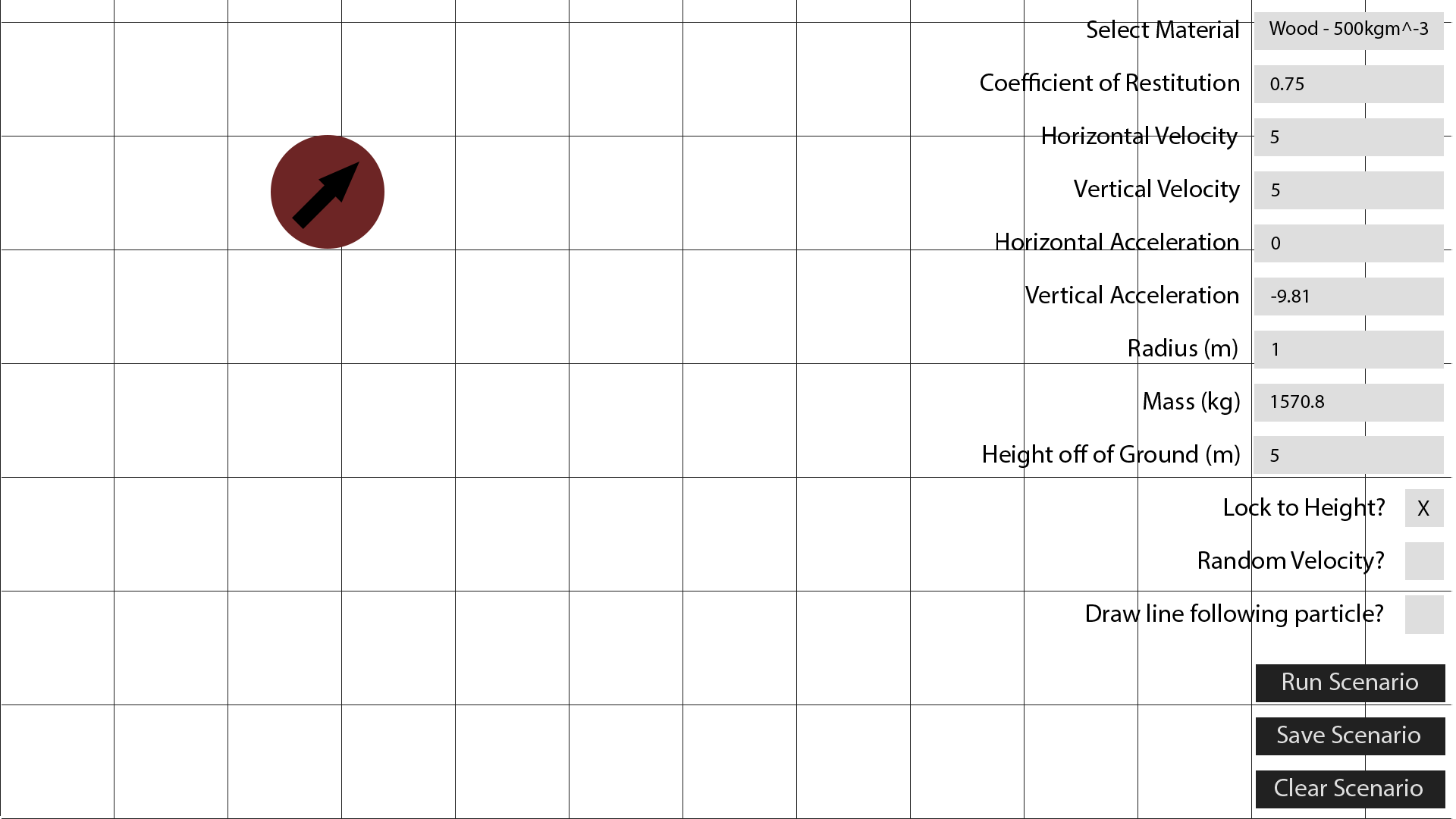
When the dropdown is closed, the arrow is pointing downwards, to indicate that it will open when it is clicked



Scenario 2 is highlighted, to show that the mouse is hovering over it, and it will be selected if the left mouse button is clicked.

When the dropdown is open, the arrow is pointing upwards, to indicate that it will close when it is clicked

#### Scenario Creation (changes from V2)



X

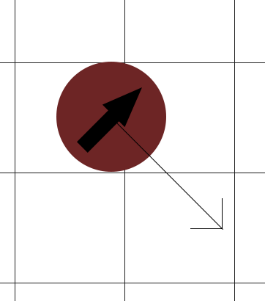
If the user so wishes, they will be able to tick this checkbox that will update the line attribute of the current particle to True, and then when the simulation is running, a line will be drawn following the particle’s motion, meaning you will be able to see how the motion of the particle has changed during the simulation.

>

User will now be able to use a dropdown menu to select a material for each particle that they place – Radius and mass will be automatically calculated and updated in the input box depending on the density of the material

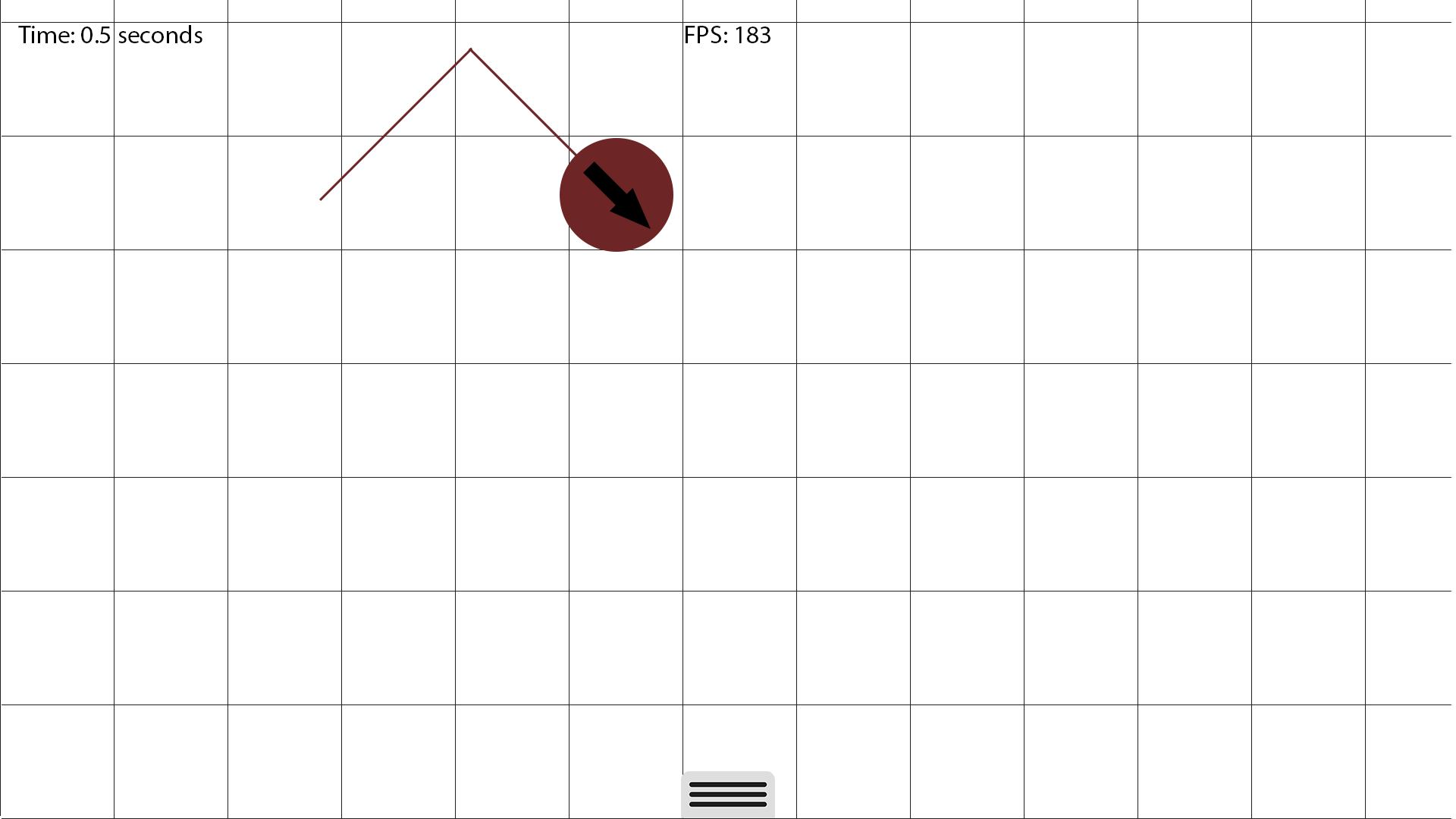
A grid will be drawn in the background, to show 1m gaps both horizontally and vertically - this will make it much easier to see how far a particle has travelled in a certain amount of time.

#### Changing the size of a particle



Arrow that originates from the centre of the particle, following the mouse cursor. When the cursor is moved further away from the particle, its size will increase (as if the user is pulling on the arrow). When the cursor moves closer to the particle, its size will decrease (as if the user is pushing on the arrow)

#### During the Simulation

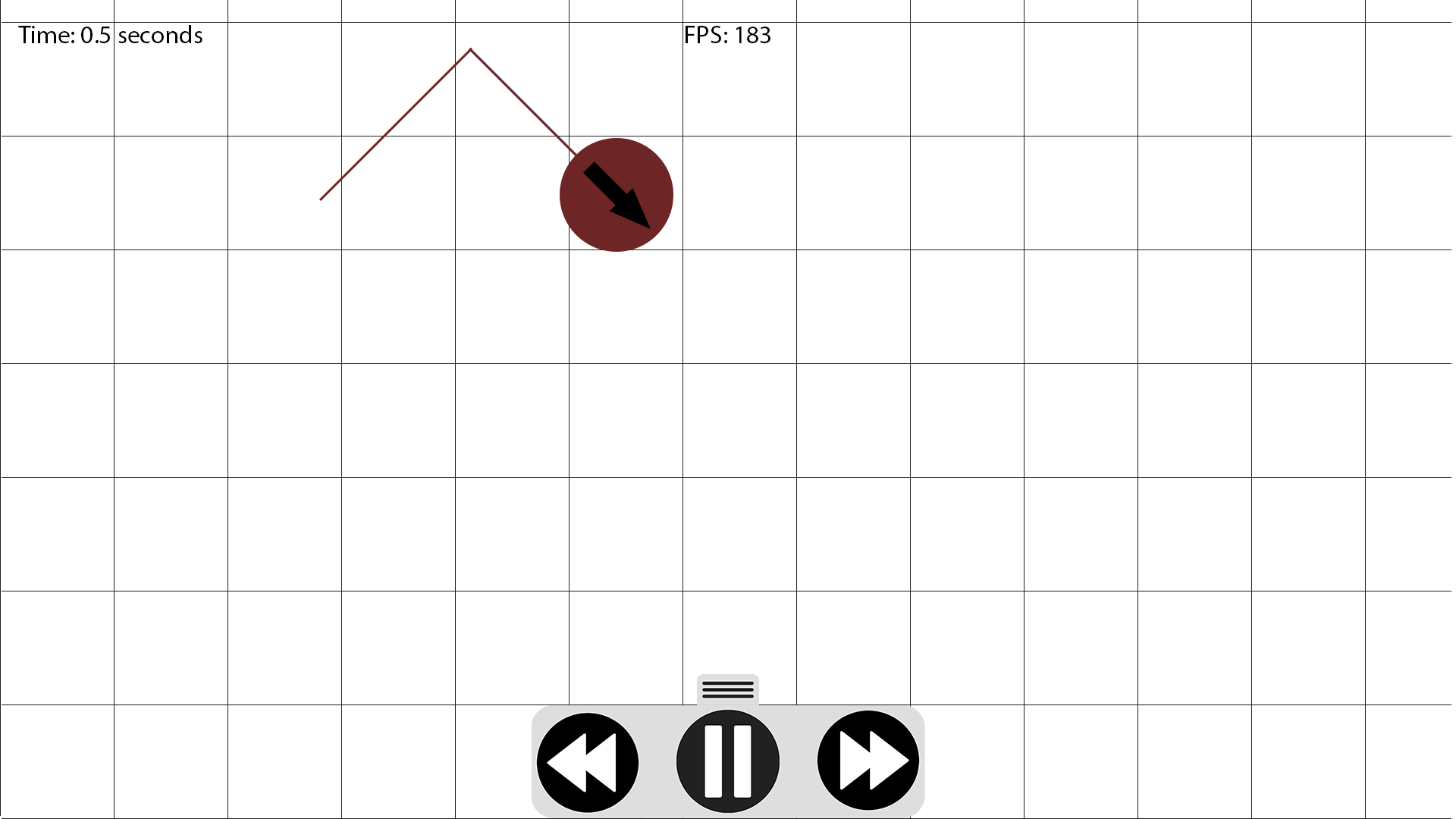


Small tab on the bottom of the screen can be clicked to show or hide time controls – so that they won’t be obscuring the bottom of the screen constantly.

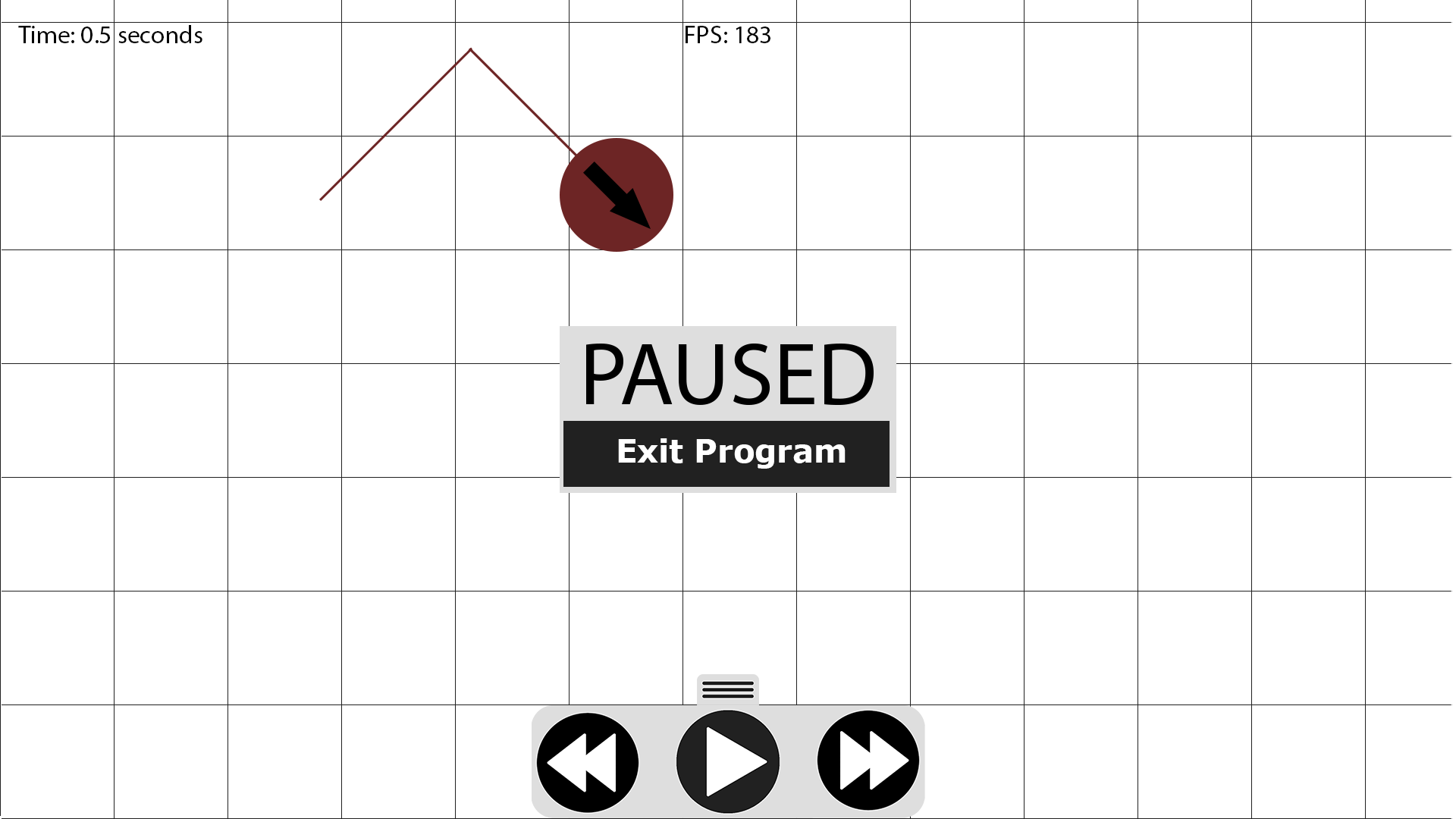
FPS counter to inform the user of how well the program is running. Higher FPS – more accurate physics

Line drawn behind particle to show path taken since start of simulation

The time indicator will be in the top-left of the screen, and will display the time (in seconds) since the start of the simulation. It will increase or decrease depending on whether the simulation is being fast-forwarded or reversed.



Once the tab has been clicked, the time controls will utilise the animation method of the container class to slide up into view over a period of 0.5 seconds. Once they have appeared, they will be circular buttons that can be used to speed up the simulation, slow it down, or pause it.



When paused, the pause button will change to a resume button.

When paused (either by pressing esc or clicking the pause button), the pause menu will appear on screen. It will use the animation method of the container class to appear from the centre out, over a period of 0.25 seconds.

### Key Functions and variables

#### Version 1

##### Functions and procedures

**movingCloser(particle1, particle2):** This function will compare the current position vectors of two particles with what their position vectors will be in the next frame, and return True if the distance between them lessens (they are moving closer) or false if it gets larger. This will help the particle .hasCollided() method to differentiate between a particle that it is colliding with (one that it is touching and moving towards) and one that it has just collided with (is still touching, but they are moving apart)

**absoluteDistance(vector1, vector2):** This function will take two vectors as arguments, and subtracts one from the other two get the difference between them. It will then use the .length() method of the vector class to return the length of the difference (the magnitude of the distance between the two vectors)

**drawDirectionArrow():** This will import an image of an arrow from the project folder, and set a scale using the following equation: It will then use pygame’s rotozoom method to rotate and scale the image using the particle direction attribute and the above equation, and place it on the particle’s centre.

arrowScale = (radius / arrow.get\_width()) \* 1.5

**main():** This function will contain a condition-controlled loop that will serve as the event loop for pygame. This event loop will start by checking for events that have occurred since last time the loop ran (button presses, mouse clicks, clicking the ‘x’ on the window), and will perform the tasks that I assign those events to. Next it will wipe the screen clean, before running the update method on the particles sprite group and updating the display. It will then loop back to the start and repeat. The speed of the loop is controlled by Pygame’s clock.tick() method, which only allows the loop to run the desired amount of times per second.

##### Key variables

**SCREEN\_WIDTH and SCREEN\_HEIGHT:** These are a pair of integers that are used by the program to set the size of the window in pixels.

**FRAME\_RATE:** This integer will store the desired amount of times (per second) that the main loop will run. It is also used to correctly scale the velocity and acceleration, so that the correct amount is added every time the update method is called.

**BG\_COLOUR:** This is a tuple containing the rgb code for the background of the simulator.

**COEFFICIENT\_RESTITUTION:** This is a float ranging from 0 up to 1 which represents the coefficient of restitution. This value is the ratio of initial to final velocity of an object when it collides with another object. By default, it will be set at 0.75, meaning that when a particle collides with the edge of the screen, it will only retain 75% of its velocity in the relevant direction.

**SCALE:** This integer stores the amount of pixels on-screen that represent one metre. This will be very useful when changing a particles coordinates, as velocity will be stored in a vector, in ms^-1, meaning it will need to be multiplied by the scale before it is divided by frame rate and added to the position vector.

**clock**: This will be a Pygame clock object, and its .tick() method will be used to control the timing of the simulation, and ensure that the main loop only runs at the frequency set by the FRAME\_RATE constant.

**particles:** This will be a Pygame sprite group containing all particle objects that are instantiated.

#### Version 2 (changed or new only)

##### Functions and procedures

**setup():** This function will contain a condition-controlled loop, with its speed controlled by Pygame’s clock, that allows the user to set up their physics scenario. It will not run the particles’ update methods, and so they won’t move anywhere, and the user will have as long as they like to customise the scenario to their liking.

**updateParticle(particle):** This function will be used by the setup() function to update the attributes of the currently selected particle, in order to ensure that any user interactions with the interface objects will be reflected in the attributes of the selected particle.

**pauseMenu()**: This function will contain a condition-controlled loop, again with its speed controlled by Pygame’s clock. The loop will only contain an event check for esc keypresses, and if one is detected, it will return from the function, back to whatever function was running when pauseMenu was called. There will be no graphical or simulation updates, meaning the simulation will be paused, and there will be some text on-screen, letting the player know that the program is paused.

**scaler(value, axis):** This function will take a value, and scale it relative to an axis – useful for consistent UI on any screen size. I will design and develop my program on a 1920x1080, and therefore the scaler function will multiply a given value by screen width/1920 if the value is being scaled on the x axis, or screen height/1080 if being scaled on the y axis.

##### Key variables

**pRef:** This will be used in the setup() function, as a reference to the most recently placed particle in the particles sprite group. This is useful as it will shorten the code significantly, as I will only have to type pRef in place of particles.sprites()[-1].

**inputList:** This will be a list containing all input boxes and checkboxes used in the setup screen. Using a list will be much more efficient than giving each input an individual name, as I will be able to iterate through the list to retrieve inputs, rather than calling the get method on each object by name.

**pgkRoot:** This will be the control object for the interface.

#### Version 3 (changed or new only)

##### Functions and procedures

**mainMenu():** This procedure will be called at the beginning of the program. Similarly to the setup procedure, it will contain a condition-controlled loop with the speed controlled by Pygame’s clock. At first, it will instantiate all main menu widgets that are shown in the UI design section of V3 decomposition.

**saveSetup():** This procedure will iterate through the particles sprite group, and write the properties of each particle to a new line in a file. This method of saving means that, when loading, the program can simply run through the file and create a new particle for each line, with those properties.

**loadSetup():** This procedure will be the counterpart to saveSetup. It will open on of the saves’ .txt files, and then

iterate through line by line, creating a particle for each line, using the attributes on that line to create it.

**sizeChange():** This procedure will run when the user presses the ‘s’ key during the setup procedure. It will draw an arrow at the mouse location, as well as a dotted line from the mouse location to the centre of the particle being resized. The particle will increase in size if the user drags the mouse away from the centre while holding the left mouse button, and it will reduce in size if the user drags the mouse towards the centre while holding the left mouse button.

**createMaterial():** This procedure will contain another condition-controlled loop, using Pygame’s clock for the speed control. It will first instantiate all necessary widgets (such as input boxes for the user to enter their desired properties of the material, and buttons to save the material.) Then, once the user has pressed the save button, the procedure will write the data of the material to a .txt file, so that the material can be accessed whenever the program is run.

##### Key Variable

There will be no key variables added in version 3 compared to version 2.

## Test Plan

### Version 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Test | Success Criteria Being Tested | Instructions | Expected Result |
| 1.1 | Are particles placed where you click? | Criteria 2 – Placeable Particles | Click anywhere on-screen using the left mouse button. | A particle should appear, centred on the mouse pointer. |
| 1.2 | Do particles bounce off of surfaces? | Criteria 6 – Momentum | Observe the particle that appears immediately. When it reaches the bottom of the window, does it bounce back upwards? | The particle should bounce back directly upwards, with a velocity equal to the velocity immediately before the collision, multiplied by the coefficient of restitution, and reversed. |
| 1.3 | Do particles collide sensibly? | Criteria 6 - Momentum | Place two particles in a way such that they will collide (one above another, or one above and slightly to the side of another). | When the two particles fall into one another, they should collide and bounce off of each other. The collision should look natural and not forced. |
| 1.4 | Does the directional arrow point in the correct direction? | Criteria 9 – UI | Place a particle. Does its arrow point downwards when it initially falls, and upwards when it rebounds off of the ground? If you place another, and they collide, are the arrows angled correctly when they move in separate directions? | The arrows should point in the direction of travel for each particles. They should not be larger than the particle, and each particle should have its own arrow. |
| 1.5 | Is the effect of gravity accurate? | Criteria 7 – Gravity | Run the program. The particle that appears immediately begins with zero velocity, eight metres (800px) above the bottom of the screen. When the particle hits the floor and bounces back upwards, the program will print the time since it was instantiated. | Using t = (sqrt(2as+u^2)-u)/a, we can calculate that the time should be 1.2771 seconds. As the program runs at 60FPS, the program will (if gravity is working correctly) print the next multiple of 1/60, which is 1.28333, to 5 d.p. |
| 1.6 | Does the program run well on less powerful computers? | Criteria 8 – Universal support | Run the program on an old pc or laptop. | The program should run, and still pass all tests that are defined in this section. |

### Version 2

In addition to the tests defined below, I will also be performing regression testing, meaning I will check version 2 against version 1’s tests, to ensure that I didn’t break anything during the development of version 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Test | Criteria Being Tested | Instructions | Expected Result |
| 2.1 | Can you type in the input boxes? | Criteria 9 - UI | In the setup screen, click on any one of the input boxes to activate it. Type on your keyboard | Hyphens(-),Decimal points(.), and any number keys that you press should appear in the input box. |
| 2.2 | Can you exceed the character limit in input boxes | Criteria 9 - UI | In the setup screen, click on any one of the input boxes to activate it. Type -9.8 | I will set the character limit at 4, and therefore you should only be able to type up to four characters. Any keypresses after -9.8 (4 characters) should be ignored. |
| 2.3 | Do the input boxes impact the particles? | Criteria 3 and 9 – Changeable properties of particles & UI | In the two velocity boxes, enter various numbers.  Next, change the downwards acceleration to -5 and place a particle(particle A) 8m off of the ground. Change downwards acceleration to -10, and place a particle next to particle A (particle B). Press ‘start simulation’. | The direction arrows on the particle you are placing should change in real time.  Using t = (sqrt(2as+u^2)-u)/a, we can calculate that particle A should hit the ground after 1.789 seconds, and particle B should hit the ground after 1.265 seconds |
| 2.4 | Do the checkboxes impact the particles? | Criteria 3 and 9 – Changeable properties of particles & UI | In the setup screen, tick the ‘Random Velocity’ checkbox and place a few particles.  Next, enter a value in the ‘Height off of ground’ input box, and tick the ‘Lock to Height’ checkbox. | When random velocity is enabled, every particle that you place should have its velocity randomised. This can be checked before starting the simulation by looking at the direction arrows.  When lock to height is enabled, the particles should only follow your mouse horizontally, and should always stay the same height off of the bottom of the screen. |
| 2.5 | Can you pause? | Criteria 5 – Simulation Settings | When the simulation is running, press ‘esc’ | The simulation should pause, and some text saying ‘Paused’ should appear on-screen. |

### Version 3

In addition to the tests defined below, I will also be performing regression testing, meaning I will check version 3 against the tests for both version 2 and version 1, to ensure that I didn’t break anything during the development of version 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Test* | *Criteria Being Tested* | *Instructions* | *Expected Result* |
| *3.1* | Is the menu animated? | Criteria 9 – UI | Load the program. | The main menu should slide in from the right. The buttons should swell when you hover over them. |
| *3.2* | Can you access the instructions? | Criteria 4 – Instructions | In the main menu, press the button labelled ‘Instructions’ | The program should transition to a page titled ‘setup instructions’. This page should have instructions on it and a button to move to the next page of instructions. |
| *3.3* | Can you load a scenario? | Criteria 5 – Simulation Settings | In the main menu, click the button labelled ‘Load Scenario’. Then, select ‘Custom Scenario 1’ in the dropdown menu, and click load. | The program should transition into the setup screen, but with six particles already on screen, with varying properties. |
| *3.4* | Can you change the size of particles? | Criteria 3 – Changeable Properties of Particles | When placing a particle, press the ‘s’ key | An arrow should appear on the mouse cursor, and when you hold the left mouse button and drag the cursor closer to the centre of the particle, the particle should increase in size. When you hold the left mouse button and drag away from the particle, it should decrease in size. |
| *3.5* | Can you edit particles once they are placed? | Criteria 3 – Changeable Properties of Particles | In the setup screen, right click on a particle that has already been placed. | A small popup menu should appear by the particle. It will contain all of the settings you can change normally; except they will only effect the particle you right clicked on |
| *3.6* | Can you control the speed of the simulation? | Criteria 5 – Simulation Settings | During the simulation, open the time controls by pressing the tab at the bottom of the screen. Press each of the buttons. | When you press the decrease button (two arrows pointing left) the speed of the sim should decrease. When you press the increase button (two arrows pointing right) the speed of the sim should increase. When you press the pause button, the sim should pause, and a pause menu should appear on screen. |

Placeholder to keep development section formatted correctly. Will update when writing design for V2

# Development

## Version One

All new programming techniques that I talk about in this section were found either on the pygame documentation (<https://www.pygame.org/docs/>) or the python 3.8.3 documentation (<https://docs.python.org/3/>)

These are all the modules that are needed for V1 of my code. I chose to give the ones with longer names abbreviated references in order to make the code much more readable. Similarly, I imported \* (all constants) from pygame.locals so that I can simply type (for example) KEYDOWN rather than pygame.locals.KEYDOWN when checking for events.

import pygame as pg  
import pygame.math as pgmath  
from pygame.locals import \*  
import math

My program uses constants for values that need to be used in multiple locations within it. This makes it much easier for me to change how my program runs (i.e. for testing compatibility I can easily change the frame rate) as I only have to change the value in one location, rather than in every location that it is used.  
SCREEN\_WIDTH = 1920  
SCREEN\_HEIGHT = 1000  
FRAME\_RATE = 60 # How many times main loop will run per second  
BG\_COLOUR = (255, 255, 255)  
COEFFICIENT\_RESTITUTION = 0.75 # How much speed is maintained after collision  
SCALE = 100 # How many pixels = 1m  
ARROW\_IMAGE = pg.image.load('arrow.png') # Used for direction arrow

self.velocity = pgmath.Vector2(v[0], v[1]) rect.x, self.rect.y)  
 if not xA: *# Assigns 0 as default x acceleration, if no value is passed* self.acceleration = pgmath.Vector2(0, yA)  
 else:  
 self.acceleration = pgmath.Vector2(xA, yA)

When I initially started working on V1, I planned to use tuples for attributes that have two values, such as position and velocity. However, upon doing further research, I discovered the pgmath.Vector2 class, that would be perfect for my needs.

Using tuples, I would need two lines rather than one in the update method, for example:

self.realX += (self.velocity[0] \* SCALE) / FRAME\_RATE  
self.realY += (self.velocity[1] \* SCALE) / FRAME\_RATE

Rather than simply adding the two vectors together, i.e.

self.pos += self.velocity \* SCALE / FRAME\_RATE

Also, if I used tuples, I would have to manually compute the dot product of two vectors for collisions, rather than simply using the built-in vector2.dot() method.

def collide(self, p2):  
 *# Particles have collided - calculate new velocities* m1, m2 = self.mass, p2.mass  
 v1, v2 = self.velocity, p2.velocity  
 pos1, pos2 = self.pos, p2.pos  
  
 selfNewVelocity = v1 - ((2 \* m2) / (m1 + m2)) \* (  
 ((v1 - v2).dot(pos1 - pos2)) / (  
 (pos1 - pos2).length()) \*\* 2) \* (pos1 - pos2)  
 p2NewVelocity = v2 - ((2 \* m1) / (m2 + m1)) \* (  
 ((v2 - v1).dot(pos2 - pos1)) / (  
 (pos2 - pos1).length()) \*\* 2) \* (pos2 - pos1)  
  
 self.velocity, p2.velocity = selfNewVelocity, p2NewVelocity

Initially, my collision function (below) was an attempt to modify equations from my A-Level Physics textbook (<https://www.hoddereducation.co.uk/subjects/science/products/16-18/aqa-a-level-physics-year-1-student-book>) – the equations that I mentioned in the research section of my analysis. This, however, proved to be unsuccessful as the equations in the textbook are used for collisions between one moving particle and one non-moving one, whereas I needed collisions between two moving particles.

def collide(p1, p2):  
 angleBetween = p1.angleTo(p2)  
  
 direction1 = p1.direction  
 direction2 = p2.direction  
  
 collisionDirection = (direction1 + direction2) / 2  
 collisionMomentum = p1.angleMomentum  
  
 p1.direction = collisionDirection + math.pi - angleBetween  
 p2.direction = collisionDirection - angleBetween  
  
 p1.angleMomentum = collisionMomentum \* math.sin(angleBetween)  
 p1.angleSpeed = p1.angleMomentum / p1.mass  
 p1.velocity[0] = p1.angleSpeed \* math.cos(p1.direction)  
 p1.velocity[1] = p1.angleSpeed \* math.sin(p1.direction)  
  
 p2.angleMomentum = collisionMomentum \* math.cos(angleBetween)  
 p2.angleSpeed = p2.angleMomentum / p2.mass  
 p2.velocity[0] = p2.angleSpeed \* math.cos(p2.direction)  
 p2.velocity[1] = p2.angleSpeed \* math.sin(p2.direction)

After doing research, I settled on using the following equations:

V represents the velocity vectors of the particles, m represents the masses, and x represents the position vectors of their centres at the time of collision. There are various equations that can be used to achieve this purpose, but I chose this pair as they do not require you to calculate angles. I made this choice as calculating angles of travel and angles between two particles will require another function, using up valuable computing power.

def drawDirectionArrow(self):  
 arrow = ARROW\_IMAGE *# Prevents having to load image every time - very slow* arrowScale = (self.radius / arrow.get\_width()) \* 1.5  
 *# Rotate and scale arrow image to fit particle* arrow = pg.transform.rotozoom(arrow, math.degrees(self.direction),  
 arrowScale)  
 arrowRect = arrow.get\_rect(center=(self.rect.x, self.rect.y))  
 screen.blit(arrow, arrowRect)

def drawDirectionArrow(self):   
 arrow = pg.image.load(**'arrow.png'**)  
 arrowScale = (self.radius / arrow.get\_width()) \* 1.5  
 *# Rotate and scale arrow image to fit particle* arrow = pg.transform.rotozoom(arrow, math.degrees(self.direction),  
 arrowScale)  
 arrowRect = arrow.get\_rect(center=(self.rect.x, self.rect.y))  
 screen.blit(arrow, arrowRect)

Above is the initial code for the drawDirectionArrow method. While this worked with no issue on my home PC, when I tested the program with it stored in my school account, it caused the program to slow down massively. After some testing, I discovered that the cause of this issue was the fact that I loaded the arrow image every time this method was called (every frame). This explains the slowdown as my home files are stored on my HDD, whereas my school files are stored on a network location, meaning the program had to retrieve the file from the cloud every frame, taking substantially longer than retrieving it from a disk. I fixed this issue by instead loading the arrow image once at the start of the program, and then setting the arrow variable equal to the image, preventing me from having to load it every time.

def updateDirection(self):  
 *# Gets direction of travel (in radians)  
 # Multiply by -1 as pygame measures angles anti-clockwise* self.direction = math.atan2(self.velocity.y, self.velocity.x) \* -1

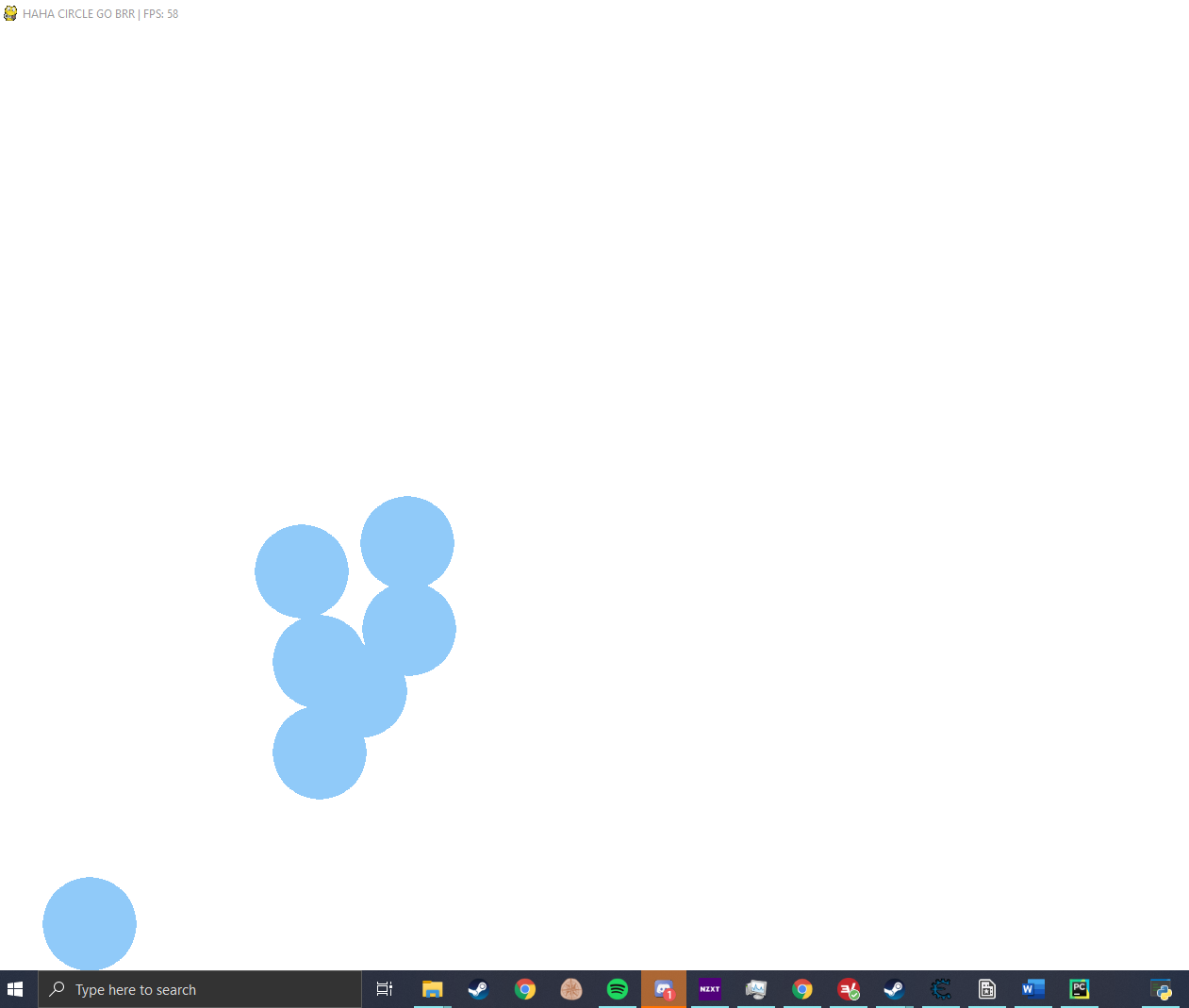
Initially, I wrote the updateDirection method using the math.atan method. However, due to the fact that this requires division of the two variables, it ended up being very prone to division by zero errors, for example if a particle was on the floor and thus had no y-velocity, which I managed to circumvent using try/except statements. Eventually, however, I came across the updated math.atan2 method, which simply takes the two values as parameters and automatically handles any division by zero, and thus condensing my method from 17 lines to 1.

def updateDirection(self):  
 *#Gets direction of travel (in radians)* try:  
 self.direction = math.atan((self.velocity[1] \* -1) / self.velocity[0])  
 if self.velocity[1] \* -1 < 0:  
 if self.velocity[0] < 0:  
 self.direction = (math.pi \* -1) + self.direction  
 else:  
 pass  
 elif self.velocity[1] \* -1 > 0:  
 if self.velocity[0] < 0:  
 self.direction = math.pi - abs(self.direction)  
 else:  
 pass  
 except ZeroDivisionError: *# If particle has no x velocity* if self.velocity[1] \* -1 < 0:  
 self.direction = (math.pi / 2) \* -1  
 elif self.velocity[1] \* -1 > 0:  
 self.direction = (math.pi / 2)

def absoluteDistance(pVector1, pVector2):  
 distance = pVector1 - pVector2  
 return distance.length()

def movingCloser(p1, p2):  
 *# Don't times by scale this time in order to get more precision with check  
 # If velocity was \*scale, there is potential that, at high velocities, the  
 # particle could 'jump' over the next, ending up further apart.* nextPos1 = p1.pos + p1.velocity / FRAME\_RATE  
 nextPos2 = p2.pos + p2.velocity / FRAME\_RATE  
  
 if absoluteDistance(nextPos1, nextPos2) < absoluteDistance(p1.pos, p2.pos):  
 return True  
  
 else:  
 return False

In order for a pair of particles to be considered ‘colliding’ they need to fulfill two conditions: They must be touching, and they must have velocities that are moving toward each other. I added this second condition due to an issue that arose in testing, in which two particles that have just collided and had the appropriate velocity change applied would occaisionally still be touching in the next frame, and so would collide again. This resulted in particles being stuck together, as shown in the screenshot below. To remedy this, I simply added a check that uses the movingCloser funtion in order to ensure that two particles are actually colliding, rather than moving apart after a collision.



if \_\_name\_\_ == **"\_\_main\_\_"**: *# If program is run as a script, this will run* pg.init()  
 pg.font.init()  
 screen = pg.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))  
 pg.display.set\_caption(**'JSanders Physics Simulation V1'**)  
 clock = pg.time.Clock()  
 particles = pg.sprite.Group()  
 main()

The \_\_name\_\_ check ensures that the code nested within it will only run if the program is directly run, as opposed to being imported by another program. It isn’t strictly necessary in this particular program, but it is generally considered good practice and has become habit for me.

### Validation

Currently, the only input to my program is a click to create a particle. However, I do not need to write any code specifically for validating this input, as there aren’t many ways that user error could cause the program to break. The only way that could cause the program to act unusually is if the user places a particle slightly off-screen, but this is handled by the section of the particle’s update method, on the next page. This works as validation as the particle will always start with a velocity of 0, therefore multiplying velocity by coefficient of restitution will still result in a velocity of 0, and so the only thing being changed is the position of the particle.

def update(self):  
 *# Checks if particle has collided with floor  
 # Velocity check in order to make sure that the particle has collided,  
 # rather than being one that has collided and was turned around in the previous frame.* if self.pos.y + self.radius >= SCREEN\_HEIGHT and self.velocity.y > 0:  
 self.velocity.y = round(  
 self.velocity.y \* COEFFICIENT\_RESTITUTION \* -1, 4)  
 self.pos.y = SCREEN\_HEIGHT - self.radius  
 print(self.time)  
  
 *# Has collided with ceiling* elif self.pos.y - self.radius <= 0 and self.velocity.y < 0:  
 self.velocity.y = round(  
 self.velocity.y \* COEFFICIENT\_RESTITUTION \* -1, 4)  
 self.pos.y = 0 + self.radius  
  
 *# Has collided with right wall* if self.pos.x + self.radius >= SCREEN\_WIDTH and self.velocity.x > 0:  
 self.velocity.x = round(  
 self.velocity.x \* COEFFICIENT\_RESTITUTION \* -1, 4)  
 self.pos.x = SCREEN\_WIDTH - self.radius  
  
 *# Has collided with left wall* elif self.pos.x - self.radius <= 0 and self.velocity.x < 0:  
 self.velocity.x = round(  
 self.velocity.x \* COEFFICIENT\_RESTITUTION \* -1, 4)  
 self.pos.x = 0 + self.radius

Checks to detect whether any part of the particle has exceeded the bounds of the screen, and if so, turns it around and slightly reduces its velocity in the relevant direction (multiplies by the coefficient of restitution – the ratio of speed that is maintained to original speed)

### Testing

* 1. **–** Particles appear on the screen with their centres at the location of my mouse pointer, for the most part. The only time when this is not true is when the mouse pointer is close enough to the edge of the screen that it would result in a particle being placed off-screen. In this case the particle will be as close to the pointer as possible without going off-screen. **This test was passed.**
  2. – To perform this test accurately, I added some print statements that show the velocity of a particle before and after it collides with a surface. Below are the results.

5.1426 \* -0.75 = -3.8695



5.7639 \* -0.75 = -4.3229

As the amount by which the program’s calculation is incorrect is inconsistent, I believe that the slight error is due to a floating-point error that occurs during the multiplication and rounding step. Due to the fact that the tiny margins of error are negligible in a GCSE/A-Level setting, I consider this **test passed**.

* 1. **A picture containing clock

     Description automatically generatedA close up of a sign

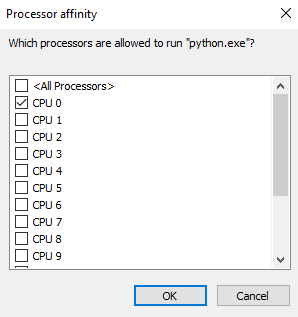
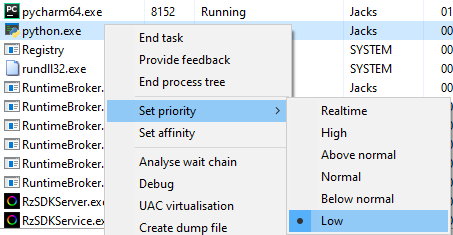
     Description automatically generated–** Before collision (Particle moving upwards has a much lower velocity)

After collision. As you can see, the particles have collided, and their directions and velocities have changed in a way that would occur in reality.

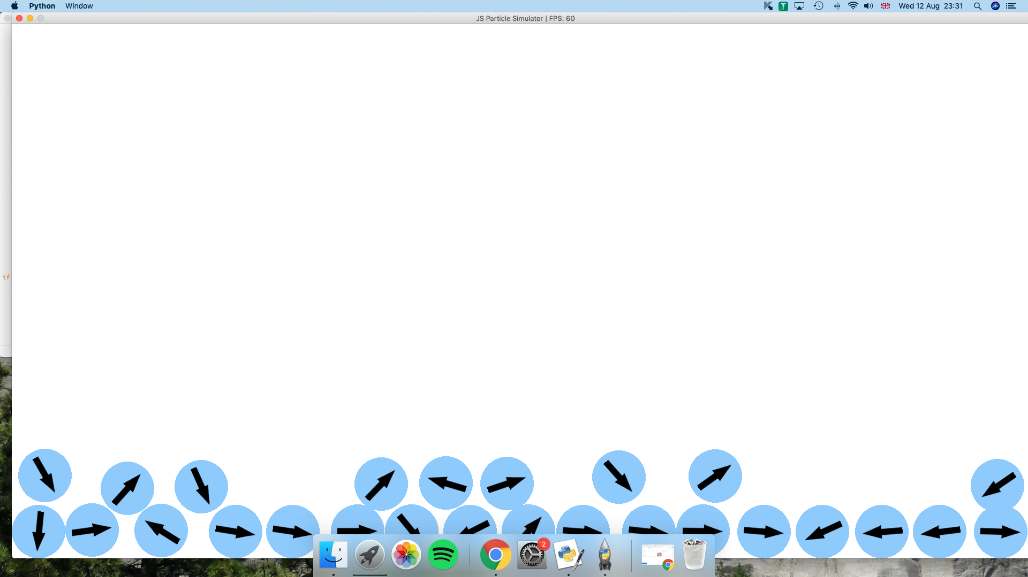
**This test has been passed.**

* 1. **–** Referring to the test data for test 1.3, you can see in the images before and after the collision that the direction arrows are pointing in the correct direction. Therefore **this test has been passed**.
  2. **–** Below is a screenshot of the console output when my code runs. The particle that is placed exactly 8m off the ground by the program falls, and as you can see from the output (printing the times that the particles hit the floor), it hits the ground at 1.28333 seconds. **The test was passed**.A picture containing bird, knife

     Description automatically generated
  3. – In order to test compatibility with low-powered hardware, I initially limited python to only one CPU core, and low priority. However, I then discovered that python only uses one core anyway, and the low priority has no noticeable impact on performance.



My second plan was to borrow a 2013 iMac, which has the added benefit of testing my program across multiple OSes. The program worked perfectly, even with upwards of 25 particles on-screen at once.



In order to test my program at a low frame rate, I implemented an artificial limiter that limited the FPS to 8. This introduced an issue as the physics engine that I created for the program was built to run at 60fps – which meant that any drop from that value results in the program seemingly running in slow-motion, as opposed to the physics still running in real time, just with fewer images displayed on-screen per second. **This test was partially passed**

A screenshot of a cell phone

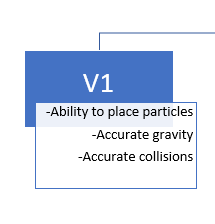
Description automatically generated

8 FPS causes the program to run at roughly 1/6 of the correct speed.

### Development of Version 1: Review

The program successfully passed all but one of the tests without issue. Test 1.6, however, highlighted an issue that didn’t come up throughout the development, as my computer is powerful enough to run the program at a constant 60FPS. The issue is that the program was written with 60fps in mind (the code is written to multiply velocity by 1/60 to get the correct position change per frame, for example) meaning that on any frame rate other than 60, the program will still act as though it were running at 60fps, and so the physics simulation will slow down, making the simulation look like it is in slow motion. To fix this issue in the next version of my program, I need to find a workaround that will account for varying frame rates, and maintain consistent physics simulation on any computer.

Currently, referring to the analysis section, the program only fully meets requirement 1. It somewhat meets criteria 8, in that runs on all OSes that I have tried it on, although it is held back from meeting this requirement by the issue that was talked about earlier.



These are the features that I stated would be included in version one in my decomposition, and all of them have been completed – evidenced by the results of tests 1.1, 1.2, 1.3, and 1.5. Something that I added that I didn’t intend to is the direction arrows that are shown on all of the particles when the program is running. This feature was added as I felt that it made the program much more interesting to watch, while not requiring much extra code.

## Version Two

I will be splitting the development of version two into two separate parts: the development of the interface module, and updating version one’s code to meet version two’s requirements.

### Development of Version Two: Interface Module (Pgkinter)

I am developing this module to mirror the features offered by Tkinter as closely as possible, with the bonus of being seamlessly integrated with Pygame. This is necessary to meet success criteria 9, as that requires the whole program to be contained within one window.

class Pgk(object):

def \_\_init\_\_(self):...

def hoverEffect(self, rgb):…

def isLight(self, rgb): *# Determines whether an rgb code is light or dark  
 """Treats the rgb code as a 3D position vector and calculates the length of  
 the line from the origin to the position vector. The longer the line,  
 the lighter the colour.  
  
 """* if sqrt(rgb[0] \*\* 2 + rgb[1] \*\* 2 + rgb[2] \*\* 2) >= 220:  
 return True  
  
 else:  
 return False

Uses Pythagoras’ Theorem to calculate the magnitude of the rgb code, treating it as a 3D vector. The longer the vector, the lighter the colour [as pure white is (255, 255, 255)]. I made the choice to consider any rgb codes that have a magnitude greater than or equal to 220 as light, and any shorter than this as dark. This result will be used in the hoverEffect method, that calculates an alternative colour for buttons to show when hovered over.

def update(self): *# Only need pgkinter.update() in main code* self.pgkGroup.update()

Update method: calls update on the interface sprite group, which in turn calls the update method of every sprite in the group.

def eventHandler(self, event):  
 *# Takes an event from the main pygame loop, and passes it to every  
 # pgkinter sprite.* handledList = [obj.handleEvent(event) for obj in  
 self.pgkGroup.sprites()]  
 for handled in handledList:  
 if handled:  
 return True  
 return False

Initially, I had the following code in each of my interface classes:

for event in pg.event.get():

*# Do something*

However, this failed due to the way that Pygame events work. In Pygame, each event is its own separate object, and once it is ‘used’ (for example when pg.event.get() is called) it is deleted. This lead to keypresses not registering correctly in input boxes, as the keypress events would be retrieved and deleted by the main event loop. My solution to this issue is to create a handleEvents method for each class in the interface, and a main eventHandler method in the control class. These methods will take an event object as a parameter, and then perform the code that would have been inside the event loop in their update methods. Below is my implementation of that solution:

for event in pg.event.get():

for obj in self.pgkGroup.sprites():

obj.handleEvent(event)

While this did fix the issue of events not registering correctly, my solution created another issue – when I clicked on any of my interface objects, a particle would be placed below it. This is because each event is now effectively being handled twice: once by the interface, and once by the main program. In order to stop this, I made the interface objects return True if the event was used by them, and then the control object will create a list from these values, and if any value in the list is True, then an interface object has handled the event. Finally, I added a check in the main event loop that will only place a particle if the event is a mouse click and the interface objects have not handled any events. eventHandler code is above, and the code for the main event loop is below:

for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
  
 if not pgkRoot.eventHandler(event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 *# Check that looks at all possibilities to ensure  
 # that the particle will be placed on-screen and not  
 # intersecting with another particle* if pRef.pos.x + pRef.radius >= SCREEN\_WIDTH \  
 or pRef.pos.x - pRef.radius <= 0 \  
 or pRef.pos.y + pRef.radius >= SCREEN\_HEIGHT \  
 or pRef.pos.y - pRef.radius <= 0 \  
 or len(pRef.hasCollided(particles)) != 0:  
 pass  
 else:  
 particles.add(Particle(50, (0, 0), (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])),  
 0, xA=0))  
 pRef = particles.sprites()[-1]

class Button(pg.sprite.Sprite):  
 def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,   
 text=None, height=None, width=None, action=None):

super().\_\_init\_\_()

Runs the constructor for the base class (Pygame’s sprite class)

The =None in the parameters of \_\_init\_\_ indicate that that parameter is not necessary for the method to work. If no parameter is passed, then it will be set to None. This allows me to add checks in the \_\_init\_\_ methods of the interface classes that will set these values to default values if nothing is passed

[...]

if not font:  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 30)  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])

[...]  
  
 self.\_\_bgColourHover = self.\_\_parent.hoverEffect(self.\_\_bgColour)

Using the control object’s hoverEffect method (self.\_\_parent is the control object)

All attributes of the Button, InputBox, and checkbox classes will be private (denoted by two underscores before the attribute name), as their attributes do not need to be accessed anywhere else in the program, and making them private ensures they will not be mistakenly modified by other parts of the program.

*# Changes text colour depending on whether the background colour is  
 # light or not.* if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (0, 0, 0))  
 else:  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (255, 255, 255))

Detects whether the button colour is light or dark, and sets the text colour accordingly, making the text high-contrast and easy to read no matter what colour the background is.

[...]

Adds button to pgkGroup sprite group – means that its update method will be run whenever pgkGroup.update() is called.

self.\_\_parent.pgkGroup.add(self)

def config(self, font=None, bgColour=None, textColour=None, text=None,  
 height=None, width=None, action=None):  
 *"""Config method allows the programmer to change any settings that may  
 have a default value after instantiation.  
  
 """* [...]

All interface objects will have a config method, allowing me to change their appearance/behaviour while the program is running. Any of these parameters that are not passed will be set to None and will not affect the attributes, only the ones that I pass will.

def defaultAction(self):  
 pass  
  
 def delete(self):  
 self.\_\_parent.pgkGroup.remove(self)

There is no method to make an object delete itself in Python (del self only deletes the local variable named ‘self’ – not the actual object), so the best I can do is make the object remove itself from the controller’s sprite group, and then write del objectName in the main program - thereby removing all references to the object, so that it will get collected by Python’s automatic garbage collection.

def update(self):  
 *# 0.5 timer check ensures that button can only be clicked once every  
 # 0.5 seconds* if self.\_\_rect.collidepoint(pg.mouse.get\_pos()) and not \

self.\_\_hovered and self.\_\_timer > 0.5:  
  
 pg.mouse.set\_visible(False) *# Hides default mouse pointer* self.\_\_hovered = True  
  
 elif not self.\_\_rect.collidepoint(pg.mouse.get\_pos()) and \  
 self.\_\_hovered:  
  
 pg.mouse.set\_visible(True) *# Shows default mouse pointer again* self.\_\_hovered = False

collidepoint is a method of Pygame’s Rect class that returns True if the rect contains the coordinates passed (in this case, the coordinates of the mouse)

This if statement will set the button’s hovered attribute to True if: the mouse pointer’s coordinates are inside of the button, the attribute is not already True, and it has been more than half a second since the button was last pressed.

Hovered will be set to false when the mouse pointer’s coordinates are no longer inside of the button, and hovered is currently True.

[...]

def handleEvent(self, event):  
 *""" Due to the nature of pygame's events, you cannot have a for loop  
 in the class update() method that checks for events, as each event can  
 only be handled once, as it is effectively destroyed on handling.  
 This means that you need a separate event handling function, which you  
 can then pass events to from the main pygame loop.  
  
 """* if self.\_\_hovered:  
 *# Only checks for events if the pointer is on the button* if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 self.\_\_action()  
 pg.mouse.set\_visible(True)  
 self.\_\_hovered = False  
 self.\_\_timer = 0  
 return True  
  
  
class Checkbox(pg.sprite.Sprite):  
  
 *# noinspection SpellCheckingInspection* def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,  
 textColour=None, inlineText=None, height=None):  
 [...]  
   
 def click(self):  
 if self.\_\_output:  
 self.\_\_output = False  
 self.\_\_outputDisplay = self.\_\_font.render(**""**, True,  
 (0, 0, 0))  
 else:  
 self.\_\_output = True  
  
 if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_outputDisplay = self.\_\_parent.pgkBlackCrossImage  
 else:  
 self.\_\_outputDisplay = self.\_\_parent.pgkWhiteCrossImage  
  
 self.\_\_outputDisplay = pg.transform.scale(  
 self.\_\_outputDisplay, (floor(self.\_\_width),  
 floor(self.\_\_height)))

Will run when the checkbox is clicked. Having this as its own method, rather than being contained in the handleEvent method allows me to run it from the main program – for example, if a checkbox needs to start checked, I can instantiate it, and immediately run the click method.

The handleEvent method will only check for events if the mouse pointer is hovering over the button. To slightly save on processing power.

def update(self):  
 [omitted as it functions very similarly to the button’s update method]  
  
 def handleEvent(self, event):  
 [...]

class InputBox(pg.sprite.Sprite):  
 def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,  
 textColour=None, inlineText=None, width=None,  
 allowLetters=None, allowNumbers=None, allowDecimal=None,  
 allowNegative=None, charLimit=None, defaultEntry=None):

[...]  
  
 **"""Need a separate check for allow[blank] as they are booleans. If I  
 used "if not [blank]", then explicitly setting them as False would  
 trigger that statement, setting the attribute to the default (True)  
  
 """** *# Acts as built-in input validation* if allowLetters is None:  
 self.\_\_allowLetters = True  
 else:  
 self.\_\_allowLetters = allowLetters  
  
 if allowNumbers is None:  
 self.\_\_allowNumbers = True  
 else:  
 self.\_\_allowNumbers = allowNumbers  
  
 *# Need an allowDecimal check in order to input floats* if allowDecimal is None:  
 self.\_\_allowDecimal = True  
 else:  
 self.\_\_allowDecimal = allowDecimal  
  
 if allowNegative is None:  
 self.\_\_allowNegative = True  
 else:  
 self.\_\_allowNegative = allowNegative

Initially, when creating the \_\_init\_\_ method for the InputBox, I treated allowLetters, allowNumbers, allowDecimal, and allowNegative the same as all other unnecessary attributes, in that the code was as follows:

if not allowX:

self.\_\_allowX = True

else:

self.\_\_allowX = allowX

However, I found that when setting allowLetters to False (as is the case in all input boxes used in V2) it would immediately be set back to True, meaning that letters could be typed and the validation was nullified. This happened because the statement if not var (in Python 3.x) calls the variable’s built in \_\_bool\_\_ function and returns the inverse. Naturally, if allowLetters is already a Boolean, then the \_\_bool\_\_ function will return allowLetters, which is False in this case. This in turn causes the allowLetters attribute to be set to True, as the if statement’s requirements are satisfied.

Using ‘if allowX is None’ fixes this issue, as instead of calling the \_\_bool\_\_ function, it only checks whether allowX is undefined (None) – which is what it will be if no value is passed during instantiation, rather than a catch-all for values that Python considers ‘False’ (0, empty lists, empty dicts, etc).

[...]

The get method returns the string contained within the input box. However, before returning, the method needs to check if the ‘|’ character is at the end of the string – as this is the character that I am using to represent the flashing cursor. If it is at the end, then it is removed, and the string without it is returned.

A try/except is also necessary, as if the user hasn’t typed anything into the input box before get() is called, self.\_\_outputText[-1] will cause an IndexError, as the -1 index does not exist.

def get(self):  
 try:  
 if self.\_\_outputText[-1] == **"|"**:  
 return self.\_\_outputText[:-1]  
 else:  
 return self.\_\_outputText  
 except IndexError:  
 return **""** [...]

def update(self):  
 try:  
 *# Timer is used to add and remove the "|" character every 0.5  
 # seconds, giving the appearance that the cursor is flashing when  
 # typing - acts as visual feedback that the inputBox is active.* if self.\_\_active and self.\_\_timer > 0.5 and \  
 self.\_\_outputText[-1] == **"|"**:  
 self.\_\_outputText = self.\_\_outputText[:-1]  
  
 elif self.\_\_active and self.\_\_timer < 0.5 and \  
 self.\_\_outputText[-1] != **"|"**:  
 self.\_\_outputText += **"|"** elif self.\_\_timer > 1:  
 self.\_\_timer = 0  
  
 except IndexError:  
 *# Index error occurs if len(output) == 0* if self.\_\_active and self.\_\_timer > 1:  
 self.\_\_outputText += **"|"** self.\_\_timer = 0

For the flashing cursor in the input box, I could have used an image, but positioning it after the text would be extremely difficult to get right. Instead of that method, I decided to use the ‘|’ character, appending it to and then removing it from the outputText string every 0.5 seconds. This means that the cursor will always be at the end of the string, without the need for calculating the coordinates at which an image would need to be placed.

The ‘|’ character will only be concatenated to the end of the string if the input box is active, the timer is less than 0.5 (seconds), and the last character in the string is not already ‘|’. It will then be removed when the timer exceeds 0.5 seconds. When the timer reaches 1 second, it will be reset to 0

This needs a try/except statement for the same reason as the get method, as it uses indexes.

*# OutputTextDisplay needs to be rendered on every update due to the  
 # fact that output can change* if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_outputTextDisplay = self.\_\_font.render(self.\_\_outputText,  
 True, (0, 0, 0))  
 else:  
 self.\_\_outputTextDisplay = self.\_\_font.render(self.\_\_outputText,  
 True,  
 (255, 255, 255))  
  
 [...]

Timer will only be incremented if the input box is active – slightly more efficient than incrementing every frame no matter what, as the timer is only needed when the input box is active.

Text colour varies depending on whether the background colour of the input box is light or dark, to ensure high-contrast and readability .

if self.\_\_active:  
 self.\_\_timer += time.time() - self.\_\_previousFrame  
 self.\_\_previousFrame = time.time()

def handleEvent(self, event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 if self.\_\_hovered and not self.\_\_active:  
 self.\_\_active = True

A mouse click event will be used by the input box for two different things, depending on the state of the input box. If the input box is not active, and the mouse is hovering over it, it will become active, and therefore will handle keypress events – until the user clicks elsewhere on the screen, and the input box will be deactivated. The second condition returns False, so that the user doesn’t have to click twice if they want to place a particle while an input box is active.

self.\_\_timer = 0  
 return True  
 elif not self.\_\_hovered and self.\_\_active:  
 *# Tries to remove "|" character from end of string,  
 # to show that the input is no longer active.* try:  
 if self.\_\_outputText[-1] == **"|"**:  
 self.\_\_outputText = self.\_\_outputText[:-1]  
 except IndexError:  
 pass  
 self.\_\_active = False

return False  
 *# Will only check keydown events if input box is active* elif self.\_\_active:  
 if event.type == KEYDOWN:  
 if event.key == K\_BACKSPACE:  
 try:  
 *# If "|" character is last in the output string,  
 # remove final two characters and add "|" back on to  
 # the end.* if self.\_\_outputText[-1] == **"|"**:  
 self.\_\_outputText = self.\_\_outputText[:-2] + **"|"** else:  
 self.\_\_outputText = self.\_\_outputText[:-1]

In all keypress event checks, an if/else statement is needed, in case the ‘|’ character is at the end of the output text (if the ‘cursor’ is shown)

except IndexError:  
 *# In case there are no characters in output* pass  
  
 [...]  
 if self.\_\_allowLetters:  
 *# Checks if key is in list of letter keys* if event.key in self.\_\_parent.pgkLetterKeys:  
 self.\_\_caps = event.mod and pg.KMOD\_CAPS  
 *# Checks for caps lock key modifier* try:

event.mod is an attribute of the Pygame keypress event object – either 1 or 0, that denotes whether or not a ‘modifier’ key was pressed during the keypress event. KMOD\_CAPS, is an attribute of Pygame itself, which will be either True or False depending on whether the ‘caps’ modifier is pressed. Setting the caps attribute of the input box to ‘event.mod and pg.KMOD\_CAPS’ ensures that it will only be True if a modifier was pressed during a keypress event, and that that modifier is the caps modifier.

[...]

if self.\_\_outputText[-1] != **"|"**:  
 if self.\_\_caps:  
 self.\_\_outputText += pg.key.name(  
 event.key).upper()  
 else:  
 self.\_\_outputText += pg.key.name(  
 event.key)  
  
 [...]

Can use pygame’s key.name method, passing the event key as a parameter, to easily get a string containing the letter of the key that was pressed.

If caps is enabled, then the .upper method will be called on that string.

### Development of Version Two: Main Program

Not much has changed in the development of the main part of the program, as most of the time was spent creating the interface module. The main additions were the pause menu and the setup screen, which I will be going over now.

This function means that I can design the interface on my screen, at a 1920x1080 screen size. I can then use this function to automatically scale each aspect of the interface in a certain axis, to be sure that it will look the same on any screen size.

*# Scales sizes relative to screen width/height - gives a consistent feel  
# across all devices  
# Needed at the start of the program, as it is used when setting constants*def scaler(toScale, axis):  
 if axis == **'x'**:  
 return int(toScale \* (SCREEN\_WIDTH / 1920))  
 else:  
 return int(toScale \* (SCREEN\_HEIGHT / 1080))

timeMultiplier = 1 / 60 *# Initial value for time between frames*

This is to correct the issue found in test 1.6 when I was testing version 1, where lower framerates caused the program to appear as if it were running in slow motion. Instead of multiplying all physics-related values by 1/60, I will create a timeMultiplier variable that will be set to the amount of time between each frame, and multiply by that instead. This also means that I can replace clock.tick(60) with clock.tick(), removing the restriction on how many times the loop can run per second, meaning that it will run at the fastest possible speed, making for even more accurate physics – as more calculations will be performed per second.

In the particle update method, instead of self.pos += self.velocity \* SCALE / FRAME\_RATE I now have self.pos += self.velocity \* SCALE \* timeMultiplier. timeMultiplier will be equal to 1/frame rate for each frame, therefore SCALE \* timeMultiplier is equal to SCALE / frameRate. The only difference is that timeMultiplier is a variabvle instead of a constant, and will be measured and changed every frame.

SCALE = scaler(100, **"x"**) *# How many pixels represent 1m - if at 1920\*1080*SMALL\_FONT = (**"Helvetica"**, scaler(18, **"y"**))  
LARGE\_FONT = (**"Helvetica"**, scaler(72, **"y"**))  
  
if SCREEN\_WIDTH / 1920 > SCREEN\_HEIGHT / 1080:  
 *# Mid font is the only one that may have issues if scaled like the  
 # others (May end up being bigger than the button that it is inside of)  
 # Therefore it picks the axis that would result in the smallest font  
 # size to prevent this* MID\_FONT = (**"Helvetica"**, scaler(30, **"y"**))  
  
else:  
 MID\_FONT = (**"Helvetica"**, scaler(30, **"x"**))

My program will only use three font sizes, in order to create a more consistent look. The different font sizes will be scaled, so that they will take up roughly the same amount of screen space no matter what screen size the program is running on.

*# Runs when user presses esc*def pauseMenu():  
 pausedFont = pg.font.SysFont(LARGE\_FONT[0], LARGE\_FONT[1])  
 pausedText = pausedFont.render(**"PAUSED"**, True, (0, 0, 0))  
 pRect = pausedText.get\_rect(center=(SCREEN\_WIDTH / 2, SCREEN\_HEIGHT / 2))  
 screen.blit(pausedText, pRect)  
 pg.display.update()  
 paused = True  
 while paused:  
 for event in pg.event.get():  
 if event.type == KEYDOWN:  
 if event.key == K\_ESCAPE:  
 return  
 if event.type == QUIT:  
 paused = False  
 pg.quit()  
 quit()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**‘JSanders Particle Simulation | FPS: '** + fps)  
 clock.tick()

The pause menu function will simply render and blit some text saying ‘Paused’ onto the centre of the screen, to give visual feedback that the program is paused. It will then run an empty loop – so that the particles will not move, but will remain visible. until esc is pressed again, at which point the function will return, and the program will continue.

updateParticle() is defined as a nested function as it is only used by the setup procedure. This means that it can access the local variables of the setup procedure that it needs (widgetList and pRef) without them needing to be passed as parameters.

def setup():  
 def updateParticle():  
 try:

*# Do i.get() excluding last item in widgetList, as that is a*

*# button, which doesn’t have a get method*  
 outList = [float(i.get()) for i in widgetList[:-1]]  
 except ValueError:  
 return  
  
 [...]

Assign values of input boxes to particle.

setting = True  
  
 offset = scaler(150, **"x"**)  
 boxWidth = scaler(125, **"x"**)  
 inputList = [  
 pgk.InputBox(pgkRoot, screen, SCREEN\_WIDTH - offset, scaler(5, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Velocity to the right (ms^-1):"**,  
 width=boxWidth, allowLetters=False, charLimit=4,  
 defaultEntry=**"0"**),  
 [...]  
 pgk.Checkbox(pgkRoot, screen,  
 SCREEN\_WIDTH - scaler(50, **"x"**), scaler(255, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Lock particle to height: "**),  
 [...]  
 ]  
  
 startButton = pgk.Button(pgkRoot, screen,  
 SCREEN\_WIDTH - scaler(350, **"x"**), scaler(355, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Done - Start Simulation"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: main(widgetList))

To begin with, I had a second button that you needed to press every time you wanted to place a particle, resulting in the UI looking like the screenshot below, on the left. However, on second thoughts, I decided that that makes scenarios too convoluted to set up – for example if you were trying to model gas particles, you would need to enter the height for one, press place, place it, and repeat. I have now settled on not needing to press a button before you can place a particle, as well as optional lock-to-height, making it easy to set up any scenario, whether you need a particle a specific distance off of the ground or not. The UI now looks like the screenshot below, on the right.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

[...]  
 pRef = particles.sprites()[-1]  
  
 while setting:  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
  
 if not pgkRoot.eventHandler(event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 *# Check that looks at all possibilities to ensure  
 # that the particle will be placed on-screen and not  
 # intersecting with another particle* if pRef.pos.x + pRef.radius >= SCREEN\_WIDTH \  
 or pRef.pos.x - pRef.radius <= 0 \  
 or pRef.pos.y + pRef.radius >= SCREEN\_HEIGHT \  
 or pRef.pos.y - pRef.radius <= 0 \  
 or len(pRef.hasCollided(particles)) != 0:  
 pass  
 else:  
 particles.add(Particle(50, (0, 0), (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])),  
 0, xA=0))  
 pRef = particles.sprites()[-1]  
  
 screen.fill(BG\_COLOUR)  
 updateParticle()  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.updateDirection()  
 sprite.drawDirectionArrow()  
 pgkRoot.update()  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'JSanders Particle Simulation | FPS: '** + fps)  
 clock.tick()

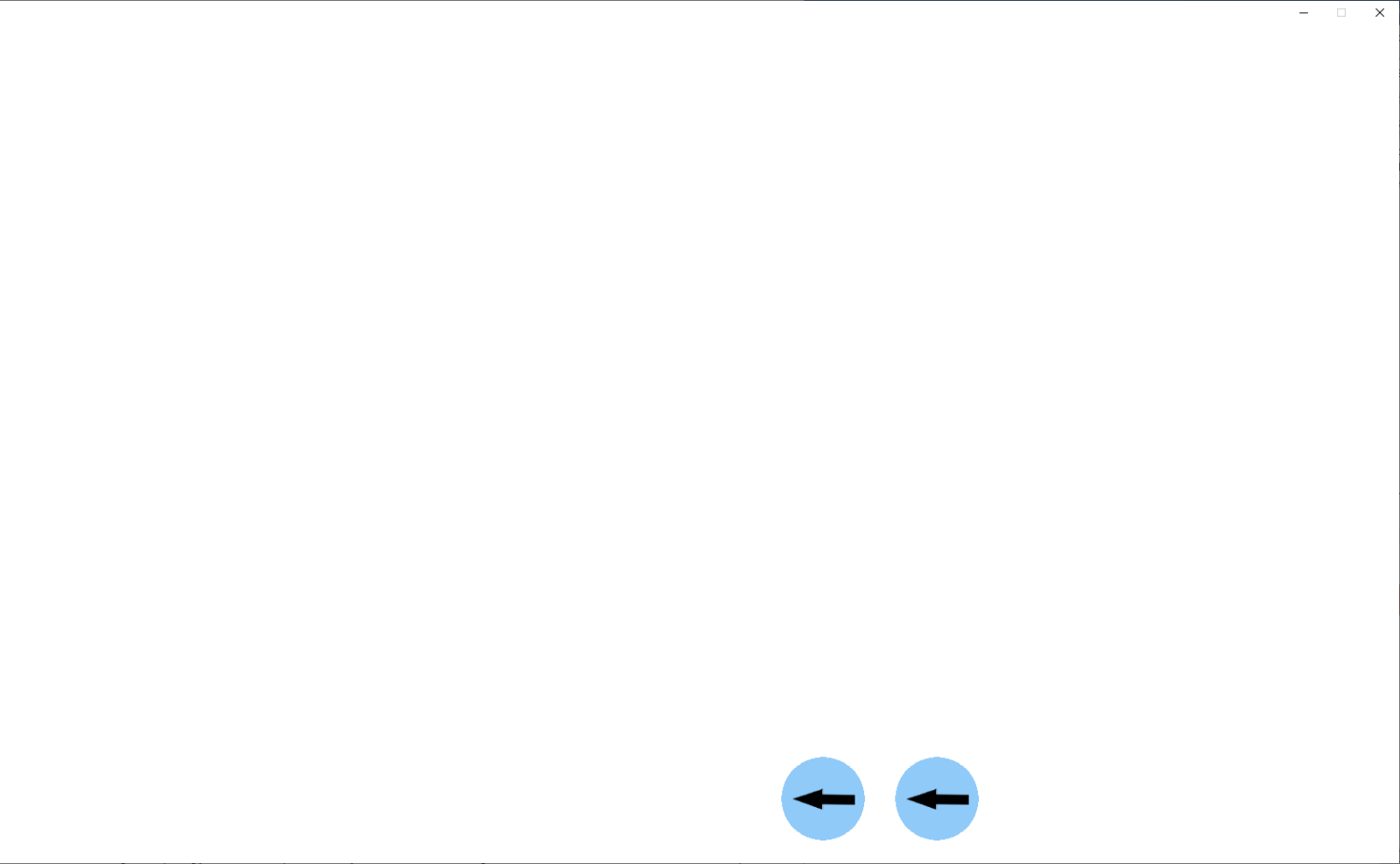
Update the attributes of the particle to match the user’s inputs.

Don’t call the update method on particles, so that they will stay still until the simulation is started.

If the particle is able to be placed (it is not intersecting with the edges of the screen or another particle), another particle will be added. This one then becomes pRef, meaning that the one that was ‘placed’ will remain in the position it was placed until the simulation starts

Before this line was added, the program had an issue – when the ‘start simulation’ button was pressed, a particle would be placed underneath the cursor. I discovered that this was because of the way that I handled placing particles – even unplaced particles are still a part of the particles sprite group. I then added this line to remove the last particle in the group, as that will always be an unplaced one – the one following the mouse cursor.

def main(widgetList):  
 *# Delete particle that gets placed on button press* particles.remove(particles.sprites()[-1])global timeMultiplier  
 running = True

  
 while running:  
 try:  
 *# If statement prevents 'jumping' of particles when user moves  
 # the window* if time.time() - previousFrame <= 0.2:  
 timeMultiplier = time.time() - previousFrame  
 *# Calculates time between frames* except NameError:  
 *# Will occur on the first frame, as there is no previous frame* pass  
  
 previousFrame = time.time()  
  
 [...]

When the Pygame window is moved by the user, the program freezes. This is an intrinsic property of Pygame and therefore I cannot change it. However, this does cause the time between frames to be much higher than it normally would, meaning that particles will jump ahead in time. This jump is inaccurate as it will not detect any collisions during it, as it is only one frame. This means that particles end up with unpredictable directions and speeds. To fix this, I added a tolerance, so as long as the time between frames is less than or equal to 0.2 seconds, the program will continue as usual. If not, then the program will use the previous time between frames, which will be at least enough for 5fps. However, the side-effect of this is that the program will not be quite accurate at frame rates less than 5 frames per second, although if the program was running at 5fps, it would be inaccurate even without this.

Particles after the ‘jump’

Particles before the ‘jump’

### Testing

#### Regression Testing – Testing against V1 criteria

**Tests 1.1 & 1.4**

****

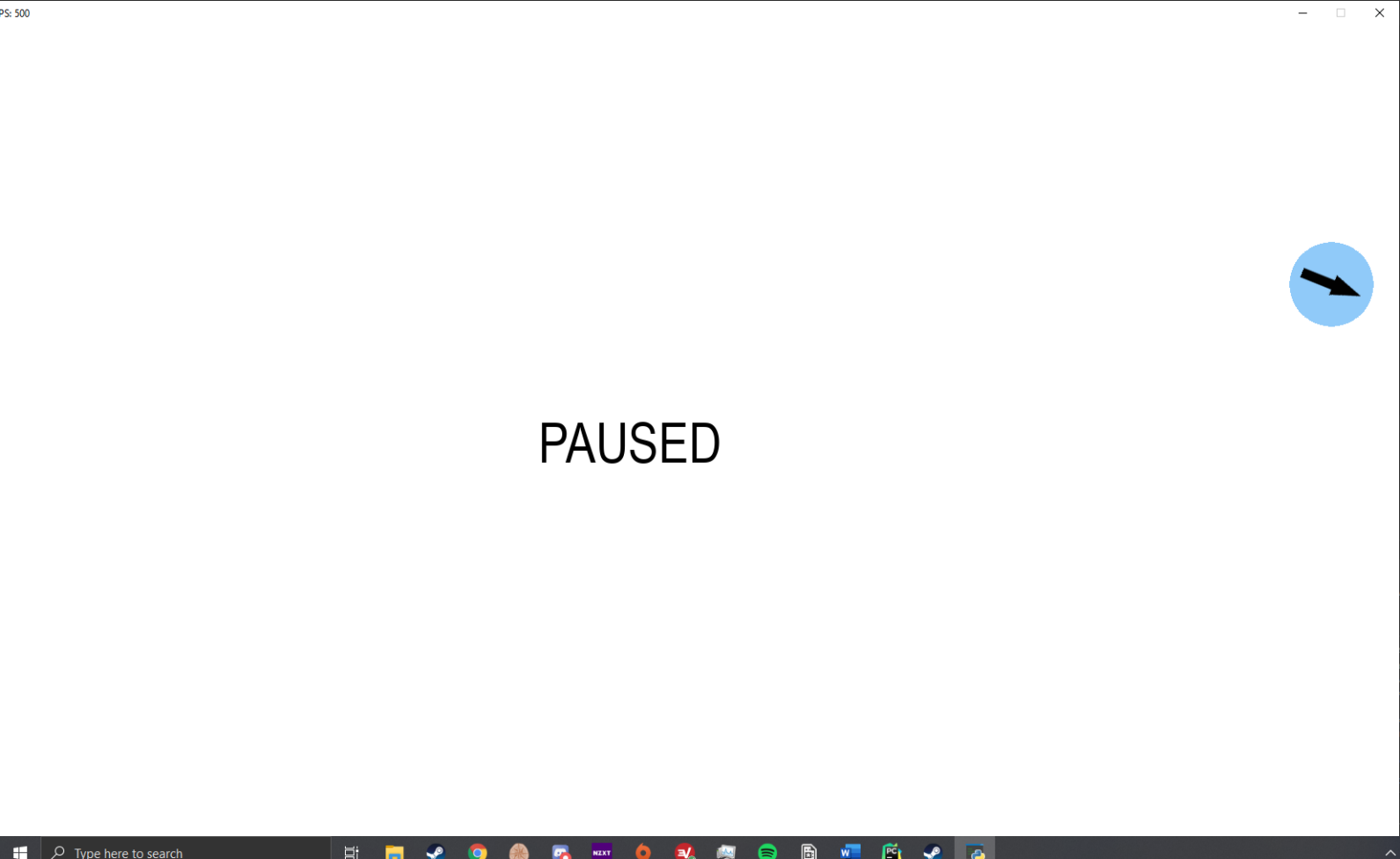
5ms^-1

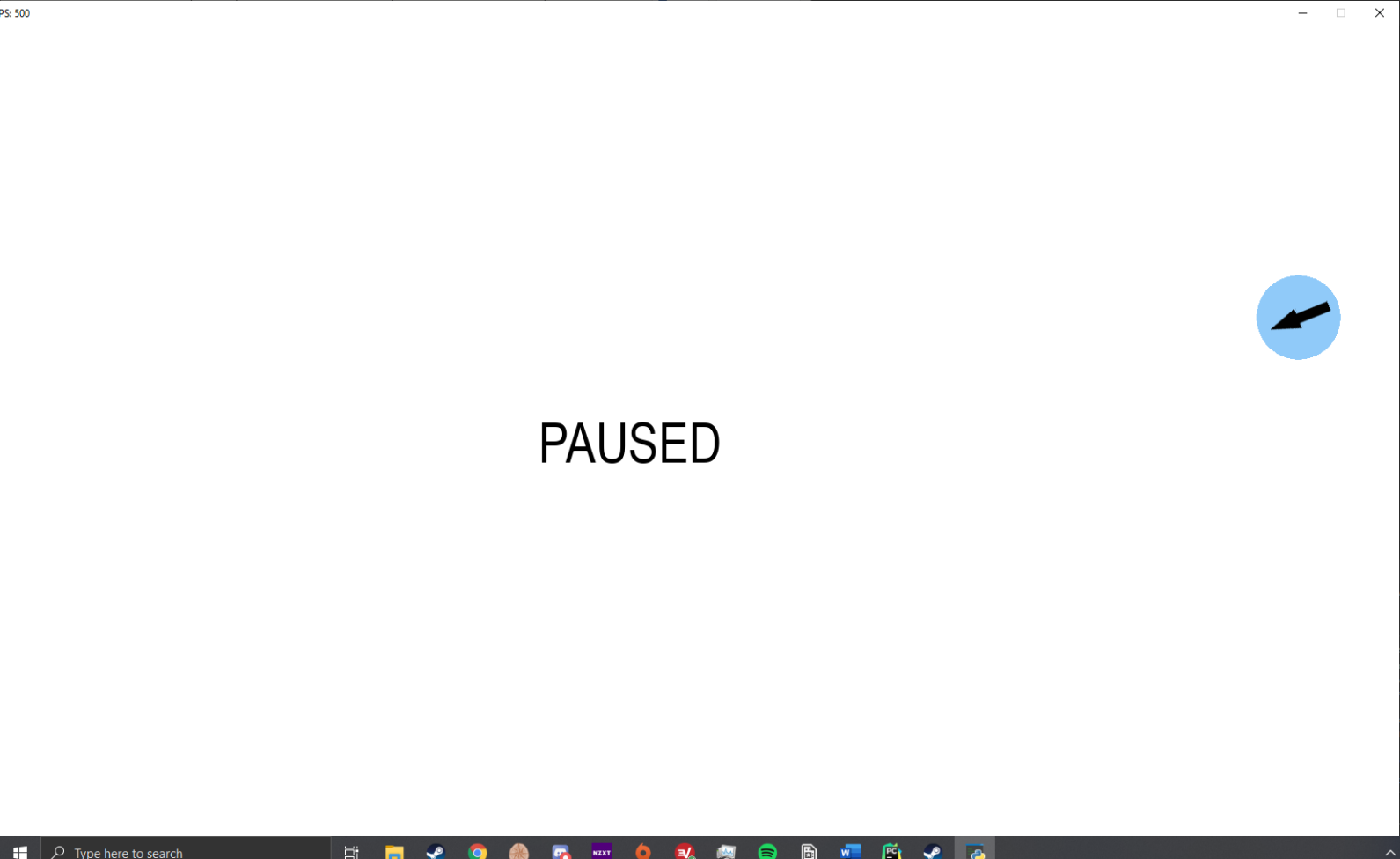
10ms^-1

**Both of these tests passed without issue**. As is shown in the diagram, the particle is placed on top of the mouse pointer, and the direction arrow lines up with the velocities of the particle.

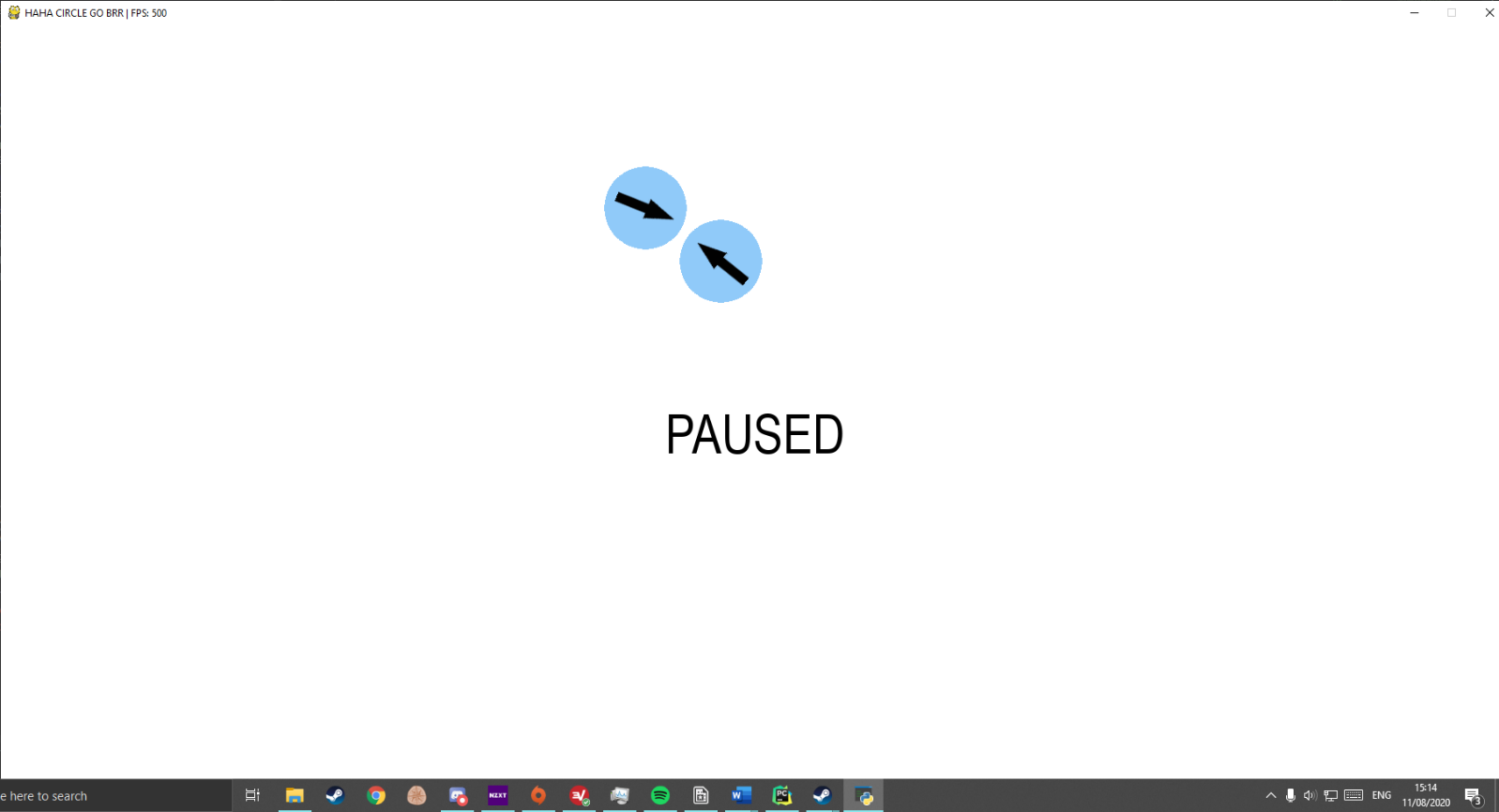
**Test 1.2**

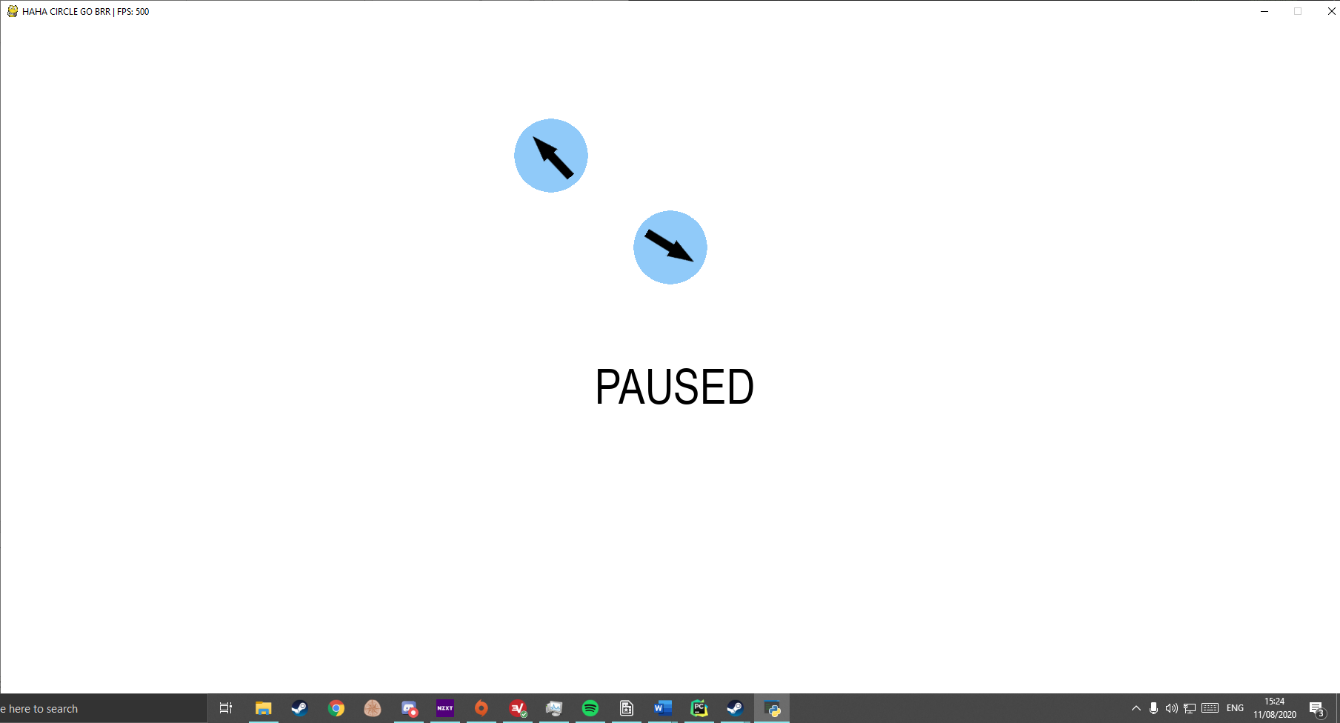
Before/after collision with wall



When the particle collided with the wall, its velocity was reversed in the horizontal axis, while its vertical velocity remained the same. This is the desired result, and as such, **the test was passed**

**Test 1.3**

Before collision (Particles have inverse vertical velocities, and slightly different horizontal velocities, although still in different directions)

After collision – Particles have collided and bounced off of one another in a way that one would expect to see in real life.

**Test 1.3 was passed.**

**Test 1.5**

As gravity is no longer implemented by default, I created a particle that had its downwards acceleration set to 9.81 ms^-2 – the same as the value that I was using for gravity in V1. I also set its position to be exactly 8 meters off of the ground, as was specified in the test instructions for test 1.5.

A picture containing bird, tree

Description automatically generated

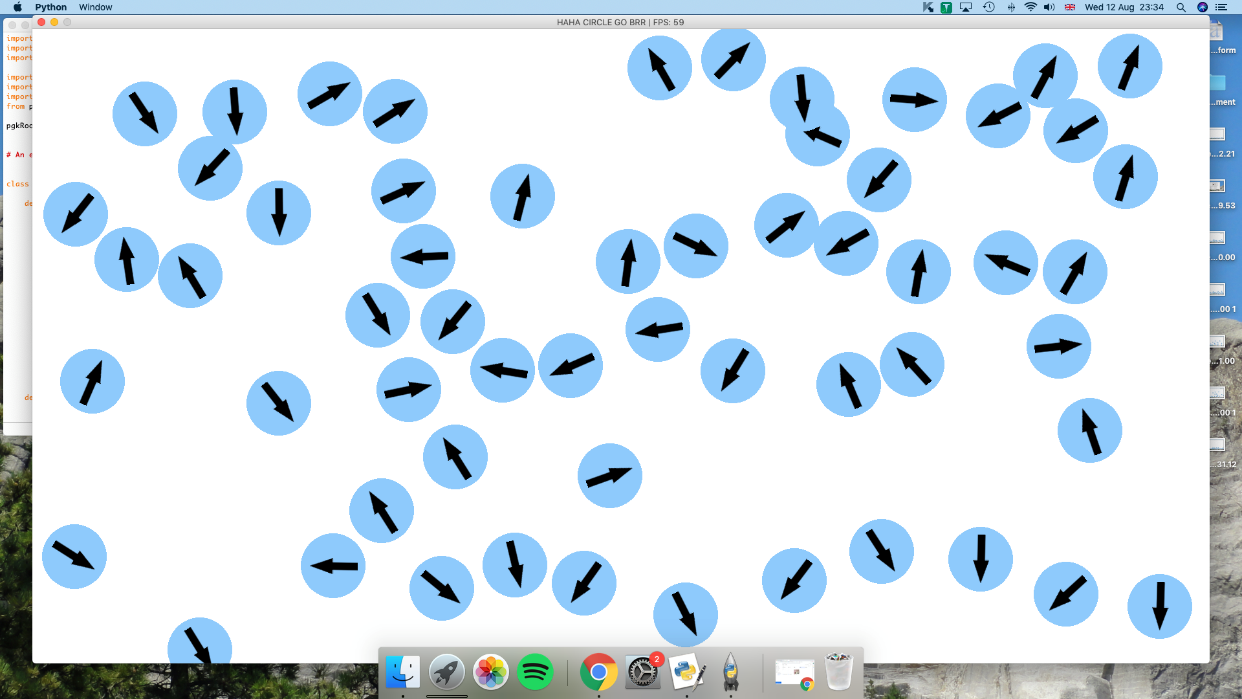
Below is a screenshot of the console output (printing the times that the particles hit the floor) when the simulation started.

**A screenshot of a cell phone

Description automatically generated**

As shown in the output, the particle hits the ground at (rounded) 1.276 seconds. This is only 0.0011 seconds faster than the calculated value. This is acceptable as the inaccuracy is so small that we can consider it negligible. **Test 1.5 was passed**

**Test 1.6**



Once again, the program ran perfectly on the Mac, even with upwards of 50 particles – although it seems that Pygame on the Mac is locked to a maximum of 60fps, as it remained at 60 even when the program was paused, and there was no processing to do.

#### Testing against V2 criteria

**Test 2.1 & 2.2**

As you can see below, the program allowed me to type in the input box. Any characters that were not numbers, hyphens(-) or decimals(.) were ignored. **This means that test 2.1 was passed.** Any characters I tried to type after the character limit of four were also ignored. **This means that test 2.2 was also passed**.

****

**Test 2.3**

**A picture containing bird, table

Description automatically generated**Following the instructions for test 2.3, I placed a particle with 5ms^-2 acceleration downwards (particle A)

And another with 10ms^-2 acceleration downwards (particle B)

A picture containing bird

Description automatically generated

Both particles were at the same height (8m) to begin with. Below is a screenshot of the console output (printing the times that the particles hit the floor)

A screenshot of a cell phone

Description automatically generated

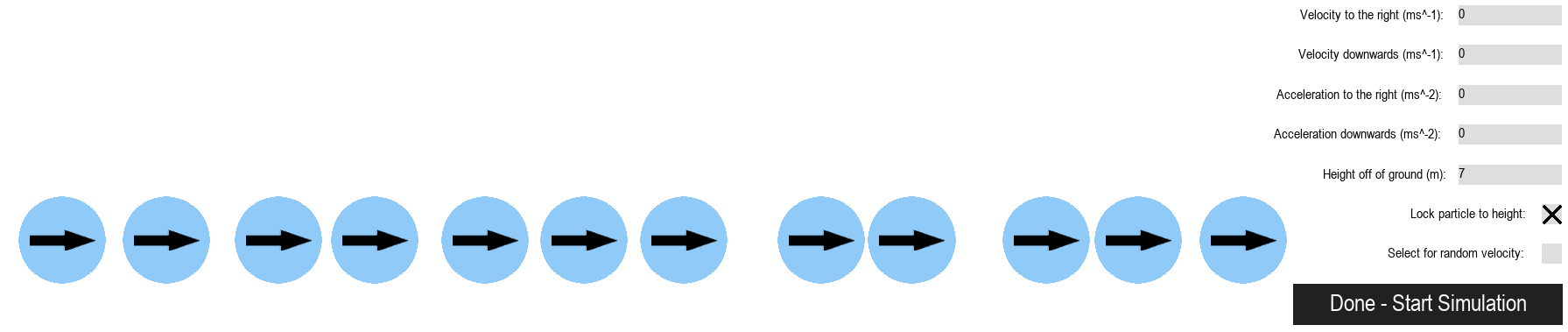
Particle A hit the floor after 1.264 seconds, and particle B hit the floor at 1.788 seconds (both times rounded to 3d.p.). Both of these values are only 0.001 seconds off of the times calculated in the test definition for test 2.3, and so **the test was passed**.

**Test 2.4**

**A picture containing drawing

Description automatically generated**

As you can see, I have left all inputs at the default - apart from the random velocity checkbox, which I have ticked.

Each particle that I placed while this checkbox was active has its own unique - random – velocity. This shows that this checkbox is working as intended.

In this screenshot, everything is left as default apart from the height off of ground (which is necessary for locking a particle to a height) and the lock to height checkbox. All particles that I placed while this checkbox was active are at the same height. While placing them, they remained at that height even if the mouse moved up or down - they only followed the mouse’s x coordinate. **This test was passed**

**Test 2.5**

A close up of a logo

Description automatically generated

When I pressed esc, all particles paused in their place, and some text came on-screen to notify me that I had paused the simulation. When esc was pressed again, the text disappeared, and all particles continued exactly as they were before the program was paused. **This test was passed**

### Development of Version 2: Review

Version 2 of the program passed all tests for versions 1 & 2 without issue, meaning that it meets all success criteria apart from 4 & 5. Referring back to the requirements for the program, version 2 means that requirements 2 and 8 are fully met, while 7 is partially met – in addition to requirement 1 being already fully met by version 1.

### Development of Version 2: Stakeholder Feedback

The following bullet points are all ideas that I have for version 3 based off of feedback that was given to me by my stakeholders.

* Add some instructions
* Add save/load functionality
* Change the size of the arrows on the particles to indicate magnitude of velocity
* Add the ability to change the appearance of particles (size/colour)
* Add the ability to zoom in and out of the simulation
* Add some time controls (fast forward, rewind)
* Add a settings menu
* Ability to save custom particle properties

## Version Three

I will be splitting the development of version three into two separate parts: updating the interface module to meet version three’s requirements, and updating version two’s main code to meet version three’s requirements.

### Development of Version Three: Interface Module (Pgkinter)

In version three, I will be adding two classes to the interface module. One of them will be a contai9ner, for handling animations. This class will mean that I can easily animate entire menus at once with just one method call, rather than having to add an animation method to each interface class. I will also be adding a dropdown menu class, that will be used in selecting a scenario to load, and selecting a material for your particle.

*# PyClipTools is an external module that allows text to be written to and*

*# read from the windows clipboard from within python*

**class** Pgk(object):

**def** \_\_init\_\_(self):  *# Initialises pgkinter - creates all necessary globals*

       [...]

For version three, I have moved all pgkinter functions into the controller object, in order to make the program entirely object-oriented.

**def** buttonDefaultAction(self):

*# Default action that will be assigned to buttons if none is assigned on*

*# instantiation*

**pass**

**def** getWidgets(self):

**return** self.pgkGroup.sprites()

**def** hoverEffect(self, rgb):

        [...]

**def** isLight(self, rgb):  *# Determines whether an rgb code is light or dark*

        [...]

*# noinspection SpellCheckingInspection*

**def** update(self):  *# Only need pgkinter.update() in main code*

        self.pgkGroup.update()

*# noinspection SpellCheckingInspection,SpellCheckingInspection*

**def** eventHandler(self, event):

        [...]

*# noinspection PyArgumentList*

**class** Button(pg.sprite.Sprite):

I have added four new attributes to the instantiation method. Image/hoverImage are assigned if I want the button to use an image rather than solid colour (for example, the time controls will use images). Container will be assigned if the button needs to be inside of a container – the button’s instantiation method will then call container.addWidget() to add itself to the container. Finally, swellOnHover dictates whether the button uses the swell/shrink functions when the mouse hovers over it. For example, the menu buttons (as they are simply text on a background) will use these functions, whereas the time controls will not as they will be using images instead.

*# noinspection SpellCheckingInspection*

**def** \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,

                 text=None, height=None, width=None,

                 action=None, image=None, hoverImage=None, container=None,

                 swellOnHover=None):

        [...]

**def** config(self, font=None, bgColour=None, text=None,

               height=None, width=None, action=None, image=None,

               hoverImage=None):

        [...]

**def** delete(self):

*# Need to set mouse back to normal, otherwise the mouse will remain*

*# hidden after the widget has been deleted*

        pg.mouse.set\_cursor(\*pg.cursors.arrow)

        self.\_\_parent.pgkGroup.remove(self)

**if** self.\_\_container:

            self.\_\_container.removeWidget(self)

**del** self

**def** draw(self):

*# Made draw its own function, as widgets need to be drawn in a*

In version 3, I have separated the draw and update methods in widget classes. This is because, with the introduction of container classes with backgrounds and foregrounds, widgets need to be drawn in different orders to the order in which they are updated by default from the pgkGroup.update() method.

*# different order if they are in a container*

**if** self.\_\_container:

            x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)

            self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)

*# Uses 0.45 \* height as 0.5 places text slightly below-centre*

            self.\_\_textRect = self.\_\_displayText.get\_rect(

                center=(x + 0.5 \* self.\_\_width, y + 0.45 \* self.\_\_height))

**if** self.\_\_hovered **and** self.\_\_timer > 0.5:

**if** self.\_\_image **is** None:

*# Draws button with hover colour variant*

                pg.draw.rect(self.\_\_screen, self.\_\_bgColourHover, self.\_\_rect)

                self.\_\_screen.blit(self.\_\_displayText, self.\_\_textRect)

**else**:

                self.\_\_screen.blit(self.\_\_hoverImage, self.\_\_rect)

            pg.mouse.set\_visible(False)

*# Draws custom mouse image over mouse location*

            self.\_\_pointerRect = self.\_\_parent.pgkPointerCursor.get\_rect(

                top=pg.mouse.get\_pos()[1])

            self.\_\_pointerRect.x = pg.mouse.get\_pos()[

                                       0] - self.\_\_pointerRect.w / 2

            self.\_\_screen.blit(self.\_\_parent.pgkPointerCursor,

                               self.\_\_pointerRect)

**elif** **not** self.\_\_hovered **or** self.\_\_timer < 0.5:

**if** self.\_\_image **is** None:

*# Draws button with regular colour*

                pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)

                self.\_\_screen.blit(self.\_\_displayText, self.\_\_textRect)

**else**:

                self.\_\_screen.blit(self.\_\_image, self.\_\_rect)

**def** getHeight(self):

**return** self.\_\_height

**def** getWidth(self):

**return** self.\_\_width

**def** getPos(self):

**return** self.\_\_rect.x, self.\_\_rect.y

**def** isHovered(self):

**return** self.\_\_hovered

*# Swell and shrink are button animations that make the button change size*

Here I have added two methods, swell and shrink. The swell method will be called from update when the mouse pointer is hovering over the button. It will calculate a distance multiplier by dividing the time that has elapsed since the last frame by the desired time for the animation (I have decided on 0.25 seconds). It then calculates the distance that the button has left to stretch in both the horizontal and vertical direction. These distances are multiplied by the distance multiplier in order to find out how much to increment the height and width of the button by this frame.

The shrink method will be called from update when the mouse pointer has *stopped* hovering over the button. It will function similarly to the swell method, but backwards.

*# when you hover over it - gives the UI a more modern and sleek feel*

**def** swell(self):

*# Works very similarly to container animations*

*# Multiplier controls how much the button swells by, in order to make*

*# the animation last a certain length of time (0.25 seconds)*

        multiplier = (time.time() - self.\_\_previousFrame) / 0.25

        widthDifference = self.\_\_origWidth \* 1.05 - self.\_\_origWidth

        heightDifference = self.\_\_origHeight \* 1.25 - self.\_\_origHeight

*# Can only add increments greater than 0/5 as pygame rect dimensions*

*# are stored as ints - anything smaller than 0.5 would have no effect*

*# on the size, and would be stuck in an infinite loop*

**if** widthDifference \* multiplier > 0.5:

            self.\_\_width += widthDifference \* multiplier

            self.\_\_coords[0] -= widthDifference \* multiplier / 2

**else**:

            self.\_\_width += 1

            self.\_\_coords[0] -= 0.5

**if** heightDifference \* multiplier > 0.5:

            self.\_\_height += heightDifference \* multiplier

            self.\_\_coords[1] -= heightDifference \* multiplier / 2

**else**:

            self.\_\_height += 1

            self.\_\_coords[1] -= 0.5

**def** shrink(self):

*# Identical to swell function, but inverse signs*

        multiplier = (time.time() - self.\_\_previousFrame) / 0.25

        widthDifference = self.\_\_origWidth \* 1.05 - self.\_\_origWidth

        heightDifference = self.\_\_origHeight \* 1.25 - self.\_\_origHeight

**if** widthDifference \* multiplier > 0.5:

            self.\_\_width -= widthDifference \* multiplier

            self.\_\_coords[0] += widthDifference \* multiplier / 2

**else**:

            self.\_\_width -= 1

            self.\_\_coords[0] += 0.5

**if** heightDifference \* multiplier > 0.5:

            self.\_\_height -= heightDifference \* multiplier

            self.\_\_coords[1] += heightDifference \* multiplier / 2

**else**:

            self.\_\_height -= 1

            self.\_\_coords[1] += 0.5

*# noinspection PyAttributeOutsideInit*

**def** update(self):

The container has a mouseMasked() method which checks the mouse’s position to see if it is obstructed by the masks. It is used here to ensure that the mouse will not be considered to be hovering over a button through a mask.

*# 0.5 timer check ensures that button can only be clicked once every*

*# 0.5 seconds*

**if** self.\_\_rect.collidepoint(pg.mouse.get\_pos()) **and** **not** self.\_\_hovered:

*# Can only be hovered over if button is not obstructed by*

*# container mask*

**if** (self.\_\_container **and** **not** self.\_\_container.mouseMasked()) **or** \

**not** self.\_\_container:

*# Sets mouse cursor to invisible*

pg.mouse.set\_visible(False)

                self.\_\_hovered = True

If the button is set to swell on hover and it is being hovered over, the program will check the button’s size to see if its current width is less than 1.25 \* its original width, and its current height is less than 1.05 \* its original height. I have implemented these checks to make sure that the button does not keep swelling forever. If one of the dimensions is larger than its upper limit, both of the dimensions will be set to their upper limit, to stop the swell method from being called.

Similarly, when the button is not hovered, its width and height will be checked against its original width and height, and the shrink method will be called if necessary.

**elif** **not** self.\_\_rect.collidepoint(

                pg.mouse.get\_pos()) **and** self.\_\_hovered:

*# Sets mouse cursor back to default*

            pg.mouse.set\_visible(True)

            self.\_\_hovered = False

**if** self.\_\_swellOnHover **and** self.\_\_hovered:

**if** self.\_\_rect.height < self.\_\_origHeight \* 1.25 **and** \

                    self.\_\_rect.width < self.\_\_origWidth \* 1.05:

                self.swell()

**elif** self.\_\_rect.height > self.\_\_origHeight \* 1.25 **or** \

                    self.\_\_rect.width > self.\_\_origWidth \* 1.05:

                self.\_\_rect.height = self.\_\_origHeight \* 1.25

                self.\_\_rect.width = self.\_\_origWidth \* 1.05

**if** self.\_\_swellOnHover **and** **not** self.\_\_hovered:

**if** self.\_\_rect.height > self.\_\_origHeight **and** \

                    self.\_\_rect.width > self.\_\_origWidth:

                self.shrink()

**if** self.\_\_rect.height < self.\_\_origHeight **or** \

                    self.\_\_rect.width < self.\_\_origWidth:

                self.\_\_rect.height = self.\_\_origHeight

                self.\_\_rect.width = self.\_\_origWidth

*# If button is not in a container, it can be drawn normally*

**if** **not** self.\_\_container:

            self.draw()

        self.\_\_timer += time.time() - self.\_\_previousFrame

        self.\_\_previousFrame = time.time()

**def** handleEvent(self, event):

        [...]

**class** Checkbox(pg.sprite.Sprite):

In the instantiation functions of all widgets aside from buttons, I have added a small bit of code that orders the widgets. As this is the checkbox instantiation, it only adds itself in front of buttons. This ordering was necessary with the addition of dropdown menus, as simply adding the widgets to the end of the sprite group led to other widgets appearing in front of a dropdown menu that was expanded, due to the fact that they were updated/drawn before the dropdown.

*# noinspection SpellCheckingInspection*

**def** \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,

                 textColour=None, inlineText=None, height=None, container=None):

        super().\_\_init\_\_()

        [...]

*# Ordering of widgets - Containers first, then labels, dropdowns,*

*# input boxes, checkboxes, then buttons. Helps with widgets handling*

*# events in the correct order - stops buttons 'hijacking' click*

*# events from dropdown menus drawn on top of them.*

        index = 0

        inGroup = False

**for** i **in** self.\_\_parent.pgkGroup.sprites():

**if** isinstance(i, Button):

                after = self.\_\_parent.pgkGroup.sprites()[index:]

**for** sprite **in** after:

                    self.\_\_parent.pgkGroup.remove(sprite)

                self.\_\_parent.pgkGroup.add(self)

**for** sprite **in** after:

                    self.\_\_parent.pgkGroup.add(sprite)

                inGroup = True

            index += 1

**if** **not** inGroup:

            self.\_\_parent.pgkGroup.add(self)

**def** config(self, font=None, bgColour=None, textColour=None,

               inlineText=None):

        [...]

**def** click(self):

**[...]**

**def** delete(self):

        [...]

**def** draw(self):

**[...]**

**def** get(self):

**return** self.\_\_output

**def** getHeight(self):

**return** self.\_\_height

**def** getWidth(self):

**return** self.\_\_width

**def** getPos(self):

**return** self.\_\_rect.x, self.\_\_rect.y

*# noinspection PyAttributeOutsideInit*

**def** update(self):

**[...]**

**def** handleEvent(self, event):

[...]

class Container(pg.sprite.Sprite):  
 def \_\_init\_\_(self, parent, screen, outlineColour=None,  
 outlineThickness=None, bg=False, bgColour=None,  
 height=None, width=None, topleft=None, topright=None,  
 bottomleft=None, bottomright=None, centre=None,  
 startVisible=True, maskColour=None):

[...]

*# In order to create the outline, I will have two rects - outlineRect  
 # and rect. outlineRect will be the colour of the outline, and will  
 # be slightly bigger than rect. It will be drawn first, with rect on  
 # top, so that only the edges will show, giving the appearance of an  
 # outline.*  
  
 self.\_\_fullHeight = self.\_\_height + self.\_\_outlineThickness \* 2  
 self.\_\_halfHeight = int(self.\_\_fullHeight / 2)  
 self.\_\_fullWidth = self.\_\_width + self.\_\_outlineThickness \* 2  
 self.\_\_halfWidth = int(self.\_\_fullWidth / 2)  
  
 self.\_\_rect = pg.Rect(0, 0, self.\_\_width, self.\_\_height)  
 self.\_\_outlineRect = pg.Rect(0, 0, self.\_\_fullWidth,  
 self.\_\_fullHeight)

Here I use an outline rect and a main rect for the container. The outline rect will be drawn first, with the main rect drawn on top of it, so that the majority is obscured. The only bits of the outline rect that are visible will form an outline around the main rect.

[...]

For the container class, there is an optional ‘startVisible’, which controls the state of the masks when the container is instantiated. If it is set to true, then the masks will be enlarged such that they will completely obscure the container, and everything within it. I programmed the container to use masks, rather than transparency values, so that I could add animations where the container is slowly revealed. This would not be possible with transparency values, as you cannot set a transparency value for part of a surface object.

The masks are initially created at (0, 0), for simplicity, but then have their positions aligned with the outline rect of the container.

*# In order to create the illusion of the container hiding widgets,  
 # I will use a 'mask' rect that is the same size as the container -  
 # this will be set to the same colour as the background in the main  
 # part of my code, making a seamless look* if startVisible:  
 self.\_\_maskLeftRect = pg.Rect(0, 0, 0, 0)  
 self.\_\_maskRightRect = pg.Rect(0, 0, 0, 0)  
 self.\_\_maskTopRect = pg.Rect(0, 0, 0, 0)  
 self.\_\_maskBottomRect = pg.Rect(0, 0, 0, 0)  
  
 elif not startVisible:  
 self.\_\_maskLeftRect = pg.Rect(0, 0, self.\_\_halfWidth,  
 self.\_\_fullHeight)  
 self.\_\_maskRightRect = pg.Rect(0, 0, self.\_\_halfWidth,  
 self.\_\_fullHeight)  
 self.\_\_maskTopRect = pg.Rect(0, 0, self.\_\_fullWidth,  
 self.\_\_halfHeight)  
 self.\_\_maskBottomRect = pg.Rect(0, 0, self.\_\_fullWidth,  
 self.\_\_halfHeight)  
  
 self.\_\_maskLeftRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskRightRect.topright = self.\_\_outlineRect.topright  
 self.\_\_maskTopRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskBottomRect.bottomleft = self.\_\_outlineRect.bottomleft  
  
 *# This attribute will be set in the startAnimation method, and will  
 # determine whether or not the container - and widgets inside of it -  
 # will be deleted once the animation has finished. This means there  
 # is no need for loads of timing variables in the main code.* self.\_\_deleteAfter = False

if len(self.\_\_parent.pgkGroup.sprites()) > 0:  
 after = self.\_\_parent.pgkGroup.sprites()[1:]  
 for sprite in after:  
 self.\_\_parent.pgkGroup.remove(sprite)  
 self.\_\_parent.pgkGroup.add(self)  
 for sprite in after:  
 self.\_\_parent.pgkGroup.add(sprite)  
  
 else:  
 self.\_\_parent.pgkGroup.add(self)  
  
 self.\_\_previousFrame = time.time()

This reorders the sprite group that contains all widgets that have been instantiated, to place the container at the back – this prevents the container from being drawn above others if they overlap.

def config(self, outlineColour=None,  
 outlineThickness=None, bg=False, bgColour=None,  
 height=None, width=None, topleft=None, topright=None,  
 bottomleft=None, bottomright=None, centre=None,  
 maskColour=None):  
  
 [...]

def addWidget(self, widget):  
 self.\_\_widgets.add(widget)  
  
 def delete(self):  
 self.\_\_parent.pgkGroup.remove(self)  
 del self  
  
 def getCorrectedCoords(self, coords):  
 newX = self.\_\_rect.topleft[0] + coords[0]  
 newY = self.\_\_rect.topleft[1] + coords[1]  
  
 return newX, newY  
  
 def getRect(self):  
 return self.\_\_rect  
  
 def animationDone(self):  
 if self.\_\_animation == [None, None, None]:  
 return True  
 else:  
 return False  
  
 def handleEvent(self, event):  
 pass  
  
 def isEmpty(self):  
 if len(self.\_\_widgets.sprites()) == 0:  
 return True  
 else:  
 return False  
  
 def isMasked(self):

This method is used by the draw methods of the widgets contained within the container. Their coordinates are passed relative to the top left corner of the container, and this method converts those relative coordinates into the absolute coordinates that Pygame uses

*# Checks if the container is fully masked*  
 if (self.\_\_maskRightRect.width == self.\_\_halfWidth and  
 self.\_\_maskLeftRect.width == self.\_\_halfWidth) or \  
 (self.\_\_maskTopRect.height == self.\_\_halfHeight and  
 self.\_\_maskBottomRect.height == self.\_\_halfHeight):  
 return True  
 else:  
 return False  
  
 def mouseMasked(self):  
 *# If the mouse is blocked by the masks, then this will be True - used  
 # to prevent clickthrough, allowing the user to interact with buttons  
 # even when they are hidden by the masks* mouseCoords = pg.mouse.get\_pos()  
 for i in [self.\_\_maskBottomRect, self.\_\_maskLeftRect,  
 self.\_\_maskTopRect, self.\_\_maskRightRect]:  
 if i.collidepoint(mouseCoords[0], mouseCoords[1]):  
 return True  
 return False  
  
 def onScreen(self):

*# Checks if the container is at least partially on-screen*  
 if self.\_\_animation[0] is None:  
 sw, sh = pg.display.get\_surface().get\_size()  
 if self.\_\_rect.bottom <= 0:  
 return False  
 elif self.\_\_rect.top >= sh:  
 return False  
 elif self.\_\_rect.topright[0] <= 0:  
 return False  
 elif self.\_\_rect.topleft[0] >= sw:  
 return False  
 else:  
 return True  
  
 return True  
  
 def removeWidget(self, widget):  
 self.\_\_widgets.remove(widget)  
  
 def startAnimation(self, type, time, inOut, startFrom=None,  
 deleteAfter=None, destination=None):  
 self.\_\_animation = [type, time, inOut]  
  
 self.\_\_deleteAfter = deleteAfter  
 if startFrom is not None:  
 *# startFrom is needed for animations that involve the whole  
 # container moving* self.\_\_animation.append(startFrom)  
  
 *# Can now specify where the container will finish its animation,  
 # instead of always finishing where it was when the animation was  
 # started. Useful if a container needs to slide off screen to the  
 # left and come back on from the right, for example.* if destination:  
 self.\_\_rect.x = destination[0]  
 self.\_\_rect.y = destination[1]  
  
 self.\_\_animation.append(self.\_\_rect.x)  
 self.\_\_animation.append(self.\_\_rect.y)  
  
 if type in [**"horizontalslide"**, **"verticalslide"**]:  
 *# Don't need masks for these animations, so set them all to be  
 # invisible* self.\_\_maskLeftRect.width = 0  
 self.\_\_maskRightRect.width = 0  
 self.\_\_maskTopRect.height = 0  
 self.\_\_maskBottomRect.height = 0  
  
 if type == **"horizontalslide"** and inOut != **"out"**:  
 self.\_\_rect.x = startFrom  
  
 elif type == **"verticalslide"** and inOut != **"out"**:  
 self.\_\_rect.y = startFrom  
  
 def centreAnimation(self):  
 *# Container, and widgets inside of it, will appear from the centre  
 # outwards.* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
  
 *# How much does the width/height change every frame?* multiplier = self.\_\_frameTime / time  
  
 *# Divide by 2 as each mask takes up half of its dimension* widthChange = int(self.\_\_fullWidth \* multiplier / 2)  
 heightChange = int(self.\_\_fullHeight \* multiplier / 2)  
  
 *# At small resolutions and high frame rates, these values will  
 # occasionally be calculated to be 0, meaning the masks will not change  
 # sizes. In that case, they need to be set to 1* if widthChange < 1:  
 widthChange = 1  
 if heightChange < 1:  
 heightChange = 1  
  
 *# If the container is appearing* if inOut == **"in"**:  
 *# Widths/heights of masks cannot go lower than 0, otherwise they  
 # would extend out the other way, resulting in longer times for  
 # disappearing animations* if self.\_\_maskLeftRect.width > 0:  
 self.\_\_maskLeftRect.width -= widthChange  
 else:  
 self.\_\_maskLeftRect.width = 0  
  
 if self.\_\_maskRightRect.width > 0:  
 self.\_\_maskRightRect.width -= widthChange  
 else:  
 self.\_\_maskRightRect.width = 0  
  
 if self.\_\_maskTopRect.height > 0:  
 self.\_\_maskTopRect.height -= heightChange  
 else:  
 self.\_\_maskTopRect.height = 0  
  
 if self.\_\_maskBottomRect.height > 0:  
 self.\_\_maskBottomRect.height -= heightChange  
 else:  
 self.\_\_maskBottomRect.height = 0  
  
 *# If all masks have completely disappeared, the animation is  
 # finished* if self.\_\_maskLeftRect.width == 0 and \  
 self.\_\_maskRightRect.width == 0 and \  
 self.\_\_maskTopRect.height == 0 and \  
 self.\_\_maskBottomRect.height == 0:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 elif inOut == **"out"**:  
 *# Same as for when the container is appearing, but invert the  
 # width/height change* [...]

This method will be called whenever the container needs to transition to a different state, such as it appearing. The parameters for the animation (type, time, whether it is animating ‘in’ or ‘out’, and locations if applicable) will be inserted into a list and stored as an attribute, which will then be accessed from the update method, which will then call the correct animation method for the type.

This method will also hide masks if necessary.

The deleteAfter argument will be set to true when this is called, if I need the container to delete itself and all widgets inside of it once its animation is complete (for example, when the program moves from the main menu to the setup screen, it uses this argument to delete the main menu after its animation has completed.)

This code controls the animation that plays when I want the container to appear from the centre outwards.

It calculates the amount by which the size of the masks should change each frame, based on the size they were at the start of the animation, the time that the animation needs to take, and the time since the last frame. It will then adjust the masks accordingly.

Hello World

Hello World

This code is similar to the centre animation, except it only uses the left- and right-hand masks.

def closeSideAnimation(self):  
 *# Appears as two 'sliding doors'* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 multiplier = self.\_\_frameTime \* 1 / time  
  
 *# This animation only deals with width, so no need to calculate  
 # height change* widthChange = self.\_\_fullWidth \* multiplier / 2  
  
 *# Set top and bottom masks to have a height of 0 - makes them  
 # invisible, as they are not needed for this animation* self.\_\_maskTopRect.height = 0  
 self.\_\_maskBottomRect.height = 0  
  
 if inOut == **"in"**:  
 *# Very similar to the centre animation, except only using  
 # maskLeft and maskRight* if self.\_\_maskLeftRect.width > 0:  
 self.\_\_maskLeftRect.width -= widthChange  
 else:  
 self.\_\_maskLeftRect.width = 0  
  
 if self.\_\_maskRightRect.width > 0:  
 self.\_\_maskRightRect.width -= widthChange  
 else:  
 self.\_\_maskRightRect.width = 0  
  
 if self.\_\_maskLeftRect.width == 0 and \  
 self.\_\_maskRightRect.width == 0:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif inOut == **"out"**:  
 *# Again, very similar to the appearance animation, except invert  
 # the width change value* [...]

Hello World

Hello World

def closeUpAnimation(self):  
 *# Appears the same as slideSideAnimation, but flipped by 90 degrees* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 multiplier = self.\_\_frameTime \* 1 / time  
  
 *# This one only deals with height, so no need to calculate widthChange* heightChange = self.\_\_fullHeight \* multiplier  
  
 *# Set the width of the left mask and the right mask to 0, making them  
 # invisible, as they are not needed for this animation* self.\_\_maskLeftRect.width = 0  
 self.\_\_maskRightRect.width = 0  
  
 if inOut == **"in"**:  
  
 if self.\_\_maskTopRect.height > 0:  
 self.\_\_maskTopRect.height -= heightChange  
 else:  
 self.\_\_maskTopRect.height = 0  
  
 if self.\_\_maskBottomRect.height > 0:  
 self.\_\_maskBottomRect.height -= heightChange  
 else:  
 self.\_\_maskBottomRect.height = 0  
  
 if self.\_\_maskTopRect.height == 0 and \  
 self.\_\_maskBottomRect.height == 0:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif inOut == **"out"**:  
 [...]

Hello World

Hello World

This code is similar to the centre animation, except it only uses the upper and lower masks.

def horizontalSlideAnimation(self):  
 *# Whole container slides in from the left or right* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 startFrom = self.\_\_animation[3]  
 destination = self.\_\_animation[4]  
 distance = destination - startFrom  
 multiplier = self.\_\_frameTime / time  
  
 *# Calculate how much the container should move this frame* posStep = abs(distance \* multiplier)  
  
 if (inOut == **"in"** and destination < startFrom) or \  
 (inOut == **"out"** and destination > startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.x > destination:  
 *# Step needs to be able to round to 1 (greater than 0.5)  
 # otherwise container won't move at all* if posStep > 0.5:  
 self.\_\_rect.x -= posStep  
 else:  
 self.\_\_rect.x -= 1  
 else:  
 self.\_\_rect.x = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif (inOut == **"in"** and destination > startFrom) or \  
 (inOut == **"out"** and destination < startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.x < destination:  
 if posStep > 0.5:  
 self.\_\_rect.x += posStep  
 else:  
 self.\_\_rect.x += 1  
 else:  
 self.\_\_rect.x = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()

This code controls the horizontal slide animation, which is different from all that have been previously mentioned, as it doesn’t utilise the masks.

This animation requires two sets of coordinates, startFrom and destination, so that the distance to travel can be calculated.

This animation changes the x coordinate of the container (and, by extension, everything within it) by an amount that is calculated each frame using the total distance to move, the time since the last frame, and the time the animation should take.

If it is animating ‘in’ then the container will move from startFrom to destination, and vice-versa for animating ‘out’. I mainly made this choice so that I could keep track of which animations moved a container onto the screen, and which moved it off of the screen.

def verticalSlideAnimation(self):  
 *# Whole container slides in from the top/bottom* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 startFrom = self.\_\_animation[3]  
 destination = self.\_\_animation[5]  
 distance = destination - startFrom  
 multiplier = self.\_\_frameTime / time  
  
 *# Calculate how much the container should move this frame* posStep = abs(distance \* multiplier)  
  
 if (inOut == **"in"** and destination < startFrom) or \  
 (inOut == **"out"** and destination > startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.y > destination:  
 *# Step needs to be able to round to 1 (greater than 0.5)  
 # otherwise container won't move at all* if posStep > 0.5:  
 self.\_\_rect.y -= posStep  
 else:  
 self.\_\_rect.y -= 1  
 else:  
 self.\_\_rect.y = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif (inOut == **"in"** and destination > startFrom) or \  
 (inOut == **"out"** and destination < startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.y < destination:  
 if posStep > 0.5:  
 self.\_\_rect.y += posStep  
 else:  
 self.\_\_rect.y += 1  
 else:  
 self.\_\_rect.y = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 def update(self):  
 *# Only draw outline rect if the outline thickness is greater than 0 -  
 # no point otherwise, as it will be completely obscured by the  
 # container rect* if self.\_\_outlineThickness > 0:  
 pg.draw.rect(self.\_\_screen, self.\_\_outlineColour,  
 self.\_\_outlineRect)  
 if self.\_\_bg:  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 *# Widgets contained within the container need to be drawn here so  
 # that they will be on top of the container, but below the masks* for widget in self.\_\_widgets:  
 widget.draw()  
  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskLeftRect)  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskRightRect)  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskTopRect)  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskBottomRect)  
  
 self.\_\_frameTime = time.time() - self.\_\_previousFrame  
 self.\_\_previousFrame = time.time()

This section of code controls the order in which the container and its widgets are drawn. The outline rect and the main rect are drawn first, followed by the widgets, followed by the masks, so that the widgets can be obscured by the masks if needs be

This animation is similar to horizontal slide, except it deals with the y coordinate of the container, instead of the x

*# All different animation types - only check if the container should  
 # be playing an animation* if self.\_\_animation[0]:  
 if self.\_\_animation[0] == **'centre'**:  
 self.centreAnimation()  
 elif self.\_\_animation[0] == **'closeside'**:  
 self.closeSideAnimation()  
 elif self.\_\_animation[0] == **'closeup'**:  
 self.closeUpAnimation()  
 elif self.\_\_animation[0] == **'horizontalslide'**:  
 self.horizontalSlideAnimation()  
 elif self.\_\_animation[0] == **'verticalslide'**:  
 self.verticalSlideAnimation()  
  
 self.\_\_maskLeftRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskRightRect.topright = self.\_\_outlineRect.topright  
 self.\_\_maskTopRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskBottomRect.bottomleft = self.\_\_outlineRect.bottomleft

Checks which animation is supposed to be playing – if any – and calls the corresponding method.

class InputBox(pg.sprite.Sprite):  
  
 def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,  
 textColour=None, inlineText=None, width=None,  
 allowLetters=True, allowNumbers=True, allowMaths=True,  
 allowSpecial=True, allowSpace=True, charLimit=None,  
 defaultEntry=None, container=None):  
 [...]

*# Acts as built-in input validation - only certain characters will be  
 # allowed to be inputted  
 # By default these attributes are set to True (in the arguments for  
 # \_\_init\_\_), meaning they will be allowed.*  
 if allowLetters:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkLetterChars)  
  
 if allowNumbers:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkNumberChars)  
  
 if allowMaths:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkMathsChars)  
  
 if allowSpecial:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkSpecialChars)  
  
 if allowSpace:  
 self.\_\_allowedChars += [**" "**]

Instead of using the parent’s letter key lists, I have decided to instead create one list of allowed characters for each input box. This prevents me from having to run a large number of if statements each frame in the handleEvent method

[...]

def config(self, font=None, bgColour=None, textColour=None, inlineText=None,  
 width=None, allowLetters=None, allowNumbers=None,  
 allowSpecial=None, charLimit=None):  
  
 *# Same functionality as other config methods* [...]

def delete(self):  
 pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 if self.\_\_container:  
 x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
 *# Aligns text with the checkbox* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 if self.\_\_hovered:  
 pg.mouse.set\_visible(False)  
 *# Draws typing cursor on location of the mouse pointer* typingRect = self.\_\_parent.pgkTypingCursor.get\_rect(  
 center=pg.mouse.get\_pos())  
 self.\_\_screen.blit(self.\_\_parent.pgkTypingCursor, typingRect)  
  
 self.\_\_screen.blit(self.\_\_outputTextDisplay, self.\_\_rect)  
 self.\_\_screen.blit(self.\_\_inlineDisplayText, self.\_\_inlineTextRect)  
  
 def get(self):  
 try:  
 if self.\_\_outputText[-1] == **"|"**:  
 return self.\_\_outputText[:-1]  
 else:  
 return self.\_\_outputText  
 except IndexError:  
 return **""** def getHeight(self):  
 return self.\_\_height  
  
 def getWidth(self):  
 return self.\_\_width  
  
 def getPos(self):  
 return (self.\_\_rect.x, self.\_\_rect.y)  
  
 def write(self, text):

Updates the position of both the main rect and the text rect. This is only necessary if the input box is inside of a container, as that is the only scenario in which these would change from what they were created as in the instantiation method (if the container is moving)

*# Changes text in input box*  
 self.\_\_outputText = text  
  
 def update(self):  
 if self.\_\_active:  
 *# Backspace needs to be placed in the update loop as it uses an  
 # updated method, which allows the user to hold it down to  
 # delete long sections of text.* keyPressed = pg.key.get\_pressed() *# Dict of pressed keys* if keyPressed[K\_BACKSPACE] and self.\_\_backspaceTimer >= \  
 self.\_\_backspaceDelay:  
 try:  
 self.\_\_outputText = self.\_\_outputText[:-1]  
  
 except IndexError:  
 *# In case there are no characters in output* pass  
 self.\_\_backspaceTimer = 0  
  
 *# Need a longer delay after first press - user won't  
 # accidentally delete multiple characters* if self.\_\_backspaceFirstPress:  
 self.\_\_backspaceDelay = 0.5  
 self.\_\_backspaceFirstPress = False  
 else:  
 self.\_\_backspaceDelay = 0.05  
  
 [...]

Will be True if backspace is pressed

I have added the ability to hold backspace to delete a long string, like you can do in most text editors. Also like in text editors, there is a small delay (self.\_\_backspaceDelay) of 0.5 seconds after deleting the first character before it continues to delete more, to prevent accidentally deleting multiple characters. After the first keypress, backspaceDelay is set to 0.05 seconds, and a character will be deleted every 0.05 seconds.

backspaceTimer is updated at the end of the update method – it adds the time elapsed since the previous frame.

*# noinspection SpellCheckingInspection* def handleEvent(self, event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 if self.\_\_hovered and not self.\_\_active:  
 self.\_\_active = True  
 self.\_\_timer = 0  
 return True  
 elif not self.\_\_hovered and self.\_\_active:  
 *# Sets cursorText to an empty string - shows that box is  
 # not active* self.\_\_cursorText = **""** self.\_\_active = False  
  
 *# Will only check keydown events if checkbox is active* if self.\_\_active:  
 if event.type == KEYUP:  
 if event.key == K\_BACKSPACE:  
 self.\_\_backspaceDelay = 0.05  
 return True  
  
 if event.type == KEYDOWN:  
 if event.key == K\_BACKSPACE:  
 self.\_\_backspaceFirstPress = True  
  
 elif event.unicode == **"**\x16**"**: *# Unicode value for ctrl+v (  
 # paste)* text = []  
 try:  
 for i in paste(): *# from pyClipTools* if i in self.\_\_allowedChars:  
 pass  
 else:  
 text.append(i)  
 text = **''**.join(text)  
  
 except TypeError:  
 text = **''** self.\_\_outputText += text  
  
 elif event.unicode in self.\_\_allowedChars:  
 self.\_\_outputText += event.unicode  
  
 try:  
 if len(self.\_\_outputText) > self.\_\_charLimit:  
 self.\_\_outputText = self.\_\_outputText[:-1]  
 else:  
 pass  
  
 except IndexError:  
 pass  
  
 return True

Using one allowedChars list in version three allows me to simply check whether the key pressed is in that list, rather than needing to check one list for letters, one for numbers, one for special characters, and so on.

When I was experimenting with Pygame’s event objects, I discovered that they had a Unicode attribute that simply returned the Unicode representation of every event (for example, pressing the E key with caps lock on would have the Unicode attribute of ‘E’). This is a very powerful attribute, as it means that I no longer have to worry about character key codes, or caps lock modifiers. It also allowed me to add the ability to paste values into the input box, as CTRL + V has its own event Unicode (\x16).

This allowed me to cut down the size of the handleEvent method from 141 lines to just 52, even with added functionality for pasting.

class Label(pg.sprite.Sprite):  
 def \_\_init\_\_(self, parent, screen, font=None, bgColour=None,  
 textColour=None, text=None, width=None, height=None,  
 container=None, topleft=None, topright=None, centre=None,  
 bottomleft=None, bottomright=None):  
 [...]  
  
 if not text:  
 self.\_\_text = **""** else:  
 *# In case I need multi-line text (string will contain newline  
 # character). Text gets broken up into a list of lines* self.\_\_text = [line for line in text.split(**'**\n**'**)]  
  
 *# Create list of rendered text surface objects* self.\_\_displayText = []  
 for text in self.\_\_text:  
 self.\_\_displayText.append(self.\_\_font.render(text, True,  
 self.\_\_textColour))  
  
 [...]

I have made the label compatible with multi-line text by simply making its text attribute a list, with each element being a line of text, rather than a simple string. To create this list, it splits the text argument it receives at each newline character (\n)

def config(self, font=None, bgColour=None, textColour=None, text=None,  
 width=None, height=None):  
  
 *# Same functionality as other config methods* [...]

def delete(self):  
 pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 if self.\_\_container:  
 x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
  
 *# Aligns text with the container* self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
 self.\_\_textRects = []  
 textGap = int(self.\_\_displayText[0].get\_rect().h \* 1.25)  
 num = 0  
  
 for i in self.\_\_displayText:  
 self.\_\_textRects.append(i.get\_rect(centerx=(self.\_\_rect.left +  
 self.\_\_width / 2),  
 top=(self.\_\_rect.top +  
 textGap \* num)))  
 num += 1  
  
 if self.\_\_bgColour is not None:  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 index = 0  
 for i in self.\_\_textRects:  
 self.\_\_screen.blit(self.\_\_displayText[index], i)  
 index += 1  
  
 def update(self):  
 pass  
  
 def handleEvent(self, event):  
 pass

This method draws each line of text on screen, with a set gap in between them

### Development of Version Three: Main Code

New Imports:

os – used to search directories for files and to reposition the pygame window.

ast – I have used the literal\_eval() method of this module to evaluate a string as a python expression (such as a list)

pathlib – used for compatibility. File paths are read differently on MacOS and Windows, this modules Path method can change how a file path is written depending on what the operating system uses.

win32api – the GetSystemMetrics method of this module is used to retrieve the resolution of the monitor, so that I can set the size of the window to simulate a full screen display

import math  
import time  
import random  
import os  
import ast   
from pathlib import Path  
  
try:  
 from win32api import GetSystemMetrics  
  
 win = True  
  
except ImportError:  
 *# If user is not on a windows system, win32api will fail to import,   
 # and the program will instead use default window size* win = False  
  
import pygame as pg  
import pygame.math as pgmath  
from pygame.locals import \*  
  
*# An external library that I made - adds tkinter features in pygame*import pgkinter as pgk

pgkRoot = pgk.Pgk()

*# Used to draw a line that follows a particle's path*class Graph(object):  
 def \_\_init\_\_(self, surface, width, height, topleft, bgColour, xLabelGap,  
 yLabelGap):  
 [...]  
  
 def draw(self):  
  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 for i in self.\_\_xLabels:  
 pg.draw.line(self.\_\_screen, (170, 170, 170),  
 (i, self.\_\_rect.top), (i, self.\_\_rect.bottom), 1)  
  
 for i in self.\_\_yLabels:  
 pg.draw.line(self.\_\_screen, (170, 170, 170),  
 (self.\_\_rect.left, i), (self.\_\_rect.right, i), 1)  
  
 for i in self.lines:  
 i.draw()  
  
 def changeLabelGap(self, xLabelGap, yLabelGap):

xLabels contains the x coordinates of each vertical line on the graph, and yLabels contains the y coordinates of each horizontal line on the grid.

*# Changes that gaps between lines on the graph*  
 self.\_\_xLabels = []  
 x = self.\_\_width  
 while x > 0:  
 self.\_\_xLabels.append(self.\_\_width - x)  
 x -= xLabelGap  
  
 self.\_\_yLabels = []  
 y = 0  
 while y < self.\_\_height:  
 self.\_\_yLabels.append(self.\_\_height - y)  
 y += yLabelGap  
  
 def clearLines(self):  
 for i in self.lines:  
 del i  
  
 self.lines = []

class Line(object):  
 def \_\_init\_\_(self, surface, graph, colour):  
 self.\_\_screen = surface  
  
 graph.lines.append(self)  
  
 self.\_\_colour = colour  
  
 self.\_\_plotCoords = []  
  
 def draw(self):  
 plots = self.\_\_plotCoords  
  
 if len(plots) > 1:  
 pg.draw.lines(self.\_\_screen, self.\_\_colour, False, plots, 2)  
  
 def addPlot(self, plot):  
 self.\_\_plotCoords.append((int(plot[0]), int(plot[1])))

Pygames draw.lines function takes a list of coordinates (plots), and draws a series of lines connecting them. This is useful as it means I can use that rather than needing to iterate through the list every frame, drawing a line between each pair of plots individually.

class Particle(pg.sprite.Sprite):  
  
 def \_\_init\_\_(self, coefficient, material, rad, density, v, colour, centre,  
 yA, xA=None):  
 [...]

This will be used to support rewinding the simulation. Each frame, the particle will store its position and velocity in a list, as a value, with the frame number as the key (not including the initial conditions, which will have a key of 0. For example, the 100th frame will be allocated a key of 100.

self.\_\_posDict = {}

[...]

def angleTo(self, p2):  
 [...]  
  
 def collide(self, p2):  
 [...]  
  
 def delete(self):  
 particles.remove(self)  
 del self  
  
 def draw(self):  
 pg.draw.circle(screen, self.colour, (self.rect.x, self.rect.y),  
 int(self.radius \* scale))  
  
 def drawDirectionArrow(self):  
 [...]  
  
 def hasCollided(self, group):  
 [...]  
  
 def scalePosition(self):  
 *# If the scale has changed since last frame, then the particle will  
 # get resized and relocated, so that it will be in the same place  
 # relative to the window and other particles.* if previousScale != scale:  
 mouseX = pg.mouse.get\_pos()[0]  
 fromMouseX = mouseX - self.pos.x  
 self.pos.x = mouseX - (fromMouseX \* (scale / previousScale))  
 fromFloor = SH - self.pos.y  
 self.pos.y = SH - (fromFloor \* (scale / previousScale))  
 self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 else:  
 pass  
  
 def updateDirection(self):  
 [...]  
  
 def updateDimension(self, rad=None, mass=None):  
 *# Apply equation v=(4/3)\*pi\*r^2 and v=m/d to update particles radius if  
 # mass is changed, and vice-versa* if rad is not None:  
 self.radius = rad  
 self.vol = (4 / 3) \* math.pi \* (self.radius \*\* 3)  
 self.mass = roundToSigFig(self.vol \* self.density, 3)  
  
 elif mass is not None:  
 self.mass = mass  
 self.vol = self.mass / self.density  
 rad = ((3 \* self.vol) / (4 \* math.pi)) \*\* (1 / 3)  
 self.radius = roundToSigFig(rad, 3)

If either radius or mass is changed by the user during the setup phase, then it will be passed as an argument to this method, which will then update the other, so that the values remain in keeping with the density of the particle’s material.

If the user zooms in or out during the setup phase, this code will change the on-screen position of the particle, to maintain correct distances between particles.

def update(self):  
 global frameNumber  
 global tNow  
  
 self.scalePosition()  
  
 *# If time is moving backwards* if TIME\_SCALES[currentTimescale] < 0:  
 try:  
 *# Sets attributes to what they were at the current frame number* p = self.posDict[frameNumber]  
 self.pos = pgmath.Vector2(p[0].x, p[0].y)  
 self.velocity = pgmath.Vector2(p[1].x, p[1].y)  
 tNow = p[2]  
  
 except KeyError:  
 pass  
  
 self.updateDirection()  
  
 self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 self.draw()  
 self.drawDirectionArrow()  
  
 *# If time is moving forward* elif TIME\_SCALES[currentTimescale] > 0:  
  
 *# Only simulate particle's motion if current frame has not  
 # already been simulated, otherwise simply retrieve its  
 # positional data from the position dictionary* if frameNumber not in self.posDict:  
 *# Checks if particle has collided with floor. Also checks  
 # that particle's velocity would take it towards the floor  
 # in order to make sure that the particle hasn't collided and  
 # was turned around in the previous frame.* [...]

If the user is reversing the simulation, the particle accesses its position dictionary, and changes its position and velocity to match whatever is stored with a key that corresponds to the current frame number (stored as a global variable, controlled in the main loop)

*#  
 # Multiply by timeMultiplier in order to increase velocity by  
 # the correct amount per second.* self.velocity += self.acceleration \* timeMultiplier  
  
 self.updateDirection()  
  
 *# Multiply by scale as velocity is ms^-1, multiply by  
 # timeMultiplier for the same reasons as before* self.pos += self.velocity \* scale \* timeMultiplier  
  
 *# Rect coordinates need to be integers* self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 self.draw()  
 self.drawDirectionArrow()  
 collisionList = self.hasCollided(particles)  
 for particle in collisionList:  
 [...]  
  
 for particle in self.recentCollisions:  
 if particle not in collisionList:  
 self.recentCollisions.remove(particle)  
 particle.recentCollisions.remove(self)  
  
 if self.line:  
 self.line.addPlot((self.pos.x, self.pos.y))

elif frameNumber in self.posDict:  
 *# If current frame has already been simulated, grab values  
 # from posDict* p = self.posDict[frameNumber]  
 self.pos = pgmath.Vector2(p[0].x, p[0].y)  
 self.velocity = pgmath.Vector2(p[1].x, p[1].y)  
 tNow = p[2]  
 self.updateDirection()  
  
 self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 self.draw()  
 self.drawDirectionArrow()

p = pgmath.Vector2(self.pos.x, self.pos.y)  
 v = pgmath.Vector2(self.velocity.x, self.velocity.y)  
  
 *# If time is set to x2 speed* if currentTimescale == 5:  
 *# Need to add data to previous 1 and 1.5 frames  
 # Interpolates what the position and velocity will be based  
 # on current velocity and acceleration* if (frameNumber - 1.5) not in self.posDict:  
 olderPos = p - (v \* scale \* timeMultiplier \* 1.5)  
 self.posDict[frameNumber - 1.5] = (olderPos, v, tNow)  
  
 if (frameNumber - 1) not in self.posDict:  
 oldPos = p - (v \* scale \* timeMultiplier)  
 self.posDict[frameNumber - 1] = (oldPos, v, tNow)  
  
 *# x1 or x2* if currentTimescale in [4, 5]:  
 *# Need to add data to previous 1/2 of a frame  
 # Interpolates what the position and velocity will be based  
 # on current velocity and acceleration* if (frameNumber - 0.5) not in self.posDict:  
 oldPos = p - (  
 v \* scale \* timeMultiplier \* 0.5)  
 self.posDict[frameNumber - 0.5] = (oldPos, v, tNow)  
  
 if frameNumber not in self.posDict:  
 *# Will always add data for the current frame if it is not  
 # already in the dictionary* self.posDict[frameNumber] = (p, v, tNow)  
  
 *# All of these checks mean that every frame (and the half frames  
 # in between them) will be included in the dictionary, allowing  
 # the user to rewind through at any speed.*

Each frame, the particle’s velocity and position vectors are copied to new vectors, which are then put into the posDict as a tuple, alongside the time elapsed since the start of the simulation.

Initially, I added the particles position and velocity attributes directly to the dictionary, without copying them over to a new vector, as follows:

self.posDict[frameNumber] = (self.pos, self.velocity, tNow)

This caused an issue, as what I didn’t realise is that this method added the vectors by reference, rather than by value. This in turn broke the rewind functionality, as all of the values in the dictionary now pointed to the same vector, meaning the particle’s position and velocity vectors were being assigned to themselves every frame, which caused no change.

The program also checks the current timescale. If it is real time (4) or double speed (5), then it must interpolate some values (as every ½ a frame needs to have values stored in the position dictionary, so that the user can rewind at half speed, if they so wish)

def absoluteDistance(pVector1, pVector2):  
 distance = pVector1 - pVector2  
 return distance.length()

def drawDottedLine(start, end):  
 *# Used when resizing particles, to create the same look as in Blender  
 # (Dotted line from the centre of the object being resized to the mouse)* xLen = end[0] - start[0]  
 yLen = end[1] - start[1]  
  
 xStep = xLen / 10  
 yStep = yLen / 10  
  
 xCoord = start[0]  
 yCoord = start[1]  
  
 for i in range(0, 5):  
 pg.draw.line(screen, (33, 33, 33), (xCoord, yCoord),  
 (xCoord + xStep, yCoord + yStep), 2)  
 xCoord += 2 \* xStep  
 yCoord += 2 \* yStep  
  
  
*# Code snippet found online - rounds a number, x, to a given number of  
# significant figures, n.*def roundToSigFig(x, n):  
 [...]  
  
  
def timeChange(speedUp):  
 global currentTimescale  
  
 *# User can only speed up if the current timescale is at most one less  
 # than the maximum. Similarly, they can only decrease the timescale if it  
 # is a least one more than the minimum. This is to prevent the timescale  
 # from going out of bounds.* if 0 <= currentTimescale < len(TIME\_SCALES) - 1 and speedUp == 1:  
 currentTimescale += 1  
 elif 0 < currentTimescale <= len(TIME\_SCALES) - 1 and speedUp == -1:  
 currentTimescale -= 1

This function is called from the fast forward/rewind buttons when the user presses one of them, and simply changes the numerical value of the timescale. Initially, I didn’t have the if/elif statements, and just used currentTimescale += speedUp, as speedUp was always going to be 1 or -1. However, this led to an issue in that the procedure was allowing time to be changed greater than the maximum (which would lead to an index error as the program tried to index the seventh element from a list that only contains six elements) or lower than the minimum (as the minimum index is 0, this would lead to -1, which would mean the timescale would loop back round and go to the maximum.)

This procedure is called from within the sizeChange procedure, and will draw a dotted line from the centre of the particle to the mouse pointer. In Pygame, there is no way to simply draw a dotted line, so instead it will draw a series of five lines, each with a small gap between them, to create the dotted line.

*# Each procedure which contains a loop needs to have at least one argument,  
# whether it is used or not, as the loop that prevents recursion needs to  
# pass an argument (it passes \*args, which cannot pass nothing). So I put a  
# dummy argument in the procedures that don't need anything to be passed to  
# them.*def mainMenu(dummyArg):

As I got further through version 3, error messages started getting longer and longer, like below:

Text

Description automatically generated with medium confidence

This error message is extremely lengthy and messy, all for one simple typo, and the complexity of these errors made it more difficult to locate bugs within my code. The errors were this long because my program had many function calls from within other functions (due to the nature of it moving from one menu screen to another), and Python’s error messages always start from <module> - the base level of a program – working through and listing all functions that the program went through before it got from <module> to the error.

To fix this issue, I stopped calling functions from within functions unless that function would return at some point. After my main code at the bottom of my program, I added this section:

nextFunction, args = mainMenu(1)  
*# Prevents recursion (For example, mainMenu would be called  
# from within main, which would be called from within setup, which would  
# be called from within mainMenu, and so on) which results in long errors*while True:  
 nextFunction, args = nextFunction(\*args)

This little loop initiates by calling mainMenu, which returns the next function to be called (either instructions, loadSetup, or setup) alongside the arguments that need to be passed to whichever function has been called, which then returns the next function and its arguments, and so on and so forth. args must be a tuple, so that it can be unpacked by the \* when nextFunction is called (each item in the tuple will be separated out into its own argument). This tuple unpacking allows me to use this loop for all of my functions, no matter how many arguments they require, as all I need to pass is an unpacked tuple, which could be any number of arguments. However, it cannot be zero arguments, as tuple unpacking requires at least one item in the tuple. This means that, to maintain compatibility with all of my code, every function needs at least one argument, hence the use of a dummy argument in mainMenu – it does nothing within the function, its sole purpose is to allow the tuple unpacking to work.

After implementing this, every function is now called from the base level of the program, rather than from within a function, preventing overly long error messages, and saving some memory, as less functions are running at the same time.

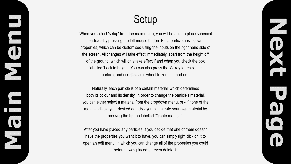
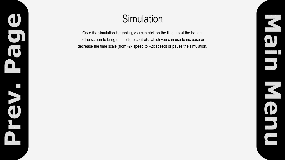
def mainMenu(dummyArg):  
 global mainmenu  
 global mainWidgets  
 global currentTimescale  
 global tNow  
 global frameNumber  
 global particleGraph  
 global scale  
 global previousScale  
  
 def endFunction(widgets, goTo, args):  
 *# Can't use buttons to set variables, so I need to use this function  
 # to set global variables* global nextFunc  
 global nextArgs  
 global mainmenu  
  
 if goTo == instructions:  
 instructions(\*args)  
 return  
  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW, True)  
  
 mainmenu, nextFunc, nextArgs = False, goTo, args  
  
 def exitProgram():  
 pg.quit()  
 quit()  
  
 global timeMultiplier  
  
 *# Reset scale and time-related globals to their default values - prevents  
 # bugs when going from a simulation to the menu, and then into another  
 # simulation* currentTimescale = 4  
 tNow = 0  
 frameNumber = 0  
 scale = scaler(100, **"x"**)  
 previousScale = scale  
  
 particleGraph.clearLines()  
 del particleGraph  
  
 particleGraph = Graph(screen, SW, SH, (0, 0), BG\_COLOUR, scale, scale)  
  
 mainContainer = pgk.Container(pgkRoot, screen, topleft=(0, 0),  
 width=SW, height=SH)  
  
 mainWidgets = [  
 pgk.Label(pgkRoot, screen, centre=(SW / 2, scaler(135, **"y"**)),  
 font=LARGE\_FONT,  
 text=**"Particle Simulator V3 by Jack Sanders"**,  
 container=mainContainer)  
 ]  
  
 buttonX = scaler(640, **"x"**)  
  
 mainWidgets += [  
 pgk.Button(pgkRoot, screen, buttonX, scaler(425, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Create a simulation"**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=lambda: endFunction(mainWidgets, setup, (1,)),  
 container=mainContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen, buttonX, scaler(560, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Load a saved simulation"**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=lambda: endFunction(mainWidgets, loadSetup,  
 (None,)),  
 container=mainContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen, buttonX, scaler(695, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Instructions"**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=lambda: endFunction(mainWidgets, instructions,  
 (mainWidgets,)),  
 container=mainContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen, buttonX, scaler(830, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Exit program :("**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=exitProgram,  
 container=mainContainer, swellOnHover=True),  
 ]  
  
 mainWidgets.append(mainContainer)  
  
 del mainContainer  
  
 mainWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"in"**, SW)  
  
 mainmenu = True  
 while mainmenu:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 mainmenu = False  
 pg.quit()  
 quit()  
  
 try:  
 *# If statement prevents 'jumping' of particles when user moves  
 # the window* if time.time() - previousFrame < 0.1:  
 timeMultiplier = (time.time() - previousFrame)  
 timeMultiplier \*= TIME\_SCALES[currentTimescale]  
 *# Calculates time between frames* previousFrame = time.time()  
 except NameError:  
 *# Will occur on the first frame, as there is no previous frame* pass  
  
 screen.fill(BG\_COLOUR)  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 *# Returns the function that runs next (setup) and the args to pass to that   
 # function (\*args requires a tuple to unpack)* return nextFunc, nextArgs

This resets the simulation controlling globals, so that each simulation starts with the default settings

Setting up widgets that are used by the main menu.

Globals needed here so they can be used by the mainMenu function, to return them to the main program. Not ideal, but buttons cannot set variables, so this is the best solution.

Function and tuple of arguments returned to main code.

def instructions(mainMenuWidgets):  
 global instructing  
  
 def changePage(goToPage, pages, mainMenuWidgets):  
 global instructing  
  
 *# First page slides to the right and gets deleted, main menu slides  
 # back in from the left, and delete the second page* if goToPage == -1:  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, SW, True)  
 mainMenuWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0)  
 for i in pages[1]:  
 i.delete()  
  
 instructing = False  
  
 *# First page slides to the left, second page slides in from the right* elif goToPage == 0:  
 pages[1][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, SW)  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0)  
  
 *# Second page slides to the right, second page slides in from the left* elif goToPage == 1:  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW)  
 pages[1][-1].startAnimation(**"horizontalslide"**, 0.5, **"in"**, SW,  
 destination=(0, 0))  
  
 *# Second page slides to the left and gets deleted, main menu slides  
 # back in from the right, and delete the first page* elif goToPage == 2:  
 pages[1][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW,  
 True)  
 mainMenuWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 0, destination=(SW, 0))  
 for i in pages[0]:  
 i.delete()  
  
 instructing = False  
  
 mainMenuWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW)  
  
 files = [**"setupInstructions.txt"**, **"mainInstructions.txt"**]  
 pageTitles = [**"Setup"**, **"Simulation"**]  
 pages = []  
 currentPage = 0  
  
 for i in files:  
 *# Iterate through instruction files, creating labels and buttons* with open(i, **"r"**) as f:  
 lines = f.readlines()  
  
 text = **''**.join(lines)  
  
 pageContainer = pgk.Container(pgkRoot, screen, topleft=(0, SW),  
 width=SW, height=SH)  
 page = [  
 pgk.Label(pgkRoot, screen, centre=(SW / 2, scaler(135, **"y"**)),  
 font=LARGE\_FONT, text=pageTitles[currentPage],  
 container=pageContainer),  
 pgk.Label(pgkRoot, screen, centre=(SW / 2,  
 SH / 2 + scaler(135, **"y"**)),  
 height=SH - scaler(135, **"x"**), width=SW \* 0.8,  
 font=MID\_FONT, text=text, container=pageContainer),  
 ]  
  
 *# Need to explicitly state the page values for the buttons - can't  
 # use currentPage + or - 1. This is because lambda passes the  
 # arguments as they are at the time of the button press. In this case  
 # that means that the previous page button will always pass 1,  
 # and the next page will always pass 3.* if currentPage == 0:  
 page += [  
 pgk.Button(pgkRoot, screen, 0, 0, height=SH, width=int(SW / 10),  
 action=lambda: changePage(-1, pages,  
 mainMenuWidgets),  
 image=L\_MENU\_IMG, hoverImage=L\_MENU\_IMG,  
 container=pageContainer),  
 pgk.Button(pgkRoot, screen, SW - int(SW / 10), 0, height=SH,  
 width=int(SW / 10),  
 action=lambda: changePage(1, pages,  
 mainMenuWidgets),  
 image=NEXT\_IMG, hoverImage=NEXT\_IMG,  
 container=pageContainer),  
 pageContainer  
 ]  
  
 else:  
 page += [  
 pgk.Button(pgkRoot, screen, 0, 0, height=SH, width=int(SW / 10),  
 action=lambda: changePage(0, pages,  
 mainMenuWidgets),  
 image=PREV\_IMG, hoverImage=PREV\_IMG,  
 container=pageContainer),  
 pgk.Button(pgkRoot, screen, SW - int(SW / 10), 0, height=SH,  
 width=int(SW / 10),  
 action=lambda: changePage(2, pages,  
 mainMenuWidgets),  
 image=R\_MENU\_IMG, hoverImage=R\_MENU\_IMG,  
 container=pageContainer),  
 pageContainer  
 ]  
  
 del pageContainer  
  
 pages.append(page)  
  
 currentPage += 1  
  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"in"**, SW,  
 destination=(0, 0))  
  
 instructing = True  
 while instructing:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 instructing = False  
 pg.quit()  
 quit()  
 screen.fill(BG\_COLOUR)  
 pgkRoot.update()  
 pg.display.update()  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()

The text that appears on the instruction screens is stored in .txt files and then read in by the program. I chose to store the instructions this way as it makes them much easier to edit than if they were hardcoded into the code.

This controls the animation for changing pages. When one page slides out to the left, another slides in from the right, and vice versa. This gives the appearance that the pages are ‘connected’ to one another, and has an appealing appearance.

One issue I found was, again, accidentally passing values by reference. If I made the buttons action changePage(currentPage – 1) Then it would use the value of currentPage at the time of the button being pressed, which would be 2, as it stops being incremented after this loop. This meant I had to instead use an if/else statement and pass the actual integers for the page that the button needs to change to.

return

def setup(dummyArg):  
 global scale  
 global previousScale   
 global setting  
 global editingParticle

*# Controls setup loop* setting = True  
 *# List of widgets used when editing a placed particle* editList = None  
 *# The particle being edited* editingParticle = None  
  
 *# How many metres are shown in the scale display* metres = 1  
  
 def endFunction(widgetList):  
 *# Starts the setupContainer's slide out animation* widgetList[-1].startAnimation(**"horizontalslide"**, 0.1, **"out"**,  
 SW + contWidth, deleteAfter=True)  
  
 def endParticleEdit(editList):  
 global editingParticle  
 *# Menu for editing a particle will disappear* editList[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
 editingParticle = Nonedef updateParticle(pRef, inputs, editing=None):  
 *# Editing argument specifies whether or not the particle being  
 # updated has already been placed (True/False)* try:  
 *# Easier to add new inputs - only need to change index here* coefficientBox = inputs[0]  
 coefficient = float(inputs[0].get())  
 xVel = float(inputs[1].get())  
 yVel = float(inputs[2].get())  
 xAccel = float(inputs[3].get())  
 yAccel = float(inputs[4].get())  
 radBox = inputs[5]  
 rad = float(radBox.get())  
 massBox = inputs[6]  
 mass = float(massBox.get())  
 height = float(inputs[7].get())  
 lockHeight = inputs[8].get()  
 randomV = inputs[9].get()  
 drawGraph = inputs[10].get()  
 material = inputs[11].get()  
 if material in MATERIALS:  
 density, colour = MATERIALS[material][0], MATERIALS[material][1]  
 else:  
 density, colour = customMaterials[material][0], \  
 customMaterials[material][1]  
  
 pRef.colour = colour  
 except ValueError:  
 return  
  
 mouseX = pg.mouse.get\_pos()[0]  
 mouseY = pg.mouse.get\_pos()[1]  
  
 if coefficient > 1:  
 coefficientBox.write(**"1"**)  
 elif coefficient < 0:  
 coefficientBox.write(**"0"**)  
  
 else:  
 pRef.restCoefficient = coefficient  
  
 if not randomV:  
 pRef.velocity.x = xVel  
 pRef.velocity.y = yVel  
 pRef.hasRandomVelocity = False  
 elif randomV and not pRef.hasRandomVelocity:  
 upper = pRef.radius \* 5  
 lower = upper \* -1  
  
 *# random.uniform instead of random.randint as uniform allows for  
 # two floating point numbers as the bounds (instead of integers)* pRef.velocity.x = roundToSigFig(random.uniform(lower, upper), 3)  
 pRef.velocity.y = roundToSigFig(random.uniform(lower, upper), 3)  
  
 inputs[1].write(str(pRef.velocity.x))  
 inputs[2].write(str(pRef.velocity.y))  
 pRef.hasRandomVelocity = True  
  
 if not drawGraph and pRef.line:  
 pRef.line = False  
  
 elif drawGraph and not pRef.line:  
 pRef.line = Line(screen, particleGraph, colour)  
  
 pRef.acceleration.x = xAccel  
 pRef.acceleration.y = yAccel  
  
 *# If user has selected that they want the particle to be locked to a  
 # certain height* if lockHeight:  
 if editing is None:  
 pRef.pos.x = mouseX  
 pRef.rect.x = mouseX  
 pRef.pos.y = SH - int(height \* scale) - pRef.radius \* scale  
 pRef.rect.y = SH - int(height \* scale) - pRef.radius \* scale  
 else:  
 if editing is None:  
 pRef.pos.x = mouseX  
 pRef.pos.y = mouseY  
 pRef.rect.x = mouseX  
 pRef.rect.y = mouseY  
  
 minRad = roundToSigFig(scaler(10, **"x"**) / scale, 3)  
 maxRad = roundToSigFig((SW / 4) / scale, 3)  
  
 *# Only update particle's radius/mass if they are not already equal to  
 # the values entered by the user* if density != pRef.density:  
 pRef.density = density  
 pRef.colour = colour  
 pRef.updateDimension(rad=rad)  
 radBox.write(str(pRef.radius))  
 massBox.write(str(pRef.mass))  
  
 elif rad != pRef.radius and minRad <= rad <= maxRad:  
 pRef.updateDimension(rad=rad)  
 massBox.write(str(pRef.mass))  
  
 elif mass != pRef.mass:  
 pRef.updateDimension(mass=mass)  
 radBox.write(str(pRef.radius))  
  
 if pRef.radius > maxRad:  
 pRef.updateDimension(rad=roundToSigFig((SW / 4) / scale, 3))  
 radBox.write(str(pRef.radius))  
 massBox.write(str(pRef.mass))  
  
 elif pRef.radius < minRad:  
 pRef.updateDimension(rad=roundToSigFig(scaler(10, **"x"**) / scale, 3))  
 radBox.write(str(pRef.radius))  
 massBox.write(str(pRef.mass))  
  
 pRef.material = material  
  
 def deleteParticle(particle):  
 particle.delete()  
  
 def clearParticles():  
 for particle in particles.sprites():  
 particle.delete()  
  
 particles.add(Particle(1, **"Wood - 800kgm^-3"**,  
 roundToSigFig((SW / 4) / scale, 3), 1, (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])), 0, xA=0))

Coefficient of restitution must be between 1 and 0 (inclusive) to avoid breaking the laws of physics.

This function is similar to what it was in version 2, but now with more properties to update, and also the ability to edit particles that have already been placed.

The particle will only follow the mouse if the user is not editing an already placed particle, to prevent them from accidentally moving a particle if they just wanted to change its velocity, for example.

Set scaled bounds for the particle’s radius, to prevent the user from making a particle that is too large for the screen, or too small to accurately simulate.

Checks whether the value the user has entered the input box is different that the particle’s current value for its radius and mass. If one of them is different, then the particle will update the others based on that value, to keep the correct density for their selected material.

If the user enters a radius that is greater than the maximum or smaller than the minimum, it will be set to whichever value they exceeded, and the particle’s mass will be updated to maintain correct density.

Deletes all particles that are on the screen, and instantiates a new dummy particle that will follow the mouse pointer.

setupContainer = pgk.Container(pgkRoot, screen,  
 topright=(SW, 0),  
 outlineThickness=0, width=scaler(400, **"x"**),  
 height=scaler(520, **"y"**))  
  
 contWidth = scaler(400, **"x"**)  
 offset = scaler(150, **"x"**)  
 boxWidth = scaler(125, **"x"**)  
 inputList = [  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(55, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Coefficient of Restitution:"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0.75"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(105, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Velocity to the right (ms^-1):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(155, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Velocity downwards (ms^-1):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(205, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Acceleration to the right (ms^-2):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(255, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Acceleration downwards (ms^-2):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(305, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Radius (m):"**, width=boxWidth, charLimit=10,  
 allowLetters=False, allowSpecial=False, allowSpace=False,  
 defaultEntry=**"0.5"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(355, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Mass (kg):"**, width=boxWidth, charLimit=10,  
 allowLetters=False, allowSpecial=False, allowSpace=False,  
 defaultEntry=**"0.5236"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(405, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Height off of ground (m):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 contWidth - scaler(50, **"x"**), scaler(455, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Lock particle to height: "**,  
 container=setupContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 contWidth - scaler(50, **"x"**), scaler(505, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Select for random velocity: "**,  
 container=setupContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 contWidth - scaler(50, **"x"**), scaler(555, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Draw line following particle's motion: "**,  
 container=setupContainer),  
  
 *# Create dropdown menu last as it needs to be drawn on top of the other  
 # inputs* pgk.Dropdown(pgkRoot, screen, contWidth - offset -  
 boxWidth, scaler(5, **"y"**), sortedCustoms + MATERIALS\_SORTED,  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Select material (scroll to see more):"**,  
 width=boxWidth \* 2, container=setupContainer)  
 ]

Instantiating the container, the input boxes, the checkboxes, and the dropdown for use in the setup loop.

*# Y distance between the buttons* buttonGap = scaler(50, **"y"**) + inputList[0].getHeight()  
  
 inputList += [  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**), scaler(605, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Create Custom Material"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: createMaterial(widgetList),  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Save Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: saveSetup(widgetList),  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap \* 2, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Load Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: loadSetup(widgetList),  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap \* 3, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Clear Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**), action=clearParticles,  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap \* 4, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Done - Start Simulation"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: endFunction(widgetList),  
 container=setupContainer, swellOnHover=True),  
 ]  
  
 widgetList = inputList + [setupContainer]  
  
 *# Remove references - Collected by garbage collection* del setupContainer  
 del inputList  
  
 widgetList[-1].startAnimation(**"horizontalslide"**, 0.25, **"in"**,  
 SW)  
  
 particles.add(Particle(1, **"Wood - 800kgm^-3"**,  
 roundToSigFig((SW / 4) / scale, 3), 1, (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])), 0, xA=0))  
  
 pRef = particles.sprites()[-1]  
  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])

Adding all of the buttons that are used by the setup loop. This was done separately to the input methods to make the code more readable

setting = True  
 while setting:  
 previousScale = scale  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
  
 if not pgkRoot.eventHandler(event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 *# Check that looks at all possibilities to ensure  
 # that the particle will be placed on-screen and not  
 # intersecting with another particle* if pRef.pos.x + pRef.radius \* scale > SW \  
 or pRef.pos.x - pRef.radius \* scale < 0 \  
 or pRef.pos.y + pRef.radius \* scale > SH \  
 or pRef.pos.y - pRef.radius \* scale < 0 \  
 or len(pRef.hasCollided(particles)) != 0:  
 pass  
 else:  
 rad = roundToSigFig((SW / 4) / scale, 3)  
 particles.add(Particle(1, **"Wood - 800kgm^-3"**,  
 **"Wood - 800kgm^-3"** rad, 10, (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])),  
 0, xA=0))  
  
 elif event.button == 3:  
 *# Editing particles after they have been placed* mouseCoords = pg.mouse.get\_pos()  
 for i in particles.sprites()[:-1]:  
 if absoluteDistance(pgmath.Vector2(mouseCoords),  
 i.pos) <= i.radius \* scale:  
 if editList:  
 for widget in editList:  
 widget.delete()  
 del editList  
 editingParticle = i  
  
 *# Create container for widgets first,  
 # then position it so that it will always be  
 # on screen.* eContainer = pgk.Container(pgkRoot, screen,  
 centre=(0, 0),  
 outlineThickness=3,  
 width=scaler(310,  
 **"x"**),  
 height=scaler(400,  
 **"y"**),  
 bg=True,  
 bgColour=(255, 255,  
 255),  
 startVisible=False)  
  
 *# Container will be positioned so that one of  
 # its corners will be in the centre of the  
 # particle* pos = i.pos  
 if pos[1] + scaler(400, **"y"**) <= SH:  
 if pos[0] + scaler(310, **"x"**) <= SW:  
 eContainer.config(topleft=pos)  
 else:  
 eContainer.config(topright=pos)  
 else:  
 if pos[0] + scaler(310, **"x"**) <= SW:  
 eContainer.config(bottomleft=pos)  
 else:  
 eContainer.config(bottomright=pos)  
  
 editContWidth = scaler(310, **"x"**)  
 offset = scaler(150, **"x"**) / 2  
 boxWidth = scaler(125, **"x"**) / 2

Positioning the container. The program checks the position of the particle, and will place the container in such a way that it will always be on screen. For example, if a particle is on the left-hand side of the screen, the container will be placed to the right of the particle. Also, the program will ensure that one of the corners of the container will always be in the centre of the particle, so that the user will easily be able to tell which particle is being edited.

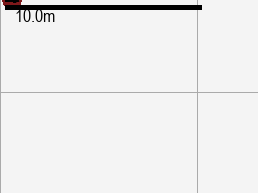
If the user right clicks, then the program will first check the location of the mouse pointer. If it is on top of a particle, then they will be able to edit that particle.

*# Creating list of widgets used in editing  
 # the particle* inputList = [  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(28, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Coefficient of "  
 "Restitution"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox([...]),  
 pgk.InputBox([...]),  
 pgk.InputBox([...]),  
 pgk.InputBox([...]),  
 pgk.InputBox([...]),  
 pgk.InputBox([...]),  
 pgk.InputBox([...]),  
 pgk.Checkbox([...]),  
 pgk.Checkbox([...]),  
 pgk.Checkbox([...])  
 ]  
  
 buttonHeight = inputList[0].getHeight() \* 2  
 fButton = pgk.Button(pgkRoot, screen,  
 editContWidth - scaler(225,  
 **"x"**),  
 scaler(303, **"y"**),  
 font=SMALL\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Finish Editing "  
 "Particle"**,  
 height=buttonHeight,  
 width=scaler(213, **"x"**),  
 action=lambda:  
 endParticleEdit(  
 editList),  
 container=eContainer,  
 swellOnHover=True)  
  
 delButton = pgk.Button(pgkRoot, screen,  
 editContWidth - scaler(  
 225, **"x"**),  
 scaler(353, **"y"**),  
 font=SMALL\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Delete Particle"**,  
 height=buttonHeight,  
 width=scaler(213, **"x"**),  
 action=lambda:  
 deleteParticle(  
 editingParticle),  
 container=eContainer,  
 swellOnHover=True)  
  
 *# Create dropdown menu last as it needs to be  
 # drawn on top of the other inputs* drop = (pgk.Dropdown(pgkRoot, screen,  
 editContWidth - offset -  
 boxWidth,  
 scaler(5, **"y"**),  
 sortedCustoms +  
 MATERIALS\_SORTED,  
 font=SMALLER\_FONT,  
 bgColour=(  
 222, 222,  
 222),  
 inlineText=**"Select "  
 "material "  
 "(scroll to see"  
 " more):"**,  
 width=boxWidth \* 2,  
 container=eContainer))

Instantiating widgets for the editing menu. Essentially the same as the main setup menu, just smaller.

Using a container means that – as well as having animation – I can set coordinates relative to the top left of the container. This is especially helpful here, as the edit menu needs to be placed next to a particle, which could be anywhere on the screen. Relative coordinates mean that I just need to code one set in, and it will look the same no matter where the container is placed. I only need to figure out the absolute coordinates once.

*# Write to the input boxes so that they so  
 # they can see the selected particle's*

 *# properties* inputList[0].write(str(i.restCoefficient))  
 inputList[1].write(str(i.velocity.x))  
 inputList[2].write(str(i.velocity.y))  
 inputList[3].write(str(i.acceleration.x))  
 inputList[4].write(str(i.acceleration.y))  
 inputList[5].write(str(i.radius))  
 inputList[6].write(str(i.mass))  
  
 if i.hasRandomVelocity:  
 inputList[9].click()  
 if i.line:  
 inputList[10].click()  
  
 drop.setSelected(i.material)  
  
 editList = inputList + [drop, fButton,  
 delButton, eContainer]  
  
 *# Delete references* del inputList  
 del drop  
 del fButton  
 del eContainer  
  
 editList[-1].startAnimation(**"centre"**, 0.25,  
 **"in"**)  
  
 *# Can only zoom in/out if no particles are being edited* elif event.type == MOUSEBUTTONDOWN and editingParticle is None:  
 *# Buttons 4 and 5 correspond to the scroll wheel up/down.  
 # These are used for changing the scale while still in  
 # the setup phase* if event.button == 4:  
 scale \*= 1.02  
  
 elif event.button == 5:  
 scale \*= 0.98  
  
 if event.type == KEYDOWN:  
 if event.key == K\_s:  
 *# Use s key (for 'scale') to change size of a particle  
 # using the mouse, rather than the input boxes* sizeChange(pRef, widgetList[5], widgetList[6], metres)  
  
 pRef = particles.sprites()[-1]  
  
 if widgetList[-1].isEmpty():  
 setting = False  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 updateParticle(pRef, widgetList)  
  
 if editingParticle is not None:  
 updateParticle(editingParticle, editList, True)  
  
 *# Finish editing if the particle is not in the sprite group (it  
 # has been deleted), and if the finishing process has not  
 # already started* if editingParticle not in particles.sprites():  
 endParticleEdit(editList)  
  
 *# Call scalePosition on all but the last sprite, as last sprite's  
 # position is determined by the location of the mouse* for sprite in particles.sprites()[:-1]:  
 sprite.scalePosition()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.updateDirection()  
 sprite.drawDirectionArrow()  
 pgkRoot.update()  
  
 scaleLength = metres \* scale  
  
 if scaleLength > scaler(200, **"x"**):  
 metres /= 10  
 scaleLength = metres \* scale  
 elif scaleLength < scaler(20, **"x"**):  
 metres \*= 10  
 scaleLength = metres \* scale  
  
 metres = roundToSigFig(metres, 1)  
  
 particleGraph.changeLabelGap(scaleLength, scaleLength)  
  
 scaleDisplay = pg.Rect(scaler(5, **"x"**), scaler(5, **"y"**), scaleLength,  
 scaler(5, **"y"**))  
  
 scaleDisplayText = fpsFont.render(**u"{0}m"**.format(str(metres)), True,  
 (0, 0, 0))  
 scaleTextRect = scaleDisplayText.get\_rect(topleft=(scaler(15, **"y"**),  
 scaler(5, **"x"**)))  
 screen.blit(scaleDisplayText, scaleTextRect)  
  
 pg.draw.rect(screen, (0, 0, 0), scaleDisplay)  
  
 fps = str(int(clock.get\_fps()))  
  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 try:  
 return main, (widgetList + editList,)  
 except TypeError:  
 return main, (widgetList,)

Controls the text that shows how many metres are represented by one square on the graph. If the squares get too small, then the graph will swap to displaying one square for 10m, and vice versa if they get too large

Calls the sizeChange function, allowing the user to change the size of a particle with their mouse.

When the user scrolls up, the scale becomes 102% of what it was before (to represent 1m, the program will use 102px instead of 100px, for example), and when they scroll down, the scale becomes 98% of what it was before. This multiplicative approach works much better than simply adding or subtracting a small amount. This is because, when the scale is increased, the amount that is being added to it gradually becomes smaller relative to the scale, making zooming in extremely slow at small scales. Also, zooming out will speed up exponentially as the amount being subtracted becomes larger and larger compared to the current scale, eventually causing a division by zero error as the scale reaches zero. This will not occur using the multiplicative method as it is impossible to reach zero by repeatedly multiplying or dividing a number, and the scale will be increasing by an amount that will remain proportional to the current scale, causing zooming to feel the same speed no matter how far you zoom in or out.

Returns a tuple made of both widgetList (the main setup widgets) and editList (the setup widgets used when editing a particle) for main to delete. Needs a try/except statement in case the user isn’t editing a particle when they click the begin simulation button, as in that case, editList would be a NoneType, which causes a TypeError when trying to add it to a list type.

def saveSetup(setupWidgets):  
 def saveToFile(saveWidgets, setupWidgets):  
 global saving  
 saveWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**), True)  
  
 fileName = **u"{0}.txt"**.format(saveWidgets[0].get())  
  
 saveData = [str(scale)]  
  
 *# Writing data from all particles to a list that will be written to  
 # the txt file* for p in particles.sprites()[:-1]:  
 if p.line:  
 line = True  
 else:  
 line = False  
 pData = [p.hasRandomVelocity, line, p.restCoefficient,  
 p.material, p.radius, p.density, p.mass, p.vol,  
 (p.velocity.x, p.velocity.y), p.colour,  
 (p.rect.x, p.rect.y), (p.acceleration.x, p.acceleration.y)]  
 saveData.append(str(pData))  
  
 with open(str(saveLocation / fileName), **"w"**) as f:  
 f.writelines(**'**\n**'**.join(saveData))  
  
 setupWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW - scaler(400, **"x"**))  
  
 saving = False  
  
 global saving  
  
 if len(particles) < 2:  
 return

To save a setup, the program creates a file that is titled [name of setup].txt. It then writes the scale of the simulation to the first line, and on every line after that, it writes data for a particle.

Exclude final particle: that is the dummy particle

takenNumbers = []  
 for file in os.listdir(saveLocation):  
 if file.endswith(**".txt"**) and file.lower().startswith(**"custom scenario"**):  
 fileNum = int(file[15:-4])  
 takenNumbers.append(fileNum)  
  
 takenNumbers.sort()  
  
 lowNum = 1  
 for num in takenNumbers:  
 if num == lowNum:  
 lowNum += 1  
 else:  
 break  
  
 setupWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**))  
  
 saveContainer = pgk.Container(pgkRoot, screen,  
 topright=(SW + scaler(400, **"x"**), 0),  
 outlineThickness=0, width=scaler(400, **"x"**),  
 height=scaler(205, **"y"**))  
  
 contWidth = scaler(400, **"x"**)  
 offset = scaler(275, **"x"**)  
 boxWidth = scaler(250, **"x"**)  
 inputList = [  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(5, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Scenario Name: "**,  
 width=boxWidth, allowSpecial=False, charLimit=35,  
 defaultEntry=**u"Custom Scenario {0}"**.format(str(lowNum)),  
 container=saveContainer),  
 ]  
  
 inputList += [  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**), scaler(55, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Save"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: saveToFile(saveWidgets, setupWidgets),  
 container=saveContainer, swellOnHover=True),  
 ]  
  
 saveWidgets = inputList + [saveContainer]  
  
 *# Remove references - Collected by garbage collection* del saveContainer  
 del inputList  
  
 saving = True  
 while saving:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 saving = False  
 pg.quit()  
 quit()  
  
 if setupWidgets[-1].animationDone():  
 saveWidgets[-1].startAnimation(**"horizontalslide"**, 0.25, **"out"**,  
 SW - contWidth)  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 *# Exclude final particle - the one that was following the mouse  
 # pointer when save button was pressed* for i in particles.sprites()[:-1]:  
 i.draw()  
 i.drawDirectionArrow()  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 return

Creating widgets used by the save loop

A scenario’s default name will be ‘custom scenario x’ where x is a unique number, ensuring that the user will not accidentally overwrite a save if they don’t give theirs a name. The program first goes through the list of files in the save directory using os.listDir(saveLocation) to find out which numbers are used, before sorting the used numbers and finding the lowest unused one.

def loadSetup(widgets):  
 def loadFromFile(loadWidgets, widgets):  
 global scale  
 global previousScale  
 global loading  
 global particleGraph  
 loadWidgets[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
  
 fileName = loadWidgets[0].get()  
  
 with open(str(saveLocation / fileName), **"r"**) as f:  
 data = f.readlines()  
  
 *# Remove newline characters from lines* newData = []  
 for line in data:  
 *# ast.literal\_eval reads the contents of the file, and evaluates the  
 # string as a python expression - in this case a list* newData.append(ast.literal\_eval(line.rstrip(**"**\n**"**)))  
  
 scale = newData[0]  
 previousScale = scale  
 particleGraph = Graph(screen, SW, SH, (0, 0), BG\_COLOUR, scale, scale)  
 for p in newData[1:]:  
 particles.add(Particle(p[2], p[3], p[4], p[5], p[8], p[9], p[10],  
 p[11][0], p[11][1]))  
  
 particles.sprites()[-1].hasRandomVelocity = p[0]  
 if p[1]:  
 particles.sprites()[-1].line = Line(screen, particleGraph, p[9])  
 else:  
 particles.sprites()[-1].line = None  
 particles.sprites()[-1].mass = p[6]  
 particles.sprites()[-1].vol = p[7]  
  
 if widgets:  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW - scaler(400, **"x"**))  
  
 particles.add(Particle(1, **"Custom Material 1 - 1.0kgm^-3"**,  
 roundToSigFig((SW / 4) / scale, 3), 1,  
 (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])), 0, xA=0))  
  
 loading = False  
  
 global loading  
  
 for particle in particles.sprites():  
 particle.delete()  
  
 if widgets:  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**))  
  
 loadContainer = pgk.Container(pgkRoot, screen, maskColour=BG\_COLOUR,  
 centre=(SW / 2, SH / 2),  
 outlineThickness=0, width=scaler(400, **"x"**),  
 height=scaler(205, **"y"**), startVisible=False)  
  
 options = os.listdir(saveLocation)  
 boxWidth = scaler(350, **"x"**)  
 *# Define inputList as an empty list initially, as it needs to be  
 # referenced by the button, which cannot be done if inputList and the  
 # button are created at the same time.* inputList = [  
 pgk.Dropdown(pgkRoot, screen, scaler(25, **"x"**), scaler(5, **"y"**),  
 options, font=SMALL\_FONT, bgColour=(222, 222, 222),  
 width=boxWidth),  
 ]  
  
 inputList += [  
 pgk.Button(pgkRoot, screen, scaler(25, **"x"**), scaler(55, **"y"**),  
 font=MID\_FONT, bgColour=(33, 33, 33), text=**"Load Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(350, **"x"**),  
 action=lambda: loadFromFile(loadWidgets, widgets),  
 container=loadContainer, swellOnHover=True),  
 ]  
  
 *# Add dropdown to container last as it needs to be drawn over the button* inputList[0].config(container=loadContainer)  
  
 loadWidgets = inputList + [loadContainer]  
  
 *# Remove references - Collected by garbage collection* del loadContainer  
 del inputList  
  
 if not widgets:  
 loadWidgets[-1].startAnimation(**"centre"**, 0.25, **"in"**)  
  
 for particle in particles.sprites():  
 particle.delete()  
  
 loading = True  
 while loading:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 loading = False  
 pg.quit()  
 quit()  
  
 if widgets and widgets[-1].animationDone():  
 loadWidgets[-1].startAnimation(**"centre"**, 0.25, **"in"**)  
  
 screen.fill(BG\_COLOUR)  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 *# If widgets is None, that means the program got to this page from the  
 # main menu, and therefore needs to move onto setup. If widgets exists,  
 # however, this function was called from within setup, and we just need a  
 # simple return statement* if not widgets:  
 return setup, (1,)  
 else:  
 return

Creating Widgets

To load a setup from a file, the filename is retrieved from the dropdown menu, and then that file is opened in read mode. Firstly, the scale is set to the first line of that file, and then the program iterates through the rest of the lines, instantiating a particle for each line, with its respective properties.

Procedure for changing the size of a particle – called when the user presses ‘s’ during the setup phase.

def sizeChange(particle, radBox, massBox, metres):  
 changing = True  
  
 *# Need to duplicate SCALE\_TOOL\_IMG as it needs to be modified with the  
 # rotozoom method of pygame's image class. Using the original image will  
 # mean that I will have to reload the image every time it is needed in  
 # order to get the original, unmodified one.* sizeArrow = SCALE\_TOOL\_IMG  
  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
  
 while changing:  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
 if pgkRoot.eventHandler(event):  
 return  
  
 elif event.type == MOUSEBUTTONDOWN:  
 if event.button == 1:  
 *# When the user initially presses the LMB,  
 # set startDistance equal to the distance between the  
 # mouse and the particle centre, and set startRad equal  
 # to the particle's current radius.* startDistance = absoluteDistance(particle.pos,  
 pgmath.Vector2(  
 pg.mouse.get\_pos()))  
 startRad = roundToSigFig(float(radBox.get()), 3)  
  
 elif event.type == KEYDOWN:  
 *# if ... in ... statement allows the user to use either the  
 # main enter key, or the enter key on the numpad.* if event.key in [K\_RETURN, K\_KP\_ENTER]:  
 return  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()

for sprite in particles.sprites():  
 sprite.draw()  
 sprite.drawDirectionArrow()  
  
 pg.mouse.set\_visible(True)  
  
 *# If the user is holding down the LMB* if pg.mouse.get\_pressed()[0]:  
 pg.mouse.set\_visible(False)  
  
 *# Draw dotted line from centre of particle to mouse pointer,* drawDottedLine(particle.pos, pg.mouse.get\_pos())  
  
 xDiff = pg.mouse.get\_pos()[0] - particle.pos.x  
 yDiff = pg.mouse.get\_pos()[1] - particle.pos.y  
  
 *# Calculating angle by which to rotate the arrow* dir = (math.atan2(yDiff, xDiff) \* -1) + math.pi / 2  
 blitArrow = pg.transform.rotate(sizeArrow, math.degrees(dir))  
  
 arrowRect = blitArrow.get\_rect(center=pg.mouse.get\_pos())  
  
 *# Blit arrow on screen in the position of the mouse pointer,  
 # to create the illusion that the pointer has changed to the arrow.* screen.blit(blitArrow, arrowRect)  
  
 posVector = pgmath.Vector2(pg.mouse.get\_pos())  
 currentDistance = absoluteDistance(particle.pos, posVector)  
 diff = currentDistance - startDistance  
  
 minRad = roundToSigFig(scaler(10, **"x"**) / scale, 3)  
 maxRad = roundToSigFig((SW / 4) / scale, 3)  
  
 changeFactor = roundToSigFig(0.5 / scale, 3)  
  
 if minRad <= startRad + (changeFactor \* diff) <= maxRad:  
 *# Equation to calculate new size of particle, based on  
 # distance that the mouse has moved away from the particle's  
 # centre (diff)* newRad = startRad + (changeFactor \* diff)  
  
 *# Update dimensions of particle and write the new dimensions  
 # to the input boxes.* particles.sprites()[-1].updateDimension(rad=newRad)  
 radBox.write(str(roundToSigFig(newRad, 3)))  
 massBox.write(str(particles.sprites()[-1].mass))  
  
 pgkRoot.update()  
  
 scaleLength = metres \* scale  
  
 scaleDisplay = pg.Rect(scaler(5, **"x"**), scaler(5, **"y"**), scaleLength,  
 scaler(5, **"y"**))  
  
 scaleDisplayText = fpsFont.render(**u"{0}m"**.format(str(metres)), True,  
 (0, 0, 0))  
 scaleTextRect = scaleDisplayText.get\_rect(topleft=(scaler(15, **"y"**),  
 scaler(5, **"x"**)))  
 screen.blit(scaleDisplayText, scaleTextRect)  
  
 pg.draw.rect(screen, (0, 0, 0), scaleDisplay)  
  
 fps = str(int(clock.get\_fps()))  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()

To change the size of the particle, the difference between the start position of the mouse and the current position of the mouse is calculated. This is done using the difference between the radii of the circles at the mouse’s position

Once this difference is calculated, it is multiplied by changeFactor to find the value by which the circle’s radius should change (while remaining within the bounds of the maximum and minimum radius. After that, the circle’s mass is updated to maintain correct density.

Smallest arrow is startDistance, largest arrow is currentDistance. The difference between them is diff

This checks if the LMB is currently pressed. I have put this outside of the events check as there is no event detected if the user is simply holding down the LMB.

A picture containing text, screenshot

Description automatically generateddef createMaterial(widgetList):  
 global materialTimer  
  
 def startExit(widgets, colour):  
 global customMaterials  
 global sortedCustoms  
 name = widgets[4].get()  
 density = widgets[3].get()  
 newMaterial = **u""" "{0} - {1}kgm^-3": [{2}, {3}],"""**.format(name,  
 density,  
 density,  
 colour)  
  
 with open(**"customMaterials.txt"**, **"r+"**) as file:  
 lines = file.readlines()[:-1]  
 lines += [newMaterial + **"**\n**"**, **"}"**]  
 *# file.truncate(0) clears the file, which is needed as otherwise  
 # lines will be added on to the end of the file, effectively  
 # duplicating everything and ruining the formatting of the  
 # dictionary* file.truncate(0)  
  
 *# Seek after truncate prevents null bytes being inserted - when  
 # truncate is used, the file tries to write from the same memory  
 # location as it was before truncation, resulting in null bytes  
 # being inserted. Seek(0) moves to the start of the file  
 # preventing null bytes* file.seek(0)  
 file.writelines(lines)  
  
 with open(**"customMaterials.txt"**, **"r"**) as file:  
 contents = file.read()  
  
 customMaterials = ast.literal\_eval(contents)  
 sortedCustoms = sorted(customMaterials)  
  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.25, **"out"**, SW,  
 deleteAfter=True)  
  
 def randomiseColour(widgets):  
 widgets[5].write(str(random.randint(0, 255)))  
 widgets[6].write(str(random.randint(0, 255)))  
 widgets[7].write(str(random.randint(0, 255)))  
  
 widgetList[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**))  
  
 materialContainer = pgk.Container(pgkRoot, screen,  
 topleft=(SW, 0), outlineThickness=0,  
 width=scaler(400, **"x"**),  
 height=scaler(520, **"y"**))  
  
 contWidth = scaler(400, **"x"**)  
 offset = scaler(150, **"x"**)  
 boxWidth = scaler(125, **"x"**)  
 rgbOffset = scaler(275, **"x"**)  
 rgbWidth = scaler(50, **"x"**)  
  
 *# A unique name will be automatically generated for each new material  
 # created, in the form of "Custom Material <number>"  
 # This loop decides what number that will be, based on how many materials  
 # are already named in that form* customNum = 1  
 for i in sortedCustoms:  
 if i.lower().startswith(**"custom material"**):  
 customNum += 1

Inside the text file

Due to the fact that the materials text file will be formatted like a python dictionary, I will need to insert the material being saved before the final curly bracket. As there is no way of simply removing the final line from the file (from within python), I had to use python’s truncate method to clear the file, before rewriting the entire dictionary to it.

Initially, I did not have file.seek(0) after truncation. This caused a bug, in that the custom materials .txt file became full of null bytes before a material, which broke the material selection. After some investigating, I discovered that python’s truncate method does not change the position within the file, meaning that it attempted to rewrite the dictionary at the position that the dictionary originally ended at, resulting in the insertion of null bytes before.

createWidgets = [  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(5, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Desired radius (m) (maximum 10m):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"1"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(55, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Desired Volume (m^3) (maximum 33.5m^3):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"1"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(105, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Desired mass (kg):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"1"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(155, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Density (kgm^-3):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(205, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Material name:"**,  
 width=boxWidth, charLimit=17,  
 defaultEntry=**"Custom Material "** + str(customNum),  
 container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - rgbOffset, scaler(255, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Material color: (R)"**,  
 width=rgbWidth, allowLetters=False, allowMaths=False,  
 allowSpecial=False, allowSpace=False,  
 charLimit=3, defaultEntry=**"126"**,  
 container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - rgbOffset + rgbWidth \* 2,  
 scaler(255, **"y"**), font=SMALL\_FONT,  
 bgColour=(222, 222, 222), inlineText=**"(G)"**,  
 width=rgbWidth, allowLetters=False, allowMaths=False,  
 allowSpecial=False, allowSpace=False,  
 charLimit=3, defaultEntry=**"25"**,  
 container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - rgbOffset + rgbWidth \* 4,  
 scaler(255, **"y"**), font=SMALL\_FONT,  
 bgColour=(222, 222, 222), inlineText=**"(B)"**,  
 width=rgbWidth, allowLetters=False, allowMaths=False,  
 allowSpecial=False, allowSpace=False,  
 charLimit=3, defaultEntry=**"27"**,  
 container=materialContainer),  
 ]  
  
 buttonGap = scaler(50, **"y"**) + createWidgets[0].getHeight()  
  
 randomButton = pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(305, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Randomise Colour"**,  
 height=createWidgets[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: randomiseColour(createWidgets),  
 container=materialContainer, swellOnHover=True)  
  
 doneButton = pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(305, **"y"**) + buttonGap, font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Finish and Save"**,  
 height=createWidgets[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: startExit(createWidgets,  
 materialColour),  
 container=materialContainer, swellOnHover=True)  
  
 *# Cannot be added to the list immediately as they rely on the list for  
 # their height attribute* createWidgets += [randomButton, doneButton, materialContainer]  
  
 *# Remove references* del materialContainer  
 del randomButton  
 del doneButton  
  
 *# Displays the current colour that the user has chosen* rgbTestRect = pg.Rect((int(SW - contWidth - scaler(60, **"x"**)),  
 int(scaler(255, **"y"**))),  
 (rgbWidth, createWidgets[0].getHeight()))  
  
 previousRad = createWidgets[0].get()  
 previousVol = None  
 previousMass = createWidgets[2].get()  
  
 isCreating = False  
  
 changing = True  
  
 while changing:  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
 if pgkRoot.eventHandler(event):  
 pass  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.drawDirectionArrow()  
 *# If animation has finished for the creation container, length will  
 # be 0 as all widgets will have been deleted* if createWidgets[-1].isEmpty():  
 del createWidgets  
  
 widgetList[-7].config(options=sortedCustoms + MATERIALS\_SORTED)  
 widgetList[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW - contWidth)  
 return  
  
 *# isCreating is used to ensure that this only runs once* if widgetList[-1].animationDone() and not isCreating:  
 *# Need to use a slide out animation, rather than in.  
 # Otherwise the container will be displayed in its final  
 # position until the animation starts - it needs to be  
 # offscreen until it starts.* createWidgets[-1].startAnimation(**"horizontalslide"**, 0.25,  
 **"out"**, SW - contWidth)  
 isCreating = True  
  
 *# Need to except ValueError here in case the user deletes all  
 # characters (calling int() on an empty string throws a ValueError)* try:  
 for i in [createWidgets[5], createWidgets[6], createWidgets[7]]:  
 if int(i.get()) > 255:  
 i.write(**"255"**)  
  
 r = int(createWidgets[5].get())  
 g = int(createWidgets[6].get())  
 b = int(createWidgets[7].get())  
  
 materialColour = (r, g, b)  
 except ValueError:  
 pass  
  
 try:  
 if createWidgets[0].get() != previousRad:  
 previousRad = createWidgets[0].get()  
  
 rad = roundToSigFig(float(createWidgets[0].get()), 3)  
 vol = roundToSigFig((4 / 3) \* math.pi \* rad \*\* 3, 3)  
  
 createWidgets[1].write(str(vol))  
 createWidgets[3].write(  
 str(roundToSigFig(float(createWidgets[2].get()) /  
 vol, 3)))  
  
 if createWidgets[1].get() != previousVol:  
 previousVol = createWidgets[1].get()  
  
 vol = roundToSigFig(float(createWidgets[1].get()), 3)  
 rad = roundToSigFig(((3 \* vol) / (4 \* math.pi)) \*\* (1 / 3), 3)  
 previousRad = str(rad)  
 createWidgets[0].write(str(rad))  
 createWidgets[3].write(  
 str(roundToSigFig(float(createWidgets[2].get()) /  
 vol, 3)))  
  
 if createWidgets[2].get() != previousMass:  
 previousMass = createWidgets[3].get()  
 createWidgets[3].write(  
 str(roundToSigFig(float(createWidgets[2].get()) /  
 float(createWidgets[1].get()),  
 3)))  
  
 *# Need to catch both errors here, as the inputs involve dividing  
 # by user input, meaning they may end up dividing by zero  
 # Also ValueError in case the user deletes everything in one input box* except (ValueError, ZeroDivisionError):  
 pass  
  
 pgkRoot.update()  
  
 fps = str(int(clock.get\_fps()))  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
 pg.draw.rect(screen, materialColour, rgbTestRect)  
  
 pg.display.update()  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
  
def main(widgetList):  
 global mainprogram  
 global tNow  
 global frameNumber  
 global timeMultiplier  
 global timeShown  
 timeShown = False  
  
 def showTimeControls(timeContainer):  
 global timeShown  
 if not timeShown:  
 timeContainer.startAnimation(**"verticalslide"**, 0.25, **"out"**,  
 SH - scaler(100, **"y"**))  
 timeShown = True  
 else:  
 timeContainer.startAnimation(**"verticalslide"**, 0.25, **"out"**, SH)  
 timeShown = False  
  
 *# Delete particle that gets placed on button press* particles.remove(particles.sprites()[-1])  
  
 timeWidgets = [  
 pgk.Container(pgkRoot, screen, centre=(SW / 2, SH + scaler(50, **"y"**)),  
 width=scaler(450, **"x"**), height=scaler(100, **"y"**))  
 ]  
  
 contWidth = scaler(450, **"x"**)  
 contHeight = scaler(100, **"y"**)  
  
 timeWidgets += [  
 pgk.Button(pgkRoot, screen, contWidth / 2 - scaler(40, **"x"**),  
 contHeight - scaler(145, **"y"**), height=scaler(40, **"y"**),  
 width=scaler(80, **"x"**),  
 action=lambda: showTimeControls(timeWidgets[0]),  
 image=TT\_IMG, hoverImage=H\_TT\_IMG,  
 container=timeWidgets[0]),  
 pgk.Button(pgkRoot, screen, contWidth / 2 - scaler(75, **"x"**),  
 contHeight - scaler(105, **"y"**), height=scaler(100, **"y"**),  
 width=scaler(150, **"x"**),  
 action=lambda: pauseMenu(timeWidgets),  
 image=PAUSE\_IMG, hoverImage=H\_PAUSE\_IMG,  
 container=timeWidgets[0]),  
 pgk.Button(pgkRoot, screen, contWidth / 2 - scaler(175, **"x"**),  
 contHeight - scaler(105, **"y"**), height=scaler(100, **"y"**),  
 width=scaler(100, **"x"**), action=lambda: timeChange(-1),  
 image=RW\_IMG, hoverImage=H\_RW\_IMG,  
 container=timeWidgets[0]),  
 pgk.Button(pgkRoot, screen, contWidth / 2 + scaler(75, **"x"**),  
 contHeight - scaler(105, **"y"**), height=scaler(100, **"y"**),  
 width=scaler(100, **"x"**), action=lambda: timeChange(1),  
 image=FF\_IMG, hoverImage=H\_FF\_IMG,  
 container=timeWidgets[0]),  
 ]  
  
 for widget in widgetList:  
 widget.delete()  
 del widget  
 *# Remove references to widgets - will get collected by Python's  
 # garbage collection* mainprogram = True  
  
 while mainprogram:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == KEYDOWN:  
 if event.key == K\_ESCAPE:  
 pauseMenu(timeWidgets)  
  
 if event.type == QUIT:  
 mainprogram = False  
 pg.quit()  
 quit()  
  
 frameNumber += TIME\_SCALES[currentTimescale]  
 if frameNumber < 1:  
 frameNumber = 1  
 try:  
 *# If statement prevents 'jumping' of particles when user moves  
 # the window* if time.time() - previousFrame < 0.1:  
 timeMultiplier = (time.time() - previousFrame)  
 timeMultiplier \*= TIME\_SCALES[currentTimescale]  
 *# Calculates time between frames* except NameError:  
 *# Will occur on the first frame, as there is no previous frame* pass  
  
 previousFrame = time.time()  
  
 if tNow <= 0 and currentTimescale < 3:  
 pauseMenu(timeWidgets)  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
 particles.update()  
  
 if timeMultiplier > 0:  
 tNow += timeMultiplier  
  
 *# Set up text that shows current time* tDisplay = round(tNow, 4)  
 timeFont = pg.font.SysFont(MID\_FONT[0], MID\_FONT[1])  
 timeText = timeFont.render(**"Time: T+"** + str(tDisplay), True, (0, 0, 0))  
 tRect = timeText.get\_rect(topleft=(10, 10))  
  
 timescaleFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 tscaleText = timescaleFont.render(**"Time Multiplier: x"** +  
 str(TIME\_SCALES[currentTimescale]),  
 True, (0, 0, 0))  
 tscaleRect = tscaleText.get\_rect(topleft=(scaler(10, **"x"**),  
 scaler(50, **"y"**)))  
  
 screen.blit(timeText, tRect)  
 screen.blit(tscaleText, tscaleRect)  
  
 pgkRoot.update()  
  
 fps = str(int(clock.get\_fps()))  
  
 *# Create text that shows the fps that the program is running at* fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
  
 clock.tick()  
  
 for i in particles.sprites():  
 i.delete()  
 return mainMenu, (1,)

Instantiating widgets used by material creation

If the user inputs a value for one of radius, volume or mass, then this calculates a new value for density (and a new value for radius/volume if one of them changes as they are both dependent on one another

Starts the animation to show/hide the time controls

Instantiates the time controls

timeMultiplier (time since last frame) is multiplied by the timescale, so that it will increase at double real time when fast forwarding, and half of real time when in slow motion

If the current simulation time is zero or lower while the program is rewinding, then the program will automatically pause, to prevent time from going negative.

If the program is moving forward in time, then the time since the start of the simulation is incremented by the time since the last frame

*# Runs when user presses esc*def pauseMenu(timeWidgets):  
 *# returnToMain ends both the pause loop and the main loop, so that the  
 # program will return back to the main menu* def returnToMain(menuWidgets, timeWidgets):  
 global paused  
 global mainprogram  
 paused = False  
 mainprogram = False  
  
 *# Get rid of the time controls* timeWidgets[0].startAnimation(**"verticalslide"**, 0.25, **"out"**, SH, True)  
  
 *# Will be true if timeWidgets has not been expanded* if timeWidgets[0].getRect().centery > SH:  
 for i in timeWidgets[1:]:  
 i.delete()  
 timeWidgets[0].delete()  
 del timeWidgets  
  
 *# Get rid of the menu* menuWidgets[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
  
 def exitPause(menuWidgets, timeWidgets):  
 *# Simply ends the pause loop, and starts the disappearing animation  
 # for the menu* global paused  
 paused = False  
 global currentTimescale  
 *# Change play button into a pause button* timeWidgets[2].config(action=lambda: pauseMenu(timeWidgets),  
 image=PAUSE\_IMG, hoverImage=H\_PAUSE\_IMG)  
  
 menuWidgets[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
 *# If current time is earlier or equal to the time that the simulation  
 # started, set timescale to 1x, as the user shouldn't be able to rewind  
 # to earlier than the beginning of the sim* if tNow <= 0:  
 currentTimescale = 4  
  
 def exitProgram():  
 pg.quit()  
 quit()  
  
 def hideStats(statList):  
 *# Menu for showing stats will disappear* statList[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
  
 global paused  
  
 pauseContainer = pgk.Container(pgkRoot, screen,  
 centre=(SW / 2, SH / 2),  
 bg=True, bgColour=BG\_COLOUR,  
 maskColour=BG\_COLOUR, outlineThickness=3,  
 outlineColour=(33, 33, 33),  
 width=scaler(345, **"x"**),  
 height=scaler(125, **"y"**), startVisible=False)  
  
 contWidth = scaler(400, **"x"**)  
 gap = scaler(55, **"y"**)  
  
 *# Need to initially create menuWidgets as an empty list, so that the  
 # buttons can reference it when passing arguments to their functions* menuWidgets = []  
 statList = []  
  
 menuWidgets += [  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(390, **"x"**), scaler(10, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Main Menu"**,  
 height=scaler(50, **"y"**),  
 width=scaler(325, **"x"**),  
 action=lambda: returnToMain(menuWidgets, timeWidgets),  
 container=pauseContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(390, **"x"**), gap + scaler(10, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Exit Program"**,  
 height=scaler(50, **"y"**),  
 width=scaler(325, **"x"**), action=exitProgram,  
 container=pauseContainer, swellOnHover=True)  
 ]  
  
 menuWidgets += [pauseContainer]  
  
 del pauseContainer  
  
 menuWidgets[-1].startAnimation(**"centre"**, 0.25, **"in"**)  
  
 *# Change pause button into a play button* timeWidgets[2].config(action=lambda: exitPause(menuWidgets, timeWidgets),  
 image=PLAY\_IMG,  
 hoverImage=H\_PLAY\_IMG)  
 pausedFont = pg.font.SysFont(LARGE\_FONT[0], LARGE\_FONT[1])  
 pausedText = pausedFont.render(**"PAUSED"**, True, (0, 0, 0))  
 pRect = pausedText.get\_rect(center=(int(SW / 2), int(scaler(380, **"y"**))))  
  
 tDisplay = round(tNow, 4)  
 timeFont = pg.font.SysFont(MID\_FONT[0], MID\_FONT[1])  
 timeText = timeFont.render(**"Time: T+"** + str(tDisplay), True, (0, 0, 0))  
 tRect = timeText.get\_rect(topleft=(scaler(10, **"x"**), scaler(10, **"y"**)))  
  
 *# The pause loop is just an empty loop - only showing the UI elements  
 # such as fps text, paused text, and time text* paused = True  
 while paused:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == KEYDOWN:  
 if event.key == K\_ESCAPE:  
 exitPause(menuWidgets, timeWidgets)  
  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 3:  
 *# Showing particle stats* [...]  
  
 if event.type == QUIT:  
 paused = False  
 pg.quit()  
 quit()  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.drawDirectionArrow()  
  
 pgkRoot.update()  
  
 timescaleFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 tscaleText = timescaleFont.render(**"Time Multiplier: x"** +  
 str(TIME\_SCALES[currentTimescale]),  
 True, (0, 0, 0))  
 tscaleRect = tscaleText.get\_rect(topleft=(scaler(10, **"x"**),  
 scaler(50, **"y"**)))  
  
 screen.blit(timeText, tRect)  
 screen.blit(tscaleText, tscaleRect)  
 screen.blit(pausedText, pRect)  
  
 fps = str(int(clock.get\_fps()))  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
  
*# noinspection PyUnboundLocalVariable*

Functions used by pause menu buttons:

**returnToMain:** starts an animation to hide the time controls and the pause menu, and sets them both to delete after their animation is complete

**exitPause:** Simply hides the pause menu, and allows the simulation to resume.

**exitProgram:** Closes the entire program.

This code is the same as the code for editing a particle after it has been placed in setup(), except I have set the canUse attribute of all the input boxes to False to prevent the user from typing into them. This allows them to pause the program at any time and check the properties of particles.

if \_\_name\_\_ == **"\_\_main\_\_"**: *# If program is run as a script, this will run* pg.init()  
 pg.font.init()  
  
  
 *# scales sizes relative to screen width/height - gives a consistent feel  
 # across all devices* def scaler(toscale, axis):  
 if axis == **'x'**:  
 return int(toscale \* (SW / 1920))  
 else:  
 return int(toscale \* (SH / 1080))  
  
  
 *# Does the opposite of scaler - needed when saving scenarios, as the  
 # values will need to be rescaled when the scenario is loaded* def descaler(toscale, axis):  
 if axis == **"x"**:  
 return int(toscale \* (1920 / SW))  
 else:  
 return int(toscale \* (1080 / SH))  
  
  
 if win:  
 SW = GetSystemMetrics(0)  
 SH = GetSystemMetrics(1)  
 else:  
 SW = 1920  
 SH = 1080  
  
 BG\_COLOUR = (244, 244, 244)  
  
 scale = scaler(100, **"x"**)  
 previousScale = scale  
  
 *# Fullscreen doesn't work with pg.mouse.set\_visible() (mouse gets  
 # centred every time function is called), so I am using a borderless  
 # window that is the same size as the screen instead. os.environ  
 # positions the window in the top left corner of the screen* os.environ[**'SDL\_VIDEO\_WINDOW\_POS'**] = **"0,0"** *# Display flags: NOFRAME to simulate a fullscreen display, and DOUBLEBUF  
 # to prevent flickering - it also slightly improves fps* screen = pg.display.set\_mode((SW, SH), NOFRAME | DOUBLEBUF)  
 screen.set\_alpha(None)  
  
 particleGraph = Graph(screen, SW, SH, (0, 0), BG\_COLOUR, scale, scale)  
  
 *# Read a dictionary in from a file* with open(**"materials.txt"**, **"r"**) as file:  
 contents = file.read()  
  
 *# ast.literal\_eval reads the contents of the file, and evaluates the  
 # string as a python expression - in this case a dictionary* MATERIALS = ast.literal\_eval(contents)  
 MATERIALS\_SORTED = sorted(MATERIALS)  
  
 with open(**"customMaterials.txt"**, **"r"**) as file:  
 contents = file.read()  
  
 customMaterials = ast.literal\_eval(contents)  
 sortedCustoms = sorted(customMaterials)  
  
 timeMultiplier = 1 / 60 *# Initial value for time between frames* TIME\_SCALES = [-2, -1, -0.5, 0.5, 1, 2]  
 currentTimescale = 4  
 tNow = 0  
 frameNumber = 0  
  
 imagesFolder = Path(**"resources/images/"**)  
 saveLocation = Path(**"Saved Scenarios/"**)  
  
 *# Used for drawDirectionArrow method* ARROW\_IMAGE = pg.image.load(  
 str(imagesFolder / **"arrow.png"**)).convert\_alpha()  
  
 SCALE\_TOOL\_IMG = pg.image.load(  
 str(imagesFolder / **"resizeCursor.png"**)).convert\_alpha()  
  
 PAUSE\_IMG = pg.image.load(  
 str(imagesFolder / **"pausedNormal.png"**)).convert\_alpha()  
 H\_PAUSE\_IMG = pg.image.load(  
 str(imagesFolder / **"pausedHovered.png"**)).convert\_alpha()  
  
 PLAY\_IMG = pg.image.load(  
 str(imagesFolder / **"playNormal.png"**)).convert\_alpha()  
 H\_PLAY\_IMG = pg.image.load(  
 str(imagesFolder / **"playHovered.png"**)).convert\_alpha()  
  
 FF\_IMG = pg.image.load(  
 str(imagesFolder / **"ffNormal.png"**)).convert\_alpha()  
 H\_FF\_IMG = pg.image.load(  
 str(imagesFolder / **"ffHovered.png"**)).convert\_alpha()  
  
 RW\_IMG = pg.transform.flip(FF\_IMG, True, False)  
 H\_RW\_IMG = pg.transform.flip(H\_FF\_IMG, True, False)  
  
 TT\_IMG = pg.image.load(  
 str(imagesFolder / **"timeTabNormal.png"**)).convert\_alpha()  
 H\_TT\_IMG = pg.image.load(  
 str(imagesFolder / **"timeTabHovered.png"**)).convert\_alpha()  
  
 PREV\_IMG = pg.image.load(  
 str(imagesFolder / **"prevPage.png"**)).convert\_alpha()  
 NEXT\_IMG = pg.image.load(  
 str(imagesFolder / **"nextPage.png"**)).convert\_alpha()  
  
 R\_MENU\_IMG = pg.image.load(  
 str(imagesFolder / **"rMainMenu.png"**)).convert\_alpha()  
 L\_MENU\_IMG = pg.image.load(  
 str(imagesFolder / **"lMainMenu.png"**)).convert\_alpha()  
  
 if SW / 1920 > SH / 1080:  
 *# Scales font sizes relative to whichever axis has been 'scaled  
 # down' more, to prevent text being larger than the widget it is in* SMALLER\_FONT = (**"Helvetica"**, scaler(12, **"y"**))  
 SMALL\_FONT = (**"Helvetica"**, scaler(18, **"y"**))  
 MID\_FONT = (**"Helvetica"**, scaler(30, **"y"**))  
 LARGE\_FONT = (**"Helvetica"**, scaler(72, **"y"**))  
  
 else:  
 SMALLER\_FONT = (**"Helvetica"**, scaler(12, **"x"**))  
 SMALL\_FONT = (**"Helvetica"**, scaler(18, **"x"**))  
 MID\_FONT = (**"Helvetica"**, scaler(30, **"x"**))  
 LARGE\_FONT = (**"Helvetica"**, scaler(72, **"x"**))  
  
 mainmenu = False  
 setting = False  
 mainprogram = False  
 paused = False  
 mainWidgets = []  
  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3'**)  
 clock = pg.time.Clock()  
 particles = pg.sprite.Group()  
  
 nextFunction, args = mainMenu(1)  
 *# Prevents recursion (For example, mainMenu would be called  
 # from within main, which would be called from within setup, which would  
 # be called from within mainMenu, and so on) which would* while True:  
 nextFunction, args = nextFunction(\*args)

As mentioned earlier, Path converts a filepath to work on whatever OS you are running the program on (Windows uses forward slashes, MacOS uses backward slashes). I have used this to set up a few paths to folders that are used throughout the program. I can then select whichever file I need just by adding that file name onto the end of this folder path with a slash and converting that to a string

os.environ[**'SDL\_VIDEO\_WINDOW\_POS'**] = **"0,0"** Sets the position of the window to the upper left corner of the screen. NOFRAME flag in the Pygame display call removes the border on the window. These combined with the fact that the window is set to fill the screen creates an appearance of the program running fullscreen. I did this as Pygame’s native fullscreen support doesn’t work well with pg.mouse.set\_visible() – it will centre the mouse every time a function is called.

Retrieves width and height of the display port.

### Testing

#### Regression Testing – Testing against V1 & V2 criteria

**Table

Description automatically generatedTests 1.1, 1.4 & 2.1**

****

5ms^-1

**All of these tests passed without issue**. As is shown in the diagram, the particle is placed on top of the mouse pointer, and the direction arrow lines up with the velocities of the particle. You can also see that I was able to type my desired velocities into the input boxes

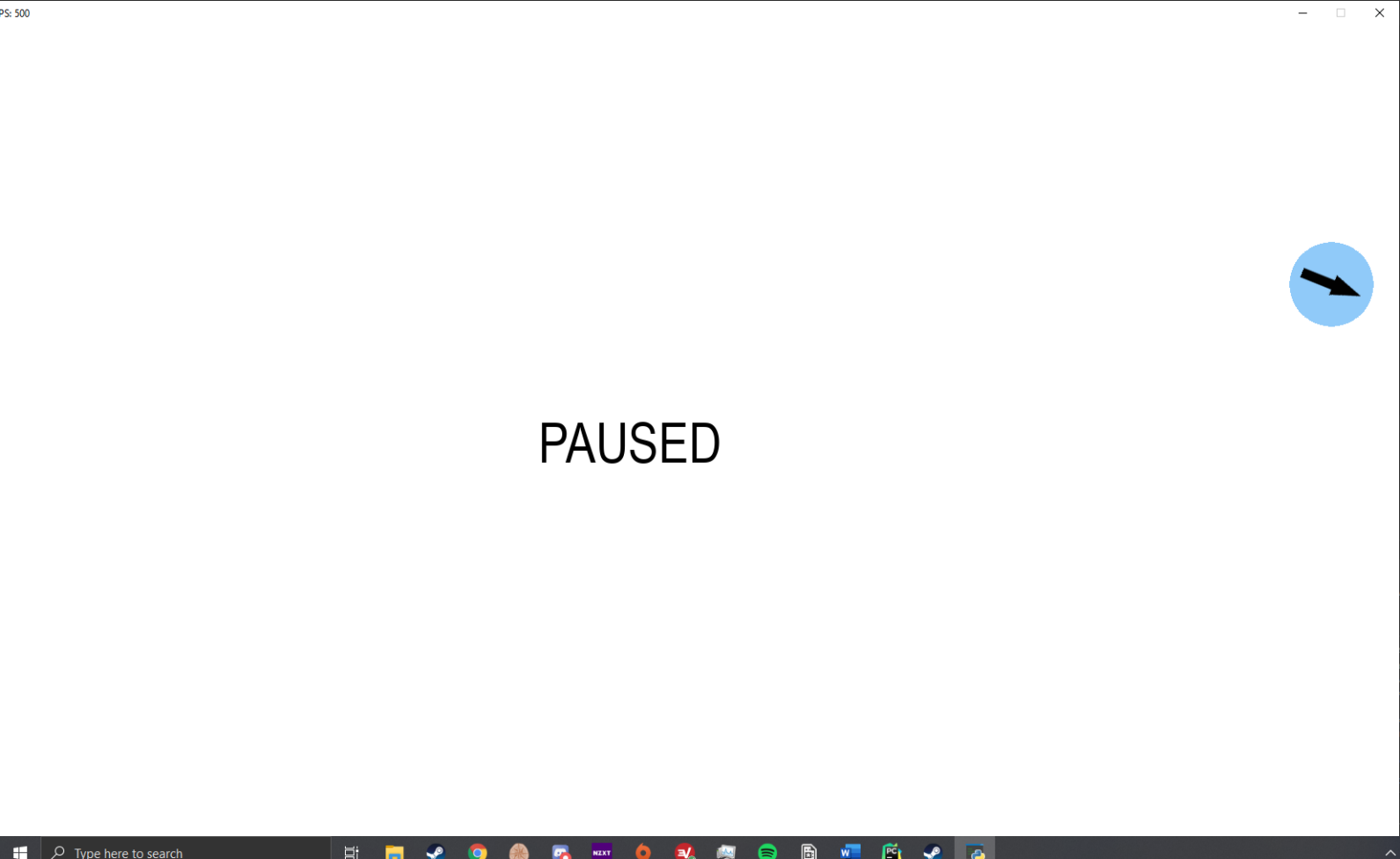
10ms^-1

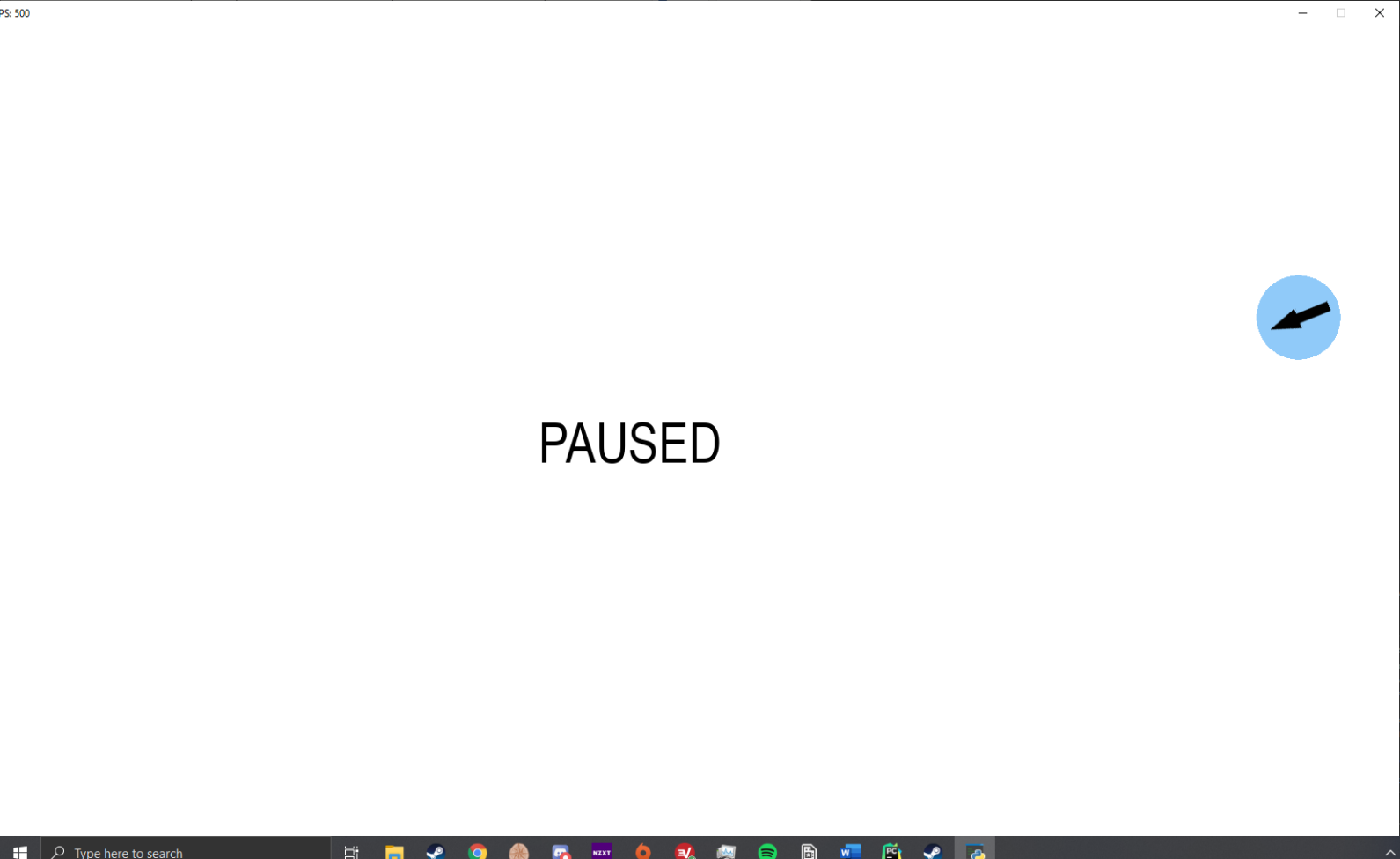
**Test 1.2**

Chart

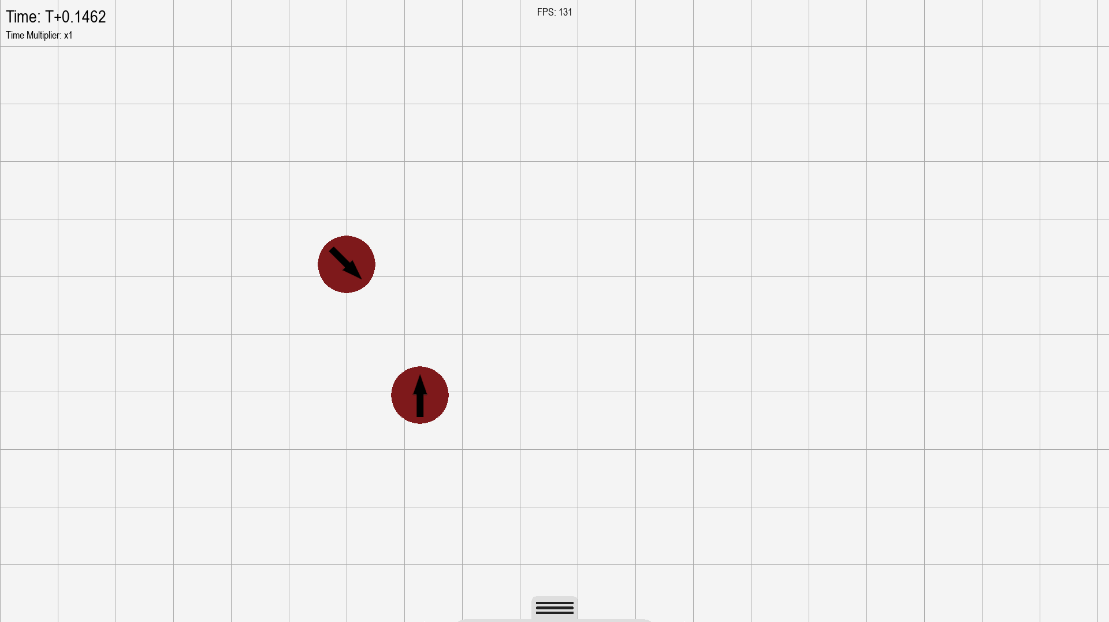
Description automatically generatedBefore/after collision with wall

A picture containing shape

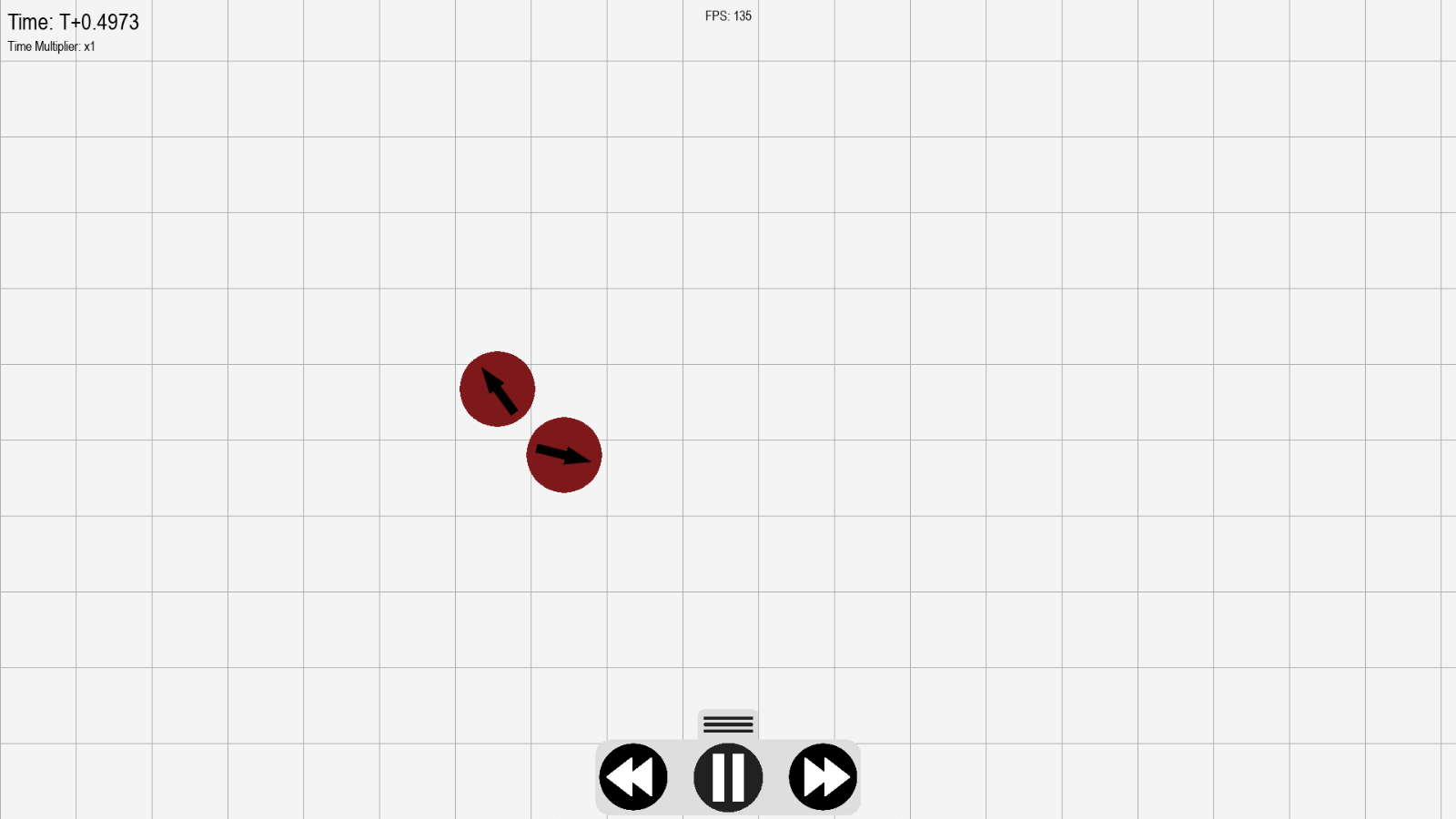
Description automatically generated

When the particle collided with the wall, its velocity was reversed in the horizontal axis, while its vertical velocity remained the same. This is the desired result, and as such, **the test was passed**

**Test 1.3**



Before collision (Particles have inverse vertical velocities, and slightly different horizontal velocities, although still in different directions)



After collision – Particles have collided and bounced off of one another in a way that one would expect to see in real life.

**Test 1.3 was passed.**

**Tests 1.5 and 2.4**

As gravity is no longer implemented by default, I created a particle that had its downwards acceleration set to 9.81 ms^-2 – the same as the value that I was using for gravity in V1 & V2. I also set its position to be exactly 8 meters off of the ground, as was specified in the test instructions for test 1.5.

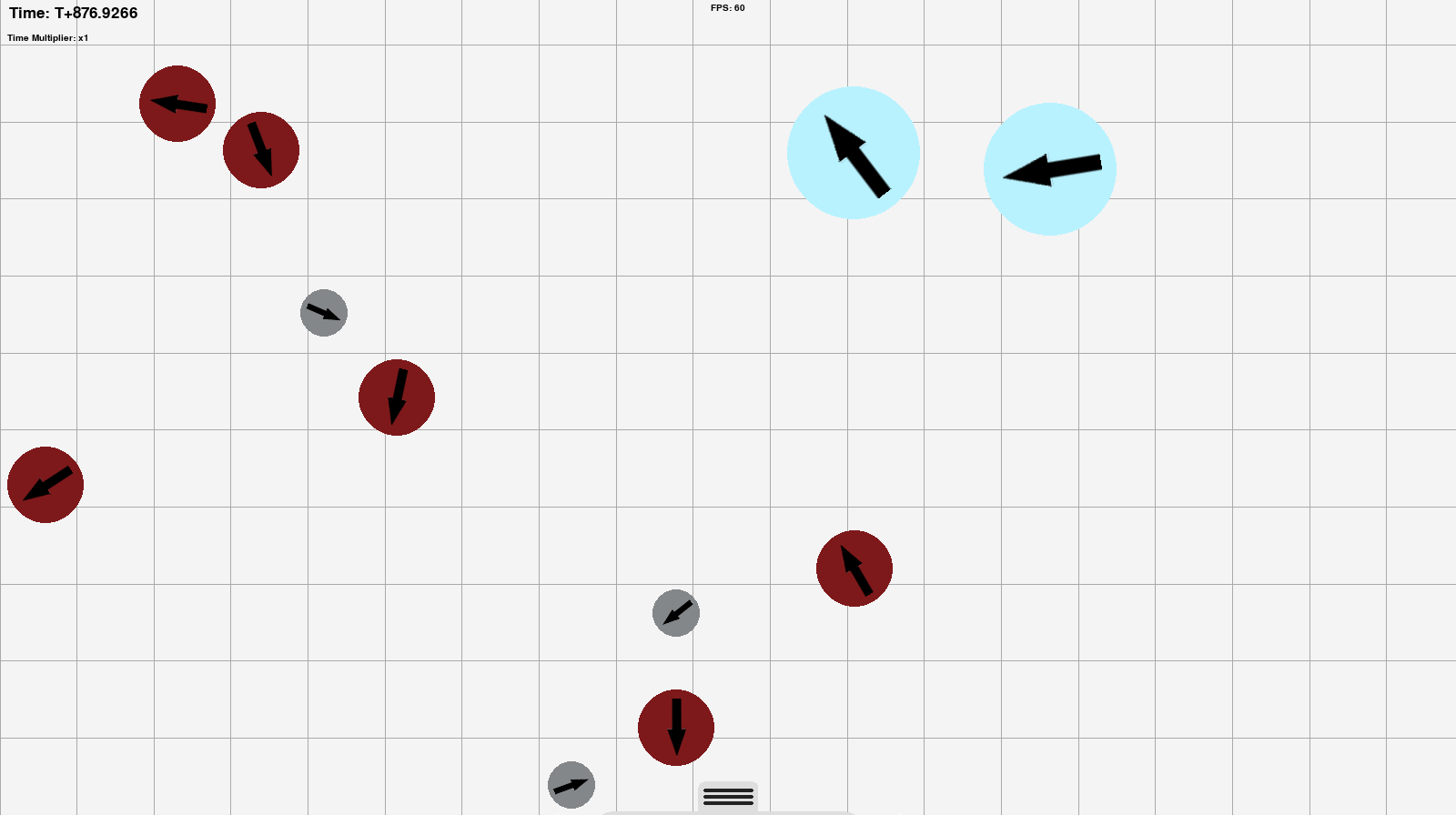
Table

Description automatically generated

Below is a screenshot of the console output (printing the times that the particles hit the floor) when the simulation started.

****

As shown in the output, the particle hits the ground at (rounded) 1.274 seconds. This is only 0.0031 seconds faster than the calculated value. This is acceptable as the inaccuracy is so small that we can consider it negligible. **Test 1.5 was passed.**

**Test 1.6**

This is a screenshot of my program running on the same Mac that I used for testing version two and version one. After running for ~880 seconds, the Mac still managed to hold a stable fps of around 55-65, and remained above 40 even when tested with a more demanding setup. All features such as fast forwarding, loading a setup, saving a setup, etc worked perfectly on the Mac. **This test was passed.**

**Test 2.2**



I was unable to follow the instructions laid out in the test specification for this test, as for version 3 I have increased the character limit on the setup screen inputs to 10. This, however, worked as I was unable to continue typing after 10 characters. **This test was passed.**

**Test 2.3**

Table

Description automatically generatedTable

Description automatically generated

Particle A Particle B

Particle B

Particle A



Particle A hits the ground after 1.785 seconds (only 0.004 seconds off of the calculated value), and particle B hits the ground after 1.267 seconds (only 0.002 seconds off of the calculated value). Both values are within 5 thousandths of a second of the calculated value, and therefore it is reasonable to say that **this test was passed.**

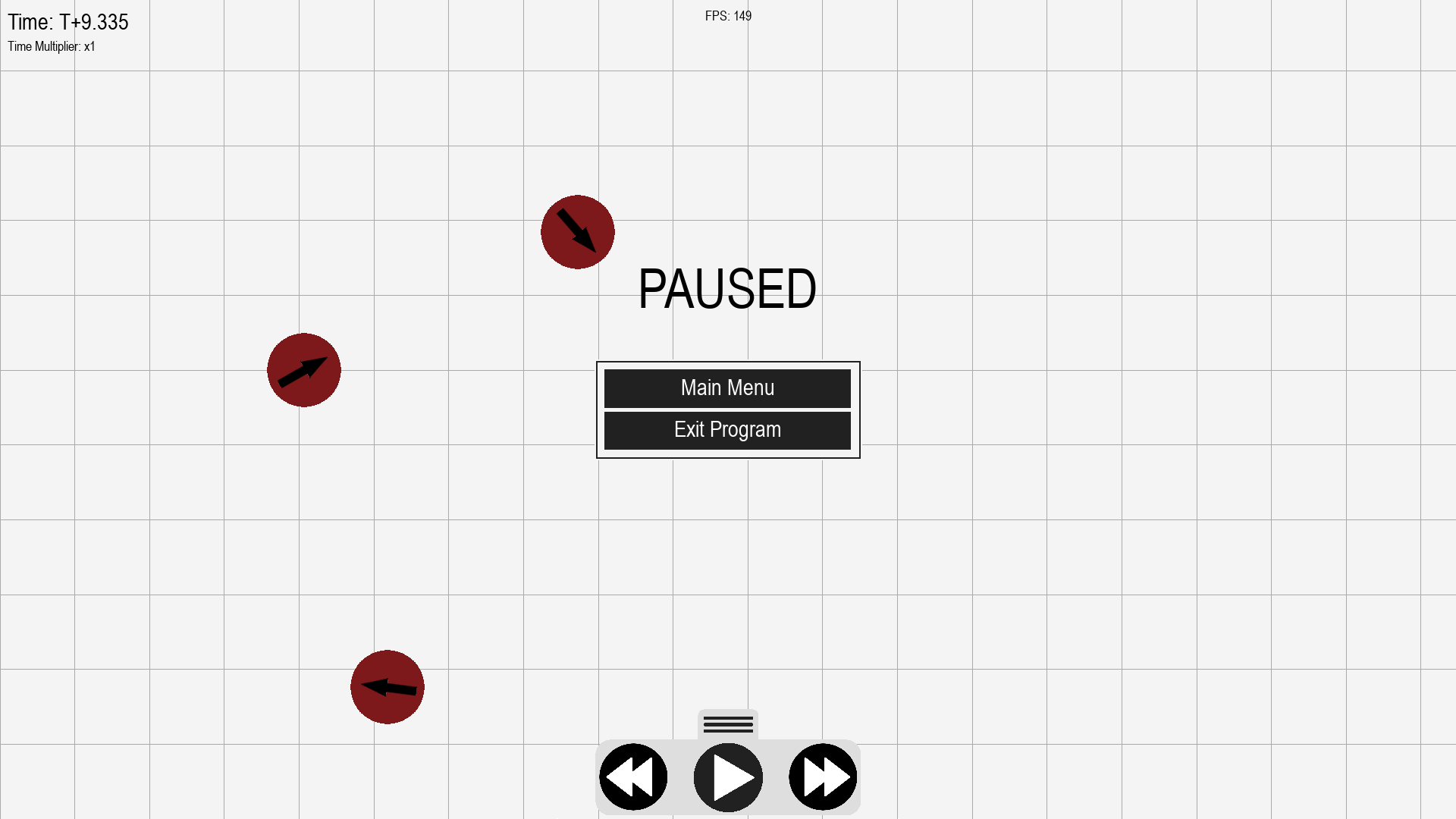
**Test 2.4**

**Chart

Description automatically generated**

All three of these particles were placed while these two checkboxes were active. By the fact that they are all on exactly the same level, you can see that the ‘lock to height’ checkbox worked, and by the fact that they all have random velocities, you can see that the ‘random velocity’ checkbox worked. **This test was passed**

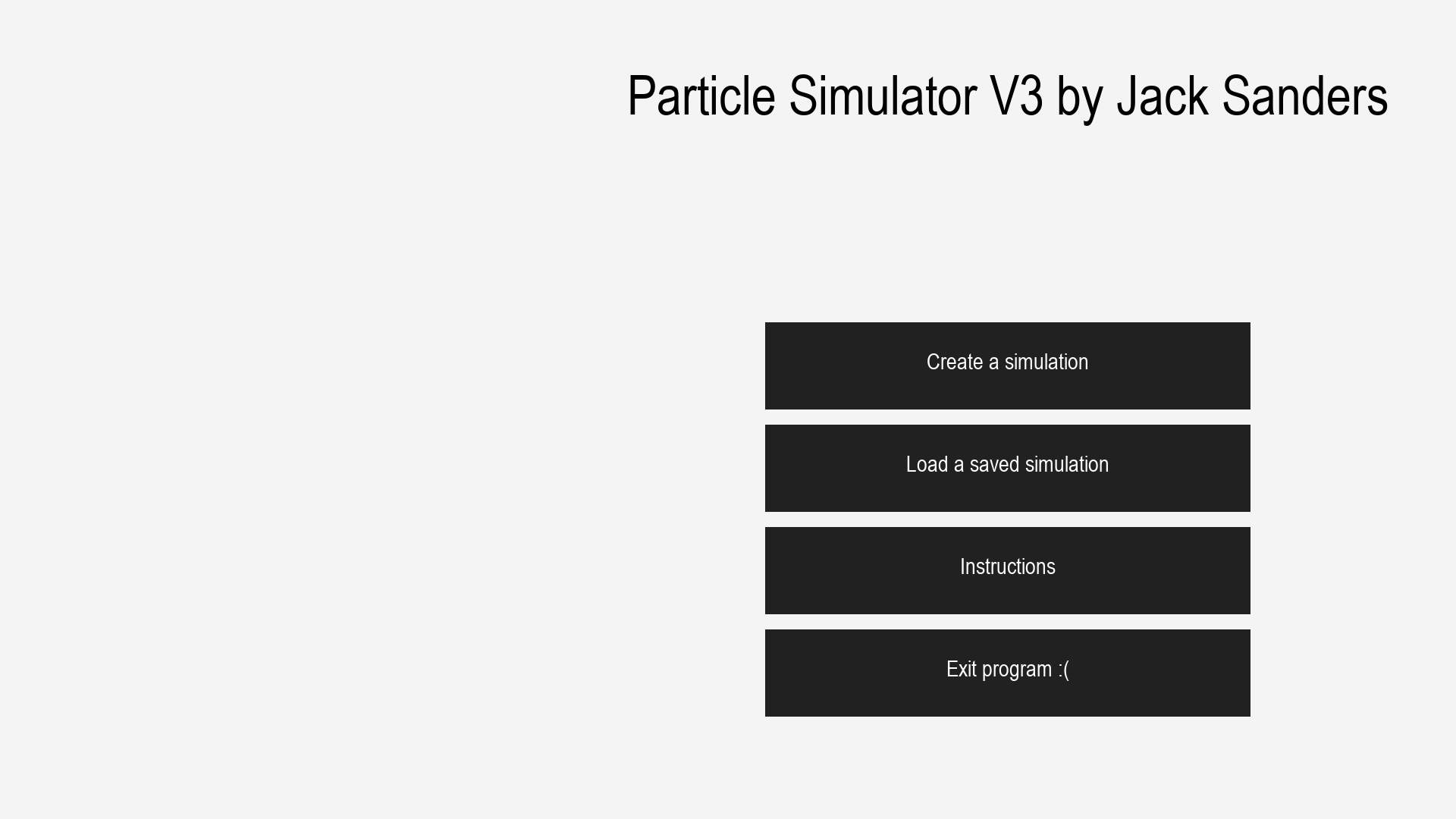
**Test 2.5**

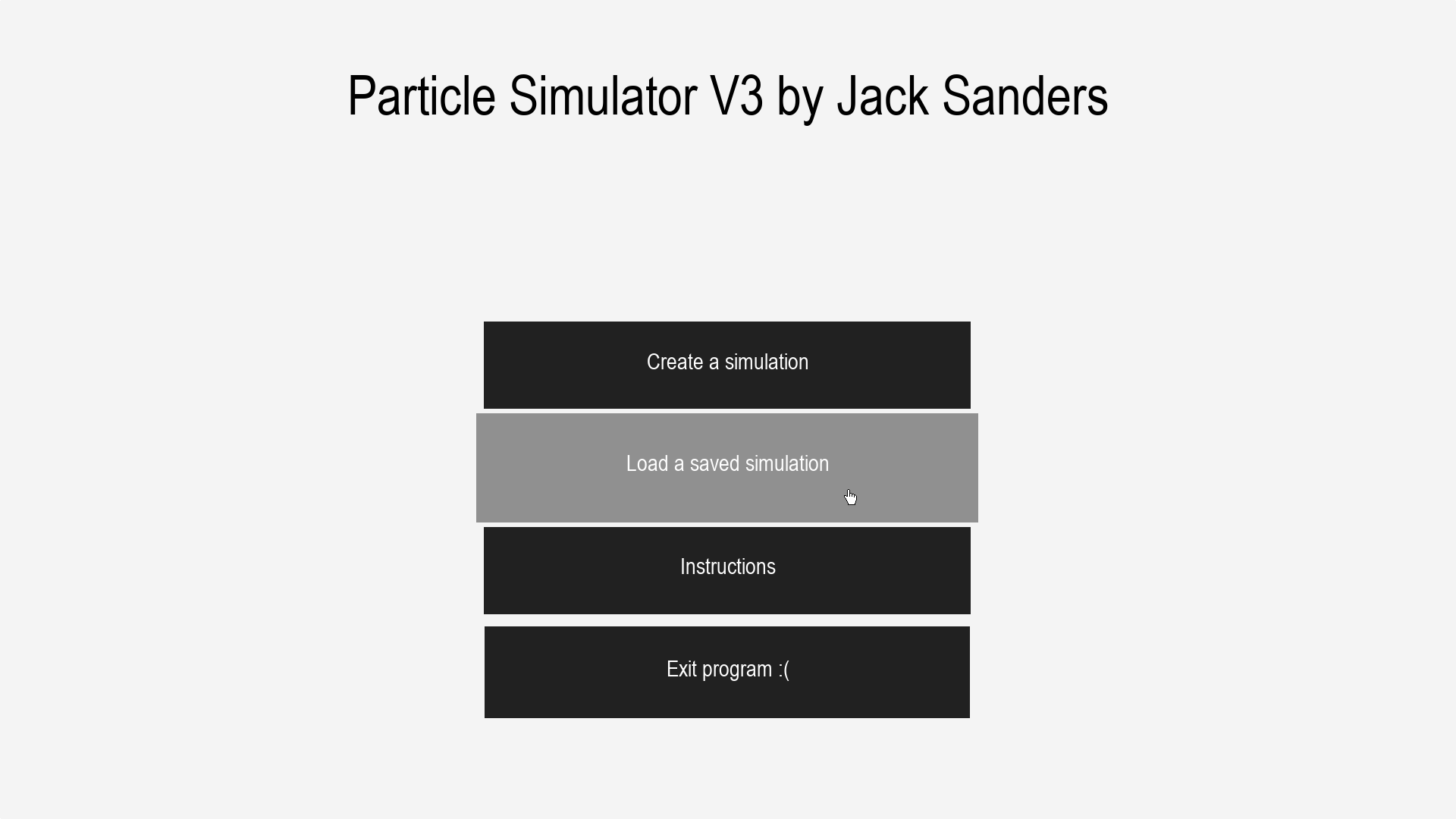


**This test was passed**

#### Testing against V3 criteria

**Test 3.1**



As you can see in the first photo, the menu does indeed start by sliding in from the right. In the second, you can see the menu as it looks when it has fully slid in, and when I hovered over a button, it swelled. **This test was passed.**

**Test 3.2**

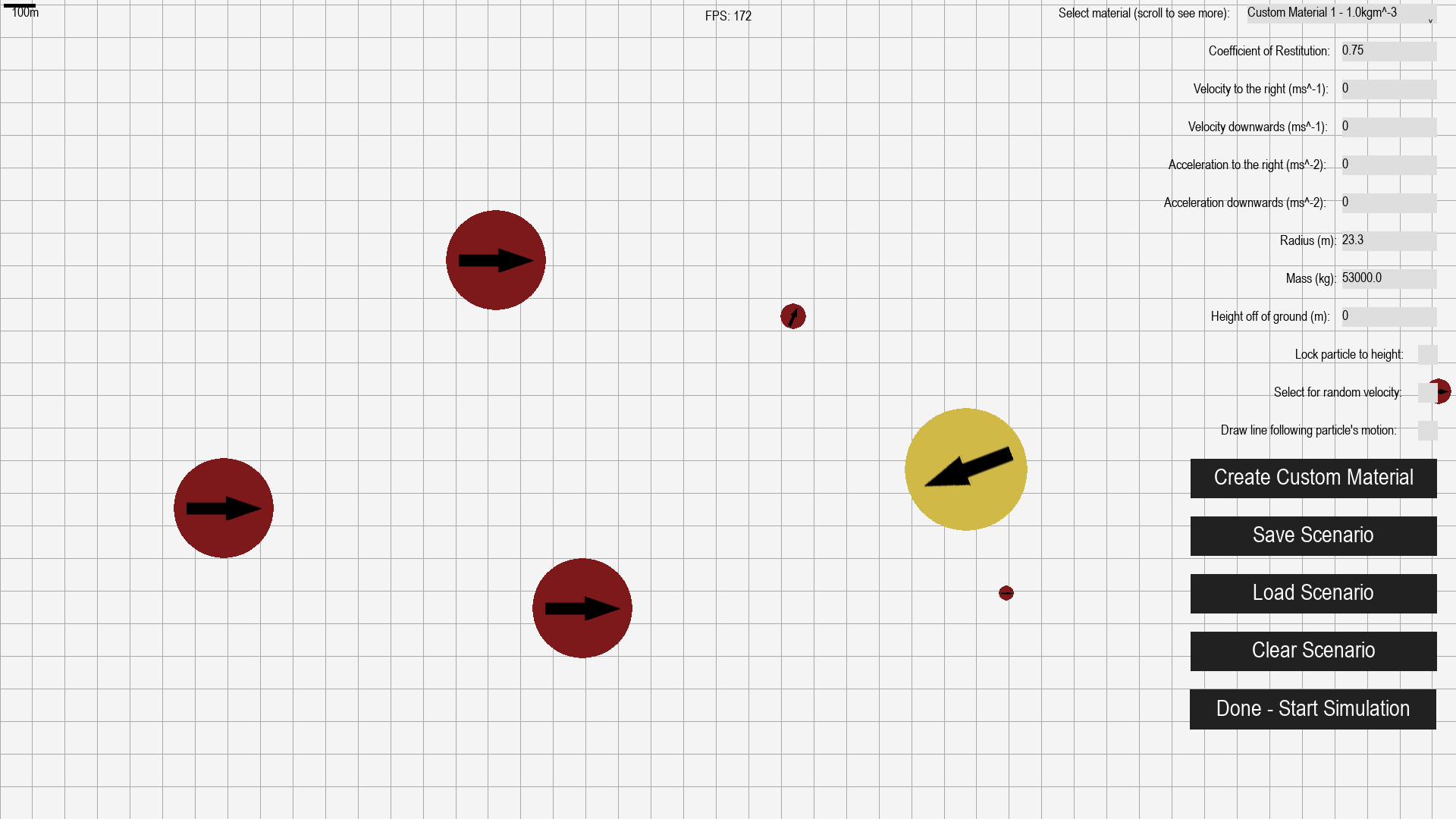


This is the page that appears when you press the ‘Instructions’ button on the main menu. **This test was passed.**

**Test 3.3**

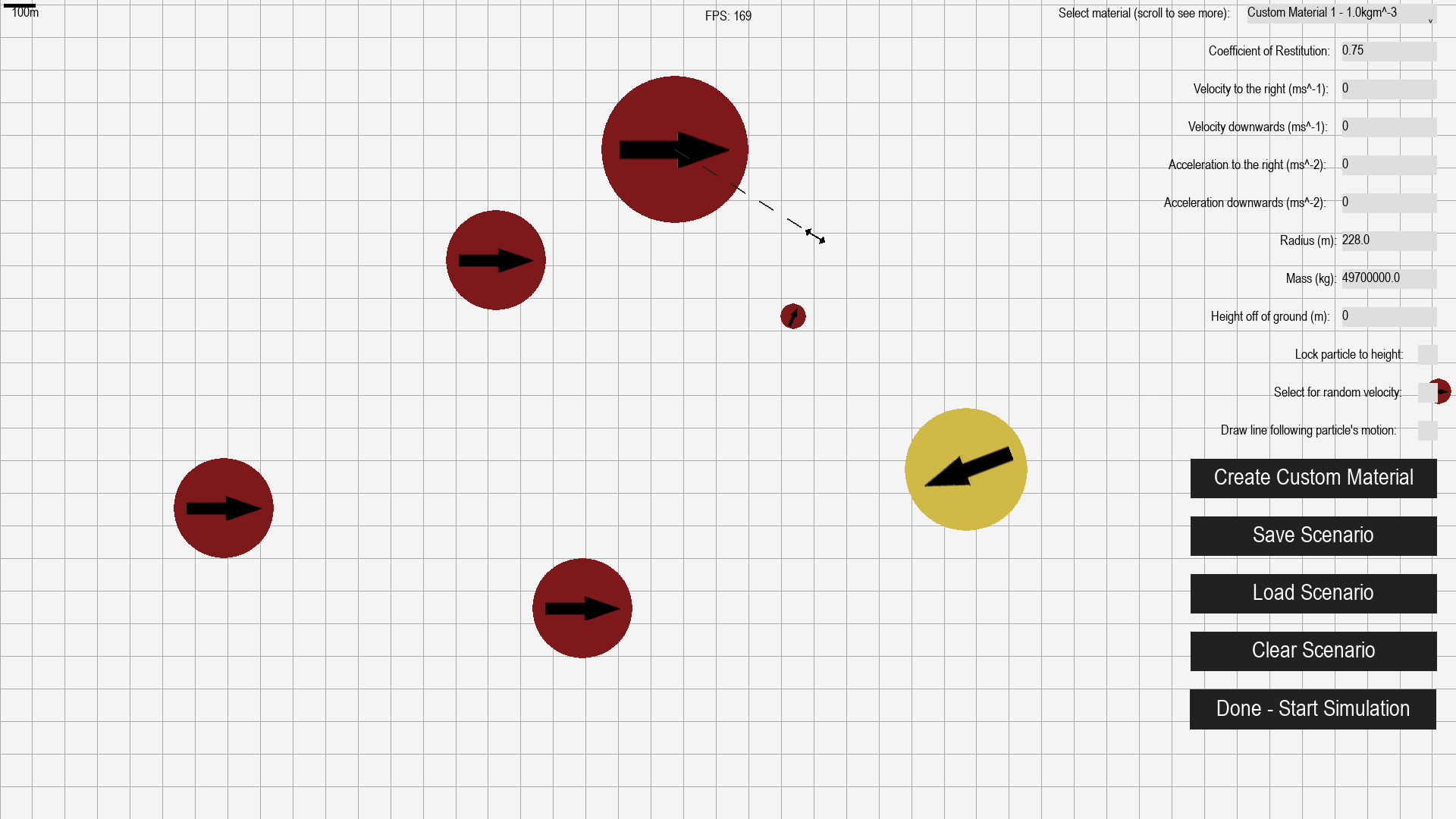
Graphical user interface, text, application, chat or text message

Description automatically generated



After pressing the ‘Load Scenario’ button, the program transitioned to the setup screen, with various particles already placed, with properties in line with what was stored in the .txt file. **This test was passed.**

**Test 3.4**



When I pressed the ‘s’ key, the mouse turned into a double-headed arrow, and a dashed line was drawn from it to the centre of the particle. When I moved the mouse, the particle changed size. **This test was passed.**

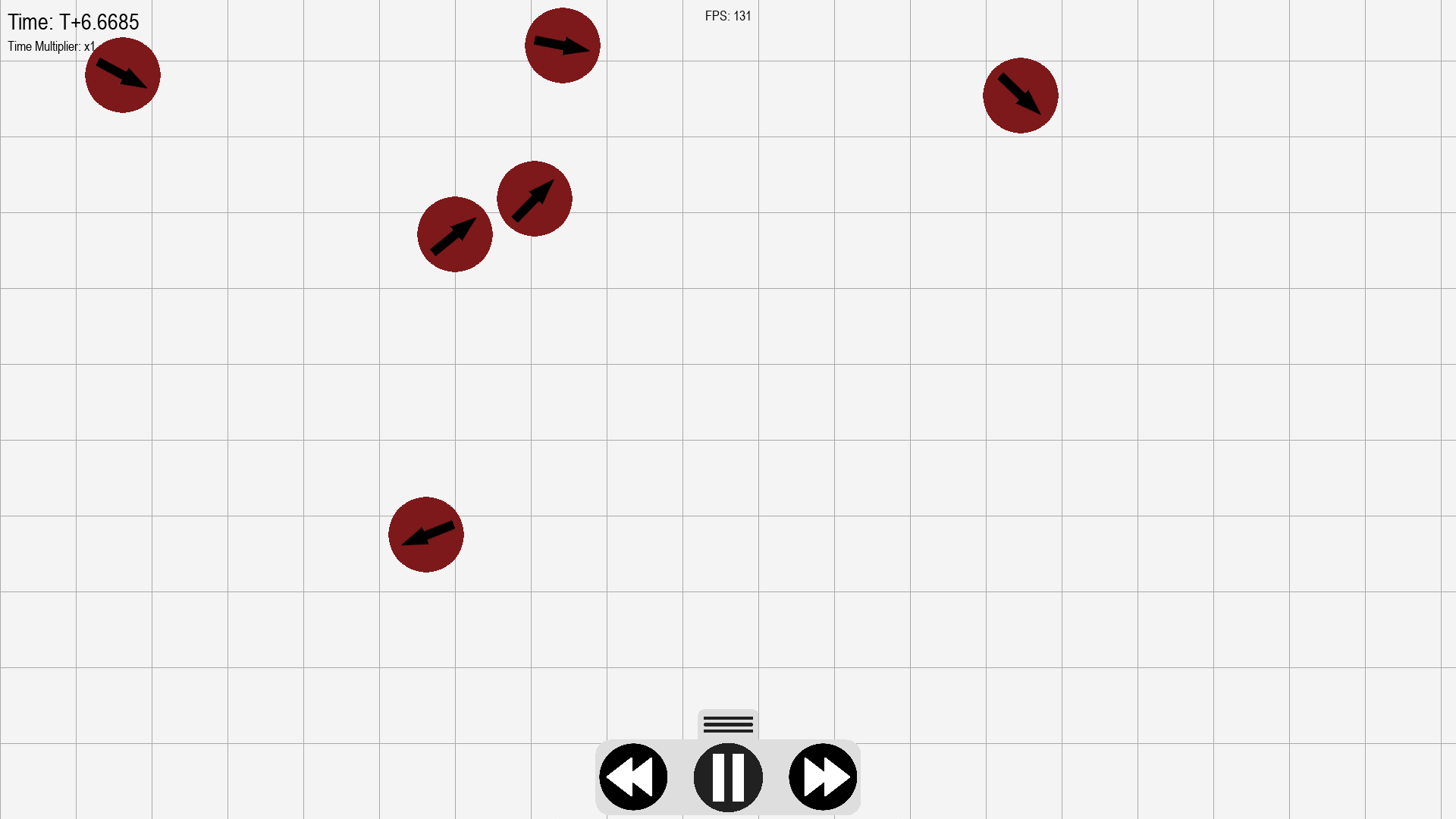
**Test 3.5**

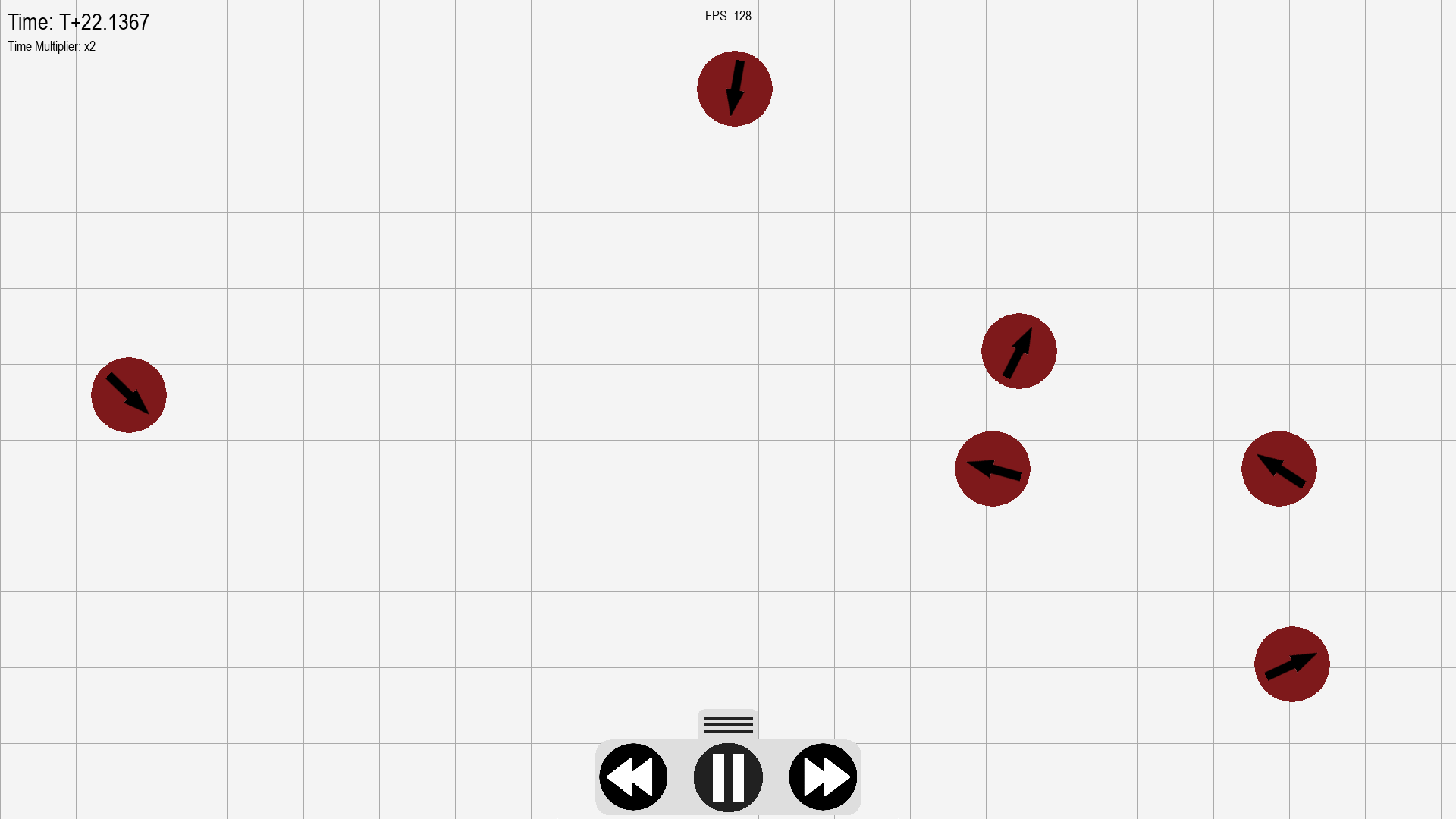
Chart

Description automatically generated with medium confidence

When I left-clicked on a particle, this menu popped up, with its bottom-left corner in the centre of the particle. When I changed the radius from 100 to 155, it increased in size. **This test was passed.**

**Test 3.6**





After I took the first screenshot, I pressed the fast forward button, and the time multiplier increased to x2. Similarly, when I pressed the rewind button, the time multiplier decreased to x0.5. When I pressed rewind again, it decreased to -0.5, and was now reversing at a slow pace. **This test was passed.**

### Development of Version 3: Review

Version 3 passed all tests from all versions 1, 2, and 3. This means that it now successfully meets all success criteria. With regards to requirements, the program meets all of them, except for requirement 4: distance markers. However, in place of distance markers, I added the ability to enable a line that draws the path of the particle behind it as it moves. This means you can see where the particle hit the floor (because of the sharp change in direction). In conjunction with this, I have added a graph display in the background that you can use to see how many metres a particle travelled before it hit the floor.

I also managed to implement all of the ideas that I gathered after receiving stakeholder feedback after development for version 2 completed.

# Evaluation

## Post Development Testing

### Regression Testing

As I haven’t changed my program since version 3, there is no need to perform additional regression testing, as all necessary tests were performed at the end of version 3, and can be evidenced on pages 129-137

### Usability/Robustness Testing

To properly test my program for ease of use and robustness, I got a friend to run it on his computer. His feedback (alongside ideas that it gave me) is below.

* “I can’t find a way to return to the editing section after starting a particle simulation”.
  + This would be a useful feature to have, and could be achieved quite easily if I simply added a few lines of code to autosave a scenario when the user presses the start simulation button to a file that will be overwritten every time a scenario is run. Alongside this, I would need a button on the simulation creation screen called ‘Load most recent autosave’ that loads the contents of this file, and maybe also a ‘Return to editing’ button on the pause menu that returns to the edit screen and automatically loads the autosave.
* “Also, the paths drawn by particles will remain when you rewind which is just a tiny issue”.
  + This issue is slightly more difficult to fi, as I would need to implement a method into the Line class that can remove plots, but it shouldn’t be too complicated. I will then just need to call that method every frame when the program is rewinding, and the lines will disappear.
* “I just set the velocity to the right to 999999999 and placed about 30 particles and it immediately crashed”.
  + A simple fix for this would be to cap the velocity relative to the screen size, as I have done with the radius.
* “Large particles cause significantly more lag than small ones”.
  + Unfortunately, this issue is not something that can be fixed without removing functionality from the program. The arrow on each particle is simply an image that gets resized and rotated depending on the size and direction of the particle. When a large particle is placed, the arrow will be scaled accordingly, resulting in a large image, which naturally causes a lot of lag. I could work around this issue by adding a ‘high-performance’ mode toggle, for use on low-power PCs, that will disable the rendering of arrows, thus preventing this issue.

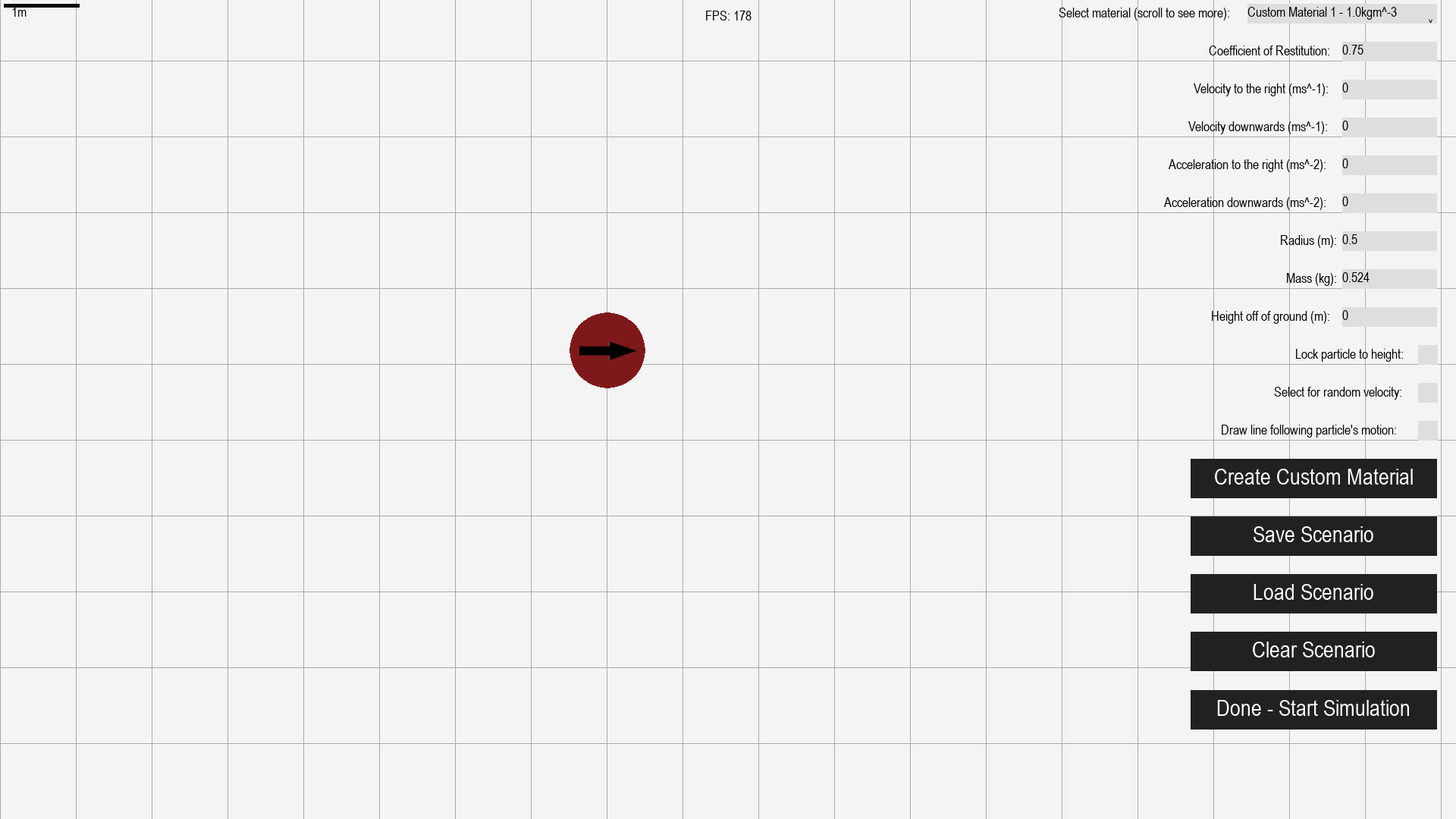
## Success Criteria

### Criteria Met

|  |  |  |
| --- | --- | --- |
|  | *Criteria* | *Met?* |
| *1* | 2D graphics | Yes |
| *2* | Placeable particles | Yes |
| *3* | Changeable properties of particles | Yes |
| *4* | Instructions | Yes |
| *5* | Simulation settings | Yes |
| *6* | Momentum | Yes |
| *7* | Gravity | Partially |
| *8* | Universal support | Yes |
| *9* | UI – Does it contain all necessary information? | Yes |

### Evidence

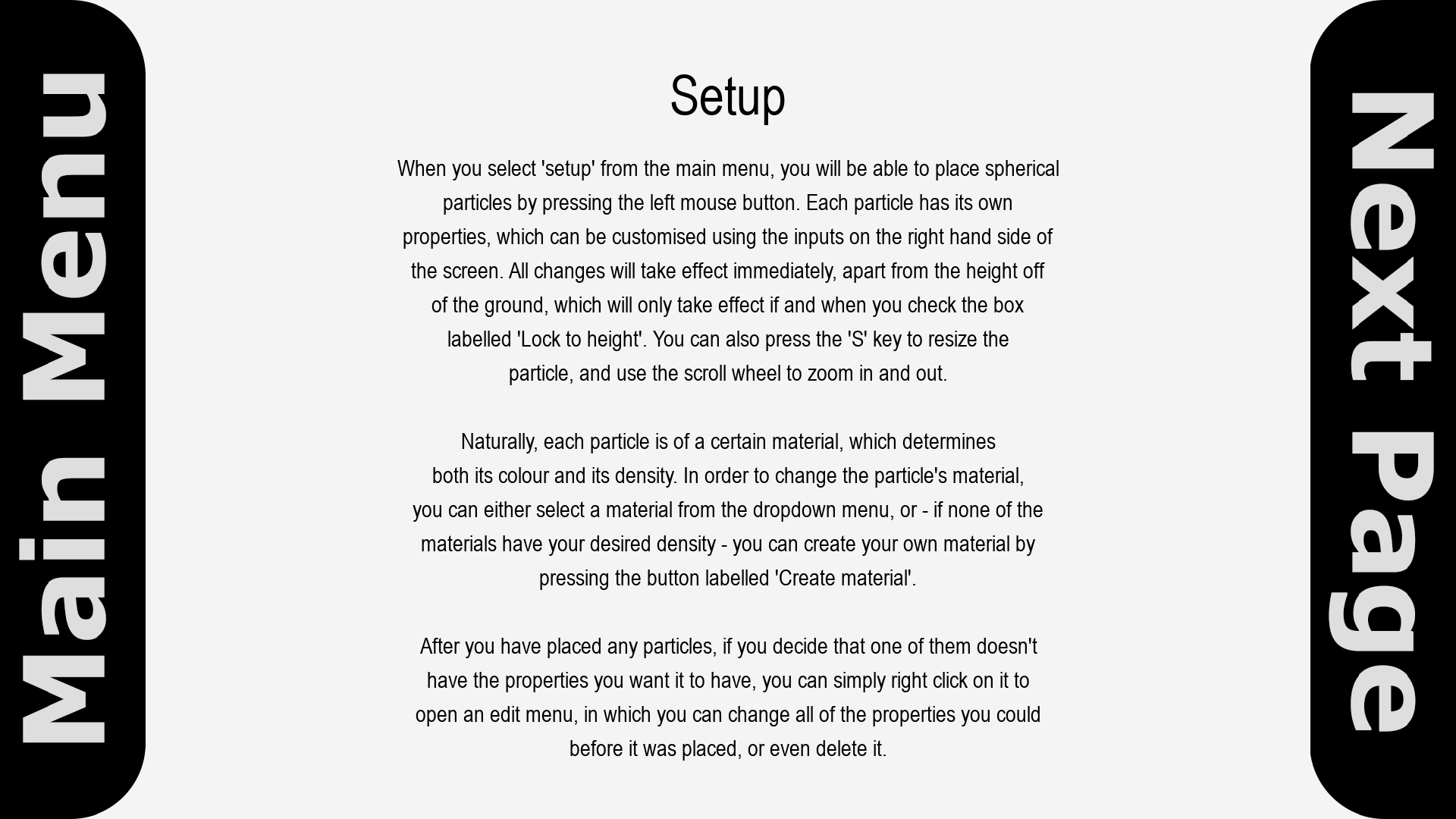
**2D graphics** – Particles are easy to see. All other elements of the UI (fps counter, properties, etc) are also easy to see, as their colours contrast well with the background.

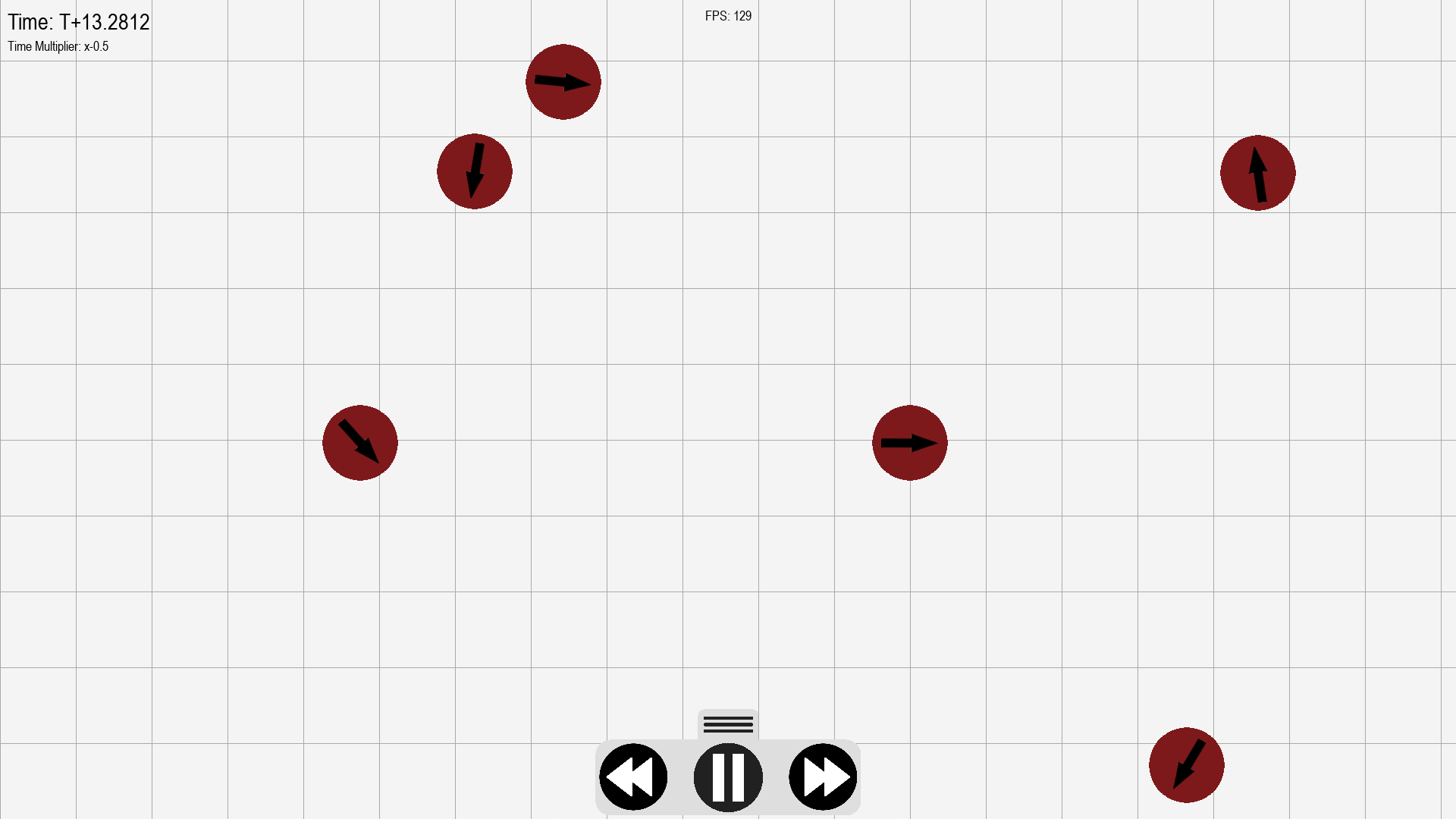


**Changeable properties of particles**

**Placeable particles**

**Instructions** – Well-written, clearly show how to use the program. The buttons are large enough to make it obvious what their function is.





**UI –** all necessary information is displayed on screen, such as: time, time multiplier, fps, and the grid.

**Simulation Settings** – note how the time multiplier in the upper left is negative – the program is rewinding.

**Momentum –** From the following two screenshots, before and after a collision between two particles of identical mass and velocity, it is clear to see that the particles’ momentum is transferred correctly. After the collision, they each move along trajectories that are in opposite directions, but are both in the same direction as the line of centres (a line connecting the centres of the two particles at the time of the collision).

Logo

Description automatically generated

A picture containing chart

Description automatically generated

**Gravity** – There is no built-in gravity in my simulator, as it is simply simulating particles, and not any attractive forces. However, since you can input custom acceleration values, it is entirely possible to simulate a scenario that requires acceleration due to gravity (such as a cannonball) by inputting a value for downwards acceleration that is constant across all of the particles you place.

**Universal support:**

First, I created a test scenario, and ran it at the highest fps my computer could achieve (around 130)

Here are some screenshots of a collision occurring at ~130fps.

Chart, bubble chart

Description automatically generatedChart, bubble chart

Description automatically generated

Next, I implemented an artificial limiter, that limited the fps right down to 15.

Here are some screenshots of the same collision occurring at 15fps.

Chart, bubble chart

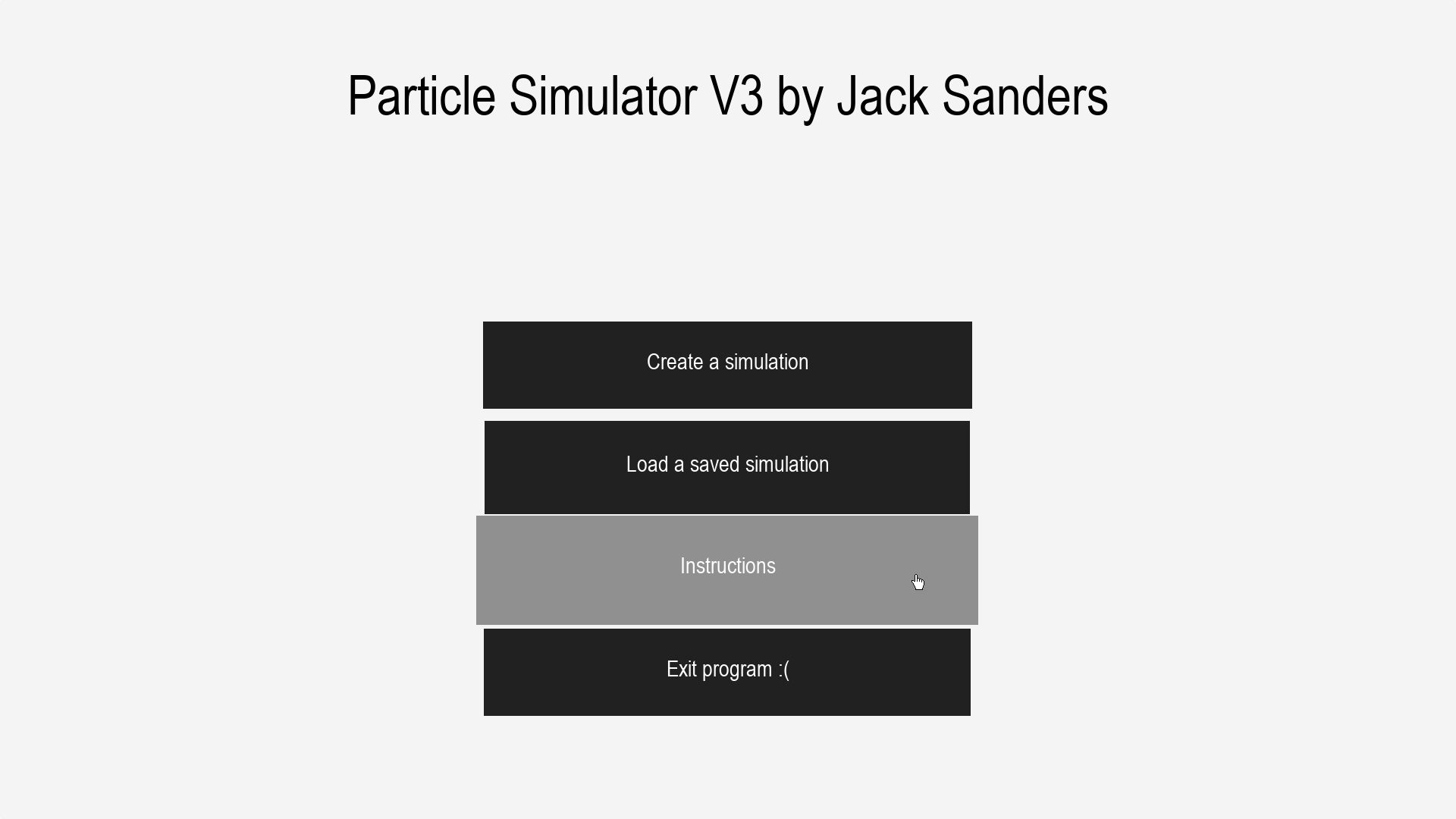
Description automatically generatedChart, bubble chart

Description automatically generated

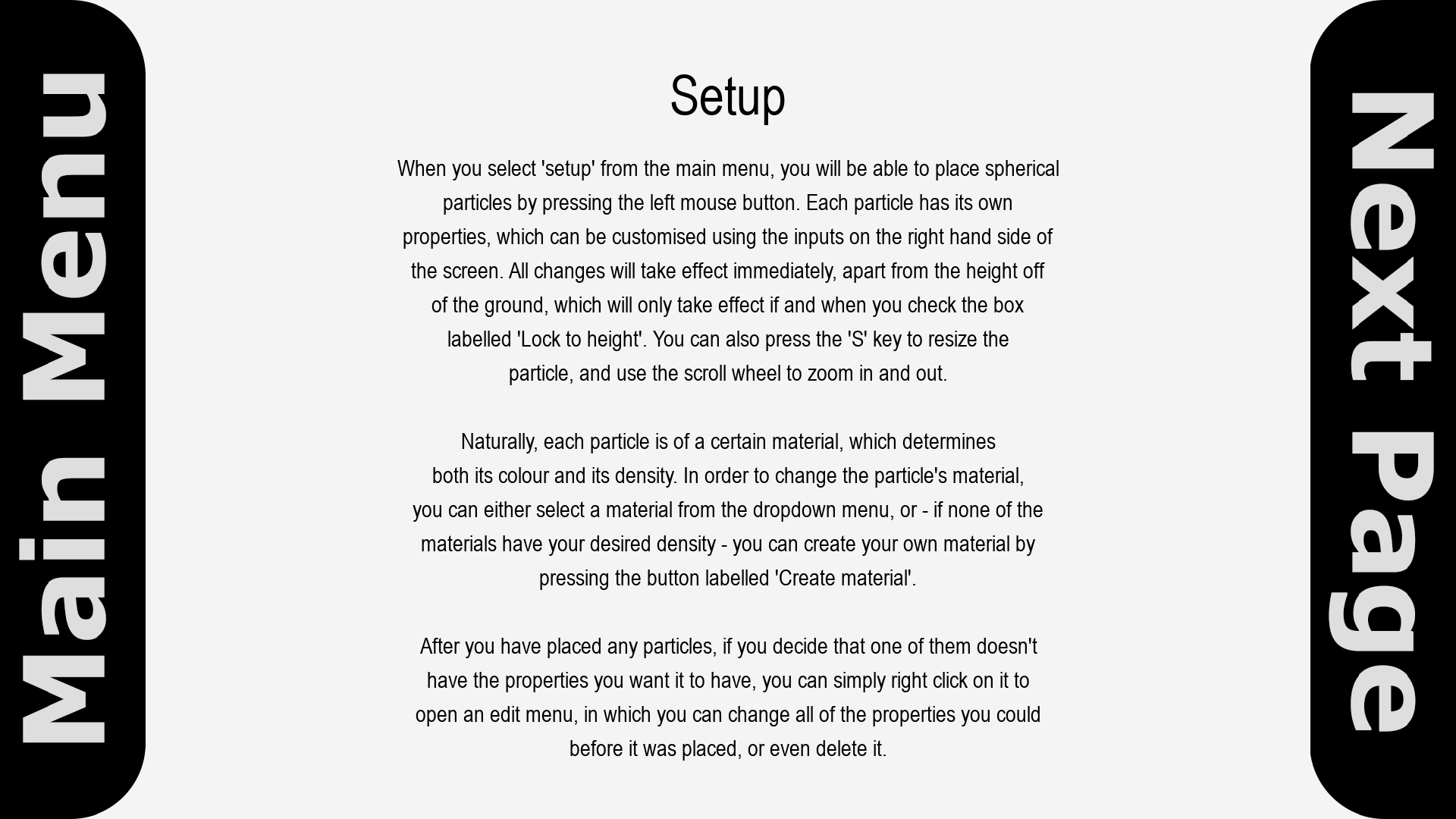
As you can see, the program behaves exactly the same, no matter what fps it runs at, meaning it has met this success criteria.

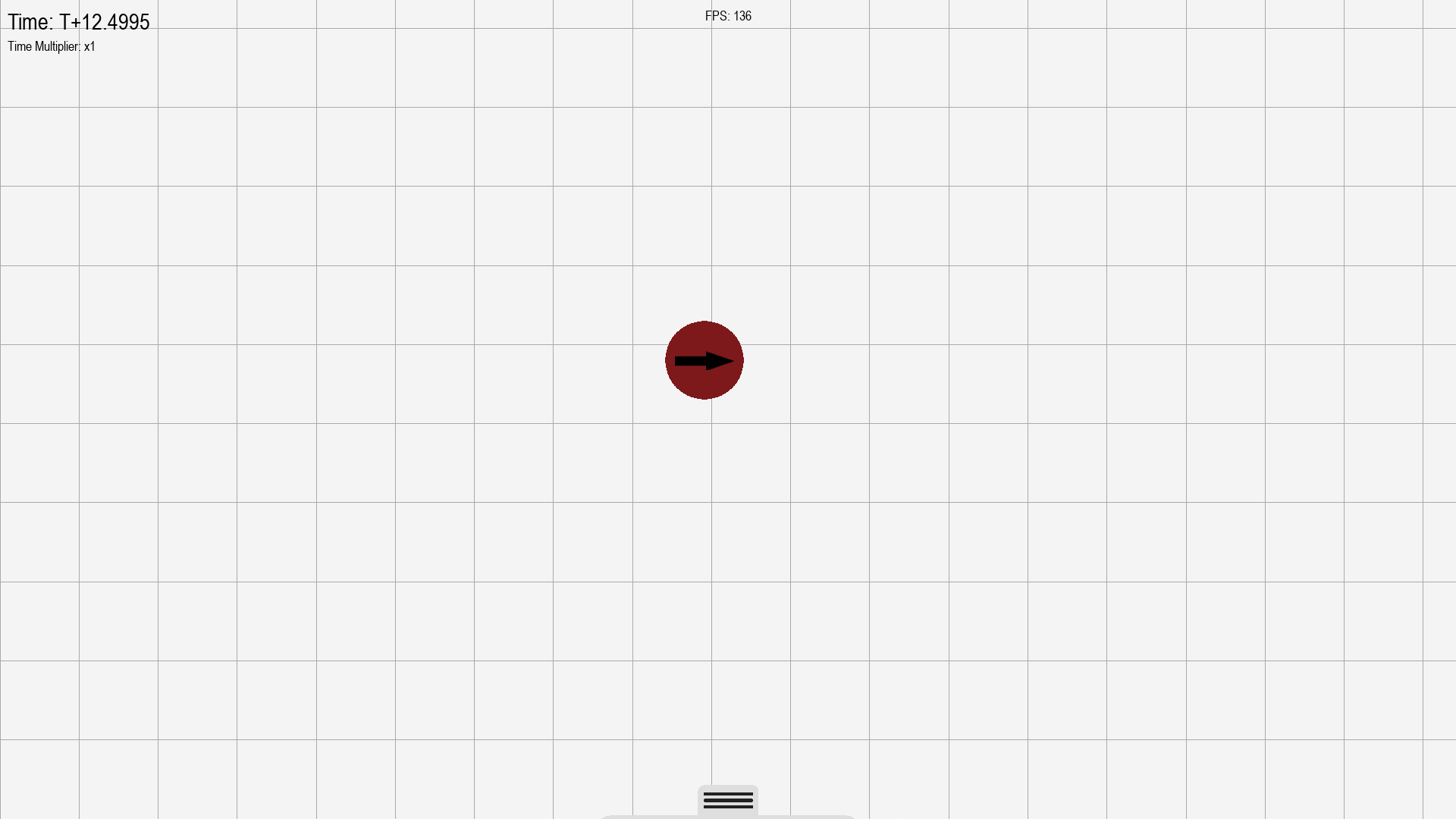
## Usability Features

The main menu screen has very large, clear buttons – making it obvious to the user where they need to click to get to the screen they want. When the user hovers over a button, it changes colour slightly and increases in size. This makes it much easier for the user to know which button they are hovering over, even if they are not particularly paying attention to the position of the mouse. This is the same for all buttons in the program, with the exception of the ones on the time controls, or the ones on the instruction menu.



Similarly, the buttons on the instruction screen are very large. While this was mainly done as an aesthetic choice, to make it look as though the user was ‘dragging’ the screen when the animation plays, it doubles as a good usability feature, providing a larger area that the user can click on to achieve the same result as a smaller button.





On the main simulation screen, the time controls start hidden, but can be shown by clicking on the tab at the bottom of the screen. In order to make this clearer, I have deliberately made it such that a small part of the body of the time controls is showing (just below and to the sides of the tab), to clearly show that the user can pull it up from there.

## Limitations

One limitation that may crop up if and when I distribute my program is that it utilises the Helvetica font pretty much everywhere. Some computers may not have Helvetica installed, meaning they will not be able to render the text, and I would need to acquire a license in order to redistribute the font.

With regards to the actual program, there aren’t really any huge limitations, only small things such as no autosaves. Currently, if you want to use the same simulation multiple times, you have to ensure that you save it, rather than simply returning directly to the edit window from the main program. While this is easily avoidable by simply saving your scenarios, it would be much better if I added an autosave feature in future versions of my program.

## Maintenance

One feature I need to add in future is an autosave functionality, and a button in the pause menu that will take you back to the setup screen, with the scenario that you just ran already in there. After adding that, I simply need to implement some minor bug fixes, such as adding a limit on how high you can set the velocity of the particles (relative to screen size), to prevent crashes. I will also need to add a removePlot method to the line class, so that a particle’s line will disappear when the program is rewinding.

Aside from these, if the stakeholders would like to add any new features, it will be very easy to do so, as my interface library is extremely modular, meaning creating new menu screens or new buttons is simple.

# Final Code

#### Pgkinter module

import time  
from pathlib import Path  
from math import sqrt, floor  
  
import pygame as pg  
from pygame.locals import \*  
  
try:  
 from pyClipTools import paste  
except ImportError:  
 def paste():  
 return **""***# PyClipTools is an external module that allows text to be written to and  
# read from the windows clipboard from within python*class Pgk(object):  
 def \_\_init\_\_(self): *# Initialises pgkinter - creates all necessary globals* pg.font.init()  
 self.pgkGroup = pg.sprite.Group()  
  
 imagesFolder = Path(**"resources/images/"**)  
 self.pgkTypingCursor = pg.image.load(  
 str(imagesFolder / **"pgkTypingCursor.png"**))  
 self.pgkPointerCursor = pg.image.load(  
 str(imagesFolder / **"pgkPointerCursor.png"**))  
 self.pgkLetterChars = set(  
 **"aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ"**)  
 self.pgkNumberChars = set(**"1234567890"**)  
 self.pgkMathsChars = set(**"-.eE"**) *# Include "e" for standard form* self.pgkSpecialChars = set(**"¬`!**\"**£$%^&\*()\_-+={[}]:;@'~#<,>.?/|**\\**¦"**)  
 self.pgkWhiteCrossImage = pg.image.load(  
 str(imagesFolder / **"pgkWhiteCross.png"**))  
 self.pgkBlackCrossImage = pg.image.load(  
 str(imagesFolder / **"pgkBlackCross.png"**))  
 print(**"Pgkinter V1.0.0 initialised successfully! Hello there!"**)  
  
 def buttonDefaultAction(self):  
 *# Default action that will be assigned to buttons if none is assigned on  
 # instantiation* pass  
  
 def getWidgets(self):  
 return self.pgkGroup.sprites()  
  
 def hoverEffect(self, rgb):  
 *# Returns an rgb code that is lighter or darker than the one passed to  
 # the function, depending on whether the original is light or dark* if self.isLight(rgb):  
 return rgb[0] \* 1 / 2, rgb[1] \* 1 / 2, rgb[2] \* 1 / 2  
 else:  
 return (  
 rgb[0] + ((255 - rgb[0]) \* 1 / 2),  
 rgb[1] + ((255 - rgb[1]) \* 1 / 2),  
 rgb[2] + ((255 - rgb[2]) \* 1 / 2))  
  
 def isLight(self, rgb): *# Determines whether an rgb code is light or dark  
 """Treats the rgb code as a 3D position vector and calculates the length of  
 the line from the origin to the position vector. The longer the line,  
 the lighter the colour.  
  
 """* if sqrt(rgb[0] \*\* 2 + rgb[1] \*\* 2 + rgb[2] \*\* 2) >= 220:  
 return True  
  
 else:  
 return False  
  
 *# noinspection SpellCheckingInspection* def update(self): *# Only need pgkinter.update() in main code* self.pgkGroup.update()  
  
 *# noinspection SpellCheckingInspection,SpellCheckingInspection* def eventHandler(self, event):  
 *# Takes an event list from the main pygame loop, and passes it to every  
 # pgkinter sprite.* handled = False  
 for obj in self.pgkGroup.sprites():  
 *# InputBoxes can handle events even after one has been handled as  
 # they need to be able to become inactive* if not handled or isinstance(obj, InputBox):  
 if obj.handleEvent(event):  
 handled = True  
 return handled  
  
  
*# noinspection PyArgumentList*class Button(pg.sprite.Sprite):  
 *# noinspection SpellCheckingInspection* def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,  
 text=None, height=None, width=None,  
 action=None, image=None, hoverImage=None, container=None,  
 swellOnHover=None):  
 super().\_\_init\_\_() *# Runs pygame sprite \_\_init\_\_() method* self.\_\_parent = parent  
 self.\_\_screen = screen  
 self.\_\_timer = 0.5 *# Starts at 0.5 so button is usable instantly* self.\_\_previousFrame = time.time()  
 self.\_\_hovered = False  
  
 try: *# Input validation* x = int(x)  
 y = int(y)  
 except ValueError:  
 raise Exception(**"Button coordinates must be integers"**)  
  
 **"""Almost all attributes have default values that will be assigned if  
 no values are passed on instantiation.  
  
 """** if font is None:  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 30)  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if bgColour is None:  
 self.\_\_bgColour = (255, 255, 255)  
 else:  
 self.\_\_bgColour = bgColour  
  
 if text is None:  
 self.\_\_text = **""** else:  
 self.\_\_text = text  
  
 *# Changes text colour depending on whether the background colour is  
 # light or not.* if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (0, 0, 0))  
 else:  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (255, 255, 255))  
  
 if not height:  
 self.\_\_height = self.\_\_displayText.get\_rect().h \* 1.25  
 else:  
 self.\_\_height = height  
  
 if not width:  
 self.\_\_width = int(pg.display.get\_surface().get\_width() / 10)  
 else:  
 self.\_\_width = width  
  
 if not action:  
 self.\_\_action = self.\_\_parent.buttonDefaultAction  
 else:  
 self.\_\_action = action  
  
 if image is None:  
 self.\_\_image = None  
 else:  
 self.\_\_image = pg.transform.scale(image, (self.\_\_width,  
 self.\_\_height))  
  
 if hoverImage is None:  
 self.\_\_hoverImage = None  
 else:  
 self.\_\_hoverImage = pg.transform.scale(hoverImage, (self.\_\_width,  
 self.\_\_height))  
  
 self.\_\_bgColourHover = self.\_\_parent.hoverEffect(self.\_\_bgColour)  
  
 self.\_\_coords = [x, y]  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 *# Uses 0.45 \* height as 0.5 places text slightly below-centre* self.\_\_textRect = self.\_\_displayText.get\_rect(  
 center=(x + 0.5 \* self.\_\_width, y + 0.45 \* self.\_\_height))  
  
 if self.\_\_image is not None:  
 self.\_\_rect = self.\_\_image.get\_rect(topleft=(x, y))  
  
 if container is None:  
 self.\_\_container = None  
 else:  
 self.\_\_container = container  
 self.\_\_container.addWidget(self)  
  
 if swellOnHover is None:  
 self.\_\_swellOnHover = False  
 else:  
 self.\_\_swellOnHover = True  
 self.\_\_origWidth = self.\_\_width  
 self.\_\_origHeight = self.\_\_height  
  
 self.\_\_parent.pgkGroup.add(self)  
  
 def config(self, font=None, bgColour=None, text=None,  
 height=None, width=None, action=None, image=None,  
 hoverImage=None):  
 *"""Config method allows the programmer to change any settings that may  
 have a default value after instantiation.  
  
 """* if not font:  
 pass  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
 if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (0, 0, 0))  
 else:  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (255, 255, 255))  
  
 if not bgColour:  
 pass  
 else:  
 self.\_\_bgColour = bgColour  
 self.\_\_bgColourHover = self.\_\_parent.hoverEffect(bgColour)  
  
 if not text:  
 pass  
 else:  
 if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_text = text  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (0, 0, 0))  
 else:  
 self.\_\_text = text  
 self.\_\_displayText = self.\_\_font.render(self.\_\_text, True,  
 (255, 255, 255))  
  
 if not height:  
 pass  
 else:  
 self.\_\_height = height  
  
 if not width:  
 pass  
 else:  
 self.\_\_width = width  
  
 if not action:  
 pass  
 else:  
 self.\_\_action = action  
  
 if image is None:  
 pass  
 else:  
 self.\_\_image = pg.transform.scale(image, (int(self.\_\_width),  
 int(self.\_\_height)))  
  
 if hoverImage is None:  
 pass  
 else:  
 self.\_\_hoverImage = pg.transform.scale(hoverImage,  
 (int(self.\_\_width),  
 int(self.\_\_height)))  
  
 self.\_\_textRect = self.\_\_displayText.get\_rect(  
 center=(self.\_\_rect.x + 0.5 \* self.\_\_width,  
 self.\_\_rect.y + 0.45 \* self.\_\_height))  
  
 def delete(self):  
 *# Need to set mouse back to normal, otherwise the mouse will remain  
 # hidden after the widget has been deleted* pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 *# Made draw its own function, as widgets need to be drawn in a  
 # different order if they are in a container* if self.\_\_container:  
 x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 *# Uses 0.45 \* height as 0.5 places text slightly below-centre* self.\_\_textRect = self.\_\_displayText.get\_rect(  
 center=(x + 0.5 \* self.\_\_width, y + 0.45 \* self.\_\_height))  
  
 if self.\_\_hovered and self.\_\_timer > 0.5:  
 if self.\_\_image is None:  
 *# Draws button with hover colour variant* pg.draw.rect(self.\_\_screen, self.\_\_bgColourHover, self.\_\_rect)  
 self.\_\_screen.blit(self.\_\_displayText, self.\_\_textRect)  
 else:  
 self.\_\_screen.blit(self.\_\_hoverImage, self.\_\_rect)  
  
 pg.mouse.set\_visible(False)  
 *# Draws custom mouse image over mouse location* self.\_\_pointerRect = self.\_\_parent.pgkPointerCursor.get\_rect(  
 top=pg.mouse.get\_pos()[1])  
 self.\_\_pointerRect.x = pg.mouse.get\_pos()[  
 0] - self.\_\_pointerRect.w / 2  
 self.\_\_screen.blit(self.\_\_parent.pgkPointerCursor,  
 self.\_\_pointerRect)  
  
 elif not self.\_\_hovered or self.\_\_timer < 0.5:  
 if self.\_\_image is None:  
 *# Draws button with regular colour* pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
 self.\_\_screen.blit(self.\_\_displayText, self.\_\_textRect)  
 else:  
 self.\_\_screen.blit(self.\_\_image, self.\_\_rect)  
  
 def getHeight(self):  
 return self.\_\_height  
  
 def getWidth(self):  
 return self.\_\_width  
  
 def getPos(self):  
 return self.\_\_rect.x, self.\_\_rect.y  
  
 def isHovered(self):  
 return self.\_\_hovered  
  
 *# Swell and shrink are button animations that make the button change size  
 # when you hover over it - gives the UI a more modern and sleek feel* def swell(self):  
 *# Works very similarly to container animations  
  
 # Multiplier controls how much the button swells by, in order to make  
 # the animation last a certain length of time (0.25 seconds)* multiplier = (time.time() - self.\_\_previousFrame) / 0.25  
  
 widthDifference = self.\_\_origWidth \* 1.05 - self.\_\_origWidth  
 heightDifference = self.\_\_origHeight \* 1.25 - self.\_\_origHeight  
  
 *# Can only add increments greater than 0/5 as pygame rect dimensions  
 # are stored as ints - anything smaller than 0.5 would have no effect  
 # on the size, and would be stuck in an infinite loop* if widthDifference \* multiplier > 0.5:  
 self.\_\_width += widthDifference \* multiplier  
 self.\_\_coords[0] -= widthDifference \* multiplier / 2  
 else:  
 self.\_\_width += 1  
 self.\_\_coords[0] -= 0.5  
  
 if heightDifference \* multiplier > 0.5:  
 self.\_\_height += heightDifference \* multiplier  
 self.\_\_coords[1] -= heightDifference \* multiplier / 2  
 else:  
 self.\_\_height += 1  
 self.\_\_coords[1] -= 0.5  
  
 def shrink(self):  
 *# Identical to swell function, but inverse signs* multiplier = (time.time() - self.\_\_previousFrame) / 0.25  
  
 widthDifference = self.\_\_origWidth \* 1.05 - self.\_\_origWidth  
 heightDifference = self.\_\_origHeight \* 1.25 - self.\_\_origHeight  
  
 if widthDifference \* multiplier > 0.5:  
 self.\_\_width -= widthDifference \* multiplier  
 self.\_\_coords[0] += widthDifference \* multiplier / 2  
 else:  
 self.\_\_width -= 1  
 self.\_\_coords[0] += 0.5  
  
 if heightDifference \* multiplier > 0.5:  
 self.\_\_height -= heightDifference \* multiplier  
 self.\_\_coords[1] += heightDifference \* multiplier / 2  
 else:  
 self.\_\_height -= 1  
 self.\_\_coords[1] += 0.5  
  
 *# noinspection PyAttributeOutsideInit* def update(self):  
 *# 0.5 timer check ensures that button can only be clicked once every  
 # 0.5 seconds* if self.\_\_rect.collidepoint(pg.mouse.get\_pos()) and not self.\_\_hovered:  
 *# Can only be hovered over if button is not obstructed by  
 # container mask* if (self.\_\_container and not self.\_\_container.mouseMasked()) or \  
 not self.\_\_container:  
 *# Sets mouse cursor to invisible* self.\_\_hovered = True  
  
 elif not self.\_\_rect.collidepoint(  
 pg.mouse.get\_pos()) and self.\_\_hovered:  
 *# Sets mouse cursor back to default* pg.mouse.set\_visible(True)  
 self.\_\_hovered = False  
  
 if self.\_\_swellOnHover and self.\_\_hovered:  
 if self.\_\_rect.height < self.\_\_origHeight \* 1.25 and \  
 self.\_\_rect.width < self.\_\_origWidth \* 1.05:  
 self.swell()  
 elif self.\_\_rect.height > self.\_\_origHeight \* 1.25 or \  
 self.\_\_rect.width > self.\_\_origWidth \* 1.05:  
 self.\_\_rect.height = self.\_\_origHeight \* 1.25  
 self.\_\_rect.width = self.\_\_origWidth \* 1.05  
  
 if self.\_\_swellOnHover and not self.\_\_hovered:  
 if self.\_\_rect.height > self.\_\_origHeight and \  
 self.\_\_rect.width > self.\_\_origWidth:  
 self.shrink()  
 if self.\_\_rect.height < self.\_\_origHeight or \  
 self.\_\_rect.width < self.\_\_origWidth:  
 self.\_\_rect.height = self.\_\_origHeight  
 self.\_\_rect.width = self.\_\_origWidth  
  
 *# If button is not in a container, it can be drawn normally* if not self.\_\_container:  
 self.draw()  
  
 self.\_\_timer += time.time() - self.\_\_previousFrame  
 self.\_\_previousFrame = time.time()  
  
 def handleEvent(self, event):  
 *""" Due to the nature of pygame's events, you cannot have a for loop  
 in the class update() method that checks for events, as each event can  
 only be handled once, as it is effectively destroyed on handling.  
 This means that you need a separate event handling function, which you  
 can then pass events to from the main pygame loop.  
  
 """* if self.\_\_hovered and self.\_\_timer > 0.5:  
 *# Only checks for events if the pointer is on the button* if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 if self.\_\_swellOnHover:  
 *# Return button to normal size* self.\_\_rect.height = self.\_\_origHeight  
 self.\_\_rect.width = self.\_\_origWidth  
 self.\_\_action()  
 pg.mouse.set\_visible(True)  
 self.\_\_hovered = False  
 self.\_\_timer = 0  
 return True  
  
  
*# noinspection PyArgumentList*class Checkbox(pg.sprite.Sprite):  
  
 *# noinspection SpellCheckingInspection* def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,  
 textColour=None, inlineText=None, height=None, container=None):  
 super().\_\_init\_\_()  
 self.\_\_parent = parent  
 self.\_\_screen = screen  
 self.\_\_hovered = False  
 self.\_\_output = False  
  
 try: *# Coordinate input validation* x = int(x)  
 y = int(y)  
 except ValueError:  
 raise Exception(**"Checkbox coordinates must be integers"**)  
  
 **"""Almost all attributes have default values that will be assigned if  
 no values are passed on instantiation.  
  
 """** if not font:  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 30)  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if not bgColour:  
 self.\_\_bgColour = (255, 255, 255)  
 else:  
 self.\_\_bgColour = bgColour  
  
 if not textColour:  
 self.\_\_textColour = (0, 0, 0)  
 else:  
 self.\_\_textColour = textColour  
  
 if not inlineText:  
 self.\_\_inlineText = **""** else:  
 self.\_\_inlineText = inlineText  
  
 *# Renders inlineText as a pygame surface object* self.\_\_inlineDisplayText = self.\_\_font.render(self.\_\_inlineText, True,  
 self.\_\_textColour)  
  
 if not height:  
 *# Height and width of checkbox scales with the font* self.\_\_height = self.\_\_inlineDisplayText.get\_rect().h \* 1.25  
 self.\_\_width = self.\_\_height  
 else:  
 self.\_\_height = self.\_\_width = height  
  
 if container is None:  
 self.\_\_container = None  
 else:  
 self.\_\_container = container  
 self.\_\_container.addWidget(self)  
  
 self.\_\_coords = (x, y)  
  
 *# Aligns text with the checkbox* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 *# Default output is blank (False)* self.\_\_outputDisplay = self.\_\_font.render(**""**, True, (0, 0, 0))  
  
 *# Ordering of widgets - Containers first, then labels, dropdowns,  
 # input boxes, checkboxes, then buttons. Helps with widgets handling  
 # events in the correct order - stops buttons 'hijacking' click  
 # events from dropdown menus drawn on top of them.* index = 0  
 inGroup = False  
 for i in self.\_\_parent.pgkGroup.sprites():  
 if isinstance(i, Button):  
 after = self.\_\_parent.pgkGroup.sprites()[index:]  
 for sprite in after:  
 self.\_\_parent.pgkGroup.remove(sprite)  
 self.\_\_parent.pgkGroup.add(self)  
 for sprite in after:  
 self.\_\_parent.pgkGroup.add(sprite)  
 inGroup = True  
 index += 1  
  
 if not inGroup:  
 self.\_\_parent.pgkGroup.add(self)  
  
 def config(self, font=None, bgColour=None, textColour=None,  
 inlineText=None):  
 *# Allows modification of attributes that were not assigned during  
 # instantiation* if not font:  
 pass  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if not bgColour:  
 pass  
 else:  
 self.\_\_bgColour = bgColour  
  
 if not textColour:  
 pass  
 else:  
 self.\_\_textColour = textColour  
  
 if not inlineText:  
 pass  
 else:  
 self.\_\_inlineText = inlineText  
  
 self.\_\_inlineDisplayText = self.\_\_font.render(self.\_\_inlineText, True,  
 self.\_\_textColour)  
  
 self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(self.\_\_rect.x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 self.\_\_rect.y + 0.45 \* self.\_\_height))  
  
 *# Needs different outputDisplay depending on whether or not checkbox  
 # has been activated* if self.\_\_output:  
 *# self.\_\_parent.isLight function checks background colour - white cross on dark  
 # backgrounds, black on light backgrounds.* if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_outputDisplay = self.\_\_parent.pgkBlackCrossImage  
 else:  
 self.\_\_outputDisplay = self.\_\_parent.pgkWhiteCrossImage  
  
 *# Scales cross to fill the checkbox, no matter what size* self.\_\_outputDisplay = pg.transform.scale(self.\_\_outputDisplay,  
 (floor(self.\_\_width),  
 floor(self.\_\_height)))  
  
 else:  
 self.\_\_outputDisplay = self.\_\_font.render(**""**, True, (0, 0, 0))  
  
 def click(self):  
 if self.\_\_output:  
 self.\_\_output = False  
 self.\_\_outputDisplay = self.\_\_font.render(**""**, True,  
 (0, 0, 0))  
 else:  
 self.\_\_output = True  
  
 if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_outputDisplay = self.\_\_parent.pgkBlackCrossImage  
 else:  
 self.\_\_outputDisplay = self.\_\_parent.pgkWhiteCrossImage  
  
 self.\_\_outputDisplay = pg.transform.scale(  
 self.\_\_outputDisplay, (floor(self.\_\_width),  
 floor(self.\_\_height)))  
  
 def delete(self):  
 pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 if self.\_\_container:  
 x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
 *# Aligns text with the checkbox* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 self.\_\_screen.blit(self.\_\_outputDisplay, self.\_\_rect)  
  
 if self.\_\_hovered:  
 *# PointerRect is modified in order to appear similarly to how the  
 # pointer appears in windows, with the top of the hand in line  
 # with the location that the mouse is pointing to.* self.\_\_pointerRect = self.\_\_parent.pgkPointerCursor.get\_rect(  
 top=pg.mouse.get\_pos()[1])  
 self.\_\_pointerRect.x = pg.mouse.get\_pos()[  
 0] - self.\_\_pointerRect.w / 2  
 self.\_\_screen.blit(self.\_\_parent.pgkPointerCursor,  
 self.\_\_pointerRect)  
  
 self.\_\_screen.blit(self.\_\_inlineDisplayText, self.\_\_inlineTextRect)  
  
 def get(self):  
 return self.\_\_output  
  
 def getHeight(self):  
 return self.\_\_height  
  
 def getWidth(self):  
 return self.\_\_width  
  
 def getPos(self):  
 return self.\_\_rect.x, self.\_\_rect.y  
  
 *# noinspection PyAttributeOutsideInit* def update(self):  
 if self.\_\_rect.collidepoint(pg.mouse.get\_pos()) and not self.\_\_hovered:  
 *# Can only be hovered over if checkbox is not obstructed by  
 # container mask* if (self.\_\_container and not self.\_\_container.mouseMasked()) or \  
 not self.\_\_container:  
 *# Sets mouse cursor to invisible* pg.mouse.set\_visible(False)  
 self.\_\_hovered = True  
  
 elif not self.\_\_rect.collidepoint(  
 pg.mouse.get\_pos()) and self.\_\_hovered:  
 *# Sets mouse cursor back to default* pg.mouse.set\_visible(True)  
 self.\_\_hovered = False  
  
 if not self.\_\_container:  
 self.draw()  
  
 def handleEvent(self, event):  
 *# Mouse click event is only checked for when the pointer is hovering  
 # over the checkbox* if self.\_\_hovered and event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 self.click()  
 return True  
  
  
class Container(pg.sprite.Sprite):  
 def \_\_init\_\_(self, parent, screen, outlineColour=None,  
 outlineThickness=None, bg=False, bgColour=None,  
 height=None, width=None, topleft=None, topright=None,  
 bottomleft=None, bottomright=None, centre=None,  
 startVisible=True, maskColour=None):  
 super().\_\_init\_\_()  
 self.\_\_parent = parent  
 self.\_\_screen = screen  
  
 self.\_\_widgets = pg.sprite.Group()  
 self.\_\_animation = [None, None, None]  
  
 if outlineColour is None:  
 self.\_\_outlineColour = (33, 33, 33)  
 else:  
 self.\_\_outlineColour = outlineColour  
  
 if outlineThickness is None:  
 self.\_\_outlineThickness = 0  
 else:  
 if outlineThickness < 0:  
 outlineThickness = 0  
 self.\_\_outlineThickness = outlineThickness  
  
 if bg:  
 self.\_\_bg = True  
 if bgColour is None:  
 self.\_\_bgColour = (222, 222, 222)  
 else:  
 self.\_\_bgColour = bgColour  
 else:  
 self.\_\_bg = False  
  
 if height is None:  
 self.\_\_height = 200  
 else:  
 self.\_\_height = height  
  
 if width is None:  
 self.\_\_width = 100  
 else:  
 self.\_\_width = width  
  
 *# In order to create the outline, I will have two rects - outlineRect  
 # and rect. outlineRect will be the colour of the outline, and will  
 # be slightly bigger than rect. It will be drawn first, with rect on  
 # top, so that only the edges will show, giving the appearance of an  
 # outline.* self.\_\_fullHeight = self.\_\_height + self.\_\_outlineThickness \* 2  
 self.\_\_halfHeight = int(self.\_\_fullHeight / 2)  
 self.\_\_fullWidth = self.\_\_width + self.\_\_outlineThickness \* 2  
 self.\_\_halfWidth = int(self.\_\_fullWidth / 2)  
  
 self.\_\_rect = pg.Rect(0, 0, self.\_\_width, self.\_\_height)  
 self.\_\_outlineRect = pg.Rect(0, 0, self.\_\_fullWidth,  
 self.\_\_fullHeight)  
  
 if maskColour is None:  
 self.\_\_maskColour = (255, 255, 255)  
 else:  
 self.\_\_maskColour = maskColour  
  
 if topleft is not None:  
 self.\_\_rect.topleft = (int(topleft[0]), int(topleft[1]))  
  
 elif topright is not None:  
 self.\_\_rect.topright = (int(topright[0]), int(topright[1]))  
  
 elif bottomleft is not None:  
 self.\_\_rect.bottomleft = (int(bottomleft[0]), int(bottomleft[1]))  
  
 elif bottomright is not None:  
 self.\_\_rect.bottomright = (int(bottomright[0]), int(bottomright[1]))  
  
 elif centre is not None:  
 self.\_\_rect.center = (int(centre[0]), int(centre[1]))  
  
 self.\_\_outlineRect.center = self.\_\_rect.center  
  
 *# In order to create the illusion of the container hiding widgets,  
 # I will use a 'mask' rect that is the same size as the container -  
 # this will be set to the same colour as the background in the main  
 # part of my code, making a seamless look* if startVisible:  
 self.\_\_maskLeftRect = pg.Rect(0, 0, 0, 0)  
 self.\_\_maskRightRect = pg.Rect(0, 0, 0, 0)  
 self.\_\_maskTopRect = pg.Rect(0, 0, 0, 0)  
 self.\_\_maskBottomRect = pg.Rect(0, 0, 0, 0)  
  
 elif not startVisible:  
 self.\_\_maskLeftRect = pg.Rect(0, 0, self.\_\_halfWidth,  
 self.\_\_fullHeight)  
 self.\_\_maskRightRect = pg.Rect(0, 0, self.\_\_halfWidth,  
 self.\_\_fullHeight)  
 self.\_\_maskTopRect = pg.Rect(0, 0, self.\_\_fullWidth,  
 self.\_\_halfHeight)  
 self.\_\_maskBottomRect = pg.Rect(0, 0, self.\_\_fullWidth,  
 self.\_\_halfHeight)  
  
 self.\_\_maskLeftRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskRightRect.topright = self.\_\_outlineRect.topright  
 self.\_\_maskTopRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskBottomRect.bottomleft = self.\_\_outlineRect.bottomleft  
  
 *# This attribute will be set in the startAnimation method, and will  
 # determine whether or not the container - and widgets inside of it -  
 # will be deleted once the animation has finished. This means there  
 # is no need for loads of timing variables in the main code.* self.\_\_deleteAfter = False  
  
 if len(self.\_\_parent.pgkGroup.sprites()) > 0:  
 after = self.\_\_parent.pgkGroup.sprites()[1:]  
 for sprite in after:  
 self.\_\_parent.pgkGroup.remove(sprite)  
 self.\_\_parent.pgkGroup.add(self)  
 for sprite in after:  
 self.\_\_parent.pgkGroup.add(sprite)  
  
 else:  
 self.\_\_parent.pgkGroup.add(self)  
  
 self.\_\_previousFrame = time.time()  
  
 def config(self, outlineColour=None,  
 outlineThickness=None, bg=False, bgColour=None,  
 height=None, width=None, topleft=None, topright=None,  
 bottomleft=None, bottomright=None, centre=None,  
 maskColour=None):  
  
 if outlineColour is None:  
 pass  
 else:  
 self.\_\_outlineColour = outlineColour  
  
 if outlineThickness is None:  
 pass  
 else:  
 if outlineThickness < 0:  
 outlineThickness = 0  
 self.\_\_outlineThickness = outlineThickness  
  
 if bg:  
 self.\_\_bg = True  
 if bgColour is None:  
 self.\_\_bgColour = (222, 222, 222)  
 else:  
 self.\_\_bgColour = bgColour  
 else:  
 pass  
  
 if height is None:  
 pass  
 else:  
 self.\_\_height = height  
  
 if width is None:  
 pass  
 else:  
 self.\_\_width = width  
  
 *# In order to create the outline, I will have two rects - outlineRect  
 # and rect. outlineRect will be the colour of the outline, and will  
 # be slightly bigger than rect. It will be drawn first, with rect on  
 # top, so that only the edges will show, giving the appearance of an  
 # outline.* self.\_\_fullHeight = self.\_\_height + self.\_\_outlineThickness \* 2  
 self.\_\_halfHeight = int(self.\_\_fullHeight / 2)  
 self.\_\_fullWidth = self.\_\_width + self.\_\_outlineThickness \* 2  
 self.\_\_halfWidth = int(self.\_\_fullWidth / 2)  
  
 if maskColour is None:  
 pass  
 else:  
 self.\_\_maskColour = maskColour  
  
 if topleft is not None:  
 self.\_\_rect.topleft = (int(topleft[0]), int(topleft[1]))  
  
 elif topright is not None:  
 self.\_\_rect.topright = (int(topright[0]), int(topright[1]))  
  
 elif bottomleft is not None:  
 self.\_\_rect.bottomleft = (int(bottomleft[0]), int(bottomleft[1]))  
  
 elif bottomright is not None:  
 self.\_\_rect.bottomright = (int(bottomright[0]), int(bottomright[1]))  
  
 elif centre is not None:  
 self.\_\_rect.center = (int(centre[0]), int(centre[1]))  
  
 self.\_\_outlineRect.center = self.\_\_rect.center  
  
 self.\_\_maskLeftRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskRightRect.topright = self.\_\_outlineRect.topright  
 self.\_\_maskTopRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskBottomRect.bottomleft = self.\_\_outlineRect.bottomleft  
  
 def addWidget(self, widget):  
 self.\_\_widgets.add(widget)  
  
 def delete(self):  
 self.\_\_parent.pgkGroup.remove(self)  
 del self  
  
 def getCorrectedCoords(self, coords):  
 newX = self.\_\_rect.topleft[0] + coords[0]  
 newY = self.\_\_rect.topleft[1] + coords[1]  
  
 return newX, newY  
  
 def getRect(self):  
 return self.\_\_rect  
  
 def animationDone(self):  
 if self.\_\_animation == [None, None, None]:  
 return True  
 else:  
 return False  
  
 def handleEvent(self, event):  
 pass  
  
 def isEmpty(self):  
 if len(self.\_\_widgets.sprites()) == 0:  
 return True  
 else:  
 return False  
  
 def isMasked(self):  
 if (self.\_\_maskRightRect.width == self.\_\_halfWidth and  
 self.\_\_maskLeftRect.width == self.\_\_halfWidth) or \  
 (self.\_\_maskTopRect.height == self.\_\_halfHeight and  
 self.\_\_maskBottomRect.height == self.\_\_halfHeight):  
 return True  
 else:  
 return False  
  
 def mouseMasked(self):  
 *# If the mouse is blocked by the masks, then this will be True - used  
 # to prevent clickthrough, allowing the user to interact with buttons  
 # even when they are hidden by the masks* mouseCoords = pg.mouse.get\_pos()  
 for i in [self.\_\_maskBottomRect, self.\_\_maskLeftRect,  
 self.\_\_maskTopRect, self.\_\_maskRightRect]:  
 if i.collidepoint(mouseCoords[0], mouseCoords[1]):  
 return True  
 return False  
  
 def onScreen(self):  
 if self.\_\_animation[0] is None:  
 sw, sh = pg.display.get\_surface().get\_size()  
 if self.\_\_rect.bottom <= 0:  
 return False  
 elif self.\_\_rect.top >= sh:  
 return False  
 elif self.\_\_rect.topright[0] <= 0:  
 return False  
 elif self.\_\_rect.topleft[0] >= sw:  
 return False  
 else:  
 return True  
  
 return True  
  
 def removeWidget(self, widget):  
 self.\_\_widgets.remove(widget)  
  
 def startAnimation(self, type, time, inOut, startFrom=None,  
 deleteAfter=None, destination=None):  
 self.\_\_animation = [type, time, inOut]  
  
 self.\_\_deleteAfter = deleteAfter  
 if startFrom is not None:  
 *# startFrom is needed for animations that involve the whole  
 # container moving* self.\_\_animation.append(startFrom)  
  
 *# Can now specify where the container will finish its animation,  
 # instead of always finishing where it was when the animation was  
 # started. Useful if a container needs to slide off screen to the  
 # left and come back on from the right, for example.* if destination:  
 self.\_\_rect.x = destination[0]  
 self.\_\_rect.y = destination[1]  
  
 self.\_\_animation.append(self.\_\_rect.x)  
 self.\_\_animation.append(self.\_\_rect.y)  
  
 if type in [**"horizontalslide"**, **"verticalslide"**]:  
 *# Don't need masks for these animations, so set them all to be  
 # invisible* self.\_\_maskLeftRect.width = 0  
 self.\_\_maskRightRect.width = 0  
 self.\_\_maskTopRect.height = 0  
 self.\_\_maskBottomRect.height = 0  
  
 if type == **"horizontalslide"** and inOut != **"out"**:  
 self.\_\_rect.x = startFrom  
  
 elif type == **"verticalslide"** and inOut != **"out"**:  
 self.\_\_rect.y = startFrom  
  
 def centreAnimation(self):  
 *# Container, and widgets inside of it, will appear from the centre  
 # outwards.* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
  
 *# How much does the width/height change every frame?* multiplier = self.\_\_frameTime / time  
  
 *# Divide by 2 as each mask takes up half of its dimension* widthChange = int(self.\_\_fullWidth \* multiplier / 2)  
 heightChange = int(self.\_\_fullHeight \* multiplier / 2)  
  
 *# At small resolutions and high frame rates, these values will  
 # occasionally be calculated to be 0, meaning the masks will not change  
 # sizes. In that case, they need to be set to 1* if widthChange < 1:  
 widthChange = 1  
 if heightChange < 1:  
 heightChange = 1  
  
 *# If the container is appearing* if inOut == **"in"**:  
 *# Widths/heights of masks cannot go lower than 0, otherwise they  
 # would extend out the other way, resulting in longer times for  
 # disappearing animations* if self.\_\_maskLeftRect.width > 0:  
 self.\_\_maskLeftRect.width -= widthChange  
 else:  
 self.\_\_maskLeftRect.width = 0  
  
 if self.\_\_maskRightRect.width > 0:  
 self.\_\_maskRightRect.width -= widthChange  
 else:  
 self.\_\_maskRightRect.width = 0  
  
 if self.\_\_maskTopRect.height > 0:  
 self.\_\_maskTopRect.height -= heightChange  
 else:  
 self.\_\_maskTopRect.height = 0  
  
 if self.\_\_maskBottomRect.height > 0:  
 self.\_\_maskBottomRect.height -= heightChange  
 else:  
 self.\_\_maskBottomRect.height = 0  
  
 *# If all masks have completely disappeared, the animation is  
 # finished* if self.\_\_maskLeftRect.width == 0 and \  
 self.\_\_maskRightRect.width == 0 and \  
 self.\_\_maskTopRect.height == 0 and \  
 self.\_\_maskBottomRect.height == 0:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 elif inOut == **"out"**:  
 *# Same as for when the container is appearing, but invert the  
 # width/height change* widthChange \*= -1  
 heightChange \*= -1  
  
 *# This time check dimensions of masks against the dimensions of  
 # the container (divided by 2 as each mask takes up half of the  
 # container)* if self.\_\_maskLeftRect.width < self.\_\_halfWidth:  
 self.\_\_maskLeftRect.width -= widthChange  
 else:  
 self.\_\_maskLeftRect.width = self.\_\_halfWidth  
  
 if self.\_\_maskRightRect.width < self.\_\_halfWidth:  
 self.\_\_maskRightRect.width -= widthChange  
 else:  
 self.\_\_maskRightRect.width = self.\_\_halfWidth  
  
 if self.\_\_maskTopRect.height < self.\_\_halfHeight:  
 self.\_\_maskTopRect.height -= heightChange  
 else:  
 self.\_\_maskTopRect.height = self.\_\_halfHeight  
  
 if self.\_\_maskBottomRect.height < self.\_\_halfHeight:  
 self.\_\_maskBottomRect.height -= heightChange  
 else:  
 self.\_\_maskBottomRect.height = self.\_\_halfHeight  
  
 *# If masks are completely blocking container, animation is finished* if self.\_\_maskLeftRect.width == self.\_\_halfWidth and \  
 self.\_\_maskRightRect.width == self.\_\_halfWidth and \  
 self.\_\_maskTopRect.height == self.\_\_halfHeight and \  
 self.\_\_maskBottomRect.height == self.\_\_halfHeight:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 def closeSideAnimation(self):  
 *# Appears as two 'sliding doors'* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 multiplier = self.\_\_frameTime \* 1 / time  
  
 *# This animation only deals with width, so no need to calculate  
 # height change* widthChange = self.\_\_fullWidth \* multiplier / 2  
  
 *# Set top and bottom masks to have a height of 0 - makes them  
 # invisible, as they are not needed for this animation* self.\_\_maskTopRect.height = 0  
 self.\_\_maskBottomRect.height = 0  
  
 if inOut == **"in"**:  
 *# Very similar to the centre animation, except only using  
 # maskLeft and maskRight* if self.\_\_maskLeftRect.width > 0:  
 self.\_\_maskLeftRect.width -= widthChange  
 else:  
 self.\_\_maskLeftRect.width = 0  
  
 if self.\_\_maskRightRect.width > 0:  
 self.\_\_maskRightRect.width -= widthChange  
 else:  
 self.\_\_maskRightRect.width = 0  
  
 if self.\_\_maskLeftRect.width == 0 and \  
 self.\_\_maskRightRect.width == 0:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif inOut == **"out"**:  
 *# Again, very similar to the appearance animation, except invert  
 # the width change value* widthChange \*= -1  
  
 if self.\_\_maskLeftRect.width < self.\_\_halfWidth:  
 self.\_\_maskLeftRect.width -= widthChange  
 else:  
 self.\_\_maskLeftRect.width = self.\_\_halfWidth  
  
 if self.\_\_maskRightRect.width < self.\_\_halfWidth:  
 self.\_\_maskRightRect.width -= widthChange  
 else:  
 self.\_\_maskRightRect.width = self.\_\_halfWidth  
  
 if self.\_\_maskLeftRect.width == self.\_\_halfWidth and \  
 self.\_\_maskRightRect.width == self.\_\_halfWidth:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 def closeUpAnimation(self):  
 *# Appears the same as slideSideAnimation, but flipped by 90 degrees* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 multiplier = self.\_\_frameTime \* 1 / time  
  
 *# This one only deals with height, so no need to calculate widthChange* heightChange = self.\_\_fullHeight \* multiplier  
  
 *# Set the width of the left mask and the right mask to 0, making them  
 # invisible, as they are not needed for this animation* self.\_\_maskLeftRect.width = 0  
 self.\_\_maskRightRect.width = 0  
  
 if inOut == **"in"**:  
  
 if self.\_\_maskTopRect.height > 0:  
 self.\_\_maskTopRect.height -= heightChange  
 else:  
 self.\_\_maskTopRect.height = 0  
  
 if self.\_\_maskBottomRect.height > 0:  
 self.\_\_maskBottomRect.height -= heightChange  
 else:  
 self.\_\_maskBottomRect.height = 0  
  
 if self.\_\_maskTopRect.height == 0 and \  
 self.\_\_maskBottomRect.height == 0:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif inOut == **"out"**:  
 heightChange \*= -1  
  
 if self.\_\_maskTopRect.height < self.\_\_halfHeight:  
 self.\_\_maskTopRect.height -= heightChange  
 else:  
 self.\_\_maskTopRect.height = self.\_\_halfHeight  
  
 if self.\_\_maskBottomRect.height < self.\_\_halfHeight:  
 self.\_\_maskBottomRect.height -= heightChange  
 else:  
 self.\_\_maskBottomRect.height = self.\_\_halfHeight  
  
 if self.\_\_maskTopRect.height == self.\_\_halfHeight and \  
 self.\_\_maskBottomRect.height == self.\_\_halfHeight:  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 def horizontalSlideAnimation(self):  
 *# Whole container slides in from the left or right* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 startFrom = self.\_\_animation[3]  
 destination = self.\_\_animation[4]  
 distance = destination - startFrom  
 multiplier = self.\_\_frameTime / time  
  
 *# Calculate how much the container should move this frame* posStep = abs(distance \* multiplier)  
  
 if (inOut == **"in"** and destination < startFrom) or \  
 (inOut == **"out"** and destination > startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.x > destination:  
 *# Step needs to be able to round to 1 (greater than 0.5)  
 # otherwise container won't move at all* if posStep > 0.5:  
 self.\_\_rect.x -= posStep  
 else:  
 self.\_\_rect.x -= 1  
 else:  
 self.\_\_rect.x = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif (inOut == **"in"** and destination > startFrom) or \  
 (inOut == **"out"** and destination < startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.x < destination:  
 if posStep > 0.5:  
 self.\_\_rect.x += posStep  
 else:  
 self.\_\_rect.x += 1  
 else:  
 self.\_\_rect.x = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 def verticalSlideAnimation(self):  
 *# Whole container slides in from the top/bottom* time = self.\_\_animation[1]  
 inOut = self.\_\_animation[2]  
 startFrom = self.\_\_animation[3]  
 destination = self.\_\_animation[5]  
 distance = destination - startFrom  
 multiplier = self.\_\_frameTime / time  
  
 *# Calculate how much the container should move this frame* posStep = abs(distance \* multiplier)  
  
 if (inOut == **"in"** and destination < startFrom) or \  
 (inOut == **"out"** and destination > startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.y > destination:  
 *# Step needs to be able to round to 1 (greater than 0.5)  
 # otherwise container won't move at all* if posStep > 0.5:  
 self.\_\_rect.y -= posStep  
 else:  
 self.\_\_rect.y -= 1  
 else:  
 self.\_\_rect.y = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
  
 elif (inOut == **"in"** and destination > startFrom) or \  
 (inOut == **"out"** and destination < startFrom):  
 if inOut == **"out"**:  
 destination = startFrom  
 if self.\_\_rect.y < destination:  
 if posStep > 0.5:  
 self.\_\_rect.y += posStep  
 else:  
 self.\_\_rect.y += 1  
 else:  
 self.\_\_rect.y = destination  
 self.\_\_animation = [None, None, None]  
  
 if self.\_\_deleteAfter:  
 for widget in self.\_\_widgets:  
 widget.delete()  
 self.delete()  
  
 def update(self):  
 *# Only draw outline rect if the outline thickness is greater than 0 -  
 # no point otherwise, as it will be completely obscured by the  
 # container rect* if self.\_\_outlineThickness > 0:  
 pg.draw.rect(self.\_\_screen, self.\_\_outlineColour,  
 self.\_\_outlineRect)  
 if self.\_\_bg:  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 *# Widgets contained within the container need to be drawn here so  
 # that they will be on top of the container, but below the masks* for widget in self.\_\_widgets:  
 widget.draw()  
  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskLeftRect)  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskRightRect)  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskTopRect)  
 pg.draw.rect(self.\_\_screen, self.\_\_maskColour, self.\_\_maskBottomRect)  
  
 self.\_\_frameTime = time.time() - self.\_\_previousFrame  
 self.\_\_previousFrame = time.time()  
  
 *# All different animation types - only check if the container should  
 # be playing an animation* if self.\_\_animation[0]:  
 if self.\_\_animation[0] == **'centre'**:  
 self.centreAnimation()  
 elif self.\_\_animation[0] == **'closeside'**:  
 self.closeSideAnimation()  
 elif self.\_\_animation[0] == **'closeup'**:  
 self.closeUpAnimation()  
 elif self.\_\_animation[0] == **'horizontalslide'**:  
 self.horizontalSlideAnimation()  
 elif self.\_\_animation[0] == **'verticalslide'**:  
 self.verticalSlideAnimation()  
  
 self.\_\_maskLeftRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskRightRect.topright = self.\_\_outlineRect.topright  
 self.\_\_maskTopRect.topleft = self.\_\_outlineRect.topleft  
 self.\_\_maskBottomRect.bottomleft = self.\_\_outlineRect.bottomleft  
  
  
*# Will be used for dropdown menus (in cases where there are multiple options  
# to select from)*class Dropdown(pg.sprite.Sprite):  
  
 def \_\_init\_\_(self, parent, screen, x, y, options, font=None,  
 bgColour=None, inlineText=None, width=None,  
 container=None):  
 super().\_\_init\_\_()  
 self.\_\_parent = parent  
 self.\_\_screen = screen  
 self.\_\_options = options  
  
 *# Need to save the original order of the options list, so that it  
 # remains the same order when items are selected* self.\_\_originalOptions = options.copy()  
  
 self.\_\_currentOption = options[0]  
  
 if font is None:  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 30)  
 self.\_\_arrowFont = pg.font.SysFont(**"Helvetica"**, 20)  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
 self.\_\_arrowFont = pg.font.SysFont(font[0], int(font[1] \* 2 / 3))  
  
 if bgColour is None:  
 self.\_\_bgColour = (255, 255, 255)  
 else:  
 self.\_\_bgColour = bgColour  
  
 if inlineText is None:  
 self.\_\_inlineText = **""** else:  
 self.\_\_inlineText = inlineText  
  
 if width is None:  
 self.\_\_width = int(pg.display.get\_surface().get\_width() / 10)  
 else:  
 self.\_\_width = width  
  
 if container is None:  
 self.\_\_container = None  
 else:  
 self.\_\_container = container  
 self.\_\_container.addWidget(self)  
  
 if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_textColour = (0, 0, 0)  
 else:  
 self.\_\_textColour = (255, 255, 255)  
  
 self.\_\_hoverColour = self.\_\_parent.hoverEffect(self.\_\_bgColour)  
  
 self.\_\_inlineDisplayText = self.\_\_font.render(self.\_\_inlineText, True,  
 self.\_\_textColour)  
  
 *# Scales height based on size of text* self.\_\_height = self.\_\_inlineDisplayText.get\_rect().h \* 1.25  
  
 *# Using a dict for rendered text objects - easier to retrieve when  
 # needed than using indices in a list* self.\_\_optionDisplays = { }  
 for i in self.\_\_options:  
 self.\_\_optionDisplays[i] = self.\_\_font.render(i, True,  
 self.\_\_textColour)  
  
 *# Using a list for rects as there only needs to be 6 (current option  
 # + 5 others)* self.\_\_rects = []  
 for i in range(0, 5):  
 self.\_\_rects.append(pg.Rect(x, y + self.\_\_height \* i, self.\_\_width,  
 self.\_\_height))  
  
 self.\_\_coords = (x, y)  
  
 *# Aligns text with input box* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_sideArrow = self.\_\_arrowFont.render(**">"**, True, self.\_\_textColour)  
 self.\_\_downArrow = pg.transform.rotate(self.\_\_sideArrow, -90)  
 self.\_\_upArrow = pg.transform.rotate(self.\_\_sideArrow, 90)  
  
 bottomRight = self.\_\_rects[0].bottomright  
 self.\_\_arrowRect = self.\_\_upArrow.get\_rect(bottomright=bottomRight)  
  
 self.\_\_expanded = False  
 self.\_\_hovered = False  
  
 self.\_\_hoverRect = self.\_\_rects[0]  
  
 *# These are the indices for the range of items that will be displayed  
 # when the dropdown is expanded* self.\_\_lower = 1  
 self.\_\_upper = 5  
  
 index = 0  
 inGroup = False  
 for i in self.\_\_parent.pgkGroup.sprites():  
 if isinstance(i, Button):  
 after = self.\_\_parent.pgkGroup.sprites()[index:]  
 for sprite in after:  
 self.\_\_parent.pgkGroup.remove(sprite)  
 self.\_\_parent.pgkGroup.add(self)  
 for sprite in after:  
 self.\_\_parent.pgkGroup.add(sprite)  
 inGroup = True  
 index += 1  
  
 if not inGroup:  
 self.\_\_parent.pgkGroup.add(self)  
  
 def config(self, options=None, font=None, bgColour=None, textColour=None,  
 inlineText=None, width=None, container=None):  
  
 if options is None:  
 pass  
 else:  
 self.\_\_options = options  
 *# Need to save the original order of the options list, so that it  
 # remains the same order when items are selected* self.\_\_originalOptions = options.copy()  
 self.\_\_currentOption = options[0]  
  
 *# Using a dict for rendered text objects - easier to retrieve when  
 # needed than using indices in a list* self.\_\_optionDisplays = { }  
 for i in self.\_\_options:  
 self.\_\_optionDisplays[i] = self.\_\_font.render(i, True,  
 self.\_\_textColour)  
  
 if font is None:  
 pass  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if bgColour is None:  
 pass  
 else:  
 self.\_\_bgColour = bgColour  
  
 if textColour is None:  
 pass  
 else:  
 self.\_\_textColour = textColour  
  
 if inlineText is None:  
 pass  
 else:  
 self.\_\_inlineText = inlineText  
  
 if width is None:  
 pass  
 else:  
 self.\_\_width = width  
  
 if container is None:  
 pass  
 else:  
 self.\_\_container = container  
 self.\_\_container.addWidget(self)  
  
 self.\_\_inlineDisplayText = self.\_\_font.render(self.\_\_inlineText, True,  
 self.\_\_textColour)  
  
 *# Scales height based on size of text* self.\_\_height = self.\_\_inlineDisplayText.get\_rect().h \* 1.25  
  
 x, y = self.\_\_coords[0], self.\_\_coords[1]  
  
 *# Aligns text with input box* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 def delete(self):  
 pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 if self.\_\_container and self.\_\_coords != tuple(  
 self.\_\_container.getCorrectedCoords(self.\_\_coords)):  
 *# Correct coordinates relative to container's topleft corner -  
 # only if container has moved since last frame in order to save  
 # performance* x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
 mult = 0  
 for rect in self.\_\_rects:  
 rect.x = x  
 rect.y = y + self.\_\_height \* mult  
 mult += 1  
  
 inlineTextWidth = self.\_\_inlineDisplayText.get\_rect().w  
 self.\_\_inlineTextRect.center = (x - inlineTextWidth \* 0.6,  
 y + 0.45 \* self.\_\_height)  
  
 bottomRight = self.\_\_rects[0].bottomright  
 self.\_\_arrowRect = self.\_\_upArrow.get\_rect(bottomright=bottomRight)  
  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rects[0])  
 self.\_\_screen.blit(self.\_\_optionDisplays[self.\_\_currentOption],  
 self.\_\_rects[0])  
  
 self.\_\_screen.blit(self.\_\_inlineDisplayText, self.\_\_inlineTextRect)  
  
 if self.\_\_expanded:  
 num = 1  
 for i in self.\_\_options[self.\_\_lower:self.\_\_upper]:  
 *# Draws the other 5 rects - and displays the text for the 5  
 # shown options on top* if self.\_\_rects[num].center == self.\_\_hoverRect.center and \  
 num != 0:  
 pg.draw.rect(self.\_\_screen, self.\_\_hoverColour,  
 self.\_\_rects[num])  
 else:  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour,  
 self.\_\_rects[num])  
 self.\_\_screen.blit(self.\_\_optionDisplays[i],  
 self.\_\_rects[num])  
 num += 1  
  
 self.\_\_screen.blit(self.\_\_upArrow, self.\_\_arrowRect)  
  
 else:  
 self.\_\_screen.blit(self.\_\_downArrow, self.\_\_arrowRect)  
  
 if self.\_\_hovered:  
 pg.mouse.set\_visible(False)  
 *# Draws custom mouse image over mouse location* self.\_\_pointerRect = self.\_\_parent.pgkPointerCursor.get\_rect(  
 top=pg.mouse.get\_pos()[1])  
 self.\_\_pointerRect.x = pg.mouse.get\_pos()[  
 0] - self.\_\_pointerRect.w / 2  
 self.\_\_screen.blit(self.\_\_parent.pgkPointerCursor,  
 self.\_\_pointerRect)  
  
 def get(self):  
 return self.\_\_currentOption  
  
 def getHeight(self):  
 return self.\_\_height  
  
 def setSelected(self, selected):  
 self.\_\_currentOption = selected  
  
 def handleEvent(self, event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 if self.\_\_hovered and not self.\_\_expanded:  
 self.\_\_expanded = True  
 return True  
 elif self.\_\_hovered and self.\_\_expanded:  
 *# If user clicks on an option, then current option will  
 # be set to the option that is currently held within the  
 # rect that the user has clicked on* index = self.\_\_rects.index(self.\_\_hoverRect)  
 if index == 0:  
 self.\_\_expanded = False  
 return True  
  
 try:  
 self.\_\_currentOption = self.\_\_options[self.\_\_lower +  
 index - 1]  
 except IndexError:  
 *# If an index error is thrown, then there aren't  
 # enough options in the dropdown menu to reach the  
 # mouse pointer. Therefore we return False as the  
 # event hasn't been handled* return False  
  
 *# Returns option list back to its original order,  
 # and moves the selected option to the front of the list  
 # (so that it will be drawn in the first rect, and shown  
 # even when the dropdown is not expanded)* self.\_\_options = self.\_\_originalOptions.copy()  
 self.\_\_options.remove(self.\_\_currentOption)  
 self.\_\_options.insert(0, self.\_\_currentOption)  
 self.\_\_expanded = False  
 return True  
 elif not self.\_\_hovered:  
 self.\_\_expanded = False  
  
 elif event.type == MOUSEBUTTONDOWN:  
 if self.\_\_expanded:  
 if event.button == 4:  
 *# Scroll wheel up* if self.\_\_lower > 1:  
 self.\_\_upper -= 1  
 self.\_\_lower -= 1  
 return True  
 elif event.button == 5:  
 *# Scroll wheel down* if self.\_\_upper < len(self.\_\_options):  
 self.\_\_upper += 1  
 self.\_\_lower += 1  
 return True  
  
 def update(self):  
 if self.\_\_rects[0].collidepoint(pg.mouse.get\_pos()) and not \  
 self.\_\_hovered:  
 *# Can only be hovered over if dropdown is not obstructed by  
 # container mask* if (self.\_\_container and not self.\_\_container.mouseMasked()) or \  
 not self.\_\_container:  
 self.\_\_hovered = True  
  
 if self.\_\_expanded:  
 self.\_\_hovered = False  
 for rect in self.\_\_rects:  
 if rect.collidepoint(pg.mouse.get\_pos()):  
 self.\_\_hovered = True  
 self.\_\_hoverRect = rect  
 if not self.\_\_hovered:  
 pg.mouse.set\_visible(True)  
  
 elif not self.\_\_rects[0].collidepoint(pg.mouse.get\_pos()) and \  
 self.\_\_hovered and not self.\_\_expanded:  
 *# Sets mouse cursor back to default* pg.mouse.set\_visible(True)  
 self.\_\_hovered = False  
  
 *# If dropdown is not in a container, it can be drawn normally* if not self.\_\_container:  
 self.draw()  
  
  
*# noinspection PyArgumentList,PyArgumentList*class InputBox(pg.sprite.Sprite):  
  
 def \_\_init\_\_(self, parent, screen, x, y, font=None, bgColour=None,  
 textColour=None, inlineText=None, width=None,  
 allowLetters=True, allowNumbers=True, allowMaths=True,  
 allowSpecial=True, allowSpace=True, charLimit=None,  
 defaultEntry=None, container=None):  
 super().\_\_init\_\_()  
 self.\_\_parent = parent  
 self.\_\_screen = screen  
 self.\_\_outputText = **""** self.\_\_cursorText = **""** self.\_\_timer = 1  
 self.\_\_backspaceTimer = 1  
 self.\_\_backspaceFirstPress = False  
 self.\_\_backspaceDelay = 0.05  
 self.\_\_previousFrame = time.time()  
 self.\_\_hovered = False  
 self.\_\_active = False  
 self.\_\_caps = False  
  
 self.\_\_allowedChars = []  
  
 try:  
 x = int(x)  
 y = int(y)  
 except ValueError:  
 raise Exception(**"InputBox coordinates must be integers"**)  
  
 if not font:  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 30)  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if not bgColour:  
 self.\_\_bgColour = (255, 255, 255)  
 else:  
 self.\_\_bgColour = bgColour  
  
 if not textColour:  
 self.\_\_textColour = (0, 0, 0)  
 else:  
 self.\_\_textColour = textColour  
  
 if not inlineText:  
 self.\_\_inlineText = **""** else:  
 self.\_\_inlineText = inlineText  
  
 if not width:  
 self.\_\_width = int(pg.display.get\_surface().get\_width() / 10)  
 else:  
 self.\_\_width = width  
  
 *# Acts as built-in input validation - only certain characters will be  
 # allowed to be inputted  
 # By default these attributes are set to True (in the arguments for  
 # \_\_init\_\_), meaning they will be allowed.* if allowLetters:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkLetterChars)  
  
 if allowNumbers:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkNumberChars)  
  
 if allowMaths:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkMathsChars)  
  
 if allowSpecial:  
 self.\_\_allowedChars += list(self.\_\_parent.pgkSpecialChars)  
  
 if allowSpace:  
 self.\_\_allowedChars += [**" "**]  
  
 if not charLimit:  
 *# By default, the character limit is set much higher than anybody  
 # would realistically need - prevents errors when comparing to  
 # string length* self.\_\_charLimit = 1000000  
 else:  
 self.\_\_charLimit = charLimit  
  
 if defaultEntry is None:  
 self.\_\_outputText = **""** else:  
 self.\_\_outputText = defaultEntry  
  
 self.\_\_inlineDisplayText = self.\_\_font.render(self.\_\_inlineText, True,  
 self.\_\_textColour)  
  
 *# Scales height based on size of text* self.\_\_height = self.\_\_inlineDisplayText.get\_rect().h \* 1.25  
  
 self.\_\_coords = (x, y)  
  
 *# Aligns text with input box* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_outputTextDisplay = self.\_\_font.render(self.\_\_outputText,  
 True, (0, 0, 0))  
 else:  
 self.\_\_outputTextDisplay = self.\_\_font.render(self.\_\_outputText,  
 True, (255, 255, 255))  
  
 if container is None:  
 self.\_\_container = None  
 else:  
 self.\_\_container = container  
 self.\_\_container.addWidget(self)  
  
 index = 0  
 inGroup = False  
 for i in self.\_\_parent.pgkGroup.sprites():  
 if isinstance(i, Button):  
 after = self.\_\_parent.pgkGroup.sprites()[index:]  
 for sprite in after:  
 self.\_\_parent.pgkGroup.remove(sprite)  
 self.\_\_parent.pgkGroup.add(self)  
 for sprite in after:  
 self.\_\_parent.pgkGroup.add(sprite)  
 inGroup = True  
 index += 1  
  
 if not inGroup:  
 self.\_\_parent.pgkGroup.add(self)  
  
 def config(self, font=None, bgColour=None, textColour=None, inlineText=None,  
 width=None, allowLetters=None, allowNumbers=None,  
 allowSpecial=None, charLimit=None):  
  
 *# Same functionality as other config methods* if not font:  
 pass  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if not bgColour:  
 pass  
 else:  
 self.\_\_bgColour = bgColour  
  
 if not textColour:  
 pass  
 else:  
 self.\_\_textColour = textColour  
  
 if not inlineText:  
 pass  
 else:  
 self.\_\_inlineText = inlineText  
  
 if not width:  
 pass  
 else:  
 self.\_\_width = width  
 self.\_\_rect = pg.Rect(self.\_\_rect.x, self.\_\_rect.y, self.\_\_width,  
 self.\_\_height)  
  
 if allowLetters is None:  
 pass  
 else:  
 self.\_\_allowLetters = allowLetters  
  
 if allowNumbers is None:  
 pass  
 else:  
 self.\_\_allowNumbers = allowNumbers  
  
 if allowSpecial is None:  
 pass  
 else:  
 self.\_\_allowSpecial = allowSpecial  
  
 if not charLimit:  
 pass  
 else:  
 self.\_\_charLimit = charLimit  
  
 self.\_\_inlineDisplayText = self.\_\_font.render(self.\_\_inlineText, True,  
 self.\_\_textColour)  
 self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(self.\_\_rect.x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 self.\_\_rect.y + 0.45 \* self.\_\_height))  
  
 def delete(self):  
 pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 if self.\_\_container:  
 x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
 *# Aligns text with the checkbox* self.\_\_inlineTextRect = self.\_\_inlineDisplayText.get\_rect(  
 center=(x - self.\_\_inlineDisplayText.get\_rect().w \* 0.6,  
 y + 0.45 \* self.\_\_height))  
  
 self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 if self.\_\_hovered:  
 pg.mouse.set\_visible(False)  
 *# Draws typing cursor on location of the mouse pointer* typingRect = self.\_\_parent.pgkTypingCursor.get\_rect(  
 center=pg.mouse.get\_pos())  
 self.\_\_screen.blit(self.\_\_parent.pgkTypingCursor, typingRect)  
  
 self.\_\_screen.blit(self.\_\_outputTextDisplay, self.\_\_rect)  
 self.\_\_screen.blit(self.\_\_inlineDisplayText, self.\_\_inlineTextRect)  
  
 def get(self):  
 try:  
 if self.\_\_outputText[-1] == **"|"**:  
 return self.\_\_outputText[:-1]  
 else:  
 return self.\_\_outputText  
 except IndexError:  
 return **""** def getHeight(self):  
 return self.\_\_height  
  
 def getWidth(self):  
 return self.\_\_width  
  
 def getPos(self):  
 return (self.\_\_rect.x, self.\_\_rect.y)  
  
 def write(self, text):  
 self.\_\_outputText = text  
  
 def update(self):  
 if self.\_\_active:  
 *# Backspace needs to be placed in the update loop as it uses an  
 # updated method, which allows the user to hold it down to  
 # delete long sections of text.* keyPressed = pg.key.get\_pressed() *# Dict of pressed keys* if keyPressed[K\_BACKSPACE] and self.\_\_backspaceTimer >= \  
 self.\_\_backspaceDelay:  
 try:  
 self.\_\_outputText = self.\_\_outputText[:-1]  
  
 except IndexError:  
 *# In case there are no characters in output* pass  
 self.\_\_backspaceTimer = 0  
  
 *# Need a longer delay after first press - user won't  
 # accidentally delete multiple characters* if self.\_\_backspaceFirstPress:  
 self.\_\_backspaceDelay = 0.5  
 self.\_\_backspaceFirstPress = False  
 else:  
 self.\_\_backspaceDelay = 0.05  
  
 *# Timer is used to add and remove the "|" character every 0.5  
 # seconds, giving the appearance that the cursor is flashing when  
 # typing - acts as feedback that the inputBox is active.* if self.\_\_active and self.\_\_timer > 0.5 and \  
 self.\_\_cursorText == **"|"**:  
 self.\_\_cursorText = **""** elif self.\_\_active and self.\_\_timer < 0.5 and \  
 self.\_\_cursorText != **"|"**:  
 self.\_\_cursorText = **"|"** elif self.\_\_timer > 1 and self.\_\_cursorText != **"|"**:  
 self.\_\_timer = 0  
  
 *# OutputTextDisplay needs to be rendered on every update due to the  
 # fact that output can change* if self.\_\_parent.isLight(self.\_\_bgColour):  
 self.\_\_outputTextDisplay = self.\_\_font.render(self.\_\_outputText +  
 self.\_\_cursorText,  
 True, (0, 0, 0))  
 else:  
 self.\_\_outputTextDisplay = self.\_\_font.render(self.\_\_outputText +  
 self.\_\_cursorText,  
 True, (255, 255, 255))  
  
 if self.\_\_rect.collidepoint(pg.mouse.get\_pos()) and not self.\_\_hovered:  
 *# Can only be hovered over if checkbox is not obstructed by  
 # container mask* if (self.\_\_container and not self.\_\_container.mouseMasked()) or \  
 not self.\_\_container:  
 pg.mouse.set\_visible(False)  
 self.\_\_hovered = True  
  
 elif not self.\_\_rect.collidepoint(  
 pg.mouse.get\_pos()) and self.\_\_hovered:  
 pg.mouse.set\_visible(True)  
 self.\_\_hovered = False  
  
 if not self.\_\_container:  
 self.draw()  
  
 self.\_\_timer += time.time() - self.\_\_previousFrame  
 self.\_\_backspaceTimer += time.time() - self.\_\_previousFrame  
 self.\_\_previousFrame = time.time()  
  
 *# noinspection SpellCheckingInspection* def handleEvent(self, event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 if self.\_\_hovered and not self.\_\_active:  
 self.\_\_active = True  
 self.\_\_timer = 0  
 return True  
 elif not self.\_\_hovered and self.\_\_active:  
 *# Sets cursorText to an empty string - shows that box is  
 # not active* self.\_\_cursorText = **""** self.\_\_active = False  
  
 *# Will only check keydown events if checkbox is active* if self.\_\_active:  
 if event.type == KEYUP:  
 if event.key == K\_BACKSPACE:  
 self.\_\_backspaceDelay = 0.05  
 return True  
  
 if event.type == KEYDOWN:  
 if event.key == K\_BACKSPACE:  
 self.\_\_backspaceFirstPress = True  
  
 elif event.unicode == **"**\x16**"**: *# Unicode value for ctrl+v (  
 # paste)* text = []  
 try:  
 for i in paste(): *# from pyClipTools* if i in self.\_\_allowedChars:  
 pass  
 else:  
 text.append(i)  
 text = **''**.join(text)  
  
 except TypeError:  
 text = **''** self.\_\_outputText += text  
  
 elif event.unicode in self.\_\_allowedChars:  
 self.\_\_outputText += event.unicode  
  
 try:  
 if len(self.\_\_outputText) > self.\_\_charLimit:  
 self.\_\_outputText = self.\_\_outputText[:-1]  
 else:  
 pass  
  
 except IndexError:  
 pass  
  
 return True  
  
  
class Label(pg.sprite.Sprite):  
 def \_\_init\_\_(self, parent, screen, font=None, bgColour=None,  
 textColour=None, text=None, width=None, height=None,  
 container=None, topleft=None, topright=None, centre=None,  
 bottomleft=None, bottomright=None):  
 super().\_\_init\_\_()  
 self.\_\_parent = parent  
 self.\_\_screen = screen  
  
 if not font:  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 30)  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if not bgColour:  
 self.\_\_bgColour = None  
 else:  
 self.\_\_bgColour = bgColour  
  
 if not textColour:  
 self.\_\_textColour = (0, 0, 0)  
 else:  
 self.\_\_textColour = textColour  
  
 if not text:  
 self.\_\_text = **""** else:  
 *# In case I need multi-line text (string will contain newline  
 # character). Text gets broken up into a list of lines* self.\_\_text = [line for line in text.split(**'**\n**'**)]  
  
 *# Create list of rendered text surface objects* self.\_\_displayText = []  
 for text in self.\_\_text:  
 self.\_\_displayText.append(self.\_\_font.render(text, True,  
 self.\_\_textColour))  
  
 if not width:  
 self.\_\_width = self.\_\_displayText[0].get\_rect().w \* 1.25  
 else:  
 self.\_\_width = width  
  
 if not height:  
 self.\_\_height = self.\_\_displayText[0].get\_rect().h \* 1.25  
 else:  
 self.\_\_height = height  
  
 self.\_\_rect = pg.Rect(0, 0, self.\_\_width, self.\_\_height)  
  
 if topleft is not None:  
 self.\_\_rect.topleft = (int(topleft[0]), int(topleft[1]))  
  
 elif topright is not None:  
 self.\_\_rect.topright = (int(topright[0]), int(topright[1]))  
  
 elif bottomleft is not None:  
 self.\_\_rect.bottomleft = (int(bottomleft[0]), int(bottomleft[1]))  
  
 elif bottomright is not None:  
 self.\_\_rect.bottomright = (int(bottomright[0]), int(bottomright[1]))  
  
 elif centre is not None:  
 self.\_\_rect.center = (int(centre[0]), int(centre[1]))  
  
 *# Positioning the text rects* self.\_\_textRects = []  
 textGap = int(self.\_\_displayText[0].get\_rect().h \* 1.25)  
 num = 0  
  
 for i in self.\_\_displayText:  
 self.\_\_textRects.append(i.get\_rect(centerx=(self.\_\_rect.left +  
 self.\_\_width / 2),  
 top=(self.\_\_rect.top +  
 textGap \* num)))  
 num += 1  
  
 self.\_\_coords = self.\_\_rect.topleft  
  
 if container is None:  
 self.\_\_container = None  
 else:  
 self.\_\_container = container  
 self.\_\_container.addWidget(self)  
  
 index = 0  
 inGroup = False  
 for i in self.\_\_parent.pgkGroup.sprites():  
 if isinstance(i, Button):  
 after = self.\_\_parent.pgkGroup.sprites()[index:]  
 for sprite in after:  
 self.\_\_parent.pgkGroup.remove(sprite)  
 self.\_\_parent.pgkGroup.add(self)  
 for sprite in after:  
 self.\_\_parent.pgkGroup.add(sprite)  
 inGroup = True  
 index += 1  
  
 if not inGroup:  
 self.\_\_parent.pgkGroup.add(self)  
  
 def config(self, font=None, bgColour=None, textColour=None, text=None,  
 width=None, height=None):  
  
 *# Same functionality as other config methods* if not font:  
 pass  
 else:  
 self.\_\_font = pg.font.SysFont(font[0], font[1])  
  
 if not bgColour:  
 pass  
 else:  
 self.\_\_bgColour = bgColour  
  
 if not textColour:  
 pass  
 else:  
 self.\_\_textColour = textColour  
  
 if not text:  
 pass  
 else:  
 self.\_\_text = [line for line in text.split(**'**\n**'**)]  
  
 self.\_\_displayText = []  
 for text in self.\_\_text:  
 self.\_\_displayText.append(self.\_\_font.render(text, True,  
 self.\_\_textColour))  
  
 if not width:  
 pass  
 else:  
 self.\_\_width = width  
  
 if not height:  
 pass  
 else:  
 self.\_\_height = height  
  
 self.\_\_rect = pg.Rect(self.\_\_rect.x, self.\_\_rect.y, self.\_\_width,  
 self.\_\_height)  
  
 self.\_\_textRects = []  
 textGap = int(self.\_\_displayText[0].get\_rect().h \* 1.25)  
 num = 0  
  
 for i in self.\_\_displayText:  
 self.\_\_textRects.append(i.get\_rect(centerx=(self.\_\_rect.left +  
 self.\_\_width / 2),  
 top=(self.\_\_rect.top +  
 textGap \* num)))  
 num += 1  
  
 def delete(self):  
 pg.mouse.set\_cursor(\*pg.cursors.arrow)  
 self.\_\_parent.pgkGroup.remove(self)  
  
 if self.\_\_container:  
 self.\_\_container.removeWidget(self)  
  
 del self  
  
 def draw(self):  
 if self.\_\_container:  
 x, y = self.\_\_container.getCorrectedCoords(self.\_\_coords)  
  
 *# Aligns text with the container* self.\_\_rect = pg.Rect(x, y, self.\_\_width, self.\_\_height)  
 self.\_\_textRects = []  
 textGap = int(self.\_\_displayText[0].get\_rect().h \* 1.25)  
 num = 0  
  
 for i in self.\_\_displayText:  
 self.\_\_textRects.append(i.get\_rect(centerx=(self.\_\_rect.left +  
 self.\_\_width / 2),  
 top=(self.\_\_rect.top +  
 textGap \* num)))  
 num += 1  
  
 if self.\_\_bgColour is not None:  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 index = 0  
 for i in self.\_\_textRects:  
 self.\_\_screen.blit(self.\_\_displayText[index], i)  
 index += 1  
  
 def update(self):  
 pass  
  
 def handleEvent(self, event):  
 pass

#### Main Code

import math  
import time  
import random  
import os  
import ast  
import winsound  
from pathlib import Path  
  
try:  
 from win32api import GetSystemMetrics  
  
 win = True  
  
except ImportError:  
 *# If user is not on a windows system, win32api will fail to import,   
 # and the program will instead use default window size* win = False  
  
import pygame as pg  
import pygame.math as pgmath  
from pygame.locals import \*  
  
*# An external library that I made - adds tkinter features in pygame*import pgkinter as pgk  
  
pgkRoot = pgk.Pgk()  
  
  
*# Used to draw a line that follows a particle's path*class Graph(object):  
 def \_\_init\_\_(self, surface, width, height, topleft, bgColour, xLabelGap,  
 yLabelGap):  
 self.\_\_screen = surface  
  
 self.\_\_width = width  
  
 self.\_\_height = height  
  
 self.\_\_bgColour = bgColour  
  
 self.\_\_rect = pg.Rect(0, 0, self.\_\_width, self.\_\_height)  
  
 self.\_\_rect.topleft = (int(topleft[0]), int(topleft[1]))  
  
 self.\_\_xLabels = []  
 x = width  
 while x > 0:  
 self.\_\_xLabels.append(width - x)  
 x -= xLabelGap  
  
 self.\_\_yLabels = []  
 y = height  
 while y > 0:  
 self.\_\_yLabels.append(height - y)  
 y -= yLabelGap  
  
 self.\_\_font = pg.font.SysFont(**"Helvetica"**, 18)  
  
 self.lines = []  
  
 def draw(self):  
  
 pg.draw.rect(self.\_\_screen, self.\_\_bgColour, self.\_\_rect)  
  
 for i in self.\_\_xLabels:  
 pg.draw.line(self.\_\_screen, (170, 170, 170),  
 (i, self.\_\_rect.top), (i, self.\_\_rect.bottom), 1)  
  
 for i in self.\_\_yLabels:  
 pg.draw.line(self.\_\_screen, (170, 170, 170),  
 (self.\_\_rect.left, i), (self.\_\_rect.right, i), 1)  
  
 for i in self.lines:  
 i.draw()  
  
 def changeLabelGap(self, xLabelGap, yLabelGap):  
 self.\_\_xLabels = []  
 x = self.\_\_width  
 while x > 0:  
 self.\_\_xLabels.append(self.\_\_width - x)  
 x -= xLabelGap  
  
 self.\_\_yLabels = []  
 y = 0  
 while y < self.\_\_height:  
 self.\_\_yLabels.append(self.\_\_height - y)  
 y += yLabelGap  
  
 def clearLines(self):  
 for i in self.lines:  
 del i  
  
 self.lines = []  
  
  
class Line(object):  
 def \_\_init\_\_(self, surface, graph, colour):  
 self.\_\_screen = surface  
  
 graph.lines.append(self)  
  
 self.\_\_colour = colour  
  
 self.\_\_plotCoords = []  
  
 def draw(self):  
 plots = self.\_\_plotCoords  
  
 if len(plots) > 1:  
 pg.draw.lines(self.\_\_screen, self.\_\_colour, False, plots, 2)  
  
 def addPlot(self, plot):  
 self.\_\_plotCoords.append((int(plot[0]), int(plot[1])))  
  
  
class Particle(pg.sprite.Sprite):  
  
 def \_\_init\_\_(self, coefficient, material, rad, density, v, colour, centre,  
 yA, xA=None):  
 super().\_\_init\_\_() *# Runs pygame sprite \_\_init\_\_() method* self.hasRandomVelocity = False  
 self.line = False  
 self.restCoefficient = coefficient  
 self.material = material  
 self.radius = rad  
 self.density = density  
 self.mass = 0  
 self.vol = 0  
 self.updateDimension(rad=self.radius)  
 self.velocity = pgmath.Vector2(v)  
 self.colour = colour  
 self.rect = pg.draw.circle(screen, self.colour, centre,  
 int(self.radius \* scale))  
 self.rect.x += int(self.radius \* scale)  
 self.rect.y += int(self.radius \* scale)  
 self.pos = pgmath.Vector2(self.rect.x, self.rect.y)  
 if not xA: *# Assigns 0 as default x acceleration, if no value is passed* self.acceleration = pgmath.Vector2(0, yA)  
 else:  
 self.acceleration = pgmath.Vector2(xA, yA)  
  
 self.direction = 0  
 self.updateDirection()  
  
 *# Dictionary storing data on each frame* self.posDict = {  
 0: (pgmath.Vector2(centre), pgmath.Vector2(v), tNow)  
 }  
 self.recentCollisions = []  
  
 def angleTo(self, p2):  
 xDistance = self.pos.x - p2.pos.x  
 yDistance = self.pos.y - p2.pos.y  
  
 *# Uses inverse tan function on two lengths. Need to add pi/2 (90deg)  
 # in order to align with pygame angles (measured from vertical)* return math.atan2(xDistance, yDistance) + math.pi / 2  
  
 def collide(self, p2):  
 collisionDir = self.angleTo(p2)  
  
 *# Particles have collided - calculate new velocities* m1, m2 = self.mass, p2.mass  
 v1, v2 = self.velocity, p2.velocity  
 pos1, pos2 = self.pos, p2.pos  
  
 selfNewVelocity = v1 - ((2 \* m2) / (m1 + m2)) \* (  
 ((v1 - v2).dot(pos1 - pos2)) / (  
 (pos1 - pos2).length()) \*\* 2) \* (pos1 - pos2)  
 p2NewVelocity = v2 - ((2 \* m1) / (m2 + m1)) \* (  
 ((v2 - v1).dot(pos2 - pos1)) / (  
 (pos2 - pos1).length()) \*\* 2) \* (pos2 - pos1)  
  
 self.velocity, p2.velocity = selfNewVelocity, p2NewVelocity  
  
 def delete(self):  
 particles.remove(self)  
 del self  
  
 def draw(self):  
 pg.draw.circle(screen, self.colour, (self.rect.x, self.rect.y),  
 int(self.radius \* scale))  
  
 def drawDirectionArrow(self):  
 arrow = ARROW\_IMAGE *# Prevents having to load image every time* arrowscale = (self.radius \* scale / arrow.get\_width()) \* 1.5  
 *# Rotate and scale arrow image to fit particle* arrow = pg.transform.rotozoom(arrow, math.degrees(self.direction),  
 arrowscale)  
 arrowRect = arrow.get\_rect(center=(self.rect.x, self.rect.y))  
 screen.blit(arrow, arrowRect)  
  
 def hasCollided(self, group):  
 collisionList = []  
 *# Checks each sprite in the group and if the sum of their radii is less  
 # than the absolute distance between their centres, they have collided.* for sprite in group.sprites():  
 totalRad = self.radius \* scale + sprite.radius \* scale  
 if absoluteDistance(self.pos, sprite.pos) <= totalRad \  
 and sprite != self:  
 collisionList.append(sprite)  
  
 return collisionList  
  
 def scalePosition(self):  
 *# If the scale has changed since last frame, then the particle will  
 # get resized and relocated, so that it will be in the same place  
 # relative to the window and other particles.* if previousScale != scale:  
 mouseX = pg.mouse.get\_pos()[0]  
 fromMouseX = mouseX - self.pos.x  
 self.pos.x = mouseX - (fromMouseX \* (scale / previousScale))  
 fromFloor = SH - self.pos.y  
 self.pos.y = SH - (fromFloor \* (scale / previousScale))  
 self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 else:  
 pass  
  
 def updateDirection(self):  
 *# Gets direction of travel (in radians)  
 # Multiply by -1 as pygame measures angles anti-clockwise.* self.direction = math.atan2(self.velocity.y, self.velocity.x) \* -1  
  
 def updateDimension(self, rad=None, mass=None):  
 *# Apply equation v=(4/3)\*pi\*r^2 and v=m/d to update particles radius if  
 # mass is changed, and vice-versa* if rad is not None:  
 self.radius = rad  
 self.vol = (4 / 3) \* math.pi \* (self.radius \*\* 3)  
 self.mass = roundToSigFig(self.vol \* self.density, 3)  
  
 elif mass is not None:  
 self.mass = mass  
 self.vol = self.mass / self.density  
 rad = ((3 \* self.vol) / (4 \* math.pi)) \*\* (1 / 3)  
 self.radius = roundToSigFig(rad, 3)  
  
 def update(self):  
 global frameNumber  
 global tNow  
  
 self.scalePosition()  
  
 *# If time is moving backwards* if TIME\_SCALES[currentTimescale] < 0:  
 try:  
 *# Sets attributes to what they were at the current frame number* p = self.posDict[frameNumber]  
 self.pos = pgmath.Vector2(p[0].x, p[0].y)  
 self.velocity = pgmath.Vector2(p[1].x, p[1].y)  
 tNow = p[2]  
  
 except KeyError:  
 pass  
  
 self.updateDirection()  
  
 self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 self.draw()  
 self.drawDirectionArrow()  
  
 *# If time is moving forward* elif TIME\_SCALES[currentTimescale] > 0:  
  
 *# Only simulate particle's motion if current frame has not  
 # already been simulated, otherwise simply retrieve its  
 # positional data from the position dictionary* if frameNumber not in self.posDict:  
 *# Checks if particle has collided with floor. Also checks  
 # that particle's velocity would take it towards the floor  
 # in order to make sure that the particle hasn't collided and  
 # was turned around in the previous frame.* if self.pos.y + self.radius \* scale >= SH and \  
 self.velocity.y \* timeMultiplier > 0:  
 self.velocity.y = roundToSigFig(  
 self.velocity.y \* self.restCoefficient \* -1, 4)  
 self.pos.y = SH - self.radius \* scale  
  
 *# Has collided with ceiling* elif self.pos.y - self.radius \* scale <= 0 and \  
 self.velocity.y \* timeMultiplier < 0:  
 self.velocity.y = roundToSigFig(  
 self.velocity.y \* self.restCoefficient \* -1, 4)  
 self.pos.y = 0 + self.radius \* scale  
  
 *# Has collided with right wall* if self.pos.x + self.radius \* scale >= SW and \  
 self.velocity.x \* timeMultiplier > 0:  
 self.velocity.x = roundToSigFig(  
 self.velocity.x \* self.restCoefficient \* -1, 4)  
 self.pos.x = SW - self.radius \* scale  
  
 *# Has collided with left wall* elif self.pos.x - self.radius \* scale <= 0 and \  
 self.velocity.x \* timeMultiplier < 0:  
 self.velocity.x = roundToSigFig(  
 self.velocity.x \* self.restCoefficient \* -1, 4)  
 self.pos.x = 0 + self.radius \* scale  
 *#  
 # Multiply by timeMultiplier in order to increase velocity by  
 # the correct amount per second.* self.velocity += self.acceleration \* timeMultiplier  
  
 self.updateDirection()  
  
 *# Multiply by scale as velocity is ms^-1, multiply by  
 # timeMultiplier for the same reasons as before* self.pos += self.velocity \* scale \* timeMultiplier  
  
 *# Rect coordinates need to be integers* self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 self.draw()  
 self.drawDirectionArrow()  
 collisionList = self.hasCollided(particles)  
 for particle in collisionList:  
 *# Self is included in collisionList, therefore != self  
 # check is required  
  
 # Also need to check if the particles are moving closer,  
 # otherwise two that collided last frame wil be treated as  
 # colliding again this frame.* if particle != self and particle not in \  
 self.recentCollisions:  
 *# Need to both add the other particle to this  
 # particle's recentCollisions list, and the other way  
 # round. If I just added the other particle to this  
 # particle's collisions list, they would  
 # occasionally 'collide' twice, as the if not in  
 # self.recentCollisions check would pass in both  
 # particles' update methods* self.collide(particle)  
 self.recentCollisions.append(particle)  
 particle.recentCollisions.append(self)  
  
 for particle in self.recentCollisions:  
 if particle not in collisionList:  
 self.recentCollisions.remove(particle)  
 particle.recentCollisions.remove(self)  
  
 if self.line:  
 self.line.addPlot((self.pos.x, self.pos.y))  
  
 elif frameNumber in self.posDict:  
 *# If current frame has already been simulated, grab values  
 # from posDict* p = self.posDict[frameNumber]  
 self.pos = pgmath.Vector2(p[0].x, p[0].y)  
 self.velocity = pgmath.Vector2(p[1].x, p[1].y)  
 tNow = p[2]  
 self.updateDirection()  
  
 self.rect.x, self.rect.y = int(self.pos.x), int(self.pos.y)  
 self.draw()  
 self.drawDirectionArrow()  
  
 p = pgmath.Vector2(self.pos.x, self.pos.y)  
 v = pgmath.Vector2(self.velocity.x, self.velocity.y)  
  
 *# If time is set to x2 speed* if currentTimescale == 5:  
 *# Need to add data to previous 1 and 1.5 frames  
 # Interpolates what the position and velocity will be based  
 # on current velocity and acceleration* if (frameNumber - 1.5) not in self.posDict:  
 olderPos = p - (v \* scale \* timeMultiplier \* 1.5)  
 self.posDict[frameNumber - 1.5] = (olderPos, v, tNow)  
  
 if (frameNumber - 1) not in self.posDict:  
 oldPos = p - (v \* scale \* timeMultiplier)  
 self.posDict[frameNumber - 1] = (oldPos, v, tNow)  
  
 *# x1 or x2* if currentTimescale in [4, 5]:  
 *# Need to add data to previous 1/2 of a frame  
 # Interpolates what the position and velocity will be based  
 # on current velocity and acceleration* if (frameNumber - 0.5) not in self.posDict:  
 oldPos = p - (  
 v \* scale \* timeMultiplier \* 0.5)  
 self.posDict[frameNumber - 0.5] = (oldPos, v, tNow)  
  
 if frameNumber not in self.posDict:  
 *# Will always add data for the current frame if it is not  
 # already in the dictionary* self.posDict[frameNumber] = (p, v, tNow)  
  
 *# All of these checks mean that every frame (and the half frames  
 # in between them) will be included in the dictionary, allowing  
 # the user to rewind through at any speed.*def absoluteDistance(pVector1, pVector2):  
 distance = pVector1 - pVector2  
 return distance.length()  
  
  
def drawDottedLine(start, end):  
 *# Used when resizing particles, to create the same look as in Blender  
 # (Dotted line from the centre of the object being resized to the mouse)* xLen = end[0] - start[0]  
 yLen = end[1] - start[1]  
  
 xStep = xLen / 10  
 yStep = yLen / 10  
  
 xCoord = start[0]  
 yCoord = start[1]  
  
 for i in range(0, 5):  
 pg.draw.line(screen, (33, 33, 33), (xCoord, yCoord),  
 (xCoord + xStep, yCoord + yStep), 2)  
 xCoord += 2 \* xStep  
 yCoord += 2 \* yStep  
  
  
*# Code snippet found online - rounds a number, x, to a given number of  
# significant figures, n.*def roundToSigFig(x, n):  
 if x != 0:  
 return round(x, -int(math.floor(math.log10(abs(x)))) + (n - 1))  
 else:  
 return 0  
  
  
def timeChange(speedUp):  
 global currentTimescale  
  
 *# User can only speed up if the current timescale is at most one less  
 # than the maximum. Similarly, they can only decrease the timescale if it  
 # is a least one more than the minimum. This is to prevent the timescale  
 # from going out of bounds.* if 0 <= currentTimescale < len(TIME\_SCALES) - 1 and speedUp == 1:  
 currentTimescale += 1  
 elif 0 < currentTimescale <= len(TIME\_SCALES) - 1 and speedUp == -1:  
 currentTimescale -= 1  
  
  
*# Each procedure which contains a loop needs to have at least one argument,  
# whether it is used or not, as the loop that prevents recursion needs to  
# pass an argument (it passes \*args, which cannot pass nothing). So I put a  
# dummy argument in the procedures that don't need anything to be passed to  
# them.*def mainMenu(dummyArg):  
 global mainmenu  
 global mainWidgets  
 global currentTimescale  
 global tNow  
 global frameNumber  
 global particleGraph  
 global scale  
 global previousScale  
  
 def endFunction(widgets, goTo, args):  
 *# Can't use buttons to set variables, so I need to use this function  
 # to set global variables* global nextFunc  
 global nextArgs  
 global mainmenu  
  
 if goTo == instructions:  
 instructions(\*args)  
 return  
  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW, True)  
  
 mainmenu, nextFunc, nextArgs = False, goTo, args  
  
 def exitProgram():  
 pg.quit()  
 quit()  
  
 global timeMultiplier  
  
 *# Reset scale and time-related globals to their default values - prevents  
 # bugs when going from a simulation to the menu, and then into another  
 # simulation* currentTimescale = 4  
 tNow = 0  
 frameNumber = 0  
  
 scale = scaler(100, **"x"**)  
 previousScale = scale  
  
 particleGraph.clearLines()  
 del particleGraph  
  
 particleGraph = Graph(screen, SW, SH, (0, 0), BG\_COLOUR, scale, scale)  
  
 mainContainer = pgk.Container(pgkRoot, screen, topleft=(0, 0),  
 width=SW, height=SH)  
  
 mainWidgets = [  
 pgk.Label(pgkRoot, screen, centre=(SW / 2, scaler(135, **"y"**)),  
 font=LARGE\_FONT,  
 text=**"Particle Simulator V3 by Jack Sanders"**,  
 container=mainContainer)  
 ]  
  
 buttonX = scaler(640, **"x"**)  
  
 mainWidgets += [  
 pgk.Button(pgkRoot, screen, buttonX, scaler(425, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Create a simulation"**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=lambda: endFunction(mainWidgets, setup, (1,)),  
 container=mainContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen, buttonX, scaler(560, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Load a saved simulation"**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=lambda: endFunction(mainWidgets, loadSetup,  
 (None,)),  
 container=mainContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen, buttonX, scaler(695, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Instructions"**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=lambda: endFunction(mainWidgets, instructions,  
 (mainWidgets,)),  
 container=mainContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen, buttonX, scaler(830, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Exit program :("**,  
 height=scaler(115, **"y"**), width=scaler(640, **"x"**),  
 action=exitProgram,  
 container=mainContainer, swellOnHover=True),  
 ]  
  
 mainWidgets.append(mainContainer)  
  
 del mainContainer  
  
 mainWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"in"**, SW)  
  
 mainmenu = True  
 while mainmenu:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 mainmenu = False  
 pg.quit()  
 quit()  
  
 try:  
 *# If statement prevents 'jumping' of particles when user moves  
 # the window* if time.time() - previousFrame < 0.1:  
 timeMultiplier = (time.time() - previousFrame)  
 timeMultiplier \*= TIME\_SCALES[currentTimescale]  
 *# Calculates time between frames* previousFrame = time.time()  
 except NameError:  
 *# Will occur on the first frame, as there is no previous frame* pass  
  
 screen.fill(BG\_COLOUR)  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 *# Returns the function that runs next (setup) and the args to pass to that   
 # function (\*args requires a tuple to unpack)* return nextFunc, nextArgs  
  
  
def instructions(mainMenuWidgets):  
 global instructing  
  
 def changePage(goToPage, pages, mainMenuWidgets):  
 global instructing  
  
 *# First page slides to the right and gets deleted, main menu slides  
 # back in from the left, and delete the second page* if goToPage == -1:  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, SW, True)  
 mainMenuWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0)  
 for i in pages[1]:  
 i.delete()  
  
 instructing = False  
  
 *# First page slides to the left, second page slides in from the right* elif goToPage == 0:  
 pages[1][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, SW)  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0)  
  
 *# Second page slides to the right, second page slides in from the left* elif goToPage == 1:  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW)  
 pages[1][-1].startAnimation(**"horizontalslide"**, 0.5, **"in"**, SW,  
 destination=(0, 0))  
  
 *# Second page slides to the left and gets deleted, main menu slides  
 # back in from the right, and delete the first page* elif goToPage == 2:  
 pages[1][-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW,  
 True)  
 mainMenuWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 0, destination=(SW, 0))  
 for i in pages[0]:  
 i.delete()  
  
 instructing = False  
  
 mainMenuWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**, 0 - SW)  
  
 files = [**"setupInstructions.txt"**, **"mainInstructions.txt"**]  
 pageTitles = [**"Setup"**, **"Simulation"**]  
 pages = []  
 currentPage = 0  
  
 for i in files:  
 *# Iterate through instruction files, creating labels and buttons* with open(i, **"r"**) as f:  
 lines = f.readlines()  
  
 text = **''**.join(lines)  
  
 pageContainer = pgk.Container(pgkRoot, screen, topleft=(0, SW),  
 width=SW, height=SH)  
 page = [  
 pgk.Label(pgkRoot, screen, centre=(SW / 2, scaler(135, **"y"**)),  
 font=LARGE\_FONT, text=pageTitles[currentPage],  
 container=pageContainer),  
 pgk.Label(pgkRoot, screen, centre=(SW / 2,  
 SH / 2 + scaler(135, **"y"**)),  
 height=SH - scaler(135, **"x"**), width=SW \* 0.8,  
 font=MID\_FONT, text=text, container=pageContainer),  
 ]  
  
 *# Need to explicitly state the page values for the buttons - can't  
 # use currentPage + or - 1. This is because lambda passes the  
 # arguments as they are at the time of the button press. In this case  
 # that means that the previous page button will always pass 1,  
 # and the next page will always pass 3.* if currentPage == 0:  
 page += [  
 pgk.Button(pgkRoot, screen, 0, 0, height=SH, width=int(SW / 10),  
 action=lambda: changePage(-1, pages,  
 mainMenuWidgets),  
 image=L\_MENU\_IMG, hoverImage=L\_MENU\_IMG,  
 container=pageContainer),  
 pgk.Button(pgkRoot, screen, SW - int(SW / 10), 0, height=SH,  
 width=int(SW / 10),  
 action=lambda: changePage(1, pages,  
 mainMenuWidgets),  
 image=NEXT\_IMG, hoverImage=NEXT\_IMG,  
 container=pageContainer),  
 pageContainer  
 ]  
  
 else:  
 page += [  
 pgk.Button(pgkRoot, screen, 0, 0, height=SH, width=int(SW / 10),  
 action=lambda: changePage(0, pages,  
 mainMenuWidgets),  
 image=PREV\_IMG, hoverImage=PREV\_IMG,  
 container=pageContainer),  
 pgk.Button(pgkRoot, screen, SW - int(SW / 10), 0, height=SH,  
 width=int(SW / 10),  
 action=lambda: changePage(2, pages,  
 mainMenuWidgets),  
 image=R\_MENU\_IMG, hoverImage=R\_MENU\_IMG,  
 container=pageContainer),  
 pageContainer  
 ]  
  
 del pageContainer  
  
 pages.append(page)  
  
 currentPage += 1  
  
 pages[0][-1].startAnimation(**"horizontalslide"**, 0.5, **"in"**, SW,  
 destination=(0, 0))  
  
 instructing = True  
 while instructing:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 instructing = False  
 pg.quit()  
 quit()  
  
 screen.fill(BG\_COLOUR)  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 return  
  
  
def setup(dummyArg):  
 global scale  
 global previousScale  
 global setting  
 global editingParticle  
 *# Controls setup loop* setting = True  
 *# List of widgets used when editing a placed particle* editList = None  
 *# The particle being edited* editingParticle = None  
  
 *# How many metres are shown in the scale display* metres = 1  
  
 def endFunction(widgetList):  
 *# Starts the setupContainer's slide out animation* widgetList[-1].startAnimation(**"horizontalslide"**, 0.1, **"out"**,  
 SW + contWidth, deleteAfter=True)  
  
 def endParticleEdit(editList):  
 global editingParticle  
 *# Menu for editing a particle will disappear* editList[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
 editingParticle = None  
  
 *# Delete all widgets - to start with a 'clean slate'  
 # for i in pgkRoot.pgkGroup.sprites():  
 # i.delete()* def updateParticle(pRef, inputs, editing=None):  
 *# Editing argument specifies whether or not the particle being  
 # updated has already been placed* try:  
 *# Easier to add new inputs - only need to change index here* coefficientBox = inputs[0]  
 coefficient = float(inputs[0].get())  
 xVel = float(inputs[1].get())  
 yVel = float(inputs[2].get())  
 xAccel = float(inputs[3].get())  
 yAccel = float(inputs[4].get())  
 radBox = inputs[5]  
 rad = float(radBox.get())  
 massBox = inputs[6]  
 mass = float(massBox.get())  
 height = float(inputs[7].get())  
 lockHeight = inputs[8].get()  
 randomV = inputs[9].get()  
 drawGraph = inputs[10].get()  
 material = inputs[11].get()  
 if material in MATERIALS:  
 density, colour = MATERIALS[material][0], MATERIALS[material][1]  
 else:  
 density, colour = customMaterials[material][0], \  
 customMaterials[material][1]  
  
 pRef.colour = colour  
 except ValueError:  
 return  
  
 mouseX = pg.mouse.get\_pos()[0]  
 mouseY = pg.mouse.get\_pos()[1]  
  
 if coefficient > 1:  
 coefficientBox.write(**"1"**)  
 elif coefficient < 0:  
 coefficientBox.write(**"0"**)  
  
 else:  
 pRef.restCoefficient = coefficient  
  
 if not randomV:  
 pRef.velocity.x = xVel  
 pRef.velocity.y = yVel  
 pRef.hasRandomVelocity = False  
 elif randomV and not pRef.hasRandomVelocity:  
 upper = pRef.radius \* 5  
 lower = upper \* -1  
  
 *# random.uniform instead of random.randint as uniform allows for  
 # two floating point numbers as the bounds* pRef.velocity.x = roundToSigFig(random.uniform(lower, upper), 3)  
 pRef.velocity.y = roundToSigFig(random.uniform(lower, upper), 3)  
  
 inputs[1].write(str(pRef.velocity.x))  
 inputs[2].write(str(pRef.velocity.y))  
 pRef.hasRandomVelocity = True  
  
 if not drawGraph and pRef.line:  
 pRef.line = False  
  
 elif drawGraph and not pRef.line:  
 pRef.line = Line(screen, particleGraph, colour)  
  
 pRef.acceleration.x = xAccel  
 pRef.acceleration.y = yAccel  
  
 *# If user has selected that they want the particle to be locked to a  
 # certain height* if lockHeight:  
 if editing is None:  
 pRef.pos.x = mouseX  
 pRef.rect.x = mouseX  
 pRef.pos.y = SH - int(height \* scale) - pRef.radius \* scale  
 pRef.rect.y = SH - int(height \* scale) - pRef.radius \* scale  
 else:  
 if editing is None:  
 pRef.pos.x = mouseX  
 pRef.pos.y = mouseY  
 pRef.rect.x = mouseX  
 pRef.rect.y = mouseY  
  
 minRad = roundToSigFig(scaler(10, **"x"**) / scale, 3)  
 maxRad = roundToSigFig((SW / 4) / scale, 3)  
  
 *# Only update particle's radius/mass if they are not already equal to  
 # the values entered by the user* if density != pRef.density:  
 pRef.density = density  
 pRef.colour = colour  
 pRef.updateDimension(rad=rad)  
 radBox.write(str(pRef.radius))  
 massBox.write(str(pRef.mass))  
  
 elif rad != pRef.radius and minRad <= rad <= maxRad:  
 pRef.updateDimension(rad=rad)  
 massBox.write(str(pRef.mass))  
  
 elif mass != pRef.mass:  
 pRef.updateDimension(mass=mass)  
 radBox.write(str(pRef.radius))  
  
 if pRef.radius > maxRad:  
 pRef.updateDimension(rad=roundToSigFig((SW / 4) / scale, 3))  
 radBox.write(str(pRef.radius))  
 massBox.write(str(pRef.mass))  
  
 elif pRef.radius < minRad:  
 pRef.updateDimension(rad=roundToSigFig(scaler(10, **"x"**) / scale, 3))  
 radBox.write(str(pRef.radius))  
 massBox.write(str(pRef.mass))  
  
 pRef.material = material  
  
 def deleteParticle(particle):  
 particle.delete()  
  
 def clearParticles():  
 for particle in particles.sprites():  
 particle.delete()  
  
 particles.add(Particle(1, **"Custom Material 1 - 1.0kgm^-3"**,  
 roundToSigFig((SW / 4) / scale, 3), 1, (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])), 0, xA=0))  
  
 setupContainer = pgk.Container(pgkRoot, screen,  
 topright=(SW, 0),  
 outlineThickness=0, width=scaler(400, **"x"**),  
 height=scaler(520, **"y"**))  
  
 contWidth = scaler(400, **"x"**)  
 offset = scaler(150, **"x"**)  
 boxWidth = scaler(125, **"x"**)  
 inputList = [  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(55, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Coefficient of Restitution:"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0.75"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(105, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Velocity to the right (ms^-1):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(155, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Velocity downwards (ms^-1):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(205, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Acceleration to the right (ms^-2):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(255, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Acceleration downwards (ms^-2):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(305, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Radius (m):"**, width=boxWidth, charLimit=10,  
 allowLetters=False, allowSpecial=False, allowSpace=False,  
 defaultEntry=**"0.5"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(355, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Mass (kg):"**, width=boxWidth, charLimit=10,  
 allowLetters=False, allowSpecial=False, allowSpace=False,  
 defaultEntry=**"0.5236"**, container=setupContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(405, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Height off of ground (m):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=setupContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 contWidth - scaler(50, **"x"**), scaler(455, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Lock particle to height: "**,  
 container=setupContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 contWidth - scaler(50, **"x"**), scaler(505, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Select for random velocity: "**,  
 container=setupContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 contWidth - scaler(50, **"x"**), scaler(555, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Draw line following particle's motion: "**,  
 container=setupContainer),  
  
 *# Create dropdown menu last as it needs to be drawn on top of the other  
 # inputs* pgk.Dropdown(pgkRoot, screen, contWidth - offset -  
 boxWidth, scaler(5, **"y"**), sortedCustoms + MATERIALS\_SORTED,  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Select material (scroll to see more):"**,  
 width=boxWidth \* 2, container=setupContainer)  
 ]  
  
 *# Y distance between the buttons* buttonGap = scaler(50, **"y"**) + inputList[0].getHeight()  
  
 inputList += [  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**), scaler(605, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Create Custom Material"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: createMaterial(widgetList),  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Save Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: saveSetup(widgetList),  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap \* 2, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Load Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: loadSetup(widgetList),  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap \* 3, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Clear Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**), action=clearParticles,  
 container=setupContainer, swellOnHover=True),  
  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(605, **"y"**) + buttonGap \* 4, font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Done - Start Simulation"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: endFunction(widgetList),  
 container=setupContainer, swellOnHover=True),  
 ]  
  
 widgetList = inputList + [setupContainer]  
  
 *# Remove references - Collected by garbage collection* del setupContainer  
 del inputList  
  
 widgetList[-1].startAnimation(**"horizontalslide"**, 0.25, **"in"**,  
 SW)  
  
 particles.add(Particle(1, **"Custom Material 1 - 1.0kgm^-3"**,  
 roundToSigFig((SW / 4) / scale, 3), 1, (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])), 0, xA=0))  
  
 pRef = particles.sprites()[-1]  
  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
  
 setting = True  
 while setting:  
 previousScale = scale  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
  
 if not pgkRoot.eventHandler(event):  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 1:  
 *# Check that looks at all possibilities to ensure  
 # that the particle will be placed on-screen and not  
 # intersecting with another particle* if pRef.pos.x + pRef.radius \* scale > SW \  
 or pRef.pos.x - pRef.radius \* scale < 0 \  
 or pRef.pos.y + pRef.radius \* scale > SH \  
 or pRef.pos.y - pRef.radius \* scale < 0 \  
 or len(pRef.hasCollided(particles)) != 0:  
 pass  
 else:  
 rad = roundToSigFig((SW / 4) / scale, 3)  
 particles.add(Particle(1, **"Wood - 800kgm^-3"**,  
 rad, 10, (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])),  
 0, xA=0))  
  
 elif event.button == 3:  
 *# Editing particles after they have been placed* mouseCoords = pg.mouse.get\_pos()  
 for i in particles.sprites()[:-1]:  
 if absoluteDistance(pgmath.Vector2(mouseCoords),  
 i.pos) <= i.radius \* scale:  
 if editList:  
 for widget in editList:  
 widget.delete()  
 del editList  
 editingParticle = i  
  
 *# Create container for widgets first,  
 # then position it so that it will always be  
 # on screen.* eContainer = pgk.Container(pgkRoot, screen,  
 centre=(0, 0),  
 outlineThickness=3,  
 width=scaler(310,  
 **"x"**),  
 height=scaler(400,  
 **"y"**),  
 bg=True,  
 bgColour=(255, 255,  
 255),  
 startVisible=False)  
  
 *# Container will be positioned so that one of  
 # its corners will be in the centre of the  
 # particle* pos = i.pos  
 if pos[1] + scaler(400, **"y"**) <= SH:  
 if pos[0] + scaler(310, **"x"**) <= SW:  
 eContainer.config(topleft=pos)  
 else:  
 eContainer.config(topright=pos)  
 else:  
 if pos[0] + scaler(310, **"x"**) <= SW:  
 eContainer.config(bottomleft=pos)  
 else:  
 eContainer.config(bottomright=pos)  
  
 editContWidth = scaler(310, **"x"**)  
 offset = scaler(150, **"x"**) / 2  
 boxWidth = scaler(125, **"x"**) / 2  
  
 *# Creating list of widgets used in editing  
 # the particle* inputList = [  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(28, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Coefficient of "  
 "Restitution"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(53, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Velocity to the "  
 "right (ms^-1):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(78, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Velocity "  
 "downwards ("  
 "ms^-1):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(103, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Acceleration to "  
 "the right ("  
 "ms^-2):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(128, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Acceleration "  
 "downwards ("  
 "ms^-2):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(153, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Radius (m):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(178, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Mass (kg):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(203, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Height off of "  
 "ground (m):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**,  
 container=eContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 editContWidth - scaler(50,  
 **"x"**),  
 scaler(228, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Lock particle to "  
 "height: "**,  
 container=eContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 editContWidth - scaler(50,  
 **"x"**),  
 scaler(253, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Select for "  
 "random velocity:"  
 " "**,  
 container=eContainer),  
 pgk.Checkbox(pgkRoot, screen,  
 editContWidth - scaler(50,  
 **"x"**),  
 scaler(278, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Draw line "  
 "following "  
 "particle's "  
 "motion: "**,  
 container=eContainer)  
 ]  
  
 buttonHeight = inputList[0].getHeight() \* 2  
 fButton = pgk.Button(pgkRoot, screen,  
 editContWidth - scaler(225,  
 **"x"**),  
 scaler(303, **"y"**),  
 font=SMALL\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Finish Editing "  
 "Particle"**,  
 height=buttonHeight,  
 width=scaler(213, **"x"**),  
 action=lambda:  
 endParticleEdit(  
 editList),  
 container=eContainer,  
 swellOnHover=True)  
  
 delButton = pgk.Button(pgkRoot, screen,  
 editContWidth - scaler(  
 225, **"x"**),  
 scaler(353, **"y"**),  
 font=SMALL\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Delete Particle"**,  
 height=buttonHeight,  
 width=scaler(213, **"x"**),  
 action=lambda:  
 deleteParticle(  
 editingParticle),  
 container=eContainer,  
 swellOnHover=True)  
  
 *# Create dropdown menu last as it needs to be  
 # drawn on top of the other inputs* drop = (pgk.Dropdown(pgkRoot, screen,  
 editContWidth - offset -  
 boxWidth,  
 scaler(5, **"y"**),  
 sortedCustoms +  
 MATERIALS\_SORTED,  
 font=SMALLER\_FONT,  
 bgColour=(  
 222, 222,  
 222),  
 inlineText=**"Select "  
 "material "  
 "(scroll to see"  
 " more):"**,  
 width=boxWidth \* 2,  
 container=eContainer))  
  
 *# Write to the input boxes so that they so  
 # the selected particle's properties* inputList[0].write(str(i.restCoefficient))  
 inputList[1].write(str(i.velocity.x))  
 inputList[2].write(str(i.velocity.y))  
 inputList[3].write(str(i.acceleration.x))  
 inputList[4].write(str(i.acceleration.y))  
 inputList[5].write(str(i.radius))  
 inputList[6].write(str(i.mass))  
  
 if i.hasRandomVelocity:  
 inputList[9].click()  
 if i.line:  
 inputList[10].click()  
  
 drop.setSelected(i.material)  
  
 editList = inputList + [drop, fButton,  
 delButton, eContainer]  
  
 *# Delete references* del inputList  
 del drop  
 del fButton  
 del eContainer  
  
 editList[-1].startAnimation(**"centre"**, 0.25,  
 **"in"**)  
  
 *# Can only zoom in/out if no particles are being edited* elif event.type == MOUSEBUTTONDOWN and editingParticle is None:  
 *# Buttons 4 and 5 correspond to the scroll wheel up/down.  
 # These are used for changing the scale while still in  
 # the setup phase* if event.button == 4:  
 scale \*= 1.02  
  
 elif event.button == 5:  
 scale \*= 0.98  
  
 if event.type == KEYDOWN:  
 if event.key == K\_s:  
 *# Use s key (for 'scale') to change size of a particle  
 # using the mouse, rather than the input boxes* sizeChange(pRef, widgetList[5], widgetList[6], metres)  
  
 pRef = particles.sprites()[-1]  
  
 if widgetList[-1].isEmpty():  
 setting = False  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 updateParticle(pRef, widgetList)  
  
 if editingParticle is not None:  
 updateParticle(editingParticle, editList, True)  
  
 *# Finish editing if the particle is not in the sprite group (it  
 # has been deleted), and if the finishing process has not  
 # already started* if editingParticle not in particles.sprites():  
 endParticleEdit(editList)  
  
 *# Call scalePosition on all but the last sprite, as last sprite's  
 # position is determined by the location of the mouse* for sprite in particles.sprites()[:-1]:  
 sprite.scalePosition()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.updateDirection()  
 sprite.drawDirectionArrow()  
 pgkRoot.update()  
  
 scaleLength = metres \* scale  
  
 if scaleLength > scaler(200, **"x"**):  
 metres /= 10  
 scaleLength = metres \* scale  
 elif scaleLength < scaler(20, **"x"**):  
 metres \*= 10  
 scaleLength = metres \* scale  
  
 metres = roundToSigFig(metres, 1)  
  
 particleGraph.changeLabelGap(scaleLength, scaleLength)  
  
 scaleDisplay = pg.Rect(scaler(5, **"x"**), scaler(5, **"y"**), scaleLength,  
 scaler(5, **"y"**))  
  
 scaleDisplayText = fpsFont.render(**u"{0}m"**.format(str(metres)), True,  
 (0, 0, 0))  
 scaleTextRect = scaleDisplayText.get\_rect(topleft=(scaler(15, **"y"**),  
 scaler(5, **"x"**)))  
 screen.blit(scaleDisplayText, scaleTextRect)  
  
 pg.draw.rect(screen, (0, 0, 0), scaleDisplay)  
  
 fps = str(int(clock.get\_fps()))  
  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 try:  
 return main, (widgetList + editList,)  
 except TypeError:  
 return main, (widgetList,)  
  
  
def saveSetup(setupWidgets):  
 def saveToFile(saveWidgets, setupWidgets):  
 global saving  
 saveWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**), True)  
  
 fileName = **u"{0}.txt"**.format(saveWidgets[0].get())  
  
 saveData = [str(scale)]  
  
 *# Writing data from all particles to a list that will be written to  
 # the txt file* for p in particles.sprites()[:-1]:  
 if p.line:  
 line = True  
 else:  
 line = False  
 pData = [p.hasRandomVelocity, line, p.restCoefficient,  
 p.material, p.radius, p.density, p.mass, p.vol,  
 (p.velocity.x, p.velocity.y), p.colour,  
 (p.rect.x, p.rect.y), (p.acceleration.x, p.acceleration.y)]  
 saveData.append(str(pData))  
  
 with open(str(saveLocation / fileName), **"w"**) as f:  
 f.writelines(**'**\n**'**.join(saveData))  
  
 setupWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW - scaler(400, **"x"**))  
  
 saving = False  
  
 global saving  
  
 if len(particles) < 2:  
 return  
  
 takenNumbers = []  
 for file in os.listdir(saveLocation):  
 if file.endswith(**".txt"**) and file.lower().startswith(**"custom scenario"**):  
 fileNum = int(file[15:-4])  
 takenNumbers.append(fileNum)  
  
 takenNumbers.sort()  
  
 lowNum = 1  
 for num in takenNumbers:  
 if num == lowNum:  
 lowNum += 1  
 else:  
 break  
  
 setupWidgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**))  
  
 saveContainer = pgk.Container(pgkRoot, screen,  
 topright=(SW + scaler(400, **"x"**), 0),  
 outlineThickness=0, width=scaler(400, **"x"**),  
 height=scaler(205, **"y"**))  
  
 contWidth = scaler(400, **"x"**)  
 offset = scaler(275, **"x"**)  
 boxWidth = scaler(250, **"x"**)  
 inputList = [  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(5, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Scenario Name: "**,  
 width=boxWidth, allowSpecial=False, charLimit=35,  
 defaultEntry=**u"Custom Scenario {0}"**.format(str(lowNum)),  
 container=saveContainer),  
 ]  
  
 inputList += [  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**), scaler(55, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Save"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: saveToFile(saveWidgets, setupWidgets),  
 container=saveContainer, swellOnHover=True),  
 ]  
  
 saveWidgets = inputList + [saveContainer]  
  
 *# Remove references - Collected by garbage collection* del saveContainer  
 del inputList  
  
 saving = True  
 while saving:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 saving = False  
 pg.quit()  
 quit()  
  
 if setupWidgets[-1].animationDone():  
 saveWidgets[-1].startAnimation(**"horizontalslide"**, 0.25, **"out"**,  
 SW - contWidth)  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 *# Exclude final particle - the one that was following the mouse  
 # pointer when save button was pressed* for i in particles.sprites()[:-1]:  
 i.draw()  
 i.drawDirectionArrow()  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 return  
  
  
def loadSetup(widgets):  
 def loadFromFile(loadWidgets, widgets):  
 global scale  
 global previousScale  
 global loading  
 global particleGraph  
 loadWidgets[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
  
 fileName = loadWidgets[0].get()  
  
 with open(str(saveLocation / fileName), **"r"**) as f:  
 data = f.readlines()  
  
 *# Remove newline characters from lines* newData = []  
 for line in data:  
 *# ast.literal\_eval reads the contents of the file, and evaluates the  
 # string as a python expression - in this case a list* newData.append(ast.literal\_eval(line.rstrip(**"**\n**"**)))  
  
 scale = newData[0]  
 previousScale = scale  
 particleGraph = Graph(screen, SW, SH, (0, 0), BG\_COLOUR, scale, scale)  
 for p in newData[1:]:  
 particles.add(Particle(p[2], p[3], p[4], p[5], p[8], p[9], p[10],  
 p[11][0], p[11][1]))  
  
 particles.sprites()[-1].hasRandomVelocity = p[0]  
 if p[1]:  
 particles.sprites()[-1].line = Line(screen, particleGraph, p[9])  
 else:  
 particles.sprites()[-1].line = None  
 particles.sprites()[-1].mass = p[6]  
 particles.sprites()[-1].vol = p[7]  
  
 if widgets:  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW - scaler(400, **"x"**))  
  
 particles.add(Particle(1, **"Custom Material 1 - 1.0kgm^-3"**,  
 roundToSigFig((SW / 4) / scale, 3), 1,  
 (0, 0),  
 (144, 202, 249),  
 (int(pg.mouse.get\_pos()[0]),  
 int(pg.mouse.get\_pos()[1])), 0, xA=0))  
  
 loading = False  
  
 global loading  
  
 for particle in particles.sprites():  
 particle.delete()  
  
 if widgets:  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**))  
  
 loadContainer = pgk.Container(pgkRoot, screen, maskColour=BG\_COLOUR,  
 centre=(SW / 2, SH / 2),  
 outlineThickness=0, width=scaler(400, **"x"**),  
 height=scaler(205, **"y"**), startVisible=False)  
  
 options = os.listdir(saveLocation)  
 boxWidth = scaler(350, **"x"**)  
 *# Define inputList as an empty list initially, as it needs to be  
 # referenced by the button, which cannot be done if inputList and the  
 # button are created at the same time.* inputList = [  
 pgk.Dropdown(pgkRoot, screen, scaler(25, **"x"**), scaler(5, **"y"**),  
 options, font=SMALL\_FONT, bgColour=(222, 222, 222),  
 width=boxWidth),  
 ]  
  
 inputList += [  
 pgk.Button(pgkRoot, screen, scaler(25, **"x"**), scaler(55, **"y"**),  
 font=MID\_FONT, bgColour=(33, 33, 33), text=**"Load Scenario"**,  
 height=inputList[0].getHeight() \* 2,  
 width=scaler(350, **"x"**),  
 action=lambda: loadFromFile(loadWidgets, widgets),  
 container=loadContainer, swellOnHover=True),  
 ]  
  
 *# Add dropdown to container last as it needs to be drawn over the button* inputList[0].config(container=loadContainer)  
  
 loadWidgets = inputList + [loadContainer]  
  
 *# Remove references - Collected by garbage collection* del loadContainer  
 del inputList  
  
 if not widgets:  
 loadWidgets[-1].startAnimation(**"centre"**, 0.25, **"in"**)  
  
 for particle in particles.sprites():  
 particle.delete()  
  
 loading = True  
 while loading:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == QUIT:  
 loading = False  
 pg.quit()  
 quit()  
  
 if widgets and widgets[-1].animationDone():  
 loadWidgets[-1].startAnimation(**"centre"**, 0.25, **"in"**)  
  
 screen.fill(BG\_COLOUR)  
  
 pgkRoot.update()  
  
 pg.display.update()  
  
 fps = str(int(clock.get\_fps()))  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
 *# If widgets is None, that means the program got to this page from the  
 # main menu, and therefore needs to move onto setup. If widgets exists,  
 # however, this function was called from within setup, and we just need a  
 # simple return statement* if not widgets:  
 return setup, (1,)  
 else:  
 return  
  
  
def sizeChange(particle, radBox, massBox, metres):  
 changing = True  
  
 *# Need to duplicate SCALE\_TOOL\_IMG as it needs to be modified with the  
 # rotozoom method of pygame's image class. Using the original image will  
 # mean that I will have to reload the image every time it is needed in  
 # order to get the original, unmodified one.* sizeArrow = SCALE\_TOOL\_IMG  
  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
  
 while changing:  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
 if pgkRoot.eventHandler(event):  
 return  
  
 elif event.type == MOUSEBUTTONDOWN:  
 if event.button == 1:  
 *# When the user initially presses the LMB,  
 # set startDistance equal to the distance between the  
 # mouse and the particle centre, and set startRad equal  
 # to the particle's current radius.* startDistance = absoluteDistance(particle.pos,  
 pgmath.Vector2(  
 pg.mouse.get\_pos()))  
 startRad = roundToSigFig(float(radBox.get()), 3)  
  
 elif event.type == KEYDOWN:  
 *# if ... in ... statement allows the user to use either the  
 # main enter key, or the enter key on the numpad.* if event.key in [K\_RETURN, K\_KP\_ENTER]:  
 return  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.drawDirectionArrow()  
  
 pg.mouse.set\_visible(True)  
  
 *# If the user is holding down the LMB* if pg.mouse.get\_pressed()[0]:  
 pg.mouse.set\_visible(False)  
  
 *# Draw dotted line from centre of particle to mouse pointer,* drawDottedLine(particle.pos, pg.mouse.get\_pos())  
  
 xDiff = pg.mouse.get\_pos()[0] - particle.pos.x  
 yDiff = pg.mouse.get\_pos()[1] - particle.pos.y  
  
 *# Calculating angle by which to rotate the arrow* dir = (math.atan2(yDiff, xDiff) \* -1) + math.pi / 2  
 blitArrow = pg.transform.rotate(sizeArrow, math.degrees(dir))  
  
 arrowRect = blitArrow.get\_rect(center=pg.mouse.get\_pos())  
  
 *# Blit arrow on screen in the position of the mouse pointer,  
 # to create the illusion that the pointer has changed to the arrow.* screen.blit(blitArrow, arrowRect)  
  
 posVector = pgmath.Vector2(pg.mouse.get\_pos())  
 currentDistance = absoluteDistance(particle.pos, posVector)  
 diff = currentDistance - startDistance  
  
 minRad = roundToSigFig(scaler(10, **"x"**) / scale, 3)  
 maxRad = roundToSigFig((SW / 4) / scale, 3)  
  
 changeFactor = roundToSigFig(0.5 / scale, 3)  
  
 if minRad <= startRad + (changeFactor \* diff) <= maxRad:  
 *# Equation to calculate new size of particle, based on  
 # distance that the mouse has moved away from the particle's  
 # centre (diff)* newRad = startRad + (changeFactor \* diff)  
  
 *# Update dimensions of particle and write the new dimensions  
 # to the input boxes.* particles.sprites()[-1].updateDimension(rad=newRad)  
 radBox.write(str(roundToSigFig(newRad, 3)))  
 massBox.write(str(particles.sprites()[-1].mass))  
  
 pgkRoot.update()  
  
 scaleLength = metres \* scale  
  
 scaleDisplay = pg.Rect(scaler(5, **"x"**), scaler(5, **"y"**), scaleLength,  
 scaler(5, **"y"**))  
  
 scaleDisplayText = fpsFont.render(**u"{0}m"**.format(str(metres)), True,  
 (0, 0, 0))  
 scaleTextRect = scaleDisplayText.get\_rect(topleft=(scaler(15, **"y"**),  
 scaler(5, **"x"**)))  
 screen.blit(scaleDisplayText, scaleTextRect)  
  
 pg.draw.rect(screen, (0, 0, 0), scaleDisplay)  
  
 fps = str(int(clock.get\_fps()))  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
  
def createMaterial(widgetList):  
 global materialTimer  
  
 def startExit(widgets, colour):  
 global customMaterials  
 global sortedCustoms  
 name = widgets[4].get()  
 density = widgets[3].get()  
 newMaterial = **u""" "{0} - {1}kgm^-3": [{2}, {3}],"""**.format(name,  
 density,  
 density,  
 colour)  
  
 with open(**"customMaterials.txt"**, **"r+"**) as file:  
 lines = file.readlines()[:-1]  
 lines += [newMaterial + **"**\n**"**, **"}"**]  
 *# file.truncate(0) clears the file, which is needed as otherwise  
 # lines will be added on to the end of the file, effectively  
 # duplicating everything and ruining the formatting of the  
 # dictionary* file.truncate(0)  
  
 *# Seek after truncate prevents null bytes being inserted - when  
 # truncate is used, the file tries to write from the same memory  
 # location as it was before truncation, resulting in null bytes  
 # being inserted. Seek(0) moves to the start of the file  
 # preventing null bytes* file.seek(0)  
 file.writelines(lines)  
  
 with open(**"customMaterials.txt"**, **"r"**) as file:  
 contents = file.read()  
  
 customMaterials = ast.literal\_eval(contents)  
 sortedCustoms = sorted(customMaterials)  
  
 widgets[-1].startAnimation(**"horizontalslide"**, 0.25, **"out"**, SW,  
 deleteAfter=True)  
  
 def randomiseColour(widgets):  
 widgets[5].write(str(random.randint(0, 255)))  
 widgets[6].write(str(random.randint(0, 255)))  
 widgets[7].write(str(random.randint(0, 255)))  
  
 widgetList[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW + scaler(350, **"x"**))  
  
 materialContainer = pgk.Container(pgkRoot, screen,  
 topleft=(SW, 0), outlineThickness=0,  
 width=scaler(400, **"x"**),  
 height=scaler(520, **"y"**))  
  
 contWidth = scaler(400, **"x"**)  
 offset = scaler(150, **"x"**)  
 boxWidth = scaler(125, **"x"**)  
 rgbOffset = scaler(275, **"x"**)  
 rgbWidth = scaler(50, **"x"**)  
  
 *# A unique name will be automatically generated for each new material  
 # created, in the form of "Custom Material <number>"  
 # This loop decides what number that will be, based on how many materials  
 # are already named in that form* customNum = 1  
 for i in sortedCustoms:  
 if i.lower().startswith(**"custom material"**):  
 customNum += 1  
  
 createWidgets = [  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(5, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Desired radius (m) (maximum 10m):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"1"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(55, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Desired Volume (m^3) (maximum 33.5m^3):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"1"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(105, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Desired mass (kg):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"1"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(155, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Density (kgm^-3):"**,  
 width=boxWidth, allowLetters=False,  
 allowSpecial=False, allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**, container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - offset, scaler(205, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Material name:"**,  
 width=boxWidth, charLimit=17,  
 defaultEntry=**"Custom Material "** + str(customNum),  
 container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - rgbOffset, scaler(255, **"y"**),  
 font=SMALL\_FONT, bgColour=(222, 222, 222),  
 inlineText=**"Material color: (R)"**,  
 width=rgbWidth, allowLetters=False, allowMaths=False,  
 allowSpecial=False, allowSpace=False,  
 charLimit=3, defaultEntry=**"126"**,  
 container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - rgbOffset + rgbWidth \* 2,  
 scaler(255, **"y"**), font=SMALL\_FONT,  
 bgColour=(222, 222, 222), inlineText=**"(G)"**,  
 width=rgbWidth, allowLetters=False, allowMaths=False,  
 allowSpecial=False, allowSpace=False,  
 charLimit=3, defaultEntry=**"25"**,  
 container=materialContainer),  
 pgk.InputBox(pgkRoot, screen, contWidth - rgbOffset + rgbWidth \* 4,  
 scaler(255, **"y"**), font=SMALL\_FONT,  
 bgColour=(222, 222, 222), inlineText=**"(B)"**,  
 width=rgbWidth, allowLetters=False, allowMaths=False,  
 allowSpecial=False, allowSpace=False,  
 charLimit=3, defaultEntry=**"27"**,  
 container=materialContainer),  
 ]  
  
 buttonGap = scaler(50, **"y"**) + createWidgets[0].getHeight()  
  
 randomButton = pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(305, **"y"**), font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Randomise Colour"**,  
 height=createWidgets[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: randomiseColour(createWidgets),  
 container=materialContainer, swellOnHover=True)  
  
 doneButton = pgk.Button(pgkRoot, screen,  
 contWidth - scaler(350, **"x"**),  
 scaler(305, **"y"**) + buttonGap, font=MID\_FONT,  
 bgColour=(33, 33, 33), text=**"Finish and Save"**,  
 height=createWidgets[0].getHeight() \* 2,  
 width=scaler(325, **"x"**),  
 action=lambda: startExit(createWidgets,  
 materialColour),  
 container=materialContainer, swellOnHover=True)  
  
 *# Cannot be added to the list immediately as they rely on the list for  
 # their height attribute* createWidgets += [randomButton, doneButton, materialContainer]  
  
 *# Remove references* del materialContainer  
 del randomButton  
 del doneButton  
  
 *# Displays the current colour that the user has chosen* rgbTestRect = pg.Rect((int(SW - contWidth - scaler(60, **"x"**)),  
 int(scaler(255, **"y"**))),  
 (rgbWidth, createWidgets[0].getHeight()))  
  
 previousRad = createWidgets[0].get()  
 previousVol = None  
 previousMass = createWidgets[2].get()  
  
 isCreating = False  
  
 changing = True  
  
 while changing:  
 for event in pg.event.get():  
 if event.type == QUIT:  
 pg.quit()  
 quit()  
 if pgkRoot.eventHandler(event):  
 pass  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.drawDirectionArrow()  
 *# If animation has finished for the creation container, length will  
 # be 0 as all widgets will have been deleted* if createWidgets[-1].isEmpty():  
 del createWidgets  
  
 widgetList[-7].config(options=sortedCustoms + MATERIALS\_SORTED)  
 widgetList[-1].startAnimation(**"horizontalslide"**, 0.5, **"out"**,  
 SW - contWidth)  
 return  
  
 *# isCreating is used to ensure that this only runs once* if widgetList[-1].animationDone() and not isCreating:  
 *# Need to use a slide out animation, rather than in.  
 # Otherwise the container will be displayed in its final  
 # position until the animation starts - it needs to be  
 # offscreen until it starts.* createWidgets[-1].startAnimation(**"horizontalslide"**, 0.25,  
 **"out"**, SW - contWidth)  
 isCreating = True  
  
 *# Need to except ValueError here in case the user deletes all  
 # characters (calling int() on an empty string throws a ValueError* try:  
 for i in [createWidgets[5], createWidgets[6], createWidgets[7]]:  
 if int(i.get()) > 255:  
 i.write(**"255"**)  
  
 r = int(createWidgets[5].get())  
 g = int(createWidgets[6].get())  
 b = int(createWidgets[7].get())  
  
 materialColour = (r, g, b)  
 except ValueError:  
 pass  
  
 try:  
 if createWidgets[0].get() != previousRad:  
 previousRad = createWidgets[0].get()  
  
 rad = roundToSigFig(float(createWidgets[0].get()), 3)  
 vol = roundToSigFig((4 / 3) \* math.pi \* rad \*\* 3, 3)  
  
 createWidgets[1].write(str(vol))  
 createWidgets[3].write(  
 str(roundToSigFig(float(createWidgets[2].get()) /  
 vol, 3)))  
  
 if createWidgets[1].get() != previousVol:  
 previousVol = createWidgets[1].get()  
  
 vol = roundToSigFig(float(createWidgets[1].get()), 3)  
 rad = roundToSigFig(((3 \* vol) / (4 \* math.pi)) \*\* (1 / 3), 3)  
 previousRad = str(rad)  
 createWidgets[0].write(str(rad))  
 createWidgets[3].write(  
 str(roundToSigFig(float(createWidgets[2].get()) /  
 vol, 3)))  
  
 if createWidgets[2].get() != previousMass:  
 previousMass = createWidgets[3].get()  
 createWidgets[3].write(  
 str(roundToSigFig(float(createWidgets[2].get()) /  
 float(createWidgets[1].get()),  
 3)))  
  
 *# Need to catch both errors here, as the inputs involve dividing  
 # by user input, meaning they may end up dividing by zero  
 # Also ValueError in case the user deletes everything in one input box* except (ValueError, ZeroDivisionError):  
 pass  
  
 pgkRoot.update()  
  
 fps = str(int(clock.get\_fps()))  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
 pg.draw.rect(screen, materialColour, rgbTestRect)  
  
 pg.display.update()  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
  
def main(widgetList):  
 global mainprogram  
 global tNow  
 global frameNumber  
 global timeMultiplier  
 global timeShown  
 timeShown = False  
  
 def showTimeControls(timeContainer):  
 global timeShown  
 if not timeShown:  
 timeContainer.startAnimation(**"verticalslide"**, 0.25, **"out"**,  
 SH - scaler(100, **"y"**))  
 timeShown = True  
 else:  
 timeContainer.startAnimation(**"verticalslide"**, 0.25, **"out"**, SH)  
 timeShown = False  
  
 *# Delete particle that gets placed on button press* particles.remove(particles.sprites()[-1])  
  
 timeWidgets = [  
 pgk.Container(pgkRoot, screen, centre=(SW / 2, SH + scaler(50, **"y"**)),  
 width=scaler(450, **"x"**), height=scaler(100, **"y"**))  
 ]  
  
 contWidth = scaler(450, **"x"**)  
 contHeight = scaler(100, **"y"**)  
  
 timeWidgets += [  
 pgk.Button(pgkRoot, screen, contWidth / 2 - scaler(40, **"x"**),  
 contHeight - scaler(145, **"y"**), height=scaler(40, **"y"**),  
 width=scaler(80, **"x"**),  
 action=lambda: showTimeControls(timeWidgets[0]),  
 image=TT\_IMG, hoverImage=H\_TT\_IMG,  
 container=timeWidgets[0]),  
 pgk.Button(pgkRoot, screen, contWidth / 2 - scaler(75, **"x"**),  
 contHeight - scaler(105, **"y"**), height=scaler(100, **"y"**),  
 width=scaler(150, **"x"**),  
 action=lambda: pauseMenu(timeWidgets),  
 image=PAUSE\_IMG, hoverImage=H\_PAUSE\_IMG,  
 container=timeWidgets[0]),  
 pgk.Button(pgkRoot, screen, contWidth / 2 - scaler(175, **"x"**),  
 contHeight - scaler(105, **"y"**), height=scaler(100, **"y"**),  
 width=scaler(100, **"x"**), action=lambda: timeChange(-1),  
 image=RW\_IMG, hoverImage=H\_RW\_IMG,  
 container=timeWidgets[0]),  
 pgk.Button(pgkRoot, screen, contWidth / 2 + scaler(75, **"x"**),  
 contHeight - scaler(105, **"y"**), height=scaler(100, **"y"**),  
 width=scaler(100, **"x"**), action=lambda: timeChange(1),  
 image=FF\_IMG, hoverImage=H\_FF\_IMG,  
 container=timeWidgets[0]),  
 ]  
  
 for widget in widgetList:  
 widget.delete()  
 del widget  
 *# Remove references to widgets - will get collected by Python's  
 # garbage collection* mainprogram = True  
  
 while mainprogram:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == KEYDOWN:  
 if event.key == K\_ESCAPE:  
 pauseMenu(timeWidgets)  
  
 if event.type == QUIT:  
 mainprogram = False  
 pg.quit()  
 quit()  
  
 frameNumber += TIME\_SCALES[currentTimescale]  
 if frameNumber < 1:  
 frameNumber = 1  
 try:  
 *# If statement prevents 'jumping' of particles when user moves  
 # the window* if time.time() - previousFrame < 0.1:  
 timeMultiplier = (time.time() - previousFrame)  
 timeMultiplier \*= TIME\_SCALES[currentTimescale]  
 *# Calculates time between frames* except NameError:  
 *# Will occur on the first frame, as there is no previous frame* pass  
  
 previousFrame = time.time()  
  
 if tNow <= 0 and currentTimescale < 3:  
 pauseMenu(timeWidgets)  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
 particles.update()  
  
 if timeMultiplier > 0:  
 tNow += timeMultiplier  
  
 *# Set up text that shows current time* tDisplay = round(tNow, 4)  
 timeFont = pg.font.SysFont(MID\_FONT[0], MID\_FONT[1])  
 timeText = timeFont.render(**"Time: T+"** + str(tDisplay), True, (0, 0, 0))  
 tRect = timeText.get\_rect(topleft=(10, 10))  
  
 timescaleFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 tscaleText = timescaleFont.render(**"Time Multiplier: x"** +  
 str(TIME\_SCALES[currentTimescale]),  
 True, (0, 0, 0))  
 tscaleRect = tscaleText.get\_rect(topleft=(scaler(10, **"x"**),  
 scaler(50, **"y"**)))  
  
 screen.blit(timeText, tRect)  
 screen.blit(tscaleText, tscaleRect)  
  
 pgkRoot.update()  
  
 fps = str(int(clock.get\_fps()))  
  
 *# Create text that shows the fps that the program is running at* fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
  
 clock.tick()  
  
 for i in particles.sprites():  
 i.delete()  
 return mainMenu, (1,)  
  
  
*# Runs when user presses esc*def pauseMenu(timeWidgets):  
 *# returnToMain ends both the pause loop and the main loop, so that the  
 # program will return back to the main menu* def returnToMain(menuWidgets, timeWidgets):  
 global paused  
 global mainprogram  
 paused = False  
 mainprogram = False  
  
 *# Get rid of the time controls* timeWidgets[0].startAnimation(**"verticalslide"**, 0.25, **"out"**, SH, True)  
  
 *# Will be true if timeWidgets has not been expanded* if timeWidgets[0].getRect().centery > SH:  
 for i in timeWidgets[1:]:  
 i.delete()  
 timeWidgets[0].delete()  
 del timeWidgets  
  
 *# Get rid of the menu* menuWidgets[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
  
 def exitPause(menuWidgets, timeWidgets):  
 *# Simply ends the pause loop, and starts the disappearing animation  
 # for the menu* global paused  
 paused = False  
 global currentTimescale  
 *# Change play button into a pause button* timeWidgets[2].config(action=lambda: pauseMenu(timeWidgets),  
 image=PAUSE\_IMG, hoverImage=H\_PAUSE\_IMG)  
  
 menuWidgets[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
 *# If current time is earlier or equal to the time that the simulation  
 # started, set timescale to 1x, as the user shouldn't be able to rewind  
 # to earlier than the beginning of the sim* if tNow <= 0:  
 currentTimescale = 4  
  
 def exitProgram():  
 pg.quit()  
 quit()  
  
 def hideStats(statList):  
 *# Menu for showing stats will disappear* statList[-1].startAnimation(**"centre"**, 0.25, **"out"**, deleteAfter=True)  
  
 global paused  
  
 pauseContainer = pgk.Container(pgkRoot, screen,  
 centre=(SW / 2, SH / 2),  
 bg=True, bgColour=BG\_COLOUR,  
 maskColour=BG\_COLOUR, outlineThickness=3,  
 outlineColour=(33, 33, 33),  
 width=scaler(345, **"x"**),  
 height=scaler(125, **"y"**), startVisible=False)  
  
 contWidth = scaler(400, **"x"**)  
 gap = scaler(55, **"y"**)  
  
 *# Need to initially create menuWidgets as an empty list, so that the  
 # buttons can reference it when passing arguments to their functions* menuWidgets = []  
 statList = []  
  
 menuWidgets += [  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(390, **"x"**), scaler(10, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Main Menu"**,  
 height=scaler(50, **"y"**),  
 width=scaler(325, **"x"**),  
 action=lambda: returnToMain(menuWidgets, timeWidgets),  
 container=pauseContainer, swellOnHover=True),  
 pgk.Button(pgkRoot, screen,  
 contWidth - scaler(390, **"x"**), gap + scaler(10, **"y"**),  
 font=MID\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Exit Program"**,  
 height=scaler(50, **"y"**),  
 width=scaler(325, **"x"**), action=exitProgram,  
 container=pauseContainer, swellOnHover=True)  
 ]  
  
 menuWidgets += [pauseContainer]  
  
 del pauseContainer  
  
 menuWidgets[-1].startAnimation(**"centre"**, 0.25, **"in"**)  
  
 *# Change pause button into a play button* timeWidgets[2].config(action=lambda: exitPause(menuWidgets, timeWidgets),  
 image=PLAY\_IMG,  
 hoverImage=H\_PLAY\_IMG)  
 pausedFont = pg.font.SysFont(LARGE\_FONT[0], LARGE\_FONT[1])  
 pausedText = pausedFont.render(**"PAUSED"**, True, (0, 0, 0))  
 pRect = pausedText.get\_rect(center=(int(SW / 2), int(scaler(380, **"y"**))))  
  
 tDisplay = round(tNow, 4)  
 timeFont = pg.font.SysFont(MID\_FONT[0], MID\_FONT[1])  
 timeText = timeFont.render(**"Time: T+"** + str(tDisplay), True, (0, 0, 0))  
 tRect = timeText.get\_rect(topleft=(scaler(10, **"x"**), scaler(10, **"y"**)))  
  
 *# The pause loop is just an empty loop - only showing the UI elements  
 # such as fps text, paused text, and time text* paused = True  
 while paused:  
 for event in pg.event.get():  
 pgkRoot.eventHandler(event)  
 if event.type == KEYDOWN:  
 if event.key == K\_ESCAPE:  
 exitPause(menuWidgets, timeWidgets)  
  
 if event.type == MOUSEBUTTONUP:  
 if event.button == 3:  
 *# Showing particle stat* mouseCoords = pg.mouse.get\_pos()  
 for i in particles.sprites():  
 if absoluteDistance(pgmath.Vector2(mouseCoords),  
 i.pos) <= i.radius \* scale:  
 if statList:  
 for widget in statList:  
 widget.delete()  
 del statList  
  
 *# Create container for widgets first,  
 # then position it so that it will always be  
 # on screen.* eContainer = pgk.Container(pgkRoot, screen,  
 centre=(0, 0),  
 outlineThickness=3,  
 width=scaler(310,  
 **"x"**),  
 height=scaler(250,  
 **"y"**),  
 bg=True,  
 bgColour=(255, 255,  
 255),  
 startVisible=False)  
  
 *# Container will be positioned so that one of  
 # its corners will be in the centre of the  
 # particle* pos = i.pos  
 if pos[1] + scaler(400, **"y"**) <= SH:  
 if pos[0] + scaler(310, **"x"**) <= SW:  
 eContainer.config(topleft=pos)  
 else:  
 eContainer.config(topright=pos)  
 else:  
 if pos[0] + scaler(310, **"x"**) <= SW:  
 eContainer.config(bottomleft=pos)  
 else:  
 eContainer.config(bottomright=pos)  
  
 editContWidth = scaler(310, **"x"**)  
 offset = scaler(150, **"x"**) / 2  
 boxWidth = scaler(125, **"x"**) / 2  
  
 *# Creating list of widgets used in editing  
 # the particle* inputList = [  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(3, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Coefficient of "  
 "Restitution"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(28, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Velocity to the "  
 "right (ms^-1):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(53, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Velocity "  
 "downwards ("  
 "ms^-1):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(78, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Acceleration to "  
 "the right ("  
 "ms^-2):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(103, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Acceleration "  
 "downwards ("  
 "ms^-2):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(128, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Radius (m):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(153, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Mass (kg):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 container=eContainer,  
 canUse=False),  
 pgk.InputBox(pgkRoot, screen,  
 editContWidth - offset,  
 scaler(178, **"y"**),  
 font=SMALLER\_FONT,  
 bgColour=(222, 222, 222),  
 inlineText=**"Height off of "  
 "ground (m):"**,  
 width=boxWidth,  
 allowLetters=False,  
 allowSpecial=False,  
 allowSpace=False, charLimit=10,  
 defaultEntry=**"0"**,  
 container=eContainer,  
 canUse=False),  
 ]  
  
 buttonHeight = inputList[0].getHeight() \* 2  
 closeButton = pgk.Button(pgkRoot, screen,  
 editContWidth - scaler(225,  
 **"x"**),  
 scaler(203, **"y"**),  
 font=SMALL\_FONT,  
 bgColour=(33, 33, 33),  
 text=**"Hide Particle Stats"**,  
 height=buttonHeight,  
 width=scaler(213, **"x"**),  
 action=lambda: hideStats(  
 statList),  
 container=eContainer,  
 swellOnHover=True)  
  
 *# Write to the input boxes so that they so  
 # the selected particle's properties* inputList[0].write(str(i.restCoefficient))  
 inputList[1].write(str(roundToSigFig(i.velocity.x,  
 3)))  
 inputList[2].write(str(roundToSigFig(i.velocity.y,  
 3)))  
 inputList[3].write(str(i.acceleration.x))  
 inputList[4].write(str(i.acceleration.y))  
 inputList[5].write(str(roundToSigFig(i.radius, 3)))  
 inputList[6].write(str(roundToSigFig(i.mass, 3)))  
 height = (SH - i.pos.y) / scale - i.radius  
 inputList[7].write(str(roundToSigFig(height, 3)))  
  
 statList = inputList + [closeButton, eContainer]  
  
 *# Delete references* del inputList  
 del eContainer  
  
 statList[-1].startAnimation(**"centre"**, 0.25,  
 **"in"**)  
  
 if event.type == QUIT:  
 paused = False  
 pg.quit()  
 quit()  
  
 screen.fill(BG\_COLOUR)  
 particleGraph.draw()  
  
 for sprite in particles.sprites():  
 sprite.draw()  
 sprite.drawDirectionArrow()  
  
 pgkRoot.update()  
  
 timescaleFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 tscaleText = timescaleFont.render(**"Time Multiplier: x"** +  
 str(TIME\_SCALES[currentTimescale]),  
 True, (0, 0, 0))  
 tscaleRect = tscaleText.get\_rect(topleft=(scaler(10, **"x"**),  
 scaler(50, **"y"**)))  
  
 screen.blit(timeText, tRect)  
 screen.blit(tscaleText, tscaleRect)  
 screen.blit(pausedText, pRect)  
  
 fps = str(int(clock.get\_fps()))  
 fpsFont = pg.font.SysFont(SMALL\_FONT[0], SMALL\_FONT[1])  
 fpsText = fpsFont.render(**u"FPS: {0}"**.format(fps), True, (0, 0, 0))  
 fpsRect = fpsText.get\_rect(midtop=(int(SW / 2), int(scaler(10, **"y"**))))  
 screen.blit(fpsText, fpsRect)  
  
 pg.display.update()  
  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3 | FPS: '** + fps)  
 clock.tick()  
  
  
*# noinspection PyUnboundLocalVariable*if \_\_name\_\_ == **"\_\_main\_\_"**: *# If program is run as a script, this will run* pg.init()  
 pg.font.init()  
  
  
 *# scales sizes relative to screen width/height - gives a consistent feel  
 # across all devices* def scaler(toscale, axis):  
 if axis == **'x'**:  
 return int(toscale \* (SW / 1920))  
 else:  
 return int(toscale \* (SH / 1080))  
  
  
 *# Does the opposite of scaler - needed when saving scenarios, as the  
 # values will need to be rescaled when the scenario is loaded* def descaler(toscale, axis):  
 if axis == **"x"**:  
 return int(toscale \* (1920 / SW))  
 else:  
 return int(toscale \* (1080 / SH))  
  
  
 if win:  
 SW = GetSystemMetrics(0)  
 SH = GetSystemMetrics(1)  
 else:  
 SW = 1920  
 SH = 1080  
  
 BG\_COLOUR = (244, 244, 244)  
  
 scale = scaler(100, **"x"**)  
 previousScale = scale  
  
 *# Fullscreen doesn't work with pg.mouse.set\_visible() (mouse gets  
 # centred every time function is called), so I am using a borderless  
 # window that is the same size as the screen instead. os.environ  
 # positions the window in the top left corner of the screen* os.environ[**'SDL\_VIDEO\_WINDOW\_POS'**] = **"0,0"** *# Display flags: NOFRAME to simulate a fullscreen display, and DOUBLEBUF  
 # to prevent flickering - it also slightly improves fps* screen = pg.display.set\_mode((SW, SH), NOFRAME | DOUBLEBUF)  
 screen.set\_alpha(None)  
  
 particleGraph = Graph(screen, SW, SH, (0, 0), BG\_COLOUR, scale, scale)  
  
 *# Read a dictionary in from a file* with open(**"materials.txt"**, **"r"**) as file:  
 contents = file.read()  
  
 *# ast.literal\_eval reads the contents of the file, and evaluates the  
 # string as a python expression - in this case a dictionary* MATERIALS = ast.literal\_eval(contents)  
 MATERIALS\_SORTED = sorted(MATERIALS)  
  
 with open(**"customMaterials.txt"**, **"r"**) as file:  
 contents = file.read()  
  
 customMaterials = ast.literal\_eval(contents)  
 sortedCustoms = sorted(customMaterials)  
  
 timeMultiplier = 1 / 60 *# Initial value for time between frames* TIME\_SCALES = [-2, -1, -0.5, 0.5, 1, 2]  
 currentTimescale = 4  
 tNow = 0  
 frameNumber = 0  
  
 imagesFolder = Path(**"resources/images/"**)  
 saveLocation = Path(**"Saved Scenarios/"**)  
  
 *# Used for drawDirectionArrow method* ARROW\_IMAGE = pg.image.load(  
 str(imagesFolder / **"arrow.png"**)).convert\_alpha()  
  
 SCALE\_TOOL\_IMG = pg.image.load(  
 str(imagesFolder / **"resizeCursor.png"**)).convert\_alpha()  
  
 PAUSE\_IMG = pg.image.load(  
 str(imagesFolder / **"pausedNormal.png"**)).convert\_alpha()  
 H\_PAUSE\_IMG = pg.image.load(  
 str(imagesFolder / **"pausedHovered.png"**)).convert\_alpha()  
  
 PLAY\_IMG = pg.image.load(  
 str(imagesFolder / **"playNormal.png"**)).convert\_alpha()  
 H\_PLAY\_IMG = pg.image.load(  
 str(imagesFolder / **"playHovered.png"**)).convert\_alpha()  
  
 FF\_IMG = pg.image.load(  
 str(imagesFolder / **"ffNormal.png"**)).convert\_alpha()  
 H\_FF\_IMG = pg.image.load(  
 str(imagesFolder / **"ffHovered.png"**)).convert\_alpha()  
  
 RW\_IMG = pg.transform.flip(FF\_IMG, True, False)  
 H\_RW\_IMG = pg.transform.flip(H\_FF\_IMG, True, False)  
  
 TT\_IMG = pg.image.load(  
 str(imagesFolder / **"timeTabNormal.png"**)).convert\_alpha()  
 H\_TT\_IMG = pg.image.load(  
 str(imagesFolder / **"timeTabHovered.png"**)).convert\_alpha()  
  
 PREV\_IMG = pg.image.load(  
 str(imagesFolder / **"prevPage.png"**)).convert\_alpha()  
 NEXT\_IMG = pg.image.load(  
 str(imagesFolder / **"nextPage.png"**)).convert\_alpha()  
  
 R\_MENU\_IMG = pg.image.load(  
 str(imagesFolder / **"rMainMenu.png"**)).convert\_alpha()  
 L\_MENU\_IMG = pg.image.load(  
 str(imagesFolder / **"lMainMenu.png"**)).convert\_alpha()  
  
 if SW / 1920 > SH / 1080:  
 *# Scales font sizes relative to whichever axis has been 'scaled  
 # down' more, to prevent text being larger than the widget it is in* SMALLER\_FONT = (**"Helvetica"**, scaler(12, **"y"**))  
 SMALL\_FONT = (**"Helvetica"**, scaler(18, **"y"**))  
 MID\_FONT = (**"Helvetica"**, scaler(30, **"y"**))  
 LARGE\_FONT = (**"Helvetica"**, scaler(72, **"y"**))  
  
 else:  
 SMALLER\_FONT = (**"Helvetica"**, scaler(12, **"x"**))  
 SMALL\_FONT = (**"Helvetica"**, scaler(18, **"x"**))  
 MID\_FONT = (**"Helvetica"**, scaler(30, **"x"**))  
 LARGE\_FONT = (**"Helvetica"**, scaler(72, **"x"**))  
  
 mainmenu = False  
 setting = False  
 mainprogram = False  
 paused = False  
 mainWidgets = []  
  
 pg.display.set\_caption(**'Jack Sanders Particle Sim V3'**)  
 clock = pg.time.Clock()  
 particles = pg.sprite.Group()  
  
 nextFunction, args = mainMenu(1)  
 *# Prevents recursion (For example, mainMenu would be called  
 # from within main, which would be called from within setup, which would  
 # be called from within mainMenu, and so on) which would* while True:  
 nextFunction, args = nextFunction(\*args)