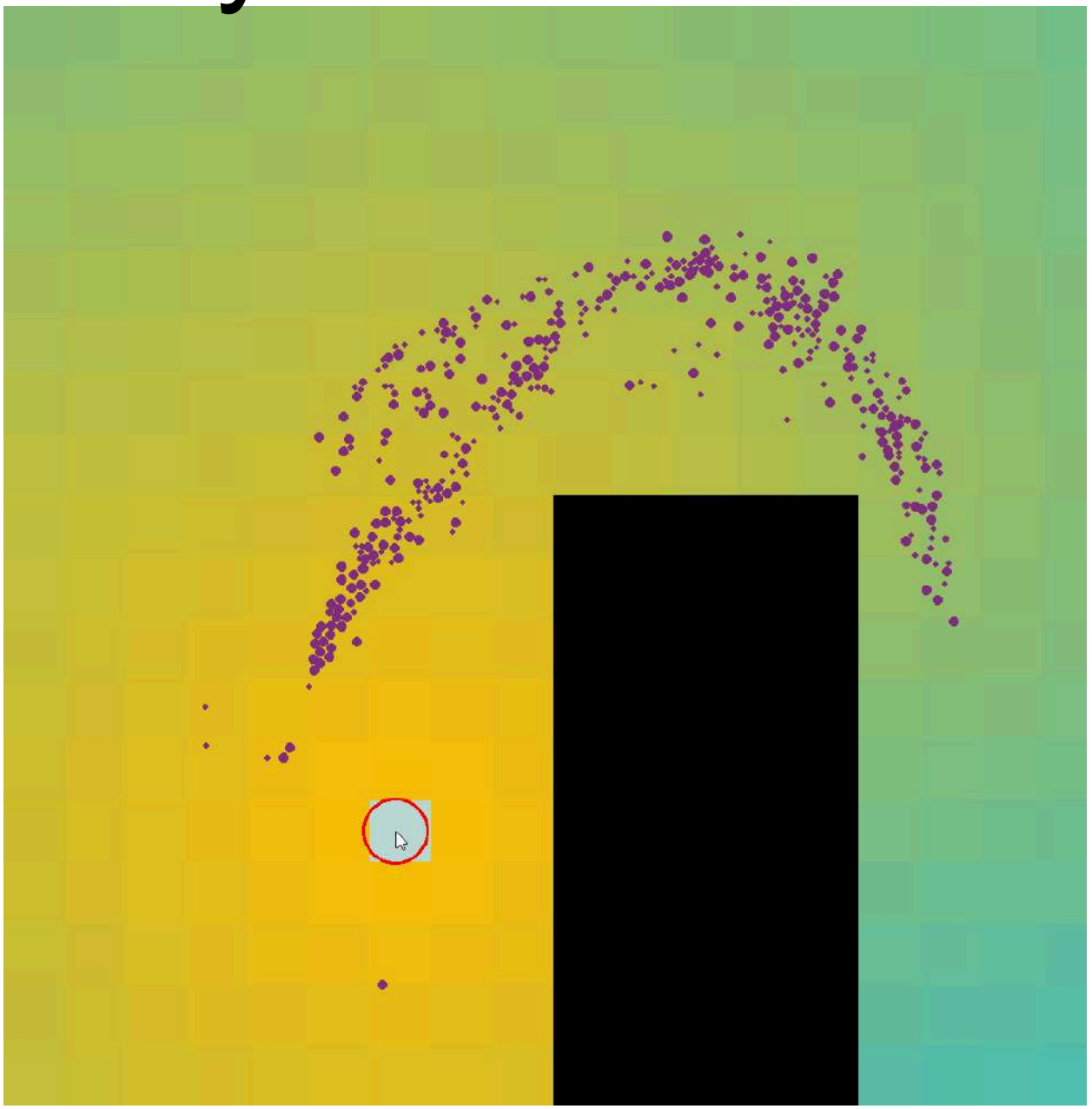
Physics Simulator



Samsul Hoque
Loughborough Grammar School
Centre Number: 25138
Candidate Number: 7246

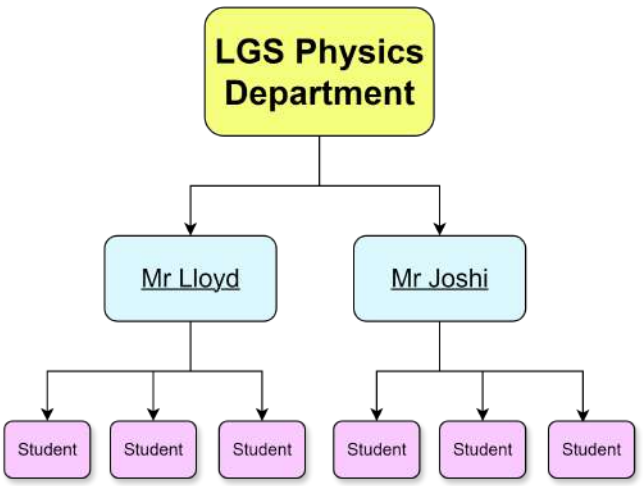
Analysis

Statement Of Problem

In a world where people spend their first 18 years in constant learning, students and schools alike are keen to spend that time wisely. The syllabus gets increasingly harder year on year and we are trying to find new ways to improve teaching. Topics such as projectile motion can feel like a blur when the education system drowns students in a sea of algebra in textbooks. Currently, my school relies heavily on more traditional teaching styles, for example the aforementioned textbooks and powerpoints. I intend to create a well-rounded simulation that covers various physics topics which would hopefully increase understanding and foster a healthier and more productive learning environment. This physics simulation project aims to provide accurate simulations suitable for an A-Level student studying physics, with the goal of not only improving grades, but also ensuring student enjoyment

End User

I am developing my project for use in classrooms. I have reached out to my physics teacher, Mr Stevens, who would like to use the program as a teaching aid. It will then be eventually distributed to other teachers in the physics department. It will act as a visualisation aid for an introduction to the supported topics. Its interactive features and adjustable variables will allow, for example, students to observe the effect a certain parameter would have. Mr Stevens may want to distribute the program to the students themselves in hope that they might engage further with the topic by performing their own experiments. A command-line interface may prove daunting to students unfamiliar with such an interface. As such, I believe a



graphical user interface would maximise engagement and be familiar with a group of senior students.

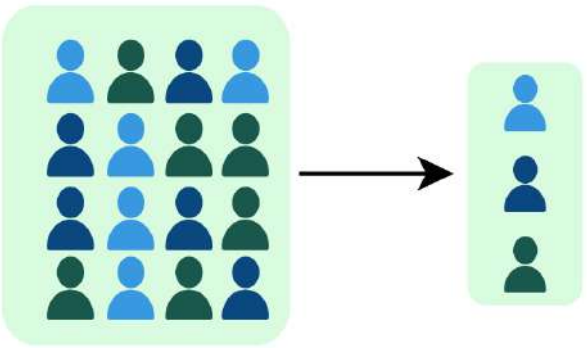
“Kids who have been working with technology from early on may be better equipped to learn in an educational environment that relies heavily on computers and other devices. And in fact, effective use of classroom technology has been shown to engage students in ways that traditional teaching methods cannot.” [8]
https://www.southernphone.com.au/blog/tech-savvy-toddlers-why-are-young-children-so-good-with-technology

Technology is deeply rooted in the younger generations and the students have almost certainly been raised around it. As such, I fully expect the students to have at the very least basic IT skills, which would be enough to use my system. The teachers, although the secondary demographic, will also have to be accommodated for as they could be using the simulation themselves as a teaching tool in lessons. The login system will be self-explanatory and simple.

Requirements Gathering

My target demographic will be students at my school who have faced similar issues. Obviously, I am a student who represents my target demographic. While I could create the program to cater to my needs and my needs only and assume that my peers will have identical views, that would be naive as everyone’s needs differ, whether it be from the topic they struggle with or their preferred learning technique. However, if I were to perform a survey, I would be able to take observations of the bigger picture and cater the program for the majority in an aim to benefit the most people. So, I will be creating a survey and distributing it to my peers and collating that information to draw conclusions which will determine the creation of the program.

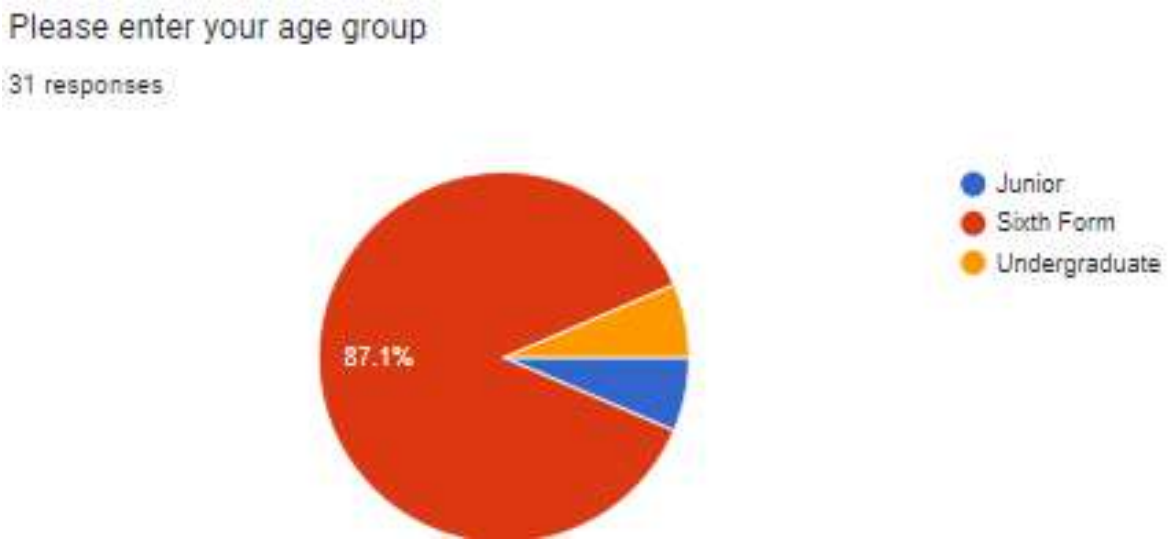
Survey Prerequisites



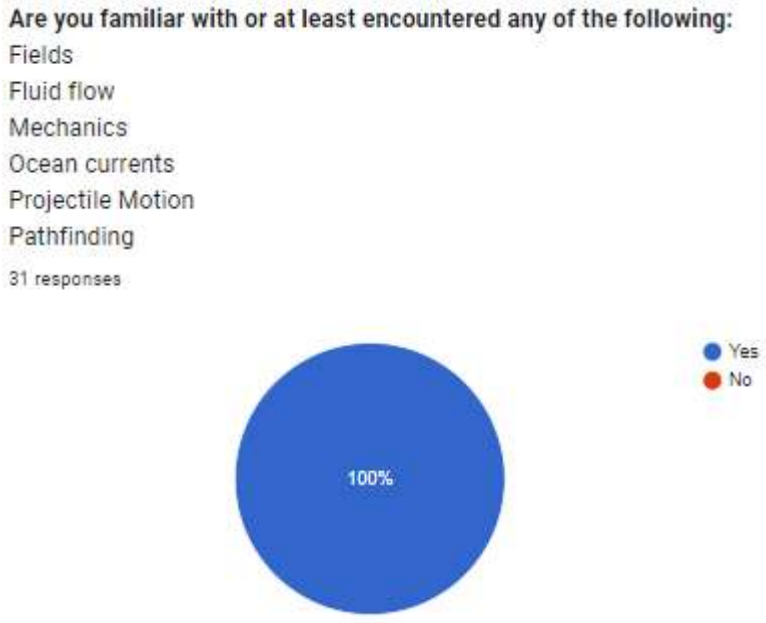
For the survey, I will be using an online form provider called Google Forms.
I had a few options at my disposal for how I would distribute my survey to people. Ideally, I wanted students who studied a variety of subjects, or in other words students who studied STEM and non-STEM subjects. However, I was aware that this could introduce bias towards non-STEM subject-takers. Therefore, I decided upon a systematic sampling technique. I created a 99 long sampling frame of all my peers who could offer a reliable answer. Then, I chose an interval of 3 and selected the person with the current index plus the interval. This provided me with 33 people who I then distributed the survey to. In order to get the number of responses I sought for, I made sure to keep the survey brief.

I received 31 responses as 2 people failed to complete the form. I first included a routine data quality check in case that the form was spammed with fake responses and to ensure that the user was not just blindly answering. Fortunately, all responses were valid so I have not had to vet the responses.

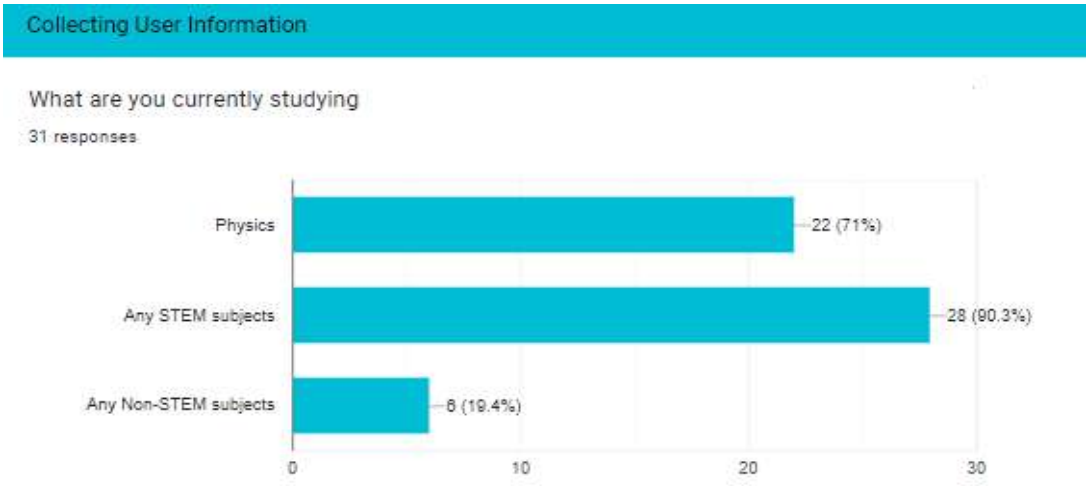
Survey Demographic



Before I asked questions relevant to the program development, I first wanted to ensure that the responses were from the target demographic. A resounding 87% of respondents were currently in Sixth Form.



My project will most likely involve more physics types of simulations. However, these concepts apply widely to many different fields. To ensure that the survey remained relevant, they were asked if they were familiar with any of the listed topics. A no response prompted the user to end the survey as I wanted responses purely from those who might benefit from the project. All respondents were familiar with at least one of the subjects.

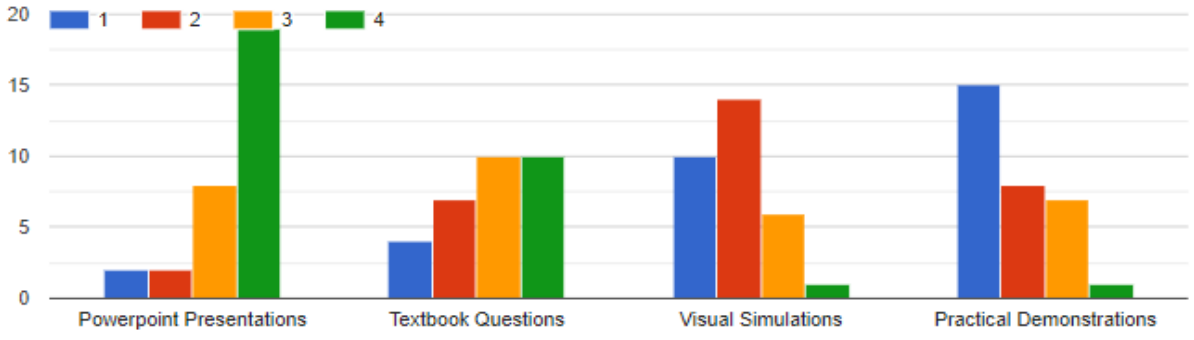


Next, they were asked which subjects they were studying. This would provide better context to the responses. My sampling frame largely consisted of the above and lower STEM subjects, so I had initially assumed that there would be very few students who also studied a non-STEM subject. As such, I was surprised to see that 25% of respondents were also studying non-STEM subjects. This would give a greater variety and I could observe if the project idea would hold up well with this alternative demographic. Unsurprisingly, 79% of those taking physics also studied other STEM subjects for A-Levels. In this year's iteration of sixth form, a popular choice has been physics alongside computer science or chemistry, and this trend has been the case for many years now.

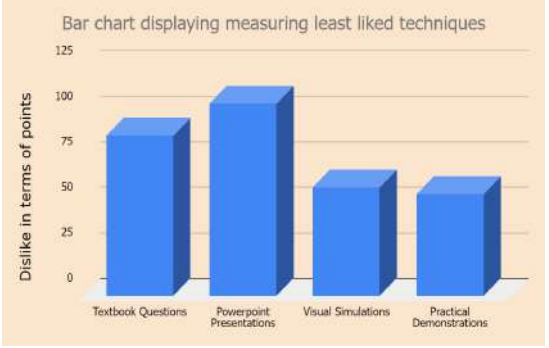
Results

Teaching Techniques Feedback

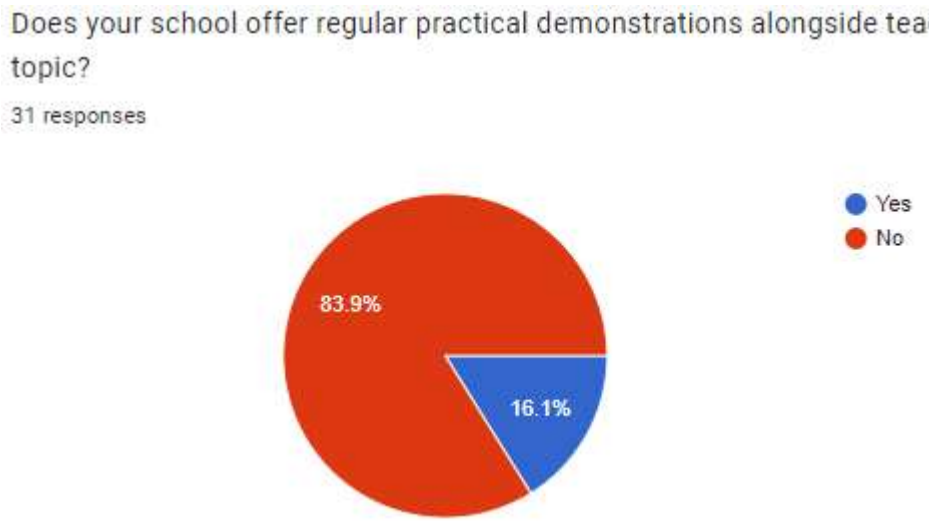
And now rank the most helpful teaching methods from 1 - 4, with 1 being the most helpful



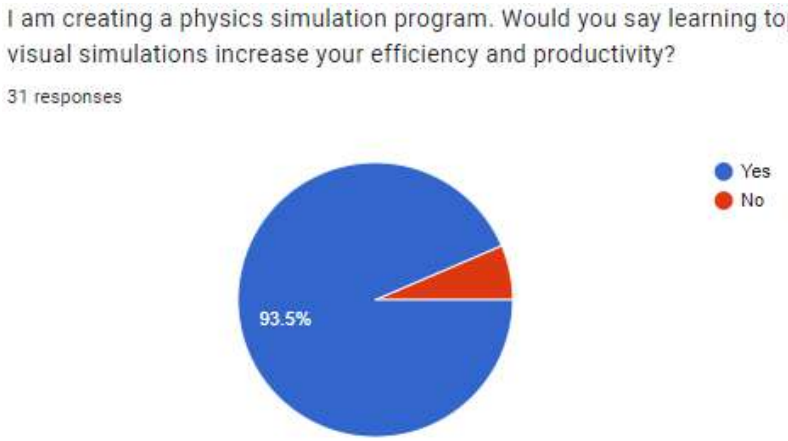
They were then asked to rank their preferred teaching methods from 1 to 4. By far the most disliked technique was powerpoint presentations. Studies have shown that these tend to have limited impact so I am not surprised to see this.



The next option was doing questions from the textbook. This technique appears to be very divisive as it was a relatively even distribution but it does seem to be skewed towards the higher scores. This suggests that there are a sizable number who benefit strongly from these textbook questions, though the small majority dislike it.
In the processed results shown in the bar chart, both visual simulations and practicals score similarly. However, this is slightly misleading. By far the most convincing technique was the practical work as 33% more people put it as their first choice. However, most then put visual simulations as their second choice, suggesting it is a viable alternative.

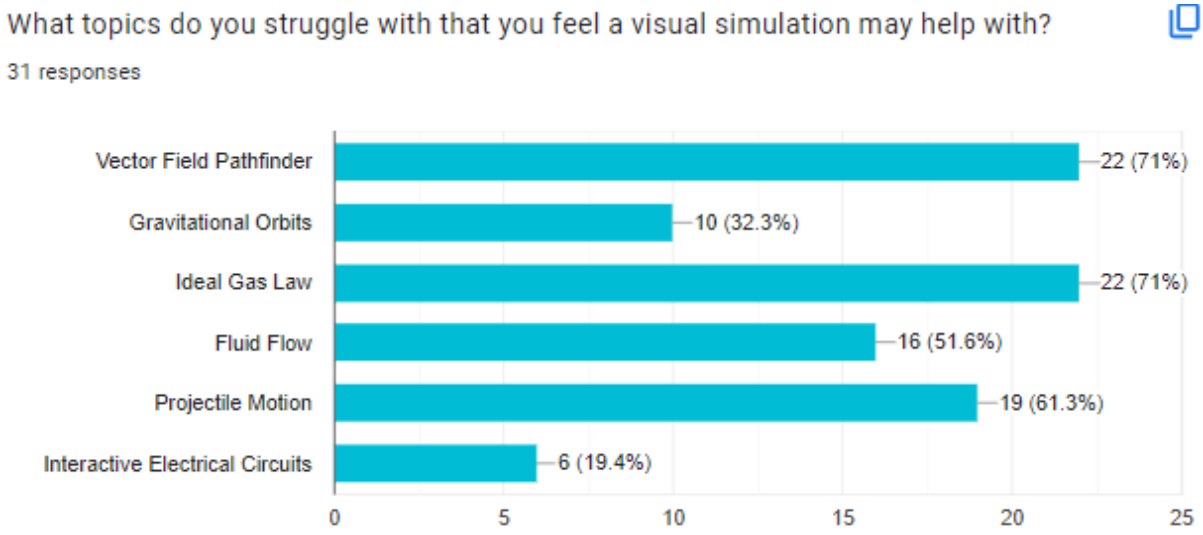


Despite being the most popular learning technique, schools do not offer regular practical experiments, with 84% not performing practical work for all topics. This is not too surprising; it is often quite cumbersome for departments to organise these on a regular basis. For example, each experiment requires a risk assessment and may have a decent cost along with it which makes it unsustainable to perform for every single topic. This means that some students are disadvantaged. As implied earlier, a visual simulation is a sensible alternative to practical demonstrations as it has no associated cost but with the interactive element that makes practical demonstrations so effective.



Confirming the above hypothesis, a resounding 94% of respondents believed that learning topics through visual simulations would increase efficiency and productivity.

Preferred Simulations Feedback



Finally, they were asked which topics the user struggles with most. I would then be able to base my choice of simulations on the responses. Joint first was a vector field pathfinder and ideal gas law with 71% of respondents . The likeness in scores may be due to the fact that a large number of students take computer science alongside physics. Whilst a quick glance might suggest otherwise, many fundamental concepts with a vector field pathfinder are found in physics, for example steering behaviours. The ideal gas law is discretely found in the A-Level syllabus and is heavily dependent on shifting parameters. As such, this makes it an ideal candidate for the program. Next with 61% was a projectile motion simulation. This is a topic that applies to the majority of all STEM subjects so I feel that it would benefit many people. Surprisingly, fluid flow made 4th place with 52%. This is a topic that is lightly touched upon in A-Level and is more profoundly studied at university. Although this would be a very challenging simulation to produce, it seems that there is a large enough number of students who would at least be interested.
I hadn't foreseen that a gravitational orbital simulation would be as unpopular as it is, with just 32% of students deciding that a visual simulation would be beneficial. This topic is a key topic in the A-Level physics syllabus and I personally found it the most challenging of the ones listed here. However, I will not be creating this seeing as it would not benefit the majority. In dead last with just 6 of the 31 deciding that it would be useful, the responses showed that an interactive electrical circuits simulation would be fairly useless. Of the possible practicals, this is one that is readily available at all schools, even at the primary school stage. What more, online resources for this very simulation are abundant and in depth, of which many have infiltrated classrooms already.

The simulations I will attempt to create are a vector field pathfinder, projectile motion simulation, an ideal gas law simulation, and fluid flow simulation, as these would benefit the majority of students.

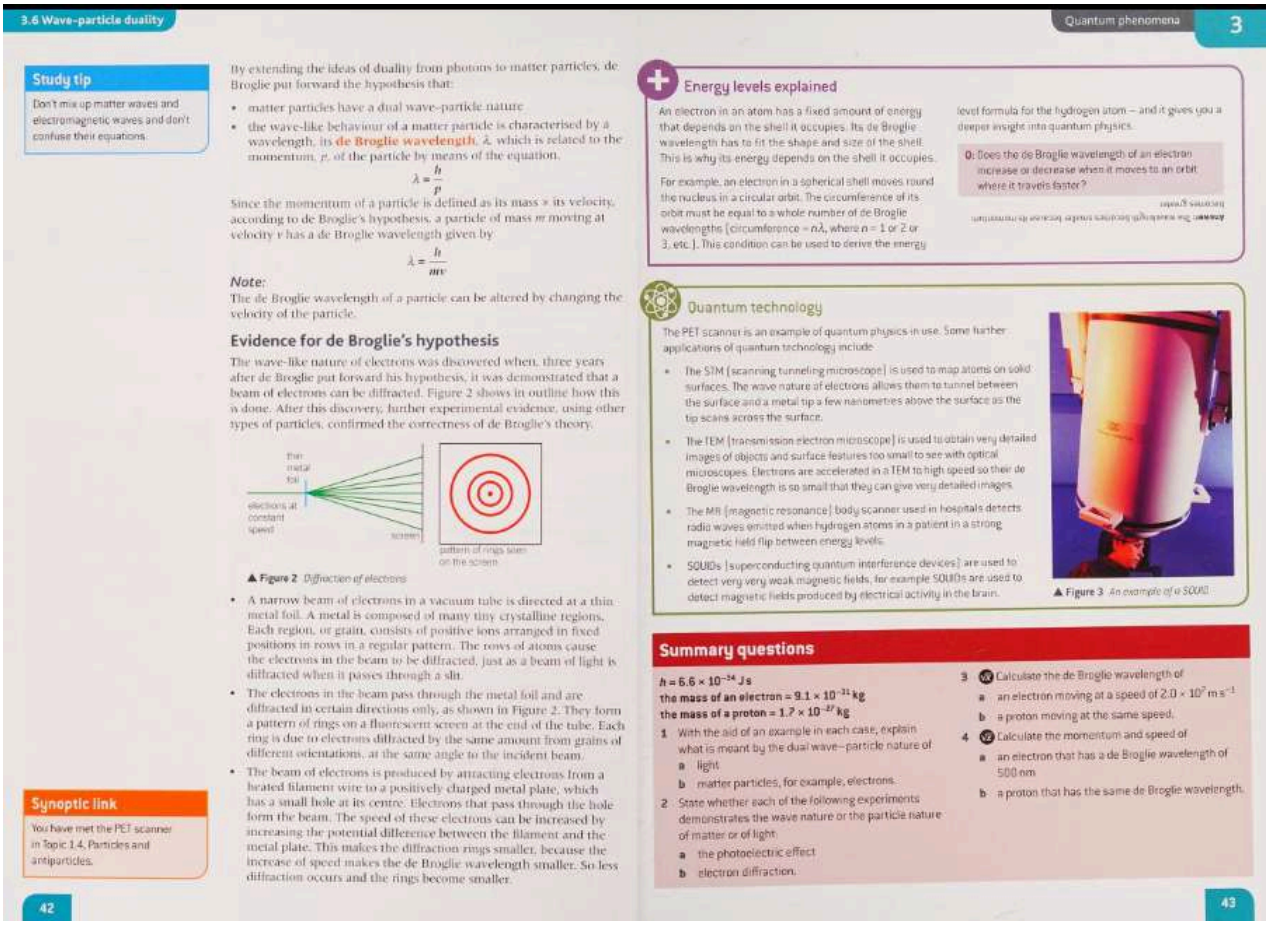
Current Systems Research

In this chapter, I will be reviewing current teaching techniques.

Current Teaching Techniques

In the requirements gathering chapter, I investigated people's opinions towards different techniques, that is textbook questions, practicals, powerpoints and visual simulations. Here, I will explore the effectiveness of these approaches. After this, I will critique a visual simulation that was high up on google search results.

Textbook Questions



3.6 Wave-particle duality

Quantum phenomena 3

Study tip

Don't mix up matter waves and electromagnetic waves and don't confuse their equations.

By extending the ideas of duality from photons to matter particles, de Broglie put forward the hypothesis that:

- matter particles have a dual wave–particle nature
- the wave-like behaviour of a matter particle is characterised by a wavelength, its **de Broglie wavelength**, λ, which is related to the momentum, p, of the particle by means of the equation

$$\lambda = \frac{h}{p}$$

Since the momentum of a particle is defined as its mass × its velocity, according to de Broglie's hypothesis, a particle of mass m moving at velocity v has a de Broglie wavelength given by

$$\lambda = \frac{h}{mv}$$

Note:
The de Broglie wavelength of a particle can be altered by changing the velocity of the particle.

Evidence for de Broglie's hypothesis

The wave-like nature of electrons was discovered when, three years after de Broglie put forward his hypothesis, it was demonstrated that a beam of electrons can be diffracted. Figure 2 shows in outline how this is done. After this discovery, further experimental evidence, using other types of particles, confirmed the correctness of de Broglie's theory.

▲ Figure 2 *Diffraction of electrons*

- A narrow beam of electrons in a vacuum tube is directed at a thin metal foil. A metal is composed of many tiny crystalline regions. Each region, or grain, consists of positive ions arranged in fixed positions in rows in a regular pattern. The rows of atoms cause the electrons in the beam to be diffracted, just as a beam of light is diffracted when it passes through a slit.
- The electrons in the beam pass through the metal foil and are diffracted in certain directions only, as shown in Figure 2. They form a pattern of rings on a fluorescent screen at the end of the tube. Each ring is due to electrons diffracted by the same amount from grains of different orientations, at the same angle to the incident beam.
- The beam of electrons is produced by attracting electrons from a heated filament wire to a positively charged metal plate, which has a small hole at its centre. Electrons that pass through the hole form the beam. The speed of these electrons can be increased by increasing the potential difference between the filament and the metal plate. This makes the diffraction rings smaller, because the increase of speed makes the de Broglie wavelength smaller. So less diffraction occurs and the rings become smaller.

Synoptic link

You have met the PET scanner in Topic 1.4, Particles and antiparticles.

Energy levels explained

An electron in an atom has a fixed amount of energy that depends on the shell it occupies. Its de Broglie wavelength has to fit the shape and size of the shell. This is why its energy depends on the shell it occupies.

For example, an electron in a spherical shell moves round the nucleus in a circular orbit. The circumference of its orbit must be equal to a whole number of de Broglie wavelengths [circumference = $n\lambda$, where n = 1 or 2 or 3, etc.]. This condition can be used to derive the energy level formula for the hydrogen atom – and it gives you a deeper insight into quantum physics.

Q: Does the de Broglie wavelength of an electron increase or decrease when it moves to an orbit where it travels faster?

Quantum technology

The PET scanner is an example of quantum physics in use. Some further applications of quantum technology include:

- The STM [scanning tunneling microscope] is used to map atoms on solid surfaces. The wave nature of electrons allows them to tunnel between the surface and a metal tip a few nanometres above the surface as the tip scans across the surface.
- The TEM [transmission electron microscope] is used to obtain very detailed images of objects and surface features too small to see with optical microscopes. Electrons are accelerated in a TEM to high speed so their de Broglie wavelength is so small that they can give very detailed images.
- The MRI [magnetic resonance] body scanner used in hospitals detects radio waves emitted when hydrogen atoms in a patient in a strong magnetic field flip between energy levels.
- SQUIDs [superconducting quantum interference devices] are used to detect very very weak magnetic fields, for example SQUIDs are used to detect magnetic fields produced by electrical activity in the brain.

▲ Figure 3 *An example of a SQUID*

Summary questions

$h = 6.6 \times 10^{-34}$ J s
the mass of an electron = 9.1×10^{-31} kg
the mass of a proton = 1.7×10^{-27} kg

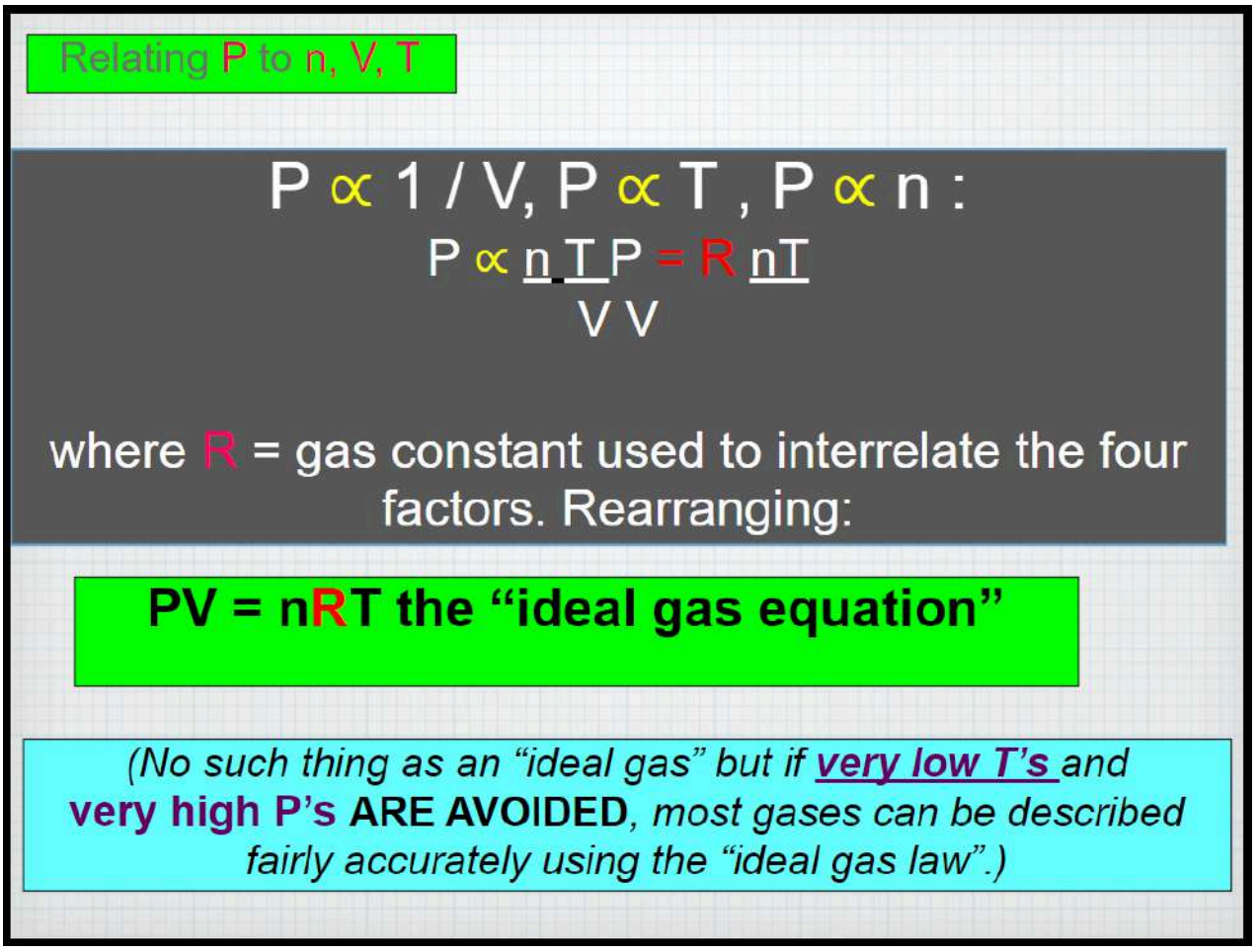
1 With the aid of an example in each case, explain what is meant by the dual wave–particle nature of
   a light
   b matter particles, for example, electrons.
2 State whether each of the following experiments demonstrates the wave nature or the particle nature of matter or of light
   a the photoelectric effect
   b electron diffraction.
3 W Calculate the de Broglie wavelength of
   a an electron moving at a speed of 2.0×10^7 m s^{-1}
   b a proton moving at the same speed.
4 W Calculate the momentum and speed of
   a an electron that has a de Broglie wavelength of 500 nm
   b a proton that has the same de Broglie wavelength.

42 43

According to my sample set (see appendix), the most used teaching technique is doing questions from the topic, as has been in the past. This is likely because textbooks are reliable sources – the textbooks are vetted by the exam boards themselves and have information directly needed for the specification. This usually means the questions too are reliable, and so wouldn't be far off what the exam board asks for in the real exam. What more, the students are usually able to complete the questions without teacher aid as the answers can be inferred from the pages prior. This promotes the student to actively engage with the material.

However, the key flaw with textbooks is that they encourage memorisation rather than understanding. Each textbook largely follows the same structure, whereby they first teach the concept, provide a couple questions on this concept, and then a final question section. Students are not encouraged nor do they have the ability to get answers to any questions they might have, and instead it turns into a rote learning exercise. If their answer does not correspond to that given by the textbook, they are led to believe they are wrong. Creativity and logical thinking feels discouraged. As concepts which rely on previously learnt information gets more complex, this flaw becomes even more pronounced. Each concept or problem is dealt with in isolation when dealing with just a textbook, so that some students struggle to make connections thanks to having to think up a new image each time. If you focus on memorising the answer, the moment you forget it you become completely lost and as such lose out on the marks; you have no recourse to try and logically think through the problem at hand and work out the answer. What more, the answer section of the textbooks feeds into this call and response mechanism. This technique feels as though it cares more about learning the exam rather than learning the concept, which does not properly reinforce a student's knowledge.

Powerpoint Presentations

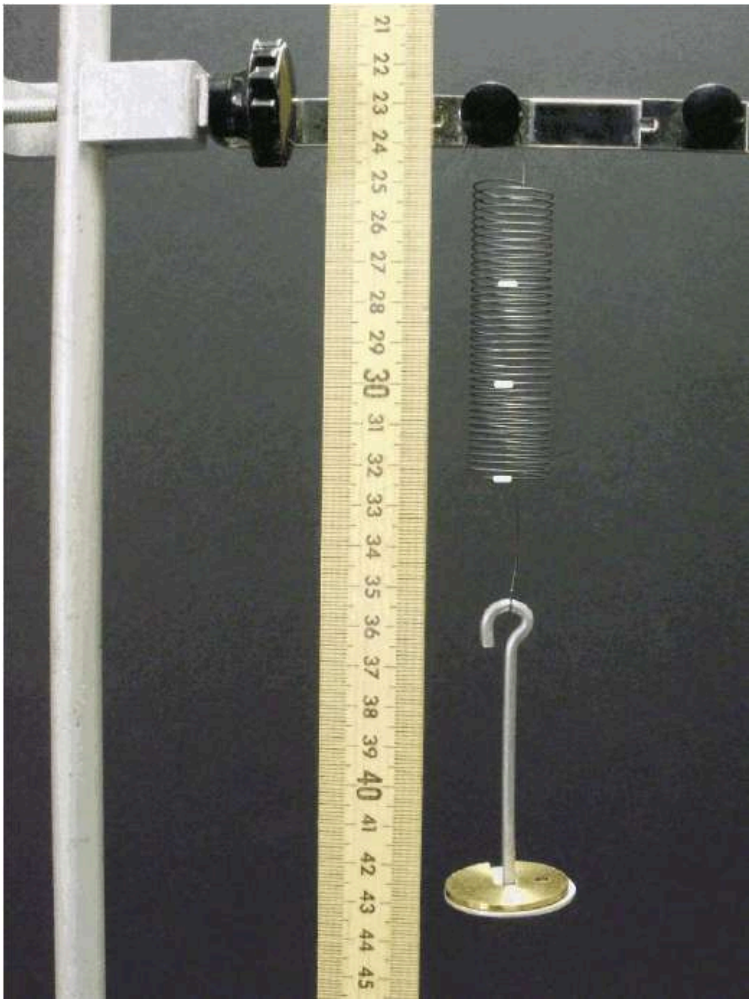


Powerpoint presentations are a widely and readily available resource which can be used to teach a class. They can easily be found online, shared between teachers, or saved on a device for many years and still hold up fine. The image is an example slide from a

powerpoint. This technique tends to drip feed information on the user so that it does not overload the user with an abundance of information. This allows the teacher to ensure that the class understands the current slide before moving on. It also allows for charts or images within slides to provide a different medium of learning, rather than just a blanket of text often found in textbooks.

I strongly believe that this technique is the weakest of the four mentioned. Firstly, it largely falls foul of the same issues as the textbook questions, in that creativity and curiosity is suppressed. Teaching can become overly reliant on powerpoints, such that students are left to monotonously read and hear the exact words off a slide, before moving on and doing the same thing for the entire topic. I said earlier that information tends to be drip fed. However, this information is rarely reinforced with continuous questions or really any sort of engagement at all, leading to very limited retention. Very often this technique relies on the pragmatism of the teacher to ensure actual learning. The learner's inactivity is the primary reason why powerpoint usage in classrooms is declining, in my opinion.

Practical Work



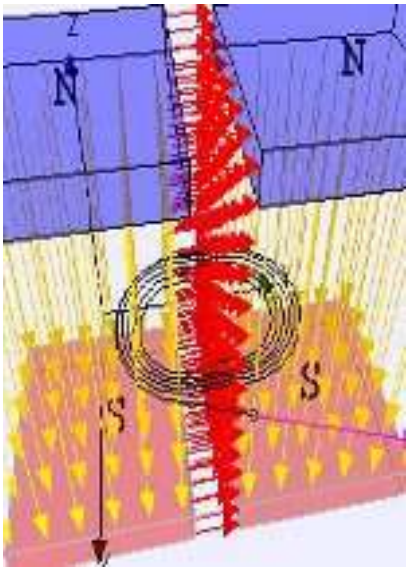
As shown in the survey, students find practical the most effective and there are countless reasons why this could be. On the right is an example of a student performing a Hooke's Law experiment. Firstly, it offers a hands-on approach to learning, forcing the user to actively engage with and apply the knowledge they have learnt, unlike in more passive techniques such as powerpoints and textbooks. Students have the freedom to experiment with certain aspects and answer any questions they might have had with the concepts, thereby promoting critical thinking rather than suppressing it. Answers to their questions can be immediately answered by the students themselves. Individual parameters can also be adjusted so that the student can observe the corresponding change in results. Moreover, they provide some actual real world context for the theory they learn, making some of the more abstract concepts more credible. Instead of having to rely on rote memorisation, a student could look back on the experiment and understand what the question is *actually* asking. Plus, some students simply can't just sit in a spot and listen to a barrage of information and be expected to absorb it all. This dynamic technique caters to students of all sorts of learning styles. With more passive techniques, a class can easily get distracted, which leads to the

class becoming increasingly erratic or increasingly sleepy. On the other hand, practical work demands teamwork and collaboration, hopefully increasing a student;s interest in the class. Exam boards clearly acknowledge the strength of this technique seeing as all three A-Level sciences have a set of required practicals that all students must complete as part of the course.

So why then is this technique not used more? The key drawback is that schools are physically unable to offer as many as they might want. An experiment requires the necessary equipment, which can become very costly. Some also require specialised equipment and facilities. Take a typical chemistry practical like the flame colours of alkali metals: each student requires a lab coat, goggles, a bunsen burner, a mat, a gauze etc etc, and also all students must work in a lab that meets specification. This can make it very exhausting for teachers to prepare one for every topic. Moreover, each practical has an arduous risk assessment process that becomes even more so if using more complex or dangerous equipment, where students are more at risk of getting hurt. These safety concerns might deter schools from conducting the more than required number of practicals. Some experiments are also practically impossible to do in a school environment, for example observing electrons in a cyclotron.

Visual Simulations - The Solution

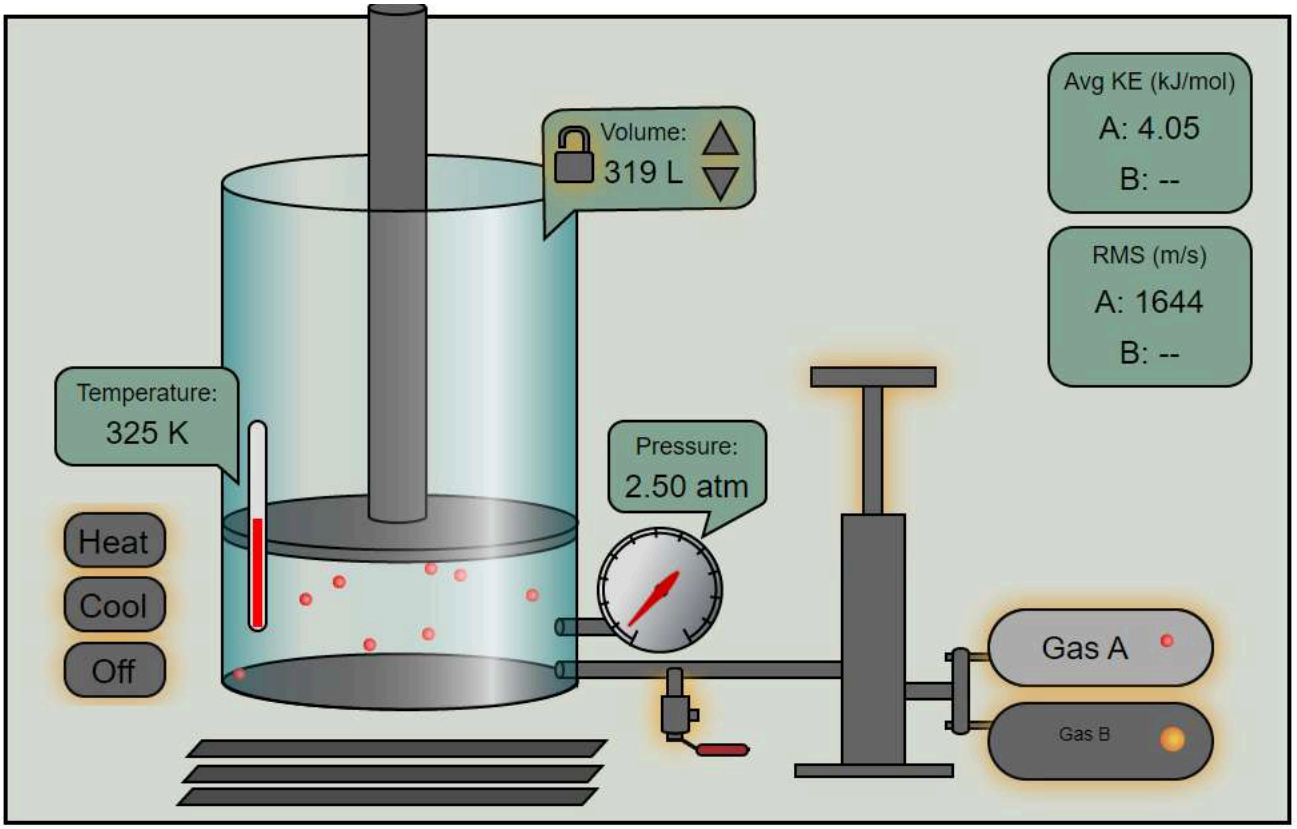
Visual simulations are an alternative solution that address nearly all of the shortcomings of the other techniques with only a slight compromise.



Firstly, they benefit from nearly all the positives mentioned in regards to practical experiments – they after all doing the same thing as an experiment, just on a display. What more, it allows for the presentation of practicals that couldn't happen in a school, such as the aforementioned cyclotron experiment. With a simulation, you can also change parameters to an extreme that would not be allowed in a school, like increasing the potential difference of a circuit to a high value. All risk assessments and needs for specific equipment are bypassed; again, it is taking place entirely within the screen. This means that a simulation, unlike practical work, can be used in nearly all situations. While I concede that a simulation is not as immersive or interactive as real life practical work, it is nonetheless miles better than anything powerpoints or textbook questions could offer. Being able to actively manipulate variables and visualise concepts is something that is painfully lacking in those techniques. Simulations therefore address their key limitations.

There are a few drawbacks. In order to allow each student to interact with the simulation themselves, they would have to have their own device. This requires the school to have the appropriate infrastructure and resources. Fortunately in my case, the school made an investment into a set of laptops, which can be provided to students for use during a lesson. The use of simulations is a newer teaching tool, meaning that the standard teacher training does not necessarily involve their usage. As such, it will take time before teachers can effectively integrate these simulations into lessons.

In conclusion, visual simulations are a powerful tool for teaching students, in particular those studying A-Level. They are undoubtedly the future, it is just a matter of having the right awareness, infrastructure, and most importantly well-made simulations before it becomes the present, the later of which I aim to produce.

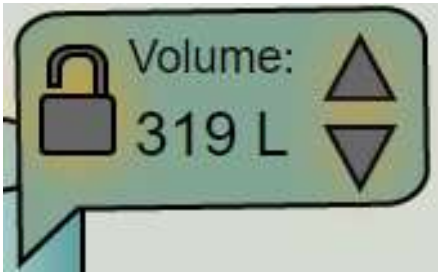


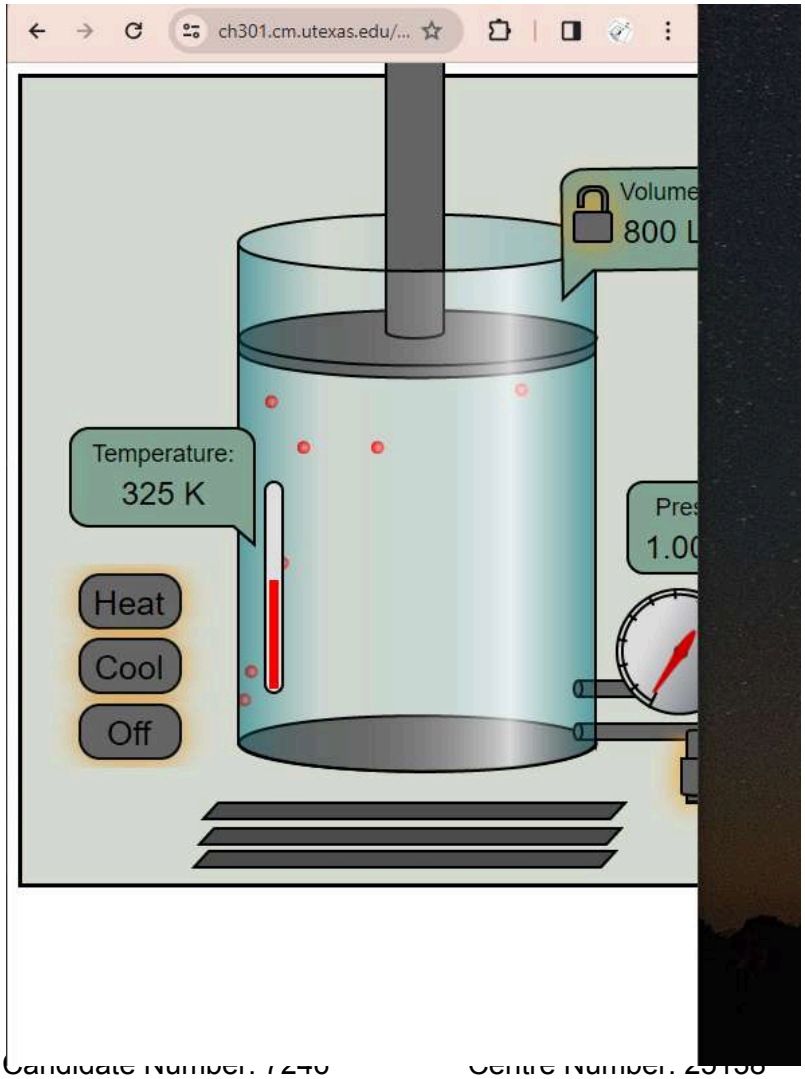
University of Texas Simulator

First Impressions

When I searched for an ideal gas simulation, the second link was a gas law simulator from the University of Texas. When I clicked on the website, I was bombarded with widgets hidden amongst a pump and container. This falls foul to overloading, as there is an

unnecessary amount of clutter in the screen. Widgets to control the simulation are scattered in between boxes which purely display information. This means that the user can get confused over what boxes can be interacted with to control the simulation. The program attempts to compensate for this oversight by placing yellow highlights around interactable buttons, but by no means is it pronounced enough to immediately tell which is interactable. The pressure display in the ideal gas law is the defining variable and so should be immediately distinguishable. However, this widget is almost identical to the other widgets, with one small difference being that it is ridiculously smaller in size. On the far right are two more widgets stating the average kinetic energy and the root mean square speed. Again, they aesthetically have little difference to the other widgets other than being further away from the centrepiece. The unexplained A and B initially seemed cryptic to me. Only after adding more particles did I realise that they referred to the two different gas types. Frustratingly, the user can't tell the difference between gas A and B until they add the respective particles. I believe it would be much better to simply label them light and heavy particles rather than just a pair of meaningless letters. I also find the colour scheme to be very bland and it does not provide enough contrast for the text to be clearly legible. For reference, on the right is the temperature control. Note that the buttons' backgrounds are close in shade to the black text, thereby camouflaging these important controls.



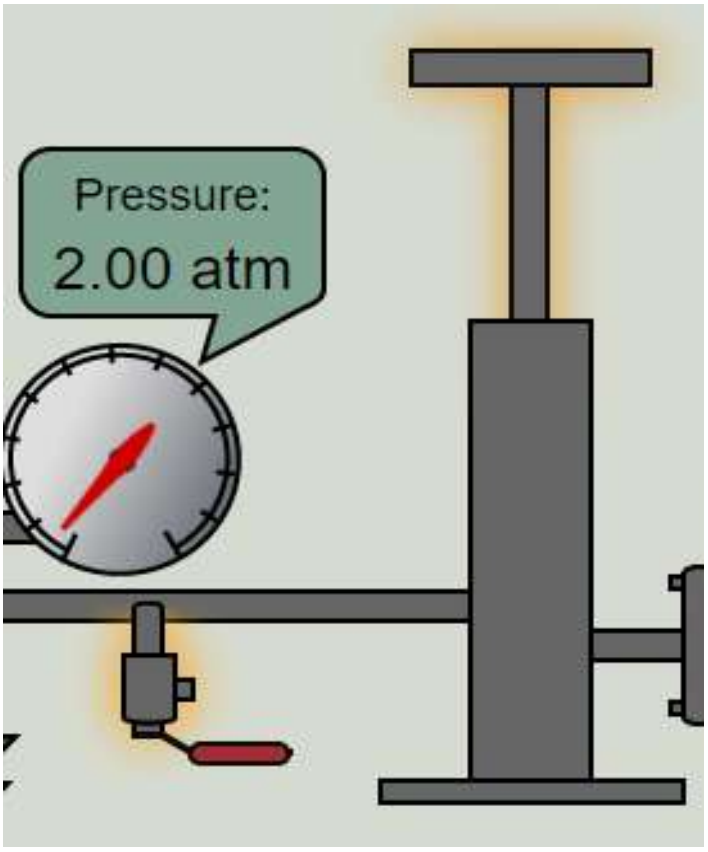


This display also does not adjust itself according to the user's display size. The simulation is a fixed size regardless of the window it's contained. In. This means that if the user, say, clicks on the simulation in split screen, some of the display will be cut off and inaccessible. For my simulation, I would therefore like to have simulations which adapt to the display dimensions, or make it so that the simulation runs in full screen

User Interactions

The function of the buttons can also be difficult to see as well. Again referencing the temperature controls, there are three buttons when I believe there should be just two. The simulation has a heater underneath the container. The ‘Heat’ button when clicked once begins increasing the temperature indefinitely and changes colour to signify this. Conversely, the ‘Cool’ option does the same but decreases the temperature indefinitely. To stop the temperature change, the user can either click on the initially selected button again, or select the off button. I found this incredibly overcomplicated. The click to toggle rather than a hold feels very clunky as it is easy for the user to miss the temperature widget to observe the temperature change. This often leads to the user overshooting their desired temperature as the temperature display feels obscured and disconnected from the buttons themselves. Changing the interaction to a hold would also mean that the redundancy in the Off button could be removed, thereby reducing the excessive overloading.

Furthermore, it is not obvious what the lock symbol on the volume widget does. Here, it keeps the volume in place. When the pressure is greater than 1 atm, it means the pressure within the container is greater than the outside. As such, a force is exerted on the box. Personally, I expect the lid to have a weight associated and so would oppose the force exerted, especially if the pressure difference is minimal. However, even with a pressure of 1.01 atm, the container instantly expanded. Instead, the program offers a vague lock symbol to keep the volume constant. Considering this is a learning tool, it seems obsolete and unintuitive to have a lid - which clearly has a weight to it - to react to the smallest of pressures. The buttons also don’t interact the way I expected them to. Similar to the temperature controls, adjusting the volume is through a toggle rather than hold. As well as the aforementioned problems, there is also the fact that this design choice makes it incredibly difficult to achieve a specific volume. For example, in the simulation I attempted to get a volume of 190 L; I was forced to perform double clicks on the buttons back and forth to zone in on 190 L.



The last major flaw with the widgets is the manner in which the particles are added and removed. In real life, you would expect that when a pump is dragged up and down, an according number of particles would be added. However, the program responds to just a click, at which point a program defined number of particles are added to the simulation. This is meant to be a teaching tool, yet they leave the user to estimate the particles in the container themselves. I also felt that the number of particles added per ‘pump’ was excessive; the simulation quickly reached some ‘particle limit’ which prevented me from adding more

particles. The increment should be much lower so as to allow the user to fine tune this parameter. The release gas mechanism is in my opinion the strangest of the interactions. Once again it suffers from the lack of mouse held down controls, as the valve temporarily opens for a fixed amount of time when clicked. I initially guessed that the programmer wanted for more particles to leave when the pressure is higher in an attempt to imitate how it would act in real life. However, I believe the implementation is unreliable and therefore flawed. I filled the container with the most particles I could use and set the temperature such that the pressure was at 7 atm. I then selected the valve and recorded the volume. As you can see from the right, it initially decrements by 100 litres, but then suddenly decides to change the interval to 50, and then 150. I can't think of a reason why this would be the desired effect. I believe it would be much simpler to simply keep the interval a constant, as this is the most straightforward technique for the user to conceptually understand.

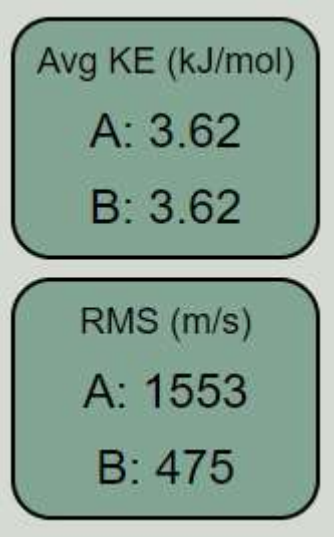
Valve Release Count	Volume / L
0	1000
1	900
2	800
3	700
4	600
5	500
6	400
7	300
8	200
9	150
10	0

Good Features

Although I am very critical of the programmer's design choices, there are nonetheless many features that are effective and would be excellent for my own implementation.

Firstly, the 3D concept is visually appealing and gives the impression of depth which many other simulations lack. However, I would argue that what it gains in aesthetics, it loses in educational value. For example, here there are no particle collisions, whereas in a 2D simulation, collisions are inevitable and so we can observe how collisions are perfectly elastic, which is an important fact with the ideal gas law.

I find the programmer's decision to use real life apparatus to be very engaging. As discussed in my requirements gathering, a main talking point for visual simulations is that they simulate a real life experiment when a student might not physically be able to perform it in real life. By using real apparatus, it shortens the gap between a simulation and performing the experiment in real life.



Finally, I think the user has chosen a very good set of parameters to present to the user. At the top right of the display, two widgets are shown which state the average kinetic energy of the particle, and the root mean square speed of the system. These provide useful insights into the properties of the particles and help to compensate for the lack of visual output from the container.

Index of /simulations

Name	Last modified	Size	Description
Parent Directory		-	
bond-strength/	2012-08-23 08:42	-	
emr/	2012-08-23 08:42	-	
gas-laws/	2012-08-23 08:42	-	
ideal-gas-law/	2013-08-30 15:12	-	
js/	2014-08-07 17:26	-	
photoelectric/	2012-08-23 08:42	-	

The above image shows how the programmer intends to make the user navigate their simulations. Here, they organise the simulations into a file system, giving each folder a name for their respective simulations. The column headings are Name, Last Modified, Size and Description. Both the SIze and Description values are empty. All bar one are no longer working as they relied on Adobe flash player, which reached its end of life a few years ago. Only the ideal gas law found in the js folder (which I can only assume stands for javascript) is working.

Pros

- ➢ This is a very straightforward and simple approach to providing a user interface.
- ➢ It is fairly intuitive with the folders being shown in the standard link colour. This would point the user towards where they must click to progress onto the next step.
- ➢ The folder names indicate vaguely what simulation is contained within.
- ➢ The folder hierarchy is decently organised, so that users would be able to quickly locate a simulation according to their preferred topic.
- ➢ The file system loads very fast thanks to the lack of processing going on.
- ➢ The date format is YYYY-MM-DD which avoids confusion that might arise with DD-MM-YYYY and MM-DD-YYYY formats.
- ➢ The addition of a last modified date also helps the user to understand which simulations have received recent updates.
- ➢ This system allows for a lot of expansion as the programmer can just append a new record onto the end of the table.
- ➢ The column headings are clickable as indicated through the programmer use of font and colour. When selected, the program organises the folder in the directory according to the selected heading. For example, clicking on Last Modified shows the most recently modified folders, and clicking once more gives the least recent.
- ➢ The theme prioritises clarity and efficiency rather than aesthetics. The screen is uncluttered and there are no visual distractions, ensuring the user focuses purely on the folders.

Cons

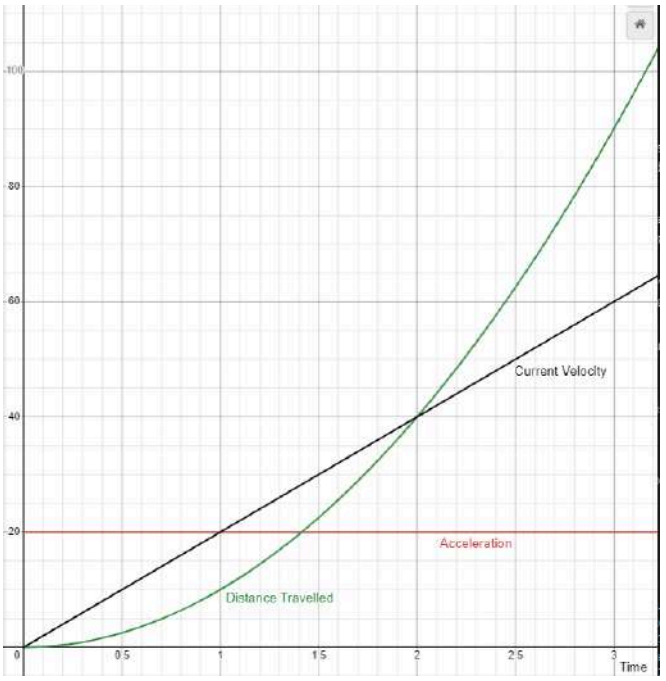
- ➢ The Size and Description headings are entirely redundant as none of the simulations have any of these properties (I checked through all folders). They should be removed to prevent overloading the user with useless clutter.
- ➢ The icons are not very descriptive. The folder gives the impression that there are more subfolders or files within. However, except for the js folder, they all link directly to the simulation. As such, I believe a different icon should be used for these to differentiate them from the js folder.
- ➢ The folder names are hardly descriptive and only offer a vague understanding of the simulation. For example, the acronym for electromagnetic radiation in the ‘emr’ simulation is very vague and most people would not know what it stands for. Plus, a simulation for electromagnetic radiation could allude to many things. As such, I believe the folder names should be more specific.
- ➢ If the number of simulations gets too long, then it can become very cumbersome to find the desired simulation, even more so without search functionality.
- ➢ The system does not respond well to different screen sizes as it remains a fixed size. On my 1920x1080 monitor, it only takes up the top left quadrant. Perhaps taking advantage of all the empty space would engage the user more.
- ➢ The theme here feels overly professional and uninspiring. I understand the programmer’s reasoning for this as it was originally designed for university students. However, it feels excessively dull and perhaps reflects the changes in graphical design since this program was produced.

In conclusion, I believe this design choice is on the extreme end of the scale. It focuses on professionalism and straightforwardness, which are not necessarily priorities for me. This simulation is also nearly 12 years old; many computers might not have had the processing power to handle more complex graphics. My target demographic are A-Level students with more than capable devices. My program can therefore afford extra embellishments and creativity to engage the students and prevent them from becoming distracted.

Modelling

Particle Fundamentals

Before I begin researching, I will explain some of the fundamentals. All objects have a mass. In free space, an object will have a displacement, which is just the distance from its origin. In my simulation, this will simply be the coordinates according to the pygame window. Differentiating displacement with respect to time gives the velocity which again all particles will have. This property gives you the speed and direction the particle is currently travelling. Differentiating again gives the acceleration. Each time step, the displacement will be incremented by the velocity vector and the velocity vector by the acceleration vector (both of which will be scaled the time step - in my simulation – if I'm aiming for 60FPS, this gives a time step of 1/60 seconds). Imagine a particle floating in free space, initially at rest. As shown on the right, I decide to apply 50N to the left and 30N to the right. The resultant force here is 20N to the left, or -20i N. From there, I can use Newton's second law to calculate the acceleration: F = ma and a = F/m. Solving for a gives an acceleration of -10i ms^-2. So, each second, the velocity will increment by -10i and the displacement by the new velocity. Observe the graph I made to showcase the relationship, where the red is the constant acceleration, the blue is the velocity (linear here) and the green the displacement. This fundamental idea applies to all my other simulations but is most evident in projectile motion.

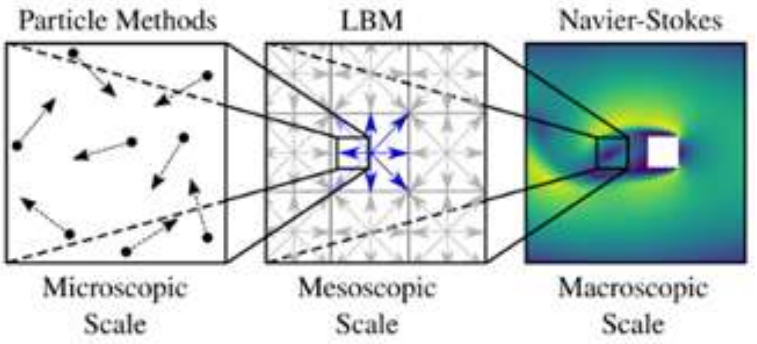


Fluid Flow Research

I would like to create a simulation which can be applicable to the real world. The final simulation should be able to resemble real life properties at face value. The user should be able to interpret, say, the motion of a container of fluid from a 2D simulation. The simulation should be both physically and mathematically accurate in all aspects, as to ensure that the user would be able to observe how different factors can affect the flow of fluid in the real world. This field is entirely new to me so here I will document the process in which I learn some of the fundamentals. In order to make this readable and concise, I will not be mentioning certain topics which might unnecessarily overcomplicate things at all. For a deeper explanation of SPH in particular, please refer to the paper used in the references.

Problem of Scale

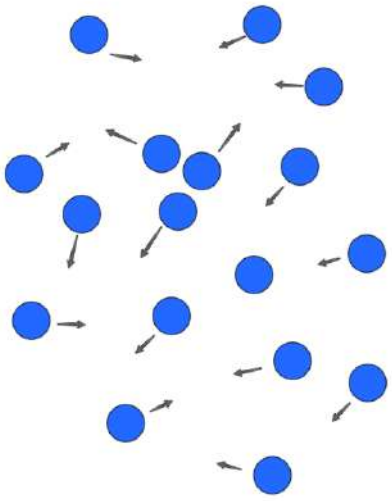
My first decision was which scale I should make the simulation at. A smaller scale represents a greater degree of depth.The model on the right largely captures how each of these options relate to each other.



Atomistic Simulation	Mesoscale Simulation	Macroscale Simulation
This is where we deal with each molecule individually.	This is where we look at pseudomolecular interactions, using probabilistic methods.	This is the most popular category as it is the most accessible for people.
This typically lies in the nanometre scale.	This is an in between scale, between atomistic scales and the more general macroscale scale	This is the most zoomed out scale of the three categories, and offers visually appealing simulations applicable to the real world
To simulate this, we would have to use Newton's equations and implement molecular dynamics	To simulate this, we would use probabilistic methods, such as the Lattice-Boltzmann theorem. This uses a grid structure as opposed to the others which look at particles to represent fluid.	To simulate this, you can choose a Lagrangian approach or an Eulerian approach. I will discuss this in more depth in the upcoming chapter.

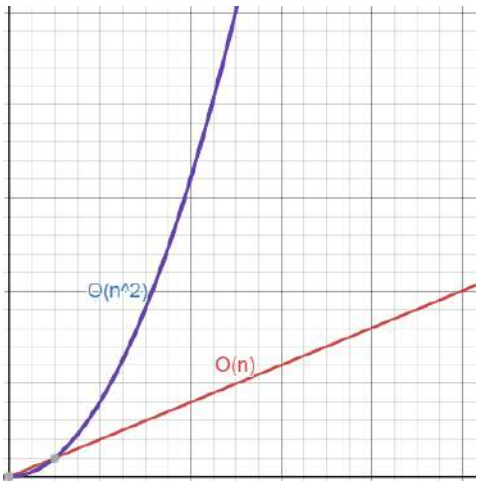
In conclusion, I am opting for a macroscale approach. My aim is to provide a simulation that offers a visually stimulating approach which can be used to perform and predict real-life practicals in the physics syllabus. Whilst an atomistic simulation would also be applicable to physics, such as the ideal gas law, I would like to make a simulation that would be closely related to fluid dynamics.

Eulerian vs Lagrangian



Sketch showing how the fluid moves around in Lagrangian method

When it comes to simulating a fluid at the macroscopic scale, there are two mainstream methods. The Lagrangian methods attempt to present fluid flow through individual particles, where we calculate the trajectory and properties of each particle separately. These particles do not represent an individual atom like an atomistic simulation, rather it represents a quantity of atoms and approximates their general properties. On the other hand, the Eulerian approach looks at a more abstract solution, where we deal with a concentration of particles and calculate the overall diffusion and convection of the given concentration of particles. Typically, a grid or a mesh is used for this. For example, each cell could have a pressure value. ‘Fluid’ or ink would then be dispersed to surrounding cells based on properties such as viscosity and pressure. This fixed grid-like structure struggles to calculate densities as well as smoothed particle hydrodynamics if the grid size is not precise enough. Instead for my program, I will implement a Lagrangian solution. A Lagrangian method, such as smoothed particle hydrodynamics, more explicitly references the Navier-Stokes equations, which is the set of equations studied by students when learning about the physics of fluid flow. Students would be able to manipulate parameters to experiment and gain a deeper understanding on the effect a given attribute of the fluid would have - for example, increasing the viscosity would lead to a thicker fluid due to the greater internal friction. The student could also apply their studies of partial differential equations into the program, as SPH requires an integration of the Navier-Stokes equation below. Moreover, this method would allow the user to easily experiment with various physics phenomena which are studied in the A-Level syllabus, such as the behaviour of a liquid in a container or ripples in water.



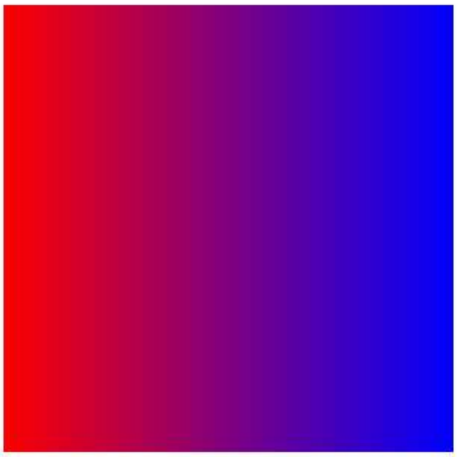
I believe a Lagrangian method is the best solution, so that students can make links with their classroom theory. Plus, when scaling the given implementations, the grid based Eulerian method is much more computationally expensive to increase the grid resolution in comparison to increasing the . Increasing the size of the grid-based approach has a time complexity of O(n^2) in comparison to the linearly

increasing particle count of lagrangian methods where it is relatively cheap to increase the number of particles (O(n)). This means that the user can scale up the Lagrangian implementation much more than they might be able to with the grid based method, thus demonstrating a better efficiency and scalability. As such, I will aim to implement a smoothed particle hydrodynamics (SPH) fluid flow simulation.

SPH Overview

Smoothed particle hydrodynamics is a method of implementing the Navier-Stokes equations. It works by giving approximations for each term in the equations, or in other words, it deals with the maths. Before I attempt to digest the Navier-Stokes equations, I would like to get a brief overview of what a basic fluid flow simulation involves. In SPH, the fluid is represented by a fixed set of particles that each represent a concentration of fluid. These particles interact with each other according to the laws of physics. Each particle will hold the expected fundamental properties: mass, displacement, and velocity. Additionally, they will have properties which will enforce the Navier-Stokes equations. Initially, I found it difficult to see how a 2d image with particles dotted around could possibly represent a real-life 3d box of water, for example; it is not immediately clear how the water's behaviour is translated into the SPH simulation. This simulation will be a birds-eye view of the container of fluid after all. Take this glass of water.

Here, we've taken a snapshot of the water sloshing against the left hand side wall of the glass. If you were to take a top-down view of the water, you would not notice this sloshing effect as it's only visible from a side angle. To communicate this property into a 2d plane, we can group together particles to create a larger density at a given point. I believe this density property translates to depth in the water. The gradient to the right of the water is a crude representation of the density or depth of the fluid from a birds-eye view where red is a larger density, and successfully communicates the depth of the water at a given point. Alternatively, you can also think of it as representing the side-view of the glass, though I would like to first implement the birds-eye view so that the program can be used to observe wavefronts for example. This makes the program a more effective tool in expressing the behaviours of fluids.



$\nabla .u = 0$

$$\rho \frac{D\vec{V}}{Dt} = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{V}$$

Nearly 200 years ago, George Stokes and Claude-Louis Navier began working on a set of equations to describe the motion of fluid. Their work was called the Navier-Stokes equations. The equations I will need are the incompressible variants. Let's say we have a set of particles. I'm looking to have the particles move such that they simulate how real water would move. By making my simulation as physically correct as possible, the simulation will be much more convincing. Note that I aim to only provide a basic fluid flow simulation, so I will not attempt to explore more complex features like turbulence.

Symbol	Meaning
ρ (the greek letter)	Density. Can be vaguely interpreted as mass as density = mass / volume Initially I had trouble understanding this concept. I found it helpful to think of similarities with how inertia is used in rotational motion, as inertia in that regard is the mass equivalent. Density can be thought of as mass when dealing with fluid flow.
du/dt	Acceleration AKA a, here shown to be the derivative of velocity with respect to time.
p	Pressure
∇ (del operator)	The gradient. It is an operator used with vectors, and so can be combined with other vector operations such as the dot and cross product. In the first equation, it means the divergence of velocity
u	Velocity AKA v.
s	Displacement or distance. Note that dx/dt = velocity and d2x/dt^2 = acceleration.

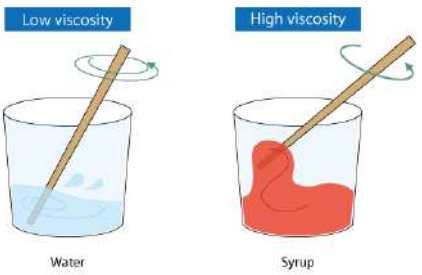
The first equation simply enforces the conversation of mass. The ∇ or nabla here is dotted with the velocity vector, and states that there is no divergence in velocity. More specifically, the velocity vector in my simulation has an i and a j vector (a horizontal and vertical component). This equation looks at how much the velocity changes in each component changes and ensures that the sum of these values cancel out to give 0. Essentially, it states that the rate of which the fluid leaves the space is equal to that which enters the space. In my SPH simulation, this is automatically met as the particles representing the fluid will be bound within the program boundaries. To simplify things, I will only be dealing with incompressible fluids, which is not too much of a problem seeing as most of the examples students deal with initially are indeed incompressible. Fortunately, in an SPH approach, the mass in the space remains the same as I am using a particle-based approach, where the particles are bound to the given space.

$$\rho \frac{D\underline{u}}{Dt} = -\nabla p + \mu \nabla^2 \underline{u} + \rho \underline{F}$$

The second equation is the most important and most challenging. Firstly, it is a partial differential equation. I have only just begun learning about ordinary differential equations in maths and the partial variety is not encountered until university. Firstly, ordinary differential equations I have encountered in maths revolve around a single variable and can be solved very often using standard methods to find equations for all variables. However, the partial differential equation here involves multiple variables and is far beyond my understanding so I will be relying on online resources. Ideally, we'd be able to integrate the du/dt to get an equation for the velocity and again to get an equation for the displacement. However these equations, given the necessary parameters, can't be solved analytically and you have to use numerical methods, such as SPH. I am struggling to understand the exact property of partial differential equations responsible for this nor the specifics of the numerical methods but the paper I am using allows me to bypass this through providing the necessary formulae

The next equation is essentially Newton's second law, with the right hand side being the sum of three different forces. One force is fairly trivial to implement in my program; the external forces (here shown as F). In my program, this will either be g to show gravity, or just left out. The other two are internal forces and will be the bulk of my effort in implementing this.
Firstly, p stands for the pressure. To explain my perspective of this, I will again be referring to the sloshing glass of water. In this current snapshot, we know that the water will try to level out and so the sloshing water will try to flow towards the other side. In other words, the region of higher density will have a higher pressure, high pressure goes to low pressure and the fluid will have a force accelerating the molecules towards the lower pressure. This force is called the pressure force.

The other force is the viscosity force. This is the fluid's resistance to movement, and is essentially the internal friction between particles. Practically speaking, thicker fluids flow slower. You can observe the influence of viscosity when pouring something like oil compared to water. In the glass of water, the water will quickly flow to the other side. If it were oil, it appears to flow more slowly and seems to be stickier.

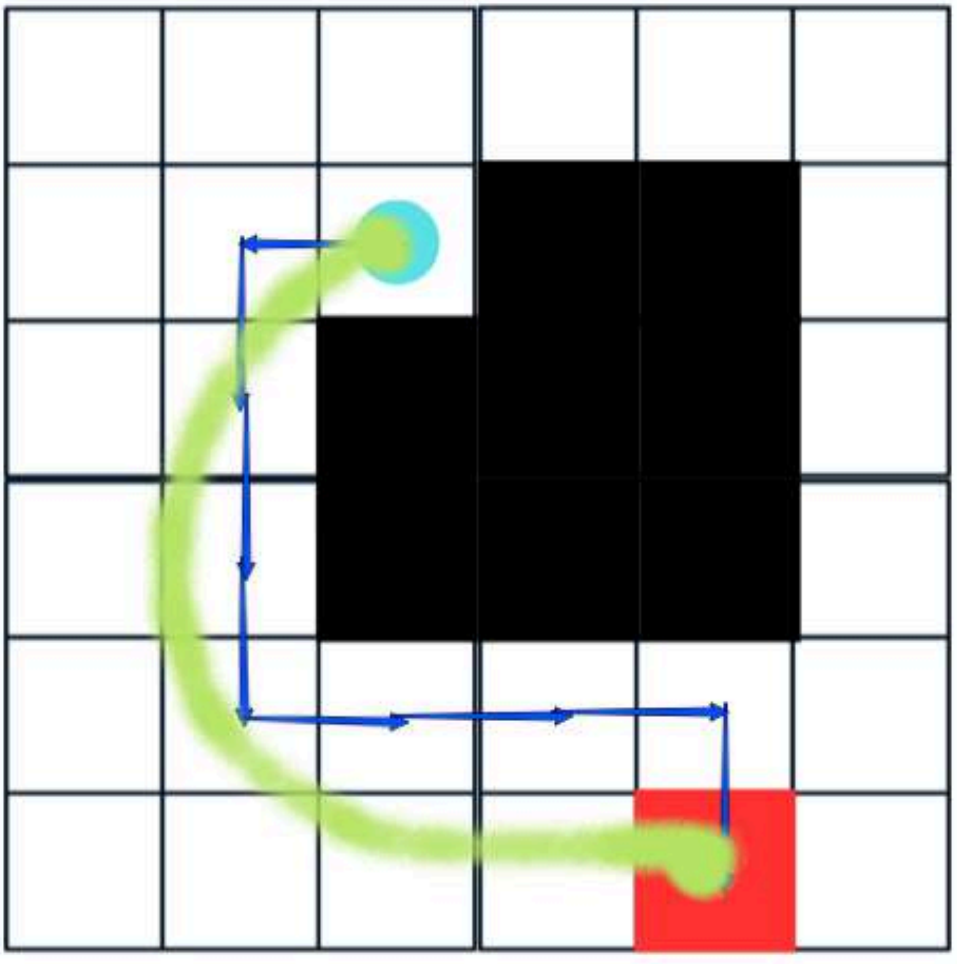


Vector Field Pathfinder

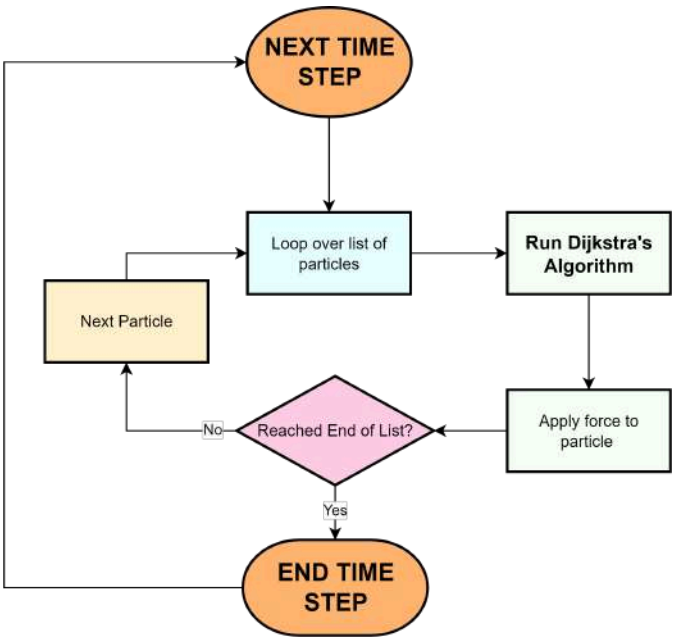
Choice of Pathfinder Justification

As seen earlier, the majority of my target demographic take computer science and physics. As such, I wanted to create a pathfinding simulation that would cater to this demographic. From personal experience, students get taught about the Dijkstra and A* pathfinding in a fair amount of depth at some point during the computer science course. These do not translate well to physics; there are two major reasons for this.

1. Firstly, particle movement can appear robotic and janky. For these to operate effectively, the particle takes a specific path calculated at run time. This removes any sort of fluid-like properties or any individuality for that matter, as the particle is forced to stay within the specified nodes, else have to recalculate the pathfinding algorithm. If the designer decides on the latter, that is recalculating the path every time step, it becomes very inefficient. As such, these less robust pathfinders are not applicable to any types of physics simulations. The below shows the conventional path a particle might take according to Dijkstra's algorithm in blue. As you can see, it is uniform and very unnatural. In green, I have attempted to display the vague path I would like the particle to take. This appears more natural and feels less predefined. What more, you could realistically believe that this is the path a particle might take in a body of water with similar obstacles, or in other words simulate current behaviour.



2. Secondly, a traditional pathfinder becomes very hardware intensive if there are many particles. Assuming that a programmer would have decided to run the algorithm every time step in order to not restrict the particle to specific nodes, I have included the flowchart on the right. As you can see, it would mean that the algorithm would have to run for each particle, each time step. While in isolation Dijkstra's algorithm is more efficient that a vector field pathfinding algorithm, it significantly deteriorate when more particles are added.



Pathfinder Properties

My ideal pathfinder would be something that could crudely resemble fluid current behaviour, and have the capacity for hundreds if not thousands of particles. A vector field pathfinder fits this near perfectly.

- ☑ The particle is able to have a velocity value which is only partially changed by an external vector, meaning there are no sharp turns as you might have seen with a Dijkstra implementation. The particle feels much more fluid-like and dynamic.
- ☑ With this pathfinder, each cell in a given grid is assigned a vector which in turn influences the particle. The particle is simply a free body; it does not perform any calculations other than getting the velocity influence from the cell. As such, increasing the number of particles does not have a minimal effect on computational efficiency.

However, there are also a few limitations which I will have to consider:

- ☐ The strength of the cell vector on the particle must be tinkered with carefully. Too large an acceleration means that particles might overshoot and miss a turn too often, especially in a maze-like environment where there are many sharp turns. This in turn tarnishes the illusion of a smooth, flowing fluid.
- ☐ The speed of the program deteriorates as the size of the grid increases. This is thanks to the nature of the breadth-first search algorithm, which is further discussed in the Key Algorithms Section.

Ideal Gas Law Research

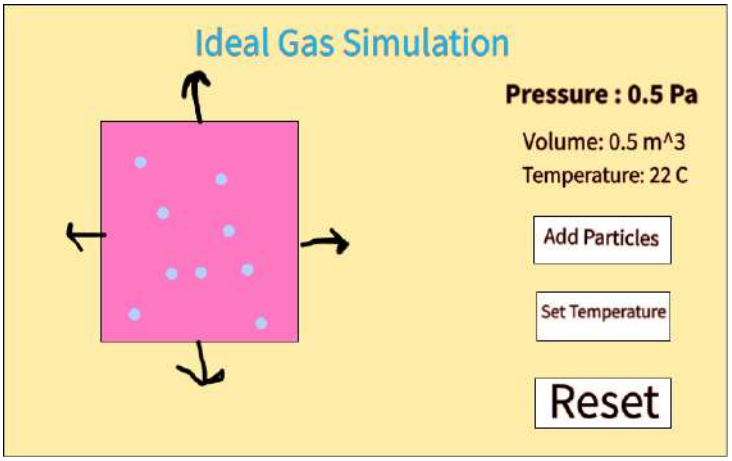
Picture a can of deodorant. Midway through a gruelling day, you decide to spray some on yourself to freshen up. All of a sudden, you feel the can cool down in the palm of your hand. What about your fridge at home; how exactly is the fridge able to turn room temperature air into colder air? These can all be explained with the ideal gas law equation, which is a combination of Boyle's law, Charles's Law and the Pressure law. Given a gas in a container, we can relate pressure, volume and temperature.

Value	Units	Additional information
Pressure	Measure in pascals, or in N/m	Calculated from the number of collisions against the container wall by the particles
Volume	Measured in m^3	The space within a container
Temperature	Commonly measured in Celsius, but can be measured with an absolute scale with Kelvin, where 0K is absolute zero. Note that celsius increases linearly with kelvin, such that 0 C = 273 K	The average kinetic energy of particles in a system
N	A number	The total number of particles in the system
k	The Boltzmann constant	This is 1.38*10^-23

The ideal gas law states that PV = nRT, or (pressure x volume) is directly proportional to (no. of moles x temperature). My aim here is to provide a container with which the user can alter its properties to observe a change in pressure.

An ideal gas has specific properties that I also must adhere to make the simulation applicable to the syllabus.

- ★ All collisions must be elastic
  I will set the coefficient of restitution to 1 to ensure no energy is lost
- ★ No intermolecular forces, bar collisions
  I will simply not implement functions of this kind.
- ★ The gas can't turn into a liquid or a solid, i.e. it must remain a gas at all times.
  In real life, if you cool down a gas enough, it will turn into a liquid, for example steam into water. In my simulation, I will not implement any support for fluids or solids or even gravity, so this is not an issue..



My simulation will have the property of pressure at its forefront. The user will then be able to change the other 4 parameters to observe how the pressure changes. I created a small graphic showing my vision for a straightforward and concise interface. Here, I have the main container on the left. Unlike the University of Texas simulation, I want to bunch up all clickable items together on the right for easy access. The user will then be able to select the buttons to alter the parameters of the simulation

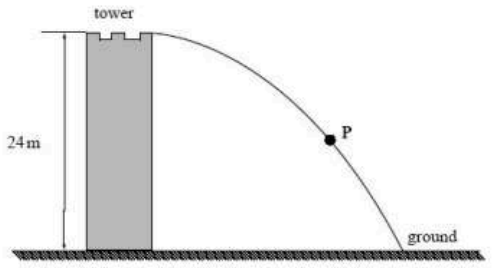
Projectile Motion Simulation

This simulation is directly linked to both the physics and maths A-Level syllabus. It revolves around predicting the path of an object given a set of constraints. I aim to provide a realistic simulation of a projectile in a 2D space.

Real Life Impracticality

On the right is an example A-Level physics question. Here, they have given a diagram to show the path of the ball, however this was very much an exception. The vast majority of questions only offer a few values and expect the student to understand. For those who are new to projectiles, this can be very difficult to visualise; teachers are often unable to perform demonstrations as these are either too small-scale to be reasonably accurate, or it becomes impractical and unsafe. My proposal is that students could use my program to calculate the initial velocity to reach a certain goal. The user will then accrue a score corresponding to the user's accuracy.

Q7. The diagram below shows the path of a ball thrown horizontally from the top of a tower of height 24 m which is surrounded by level ground.



(a) Using two labelled arrows, show on the diagram above the direction of the velocity, *v*, and the acceleration, *a*, of the ball when it is at point **P**.

(2)

(b) (i) Calculate the time taken from when the ball is thrown to when it first hits the ground. Assume air resistance is negligible.

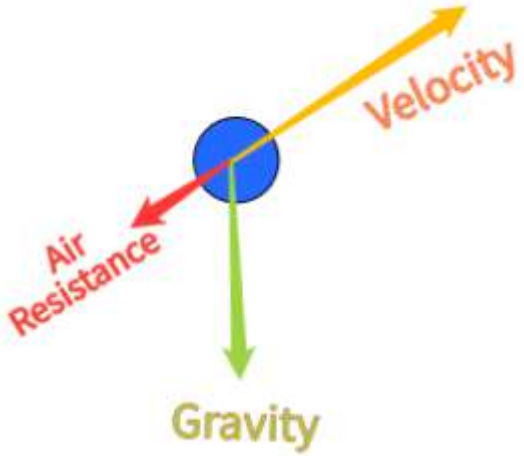
Answer ..... s

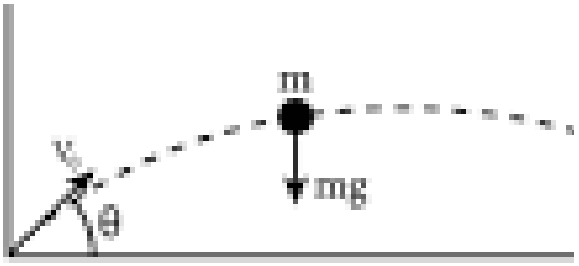
(2)

(ii) The ball hits the ground 27 m from the base of the tower. Calculate the speed at which the ball is thrown.

Answer ..... m s^{-1}

(2)
(Total 6 marks)





When an object is in the air, it experiences a number of forces. For simplicity, we divide the forces into the components – vertical and horizontal – using trigonometric functions.

Firstly, the earth exerts a force on the object known as gravity, which pulls the object down towards the ground. As discussed in the particle fundamentals chapter, this produces an acceleration downwards, which influences the vertical velocity component of the object. Every second, the object accelerates 9.8m/s^2 downwards. As it uses constant acceleration, we can apply the law of kinematics, crudely known as SUVAT, for displacement, initial velocity, final velocity, acceleration, and time. Given three of these parameters, you can use these equations to find the other two.

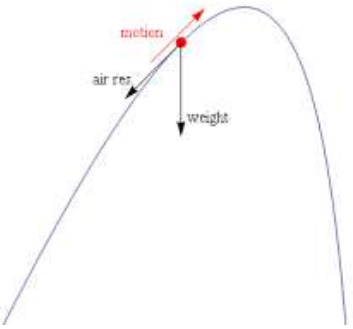
$$v = u + at$$

$$s = ut + \frac{1}{2}at^2$$

$$v^2 = u^2 + 2as$$

With regards to the horizontal components, the forces tend to be in equilibrium. This means that the horizontal velocity remains the same and there is no overall acceleration. This means we can apply the most basic formulae, such as s = v/t.

Applying the above concepts means that all projectile motion can be modelled with a quadratic. Generally, air resistance or drag is not considered in questions to maintain simplicity. However, this is an important factor in real life which opposes the motion of the particle, and therefore alters the quadratic shape. To maintain realism, I intend to provide the user with the ability to enable air resistance. While this limits the educational aspect, it provides extra context to how the projectile would function in the real world. Calculating air resistance accurately is a complex task as there are a number of factors at play such as laminar and turbulent flow. However, I have found a suitably accurate formula on the right. Air resistance is calculated to be half of the velocity squared, multiplied by drag coefficient, density and surface area. Note that all bar the velocity can be reduced into a single constant. As such, in my simulation, I will use F = kv^2, where k is a constant which I will tinker with until I reach the desired effect.



$$\vec{F}_r = \frac{1}{2}D\rho A\vec{v}^2$$

Programming Choices

Programming Language

Python	
Pros	**Cons**
Python has an extensive standard library which would vastly accelerate development time.	The user would typically have to run the file as a .py as it does not export the file as an executable by default. This means that it would not run as fast as other programming languages might.
Python is the language I am most familiar with, meaning I have already learnt most of the fundamental concepts needed to create a physics simulation.	Due to being a high-level language, python has many more memory inefficiencies compared to other languages

C++	
Pros	**Cons**
C++ is a low-level language, meaning an instruction is C++ more closely aligns with machine code instructions. This means that I could have a greater degree of control over the hardware, so that I might be able to save on memory and efficiency.	C++ is a much harder language to develop programs in. For example, it's harder to debug, you must manually manage the memory, and there is a greater risk to bugs due to more direct access to the computer hardware
C++ also has a strong standard library. However, I still prefer the more assortment provided with python	A function in C++ is more complicated to both understand and develop in comparison to other languages. This means that C++ typically needs more lines of code to achieve the same thing that python would be able to do with less code

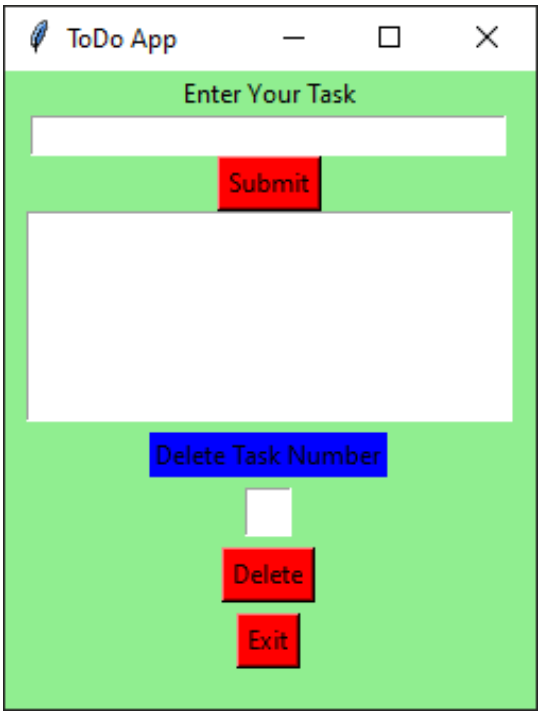
In conclusion, I will use python to develop the program. The key reason is that I am much more familiar with python. Moreover, the functionality provided in the python standard library will prove very helpful. For example, there are a number of key modules which C++ does not have a strong enough alternative for.

Graphical Modules

With python, there are a number of graphical modules which each contain lots of functionality. As such, I had a difficult choice in deciding which to use in my program. There are three modules which I am considering: Tkinter, Pygame, PyQt6

Tkinter	**Pygame**	**PyQt6**
Tkinter uses an event loop that continuously listens for events. These events might be bound to specific things, for example, clicking a button.	Pygame also uses an event loop. However, you check for events in the event loop and handle them as they come.	The event loop is integrated into the base files so you don't have to check for events.
The styling options in Tkinter are largely lacklustre. It has no support for CSS and the built-in options are disappointing	Similar to Tkinter, there is no support for CSS. The styling options are more manual.	PyQt6 has many different styling options thanks to its ability to handle CSS, as well as a number of built-in methods. Plus, it has a wide array of widgets such as sliders and
Tkinter comes with some basic GUI things, such as buttons and text fields. This would make it ideal for a lightweight UI	Pygame is more oriented towards game development, due to its built-in functionality. In particular, pygame has a clock which is incremented every time step. This would be ideal for simulations	PyQt6 has an enormous range of features and comes with logical mechanisms such as with signals or layouts.

In conclusion, pygame is a must-have thanks to its compatibility with simulations, which revolve around drawing particles and pathing their movement with respect to time. However, it is clear that while delivering in the graphical department, pygame falls short in providing a coherent UI that is appealing and easily understandable to the user. As such, I believe a hybrid approach would be best. I have included an example of the TKinter GUI on the right. I am unimpressed by the styling and functionality. Although PyQt6 will be the more complex to learn, I believe the positives in functionality (layers, signals, css, etc.) and aesthetics is worth it. Therefore, I will use PyQt6 for the main user interface to navigate to the simulations, which will use the pygame module.



Python Library

There are a few modules from the built-in library that I will use. These will greatly speed up the development time and improve the efficiency of the program immensely.

Module	Reasoning
numpy	By far the most important module to this program is Numpy. The main reason why I believe this module is so important is that it supports vectorised operations. There are a few areas of the program where dense, mathematical computation is necessary. These vectorised operation will result in faster computations and reduced execution time, leading to a better user experience. Furthermore, Numpy comes with lots of the more run-of-the-mill functions and values. For example, np.inf gives infinity, and np.pi gives pi, the constant.
Mysql .connector	This will be used to send SQL queries to the database server and fetch results if applicable

re	Short for regular expressions. This will enable me to quickly perform more complex pattern matching. This will be used to ensure that any inputted emails or the likes are formatted correctly and valid.
	`elif not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$", self.email.text()):`
	`  detailed_text += "Email is not valid.\n"`
	`  valid = False`

random	Uniformity is boring and the simulations will quickly go stale if done in this way. In order to make variation within the simulation, I will be using this module to pseudo randomly generate numbers, for example with the initial positions of particles.
hashlib	This is a module that's equipped with hashing algorithms. I will use its sha256 function with my login system to store a hashed version of the user's password for security purposes.

sys	This will purely be used when exiting the program, or more specifically when exiting the PyQt6 application.
	`app = QApplication(sys.argv)`
	`window = MainWindow()`
	`window.show()`
	`sys.exit(app.exec())`

Objectives

I have created a set of objectives which I will aim to complete by the end of the project.

Main Window

1) Provide a user interface for the user

1.1) The UI should be simple and straightforward being careful to minimise overloading the user

1.2) The UI should be full screen to prevent the student becoming distracted by other programs

1.3) The program display correctly for all resolutions

1.4) The user should be able to create an account with the program.

1.5) The user should be able to log in to a previously made account

1.6) All user inputs such as email address should be fully validated

` 1.7) Any parameters the user might select through using the simulations should be stored for future use, and the program should pre-load these parameters the next time they log in to the program

1.8) A teacher should be able to have extra access rights, such as view a student's progress within their class or edit levels

1.9) The user interface should provide engaging and helpful responses, for example if the user fails an email login.

Fluid Flow Simulation

2) Create an SPH fluid flow accurate to the Navier-Stokes equations

2.1) Create a particle that will react accordingly upon receiving a force; for example a larger mass should have a slower acceleration as per Newton's 2nd law

2.2) Handle boundary collisions appropriately

2.3) Have a spatial map that will accurately track the positions of particles

2.4) Be able to efficiently calculate density and pressure values for all particles

2.5) Be able to calculate pressure forces and viscosity forces

2.6) Correctly impose pressure and viscosity forces onto the particles

2.7) Be able to toggle between adding a gravity force

2.8) Allow the user to change the parameters of the simulation

2.9) Allow the user to apply a repulsive force at a chosen point

2.10) The program should be able to run at at least 30 frames per second.

Vector Field Pathfinder

3) Create a vector field pathfinder simulation

3.1) Particles should exhibit fluid-like properties

3.2) The program should generate a distance field.

3.3) The program should generate an accurate velocity field.

3.4) The user should be able to dynamically change cells into obstacles which block particles.

3.5) The user should be able to add new particles with different properties such as mass and density which react accordingly

3.6) The program should have a strong and intuitive UI.

3.7) The simulation should function as expected regardless of the user's settings.

3.8) The particles should have some natural variation to prevent a blank uniformity

3.9) The program should run efficiently with a reasonable set of parameters

Projectile Motion Simulation

4) Create a program which simulates the motion of projectiles in 2D.

4.1) Create balls which follow the laws of motion accurately.

4.2) The user should be able to project a ball with a inputted velocity

4.3) The simulation should have walls or obstacles

4.4) The simulation should provide live data for the currently in motion projectile.

4.5) The user should have the option for extra features such as air resistance

4.6) The user should have a reward system

4.8) The simulation should be easy to use

Ideal Gas Law

5) Create a simulation presenting the ideal gas law

5.1) The particles should follow the same rules as outlined in the ideal gas law.

5.2) The simulation should have an organised and validated interface

5.3) A pressure value should be given to the user in a sensible format

5.4) The user should be able to change the volume of the box

5.5) The user should be able to change the temperature of the box

5.6) The user should be able to add more particles to the box

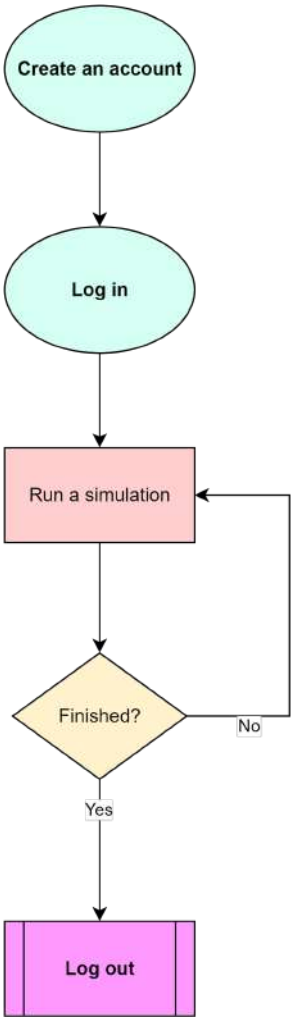
5.7) The properties of the box should be shown to the user in an easy to read format

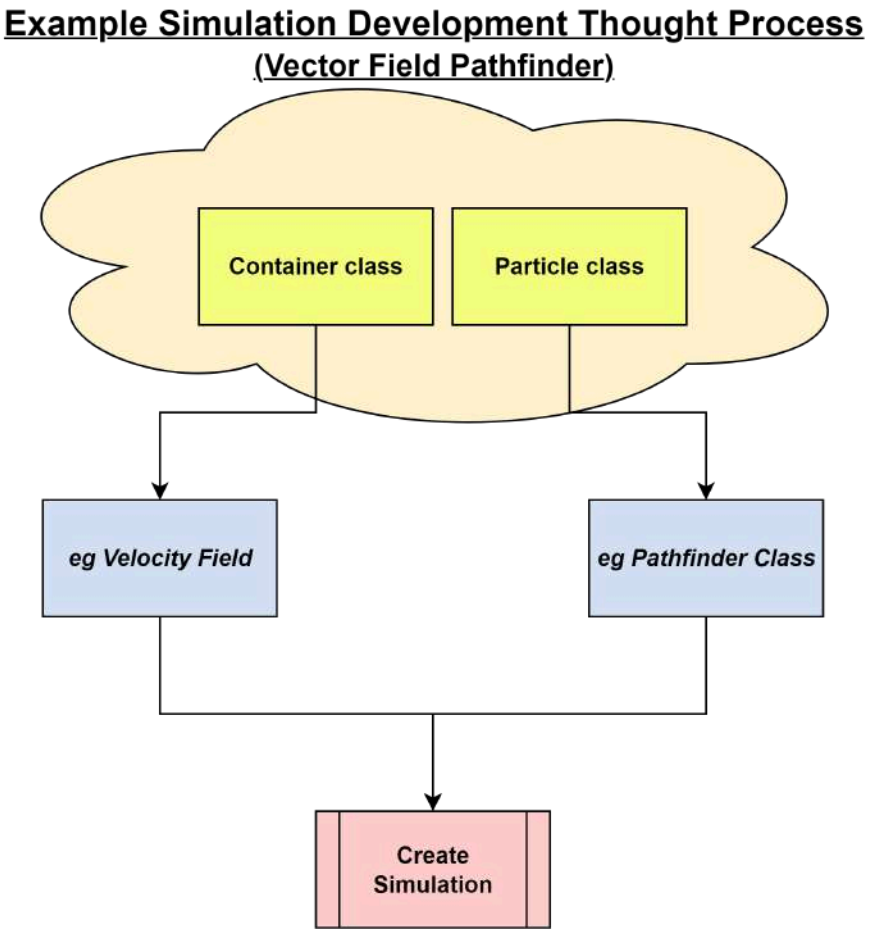
Design

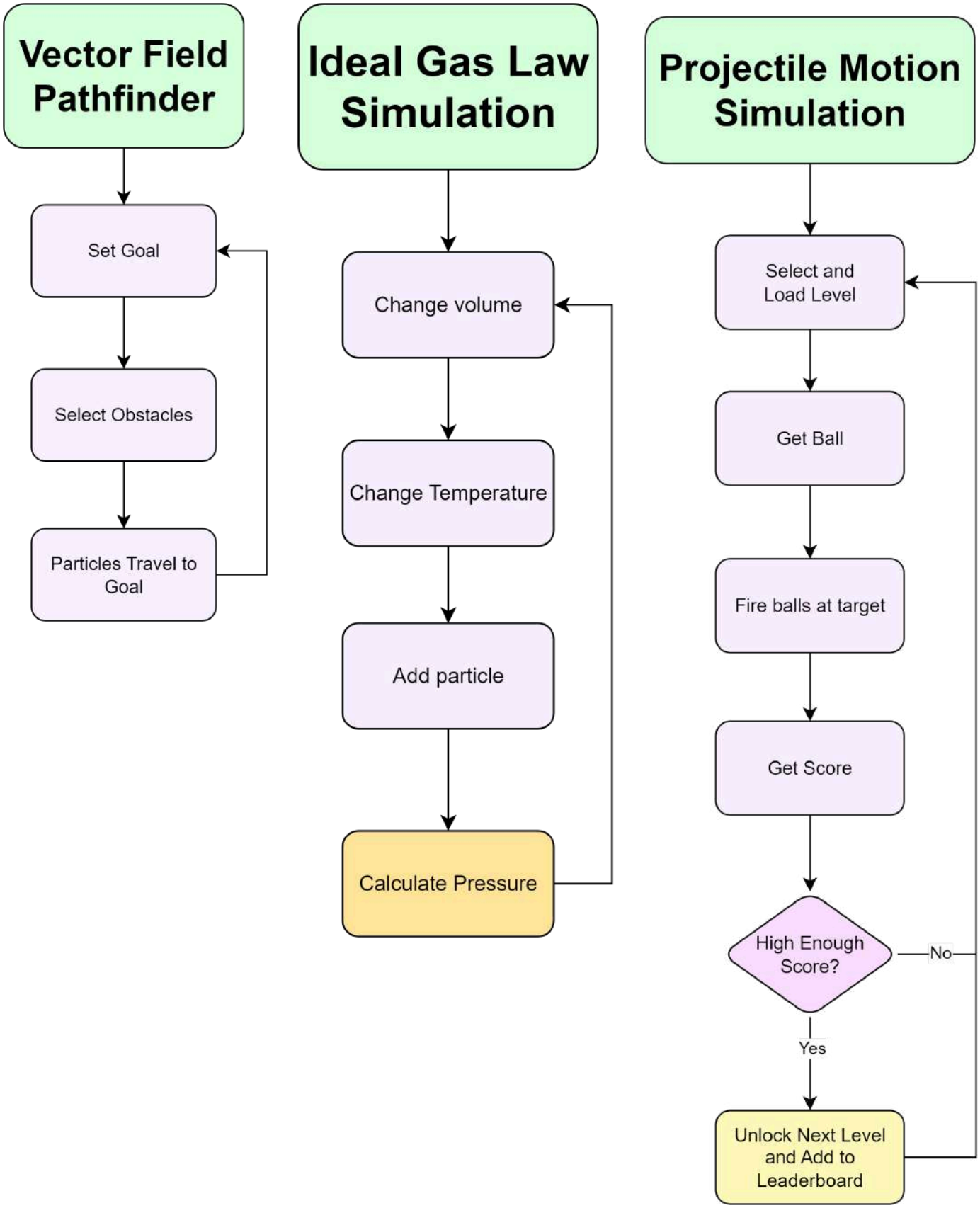
High-Level Overview

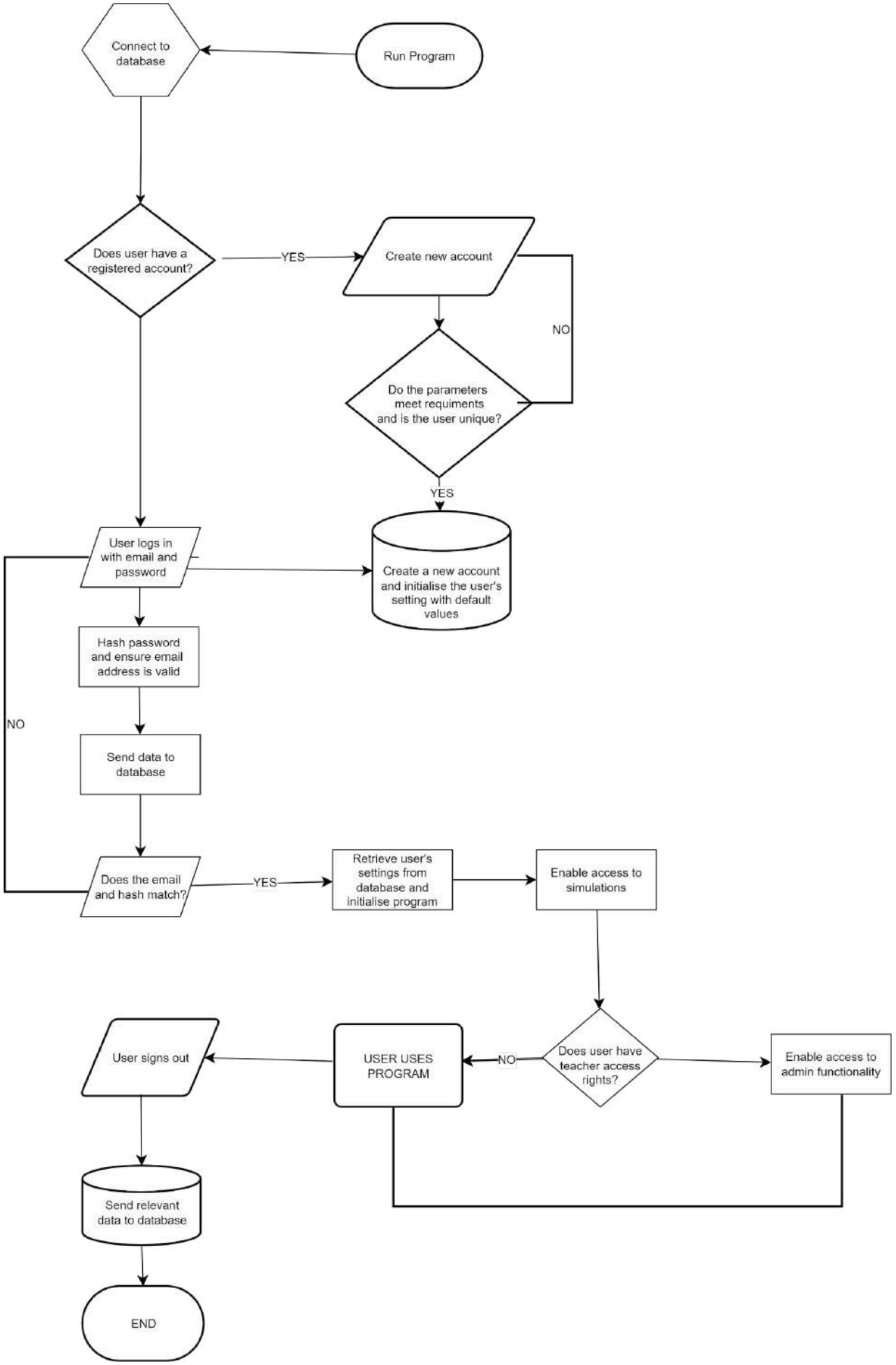
The main selling point of the program is the simulations. However, I will create a login system which will contribute to the utility of the program. For example, the user will be able to have the inputs saved for the next usage, or be able to see a leaderboard comparing them to the rest of the class.

I will rely heavily on modular programming to improve functionality and help further development. The simulations fundamentally rely on a container and a set of particles. As such, it makes sense to create base classes from which I can branch out and create individualised simulations.






Vector Field Pathfinder
Set Goal
Select Obstacles
Particles Travel to Goal
Ideal Gas Law Simulation
Change volume
Change Temperature
Add particle
Calculate Pressure
Projectile Motion Simulation
Select and Load Level
Get Ball
Fire balls at target
Get Score
High Enough Score?
No
Yes
Unlock Next Level and Add to Leaderboard


Connect to database
Run Program
Does user have a registered account?
YES
Create new account
NO
Do the parameters meet requiments and is the user unique?
YES
User logs in with email and password
Create a new account and initialise the user's setting with default values
Hash password and ensure email address is valid
NO
Send data to database
Does the email and hash match?
YES
Retrieve user's settings from database and initialise program
Enable access to simulations
User signs out
USER USES PROGRAM
NO
Does user have teacher access rights?
Enable access to admin functionality
Send relevant data to database
END

Data Dictionary

Please note that there may be some inaccuracies as the program may have changed since this was finalised.

baseClasses.py		
Variable	**Data Type**	**Intended Use**
class Particle		
vector_field	Object	The object the particle is bound to. Used to access its methods and values
damping	Float	Upon registering a collision, a particle will have its velocity multiplied by this value as a form of damping
mass	Float	The mass of the particle. Typically used when dealing with Newton's second law calculations
radius	Integer	The particle will be displayed as a circle of this radius
velocity	Numpy array; data type: float	Stores the current velocity the particle is travelling at.
position	Numpy array; data type: float	Stores the current position of the particle relative to the top left of the display
next_position	Numpy array; data type: float	Stores the projected position of the particle in the next time step. Can be used to ensure boundary conditions are met
class Cell		
cellList	set	Stores the particle objects currently within the cell's boundaries. It can be used to find neighbouring particles, as it can return all the particles within its borders
velocity	Numpy array; datatype: float	Stores the current velocity vector of the cell.
isBlocked	Boolean	Stores the blocked state of the cell

class SpatialMap

Variable	Data Type	Intended Use
noOfRows	Integer	Stores number of rows in the grid
noOfCols	Integer	Stores number of columns in the grid
draw_grid	Boolean	Used to check if grid information (typically grid lines) should be displayed
grid	Numpy array; data type: Object Cell	Stores a 1D array of each cell in the grid. The grid corresponds to the boxes shown on the display

pathfinderSimulation.py

Class VelocityField

blocked_cells	Set	Used to store the coordinates of any cells that are meant to be obstacles. For use when executing functions related to obstacles
goal	Numpy array	Used to store the coordinates of the currently selected goal
Cell_width (deprecated)	Float	Contains the width of a blocked cell. Need for the now deprecated collision avoidance steering behaviour
is_adding_particles	Boolean	Used to check if the user's right click should add a particle
is_adding_cells	Boolean	Used to check if the user's right click will toggle a blocked cell
enable_collisions_between_particle	Boolean	Used to enable collision behaviours between particles
draw_heatmap	Boolean	Used to check if the distance field and the according colour heatmap should be displayed
particle_to_add_radiues	integer	Particles which the user adds in future will have a radius of this size

cell_distances	Numpy array	An array with the same dimension as self.grid. It will be used to hold the distances of each corresponding cell when generating the distance field
particle_max_velocity	int	A maximum velocity which the steering calculation will use to dictate the strength of the steering force
Class Pathfinder		
damping	Float	Dictates the factor the velocity reduces by due to damping forces

projectileMotionSimulation.py

Class ProjectileParticle

Variable	Data Type	Intended Use
acceleration	Numpy array, data type:float	Holds the acceleration values of the particle
damping	float	Dictates the factor the velocity reduces by due to damping forces
colour	tuple	The colour of the ball when displayed on the screen
hit_goal	boolean	States if the ball has come in contact with the target
Class Obstacle		
position	Numpy array	The top left point of the particle according to pygame coordinates
width	int	The width of the obstacle
height	int	The height of the obstacle
goal	boolean	States if the obstacle is the goal. If it is a goal, the obstacle will be handled as a target for the projectiles. For example, it will contribute to the user score and particles will ‘stick’ to it.

Class Container		
particles	list	A list containing all currently registered particle objects.
selected_particle	int	The index of the particle which the user is currently interacting with according to the particle list
projected_particle_velocity_multiplier	int	The multiplier dictating how many times more the power of the pull back on the particle's velocity when firing a projectile
draw_line_to_mouse	boolean	Dictates if a visual indicator should be shown when the user is firing a particle
colliding_balls_pairs	List; data type: tuple	Holds a tuple of balls which in that current time frame have collided with each other
drag_coefficient	float	The drag coefficient, which is used in calculating the air resistance force
g	float	The real life value for the acceleration due to gravity
penetration_factor	float	The factor the velocity reduces upon hitting the target
px_to_metres_factor	float	The conversion ratio from px to "metres"
toggle_velocty_display	boolean	Dictates whether the projected velocity is in vector form or as a speed
show_coordinates	boolean	Dictates if the cartesian coordinates of the pointer should be displayed
score	int	Holds the user score, which is added to when the user hits the target with a projectile
obstacles	list	Holds all obstacle objects

idealGasLawSimulation.py		
Class GasParticle		
colour	tuple	When displaying the ball to the screen without an image, the ball will have this colour

Class Container		
particles	list	A data structure holding the particle objects
selected_particle	int	The index according to self.particles of the particle currently being interacted with by the user
projected_velocity_multiplier	int	The multiplier by which increases the strength of the projection when firing particles
dimensions	Numpy array	The dimensions of the container holding the particles
font	SysFont object	A pygame object which initialises a font
collision_count	Int	Counts the number of collisions between particles
collision_spark	boolean	Dictates whether the program should draw a line or ‘spark’ between colliding particles
colliding_balls_pairs	list	A data structure holding tuples containing two particle objects which have undergone a collision event
wall_selected	int	An index which corresponds to a side of the container walls.
wall_radius	int	Sets the thickness of the container walls
pressure_display	Widget object	Widget for the displaying the pressure of the container
temp_slider	Widget object	Used by the user to adjust the temperature value
particle_button	Widget object	Used by the user to toggle between adding heavy and light particles
collision_counter	Widget object	A widget displaying the number of collisions in the container
reset_button	Widget object	A widget which, when clicked, re-initialises the container. More specifically, the temperature, root mean square velocity, and number of particles will all be reset to its initial values. However, the container dimensions will remain as set by the user.
widgets	list	A data structure containing all widget objects. Used when updating and drawing them onto the screen.

temperature	int	Stores the temperature of the container in kelvin. Can be adjusted by the user.
initial_temperature	int	Holds the initial temperature of the container in kelvin
Class Widget		
position	Numpy array	Contains the origin of the widget
size	Numpy array	Contains the width and height of the widget
colour	Numpy array	Dictates the base colour of the widget.
default_colour	Numpy array	Stores the original colour value in case the program needs to revert to its original state
slider	boolean	Boolean checking to see if the object is a slider. If False, the object will typically be a button.
parent	Container object	If necessary, this will hold the parent object in case the widget needs to access its attributes
text	string	Stores the text that is to be displayed
alt_text	string	After a specific user event, the alt_text can be used to switch the text output of the widget
font	SysFont object	A pygame object which initialises a font
hover	boolean	Should the widget's appearance change if the user hovers their cursor over it?
dynamic	boolean	Dictates if the position of the widget is actively changing during execution

Fluid Flow Simulation

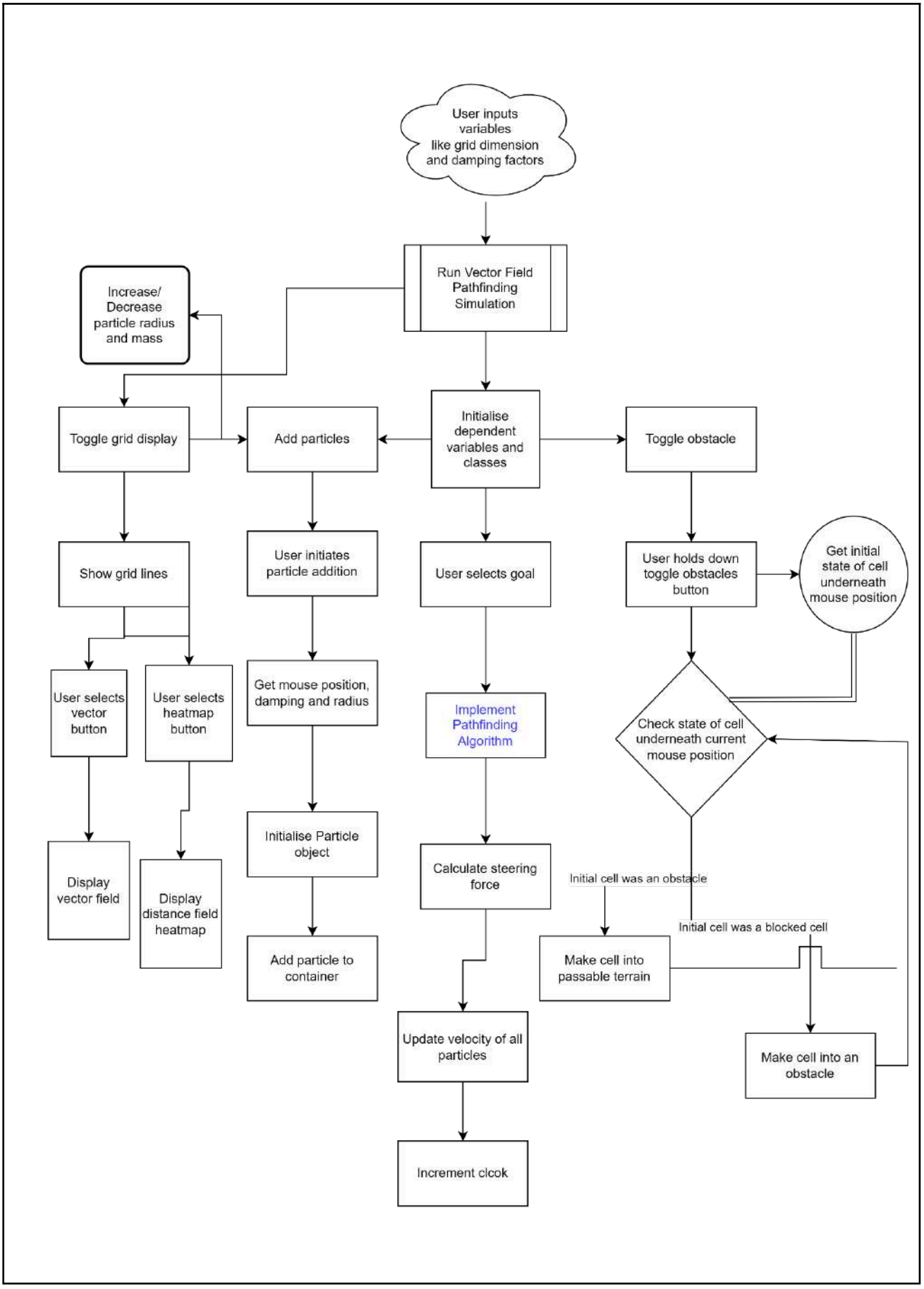
Class FluidParticle		
damping	int	Dictates the factor of energy the particle will lose upon collision with the boundary
radius	int	Determines the radius of the particle when it is outputted onto the screen

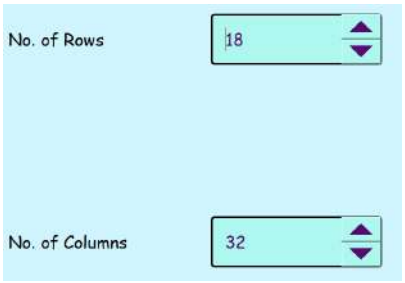
mass	float	The mass of the particle. It is necessary when performing most of the calculations
spatial_map	FluidSpatialMap object	Holds the spatial map which the particle is bound to. Can be used to access its attributes
force	Numpy array	Holds the sum of the forces which are acting on the particle in the current time step. Each step, these forces will be applied to the particle
velocity	Numpy array	The current velocity of the particle
position	Numpy array	The current position of the particle according to the pygame coordinate system.
pressure_force	Numpy array	Stores the force on the particle from the pressure
density	float	Stores the density of the particle.
pressure	float	Stores the pressure value of the particle. Calculate from its density and the rest density
Class FluidSpatialMap		
rest_density	int	States the rest density of the fluid. It is used when calculating the pressure via the ideal gas laws
smoothing_radius	int	Dictates the smoothing radius which is the maximum distance the algorithm will consider for pressure, density etc contributions.
kernel	SmoothingKernel object	A more general smoothing kernel which uses cubic spline
pressure_kernel	SmoothingKernel object	A smoothing kernel which uses Debrun's spiky kernel to calculate the pressure contributions from nearby particles
Class SmoothingKernel		
cubic_spline	boolean	States if the cubic_spline kernel is being used
poly_6	boolean	States if the poly_6 kernel is being used
gaussian	boolean	States if the Gaussian kernel is being used. This kernel looks at the contribution from all particles regardless of how close they are
spiky	boolean	States if Debrun's spiky kernel is being used

test	boolean	A test kernel with an easy to understand formula
normalisation_constant	float	A constant unique to each kernel which is used to normalise the final output

Input Process Output Tables

INPUT	PROCESS	STORAGE	OUTPUT
Vector Field Pathfinder			


User inputs variables like grid dimension and damping factors
Run Vector Field Pathfinding Simulation
Increase/ Decrease particle radius and mass
Toggle grid display
Add particles
Initialise dependent variables and classes
Toggle obstacle
Show grid lines
User initiates particle addition
User selects goal
User holds down toggle obstacles button
Get initial state of cell underneath mouse position
User selects vector button
User selects heatmap button
Get mouse position, damping and radius
Implement Pathfinding Algorithm
Check state of cell underneath current mouse position
Display vector field
Display distance field heatmap
Initialise Particle object
Calculate steering force
Initial cell was an obstacle
Initial cell was a blocked cell
Add particle to container
Make cell into passable terrain
Update velocity of all particles
Make cell into an obstacle
Increment clcok

The user requests to change the grid dimensions	The user will adjust the grid dimensions in the main PyQt6 window by adjusting the slider, prior to running the simulation. This is passed as a parameter into the simulation	The grid size will be stored in the main UI, and then in the database upon the closure of the window	The size of each cell in the grid will adjust to accommodate for this change in the initialisation of the simulation
			
The user selects a new goal.	The coordinates of the mouse position are hashed into the relevant coordinate for the new goal. The program regenerates the distance heatmap based on this new goal, and then calculates the new vectors for each cell	Each cell now has a new normalised vector which represents its velocity	The goal cell will be highlighted and the colour heatmap will change according to the new distance field.

```
    click_event = pygame.mouse.get_pressed()
    if any(click_event):
        pos_cell_index =
vector_field.index_to_coord(vector_field.hash_position(pygame.mouse.get_pos())
)
        if click_event[0]: # LEFT CLICK
            vector_field.update_velocity_field(pos_cell_index)
```

The user requests to change a free cell into a blocked cell, or an obstacle	The program finds the relevant coordinates and changes its properties to make it impassable and have a repulsion force to the particles	The cell's coordinates are added to the blocked cells list	The cell will change into a dark colour which clearly distincts it from a free cell

```
    def toggle_blocked_cell(self, coord):
        if self.is_adding_cells:
            if coord not in self.blocked_cells:
                self.blocked_cells.add(coord)
        else:
            if coord in self.blocked_cells:
                self.blocked_cells.remove(coord)
```

The user requests to add a new particle	The user right clicks on a point and its coordinates are found. A new particle object is created with the relevant properties	This new particle is added to the particles list in the container class.	A new particle will be added to the display, and will follow the correct movement behaviours according to the vector field.

```
    elif click_event[2]: # RIGHT CLICK
        if not vector_field.is_adding_particles:
            vector_field.toggle_blocked_cell(pos_cell_index)
        else:
            vector_field.add_particle(pygame.mouse.get_pos())
```

The user requests to adjust the size of future particles	The user presses the relevant button to either increase or decrease the radius of the particle	The particle radius variable will be adjusted and future objects will be ad	When the user requests to add a particle object, it will display the new updated radius

```
    elif event.key == pygame.K_EQUALS: # plus symbol
        vector_field.particle_to_add_radius += 1
    elif event.key == pygame.K_MINUS: # minus symbol
        vector_field.particle_to_add_radius =
max(vector_field.particle_to_add_radius - 1, 3)
```

The user requests to toggle the colour gradient for the distance field	When set to True, each cell is given a colour value according to their normalised distance from the goal	The state of the variable dictating whether to display the heatmap is toggled	The heatmap will be shown to the user, or vice versa

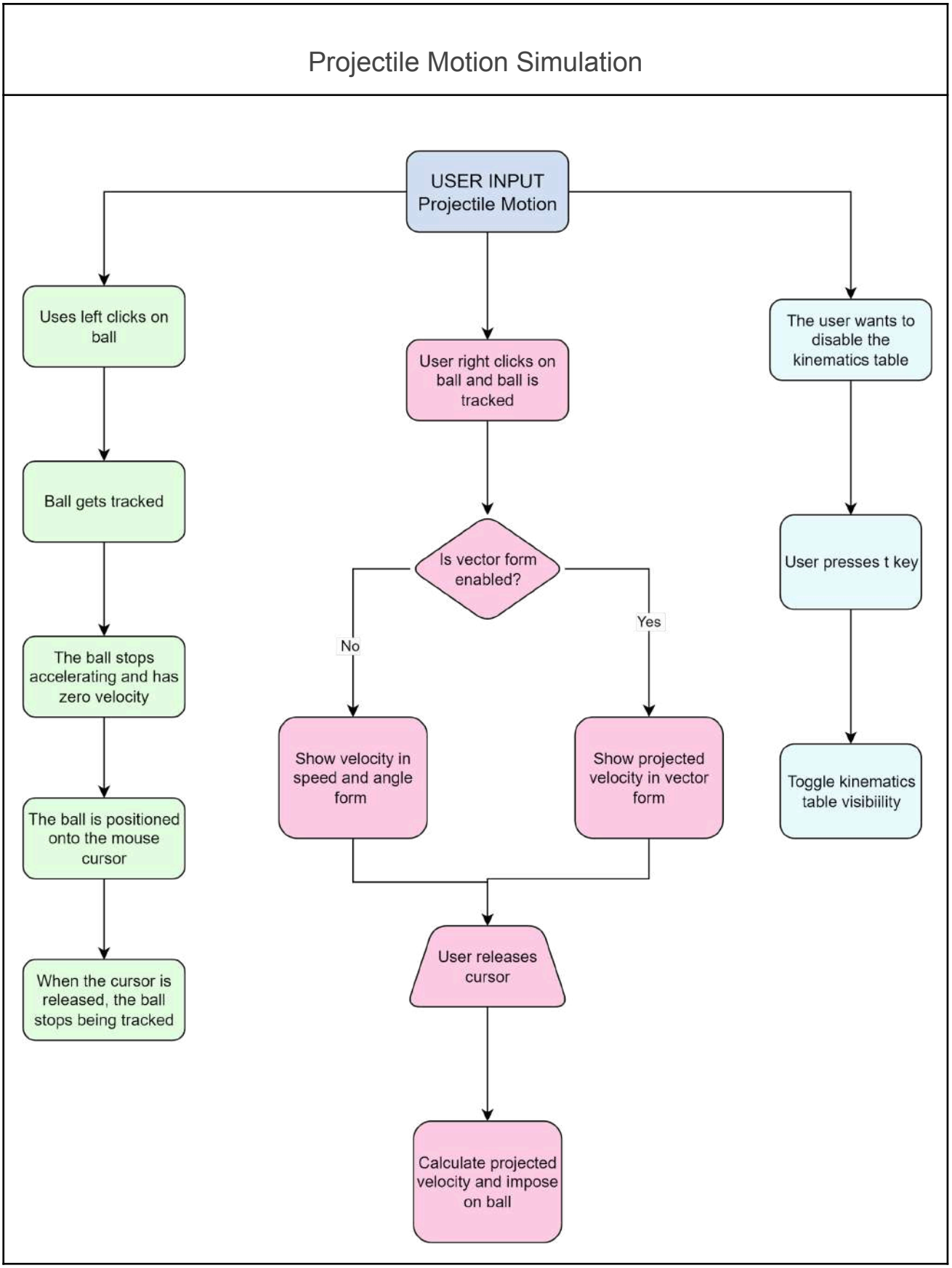
```
    def display_heatmap(self, screen): # drawing a gradient depending on the
distance
        max_distance = np.max(list(filter(lambda x: np.isfinite(x),
self.cell_distances)))

        for i in range(self.noOfRows):
            for j in range(self.noOfCols):
                distance = self.cell_distances[self.coord_to_index((i, j))]
                if distance > 0:
                    if np.isinf(distance):
                        pygame.draw.rect(screen, (255,160,255), (i *
box_width, j * box_height, box_width, box_height))
                    else:
                        norm_distance = distance / max_distance
                        colour = np.array([255 * (1 - norm_distance), 190, 255
* (norm_distance)], dtype=int)
                        pygame.draw.rect(screen, colour,
```

		(i * box_width, j * box_height, box_width, box_height))	
The user requests to view the properties of the individual cells	The program will display the grid lines. The program will also display the distance to the goal of each cell as well as the direction of its force.	The state of the variable dictating whether to display the grid information is toggled	The program will now overlay the background with distance and the direction of the force values, as well as indicating the borders of each cell

```
    for coord, cell, distance in zip(vector_field.get_grid_coords(), vector_field.grid, vector_field.cell_distances):
        # print(cell.velocity)
        boxCentre = np.array([coord[0] + box_width/2, coord[1] + box_height/2])
        lineRadius = (box_width/2.2) * cell.velocity
        if not any(np.isnan(cell.velocity)):
            pygame.draw.line(screen, "#ff3542", (boxCentre), boxCentre+lineRadius)
        if draw_distances and distance > 0:
            number = font.render(f"{distance:.1f}", True, (255, 255, 255))
            screen.blit(number, boxCentre - (box_width//4))
```

The user request to enable particle collisions	The user presses the relevant button the enable collisions. The program will cycle through the particle's neighbours and resolve the first collision it encounters, for each particle. This is to ensure speed, as well as providing a more particle-like and visually pleasing results	The state of the variable dictating whether collisions are enabled is toggled.	The program will now resolve static collisions to a light degree. As a result, particles will appear to endlessly vibrate around a fixed position when many particles are overlapping, and particles will no longer overlap in one-to-one collisions.

```
    elif event.key == pygame.K_c: # toggle collisions. turn off to reduce latency
        vector_field.enable_collision_between_particles = not vector_field.enable_collision_between_particles
```


Projectile Motion Simulation
USER INPUT Projectile Motion
Uses left clicks on ball
Ball gets tracked
The ball stops accelerating and has zero velocity
The ball is positioned onto the mouse cursor
When the cursor is released, the ball stops being tracked
User right clicks on ball and ball is tracked
Is vector form enabled?
No
Yes
Show velocity in speed and angle form
Show projected velocity in vector form
User releases cursor
Calculate projected velocity and impose on ball
The user wants to disable the kinematics table
User presses t key
Toggle kinematics table visibiility

The user drags a ball	The user selects a ball with the left mouse button. The position vector of the particle is changed to that of the cursor's. The velocity and acceleration of the particle is zeroed.	The particle's index according to the container's list of particles is stored to keep track of the currently selected particle.	The particle appears to follow the mouse cursor at all times.

```
    def drag_particle(self, mouse_pos):
        for index, particle in enumerate(self.particles):
            if not particle.radius < mouse_pos[0] < screen_width -
particle.radius and particle.radius < mouse_pos[1] < screen_height -
particle.radius:
                continue
            distance = particle.vector_field.get_magnitude(np.array(mouse_pos)
- particle.position)
            if distance < particle.radius:
                particle.velocity = particle.velocity * 0
                self.selected_particle = index
                return
```

The user releases a ball after dragging it.	The user releases the left mouse button. The particle's acceleration is changed to account for gravity. The velocity of particle can now change because the update() method is re-enabled.	The variable keeping track of the currently selected particle is reset to None.	The particle resumes standard functionality. For example, the particle will accelerate downwards each time step.

```
    def drop_particle(self):
        self.particles[self.selected_particle].velocity *= 0
        self.selected_particle = None
```

The user begins to project the particle	The user selects the ball with the right mouse button. The ball becomes stationary and stops accelerating	The index of the currently projected particle is stored.	The particle stops moving and stops accelerating
The user increases the power of the the projection when firing a particle	The user moves the mouse button to indicate the power and direction they would like to fire the particle. The projected velocity and angle is calculated according to the distance between the particle centre and mouse position. The velocity are then multiplied by a factor to translate from px to metres.	Values are shown above the selected particle.	Depending on the state of the velocity display, either the projected velocity in vector form or speed and angle from the i vector is shown above the currently selected particle. A line is drawn from the particle's centre to the mouse cursor.

```
    if vector_field.toggle_velocity_display:
        display_params = f"{int(projected_velocity[0])}i\u0302 +
{int(-projected_velocity[1])}j\u0302"
    else:
        display_params =
f"{vector_field.get_magnitude(projected_velocity).astype(int)} m/s | \u03B1 =
{int(np.arctan2(projected_velocity[1], projected_velocity[0]) * -180 /
np.pi)}\u00B0"
```

The user fires the particle	The projected velocity is again calculated using the distance between cursor and particle. This is then multiplied by the projected_particle_velocity_multiplier	The particle's velocity is replaced with the new calculated velocity.	The line showing the drag back is no longer shown. The particle travels in the direction and with the magnitude specified when dragging the particle back.

```
    def release_projected_particle(self, mouse_pos):
        particle = self.particles[self.selected_particle]
        particle.velocity = (particle.position - np.array(mouse_pos)) *
self.projected_particle_velocity_multiplier
        self.draw_line_to_mouse = False
        self.selected_particle = None
```

The ball hits the target	The user ideally fires a projectile at the target. Each time step, The velocity is reduced by a factor determined by the penetration_factor.	The hit_goal attribute changes to True to signify it is within the target. The acceleration is zeroed whilst within the target boundaries.	The ball appears to slow down when within the target boundaries

```
    def collision_event_goal(self, screen):
        goal = self.vector_field.goal
        if self.entirely_in_obstacle_check2(goal.position, goal.width):
            self.velocity = self.velocity * (1 -
self.vector_field.penetration_factor)
            self.acceleration *= 0
            self.hit_goal = True
            self.colour = (25,125,195)
            if np.allclose(self.velocity, np.zeros_like(self.velocity),
atol=2):
                self.vector_field.selected_particle = None
                try:
                    self.vector_field.calculate_points(self)
                    print(self.vector_field.score)
                    self.vector_field.particles.remove(self)
                    self.vector_field.splattered_particles.append(self)
```

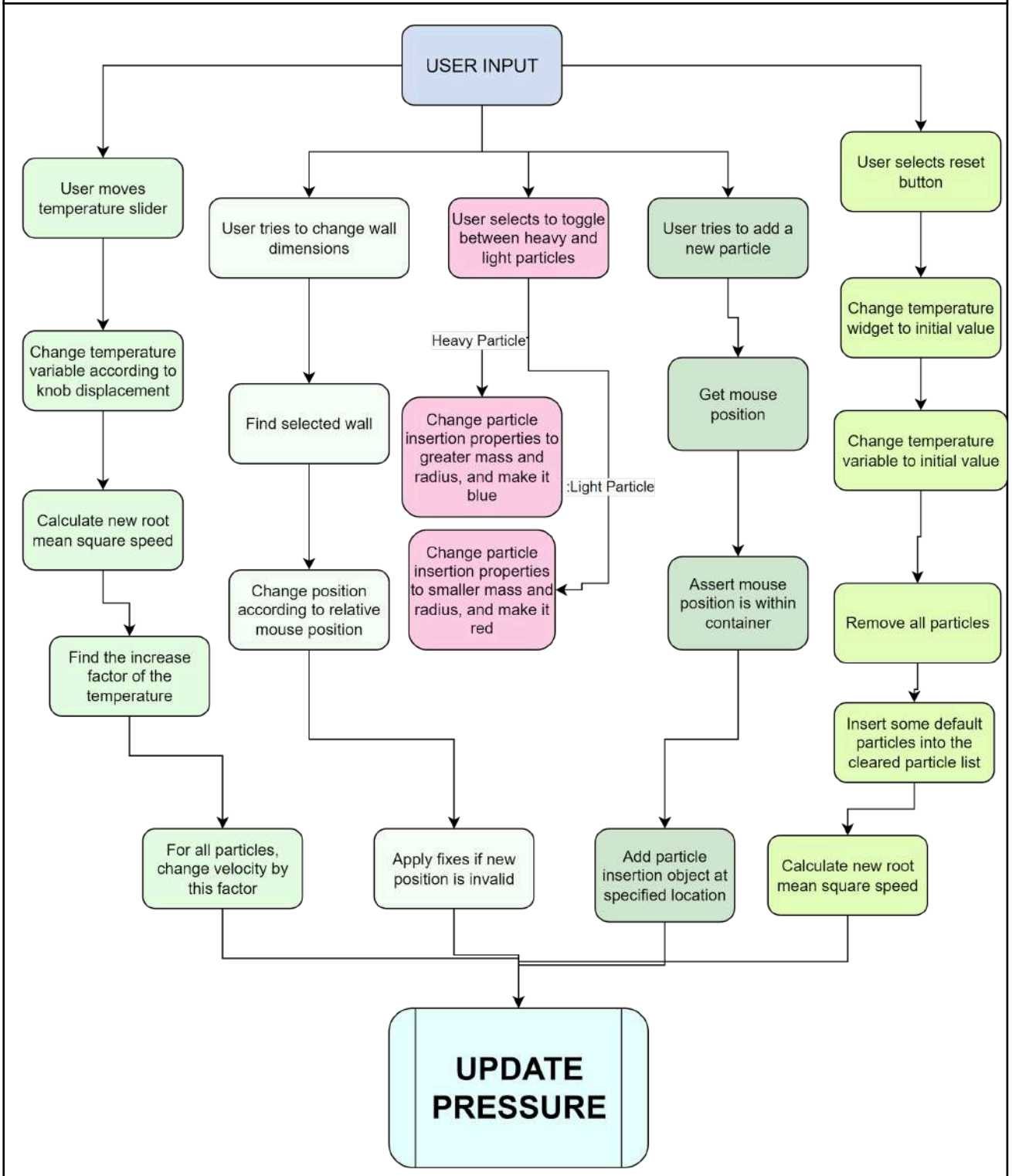
```
                except:
                    return
        else:
            self.hit_goal = False
            self.acceleration = np.array([0, self.vector_field.g])
```

The ball comes to rest within the target	When the velocity reaches zero and the particle is within the target, the particle has officially 'hit' the target. The particle is removed from the particles list to stop it updating every time step	When the ball comes to rest, points are added to the score depending on the distance. The particle is added to the splattered_particles list.	The ball is replaced with a splat image. The splat image is displayed on top of the target, where the particle landed. The score is updated to signify hitting the target

```
    def calculate_points(self, particle):
        multiplier = abs((self.get_magnitude(self.goal.position -
particle.position) / self.goal.width))
        points = int(100 * (1 - multiplier ** 2))
        self.add_points(points)
```

INPUT	PROCESS	STORAGE	OUTPUT

Ideal Gas Law Simulation



The user requests to increase the temperature of the container	The user drags the slider up and down, and the temperature will adjust according to the difference between the knob's rest position and the current position.	Moving the slider will change the temperature value of the container	If the temperature increases, we can look at the ideal gas law (PV =nRT) and observe that the pressure will increase. As temperature is the average kinetic energy of the particles, the average speed will increase accordingly.

```
    def update(self):
        if self.is_clicked:
            if self.slider:
                self.knob[0] = max(min(pygame.mouse.get_pos()[0],
self.knob_rest_pos[0] + 0.5 * self.size[0]), self.position[0])
                self.knob_value += 0.005 * (self.knob_rest_pos[0] -
self.knob[0])
                self.knob_value = max(1, self.knob_value)
                self.parent.temperature_change(self.knob_value)
```

The user requests to add a heavy particle	The user clicks on the particle button to toggle to heavy particles. The user clicks within the container walls. The program adds a particle of a relatively larger mass and radius and the particle has standard functionality	The new particle is added to the container's particle list. The particle has larger radius and mass attributes, and an initial zero velocity vector	A new heavy particle will be displayed at the current mouse position. It will have a higher inertia, so a light particle would not have as much effect upon a collision event.
The user requests to add a light particle	The user clicks on the particle button to toggle to heavy particles. The user clicks within the container walls. The program adds a particle of a relatively smaller mass and radius and the particle has standard functionality	The new particle is added to the container's particle list. The particle has smaller radius and mass attributes, and an initial zero velocity vector	A new light particle will be displayed at the current mouse position. It will have lower inertia, so when it collides with a heavier particle, the particle will appear to be zippier

```
def add_particle(self, mouse_position):  # Add heavy or light particle at cursor
    if self.particle_button.is_clicked:
        obj = GasParticle(0.06, 5, self, position=np.array(mouse_position,
dtype=float))
        obj.colour = np.array([255, 60, 60])
    else:
        obj = GasParticle(0.1, 8, self, position=np.array(mouse_position,
dtype=float))

    self.particles.append(obj)
    self.pressure_display.text = f"{self.calculate_pressure() * 1000:<.2f} mPa"
```

The user	The user right clicks on a given	If the new dimensions	The selected container

requests to increase the size of the container	wall. The program calculates the selected wall, and the wall moves along with the mouse position. The pressure value is recalculated while a wall is selected to ensure that the pressure adjusts accordingly	meet the standard checks, then they are imposed onto the container object's self.dimensions.	wall is changed according to the mouse position. For example, the user selects the left wall. The wall then aligns with the relative change in the horizontal component of the mouse position. Particles will now be bound within the boundaries of the new walls.

```
    def change_wall_dimensions(self, change):
        index = self.wall_selected
        dim = self.dimensions.copy()
        if index % 2:
            dim[index] += change[1]
        else:
            dim[index] += change[0]
        if not dim[2] + self.wall_radius < screen_width * 0.75:
            dim[2] = screen_width * 0.75 - self.wall_radius
        if 20 < dim[2] - dim[0] and dim[3] - dim[1] > 20:
            self.dimensions = dim
        self.pressure_display.text = f"{round(self.calculate_pressure())} Pa"
```

The user requests to reset the container.	The user selects the reset button. The program runs the initialise_container method of the container object. The dimensions of the container remain unchanged, as requested in peer testing.	The self.initial_temperature is copied into self.temperature. The particles list is cleared of all particles. New particles are then added to the system with random positions and with the same initial base velocity.	The container's walls are unchanged. The number of particles reset to the initial value. Their speeds and so the temperature also resets. The pressure value subsequently changes accordingly.

```
   def initialise_container(self):
        dim = self.dimensions
        base_v = 200
        self.temp_slider.knob_value = self.initial_temperature
        self.temperature = self.initial_temperature

        self.particles.clear()
        self.particles.extend([GasParticle(1, 8, self, 1.00,
position=np.array(
            [randint(dim[0], dim[2]), randint(dim[1], dim[3])]),
velocity=np.array(
```

```
            [randint(-base_v, base_v), randint(-base_v, base_v)],
dtype=float)) for _ in range(50)])  # eccentricity
        self.rms_velocity = self.calculate_rms_velocity()
        self.pressure_display.text = f"{round(self.calculate_pressure())} Pa"
```

The user starts the stopwatch.	The user selects the start button on the stopwatch. The program keeps track of the number of collisions while the stopwatch is running.	The number of collisions are stored as a container attribute. The time elapsed is stored	The current collision count over the time elapsed is displayed to the user, as is the current time
The user stops the stopwatch	The user selects the stop button on the stopwatch. The program pauses counting the number of collisions and pauses the stopwatch.	The number of collisions remains static. The internal timer of the program resets to 0 seconds.	The number of collisions during the time elapsed is displayed to the user.
Stopwatch no longer in the plans so please ignore.			

Key Algorithm Explanations

Universal Algorithms

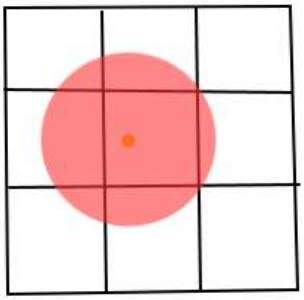
Fixed-Radius near-neighbour problem.

One if not the most computationally expensive part of my project is finding which particles lie within the radius of

another. This concept is essential to all of the simulations containing inter-particle forces. For the current investigation, I will make reference to fluid flow as this is where it is most computationally important, but this concept applies to other cases such as with collisions in the ideal gas law simulation. To get a value from every single particle within the container is incredibly inefficient and is often not viable. This dilemma is known as the fixed-radius near-neighbour problem. In my simulation, if a particle lies within a fixed radius (in the case of my fluid flow simulation on the right, the smoothing radius) of a given particle, then that particle will be included in various pressure and viscosity calculations and as such will impact the particles' motion; all those not within the smoothing radius will be ignored. There are multiple algorithms that attempt to offer an efficient solution but each have their own drawbacks. With my project, I aim to have a solution which will be fast for a relatively small number of particles within a given smoothing radius, as I believe that in comparison to other fluid flow simulations, the program will be unsuitable for thousands of particles due to the limitations in hardware and efficiencies of the programming language.

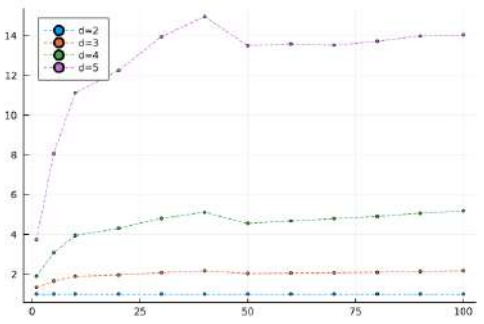
The first solution which I had initially implemented was the trivial approach. Here, we take the list containing every particle and loop through the entire list. As mentioned before, we calculate the distance between the main particle and the particle in question. If that distance is within the smoothing radius, then that particle is appended to the neighbouring_particles list, which will be eventually returned for use in density calculations. With a time complexity of O(n^2), this is highly inefficient and is by far the worst solution. When implemented, my computer failed to load the next frame within 20 seconds.

```
def brute_force(particle, particleList):
    neighbouring_particles = []
    for neighbour in particleList:
        distance = (particle.position - neighbour.position).magnitude()
        if distance < smoothing_radius:  # The smoothing radius is predefined
            neighbouring_particles.append(neighbour)

    return neighbouring_particles
```



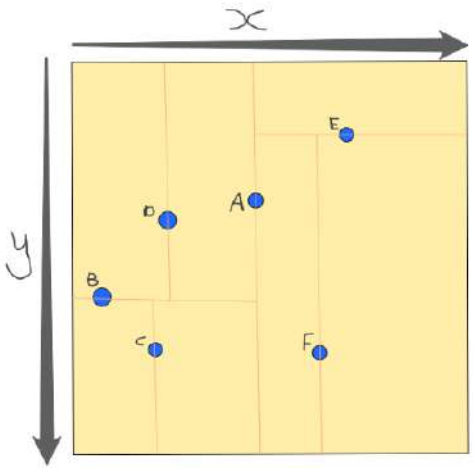
Another proposal is using cell lists for a spatial map. This is where the screen space is partitioned into a grid in which each cell has width and height *h*, where *h* is at least the smoothing radius. This value is chosen as it means that for any particle with its centre within a cell, all of its neighbours will lie within the 3x3 grid, as shown on the left. This restricts the region with which we have to perform the distance check, thereby increasing efficiency tenfold. One implementation is creating a cell list of length "number of rows x number of columns" for each cell in the grid. Then, we can use a simple formula to assign each particle to its associated cell. This has a time complexity of O(n) as the number of iterations is linear to the number of particles, but this is misleading as cell lists suffer greatly from an increase in grid dimensions, as the number of possible cells increases exponentially. ***This means that if the dimensions are large enough with a fixed smoothing radius, then this implementation could be less efficient than the naive approach.***

Also, it can be the case that some cells remain empty, which means an excess amount of unused memory. A more complex alternative to the aforementioned is to use a dense spatial hash table. We define two cell lists with a chosen length. We can then use a hashing function to count the number of particles in a cell and store that in a list. These act as pointers to the particle map. Then, I

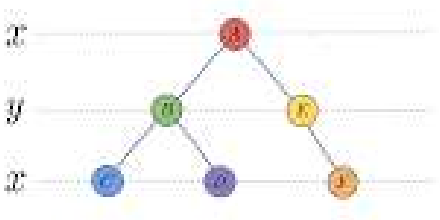


The ratio to $d = 2$ results as a function of number of points n for dimensions $d = 2, \ldots, 5$ and radius $r = 0.01$.

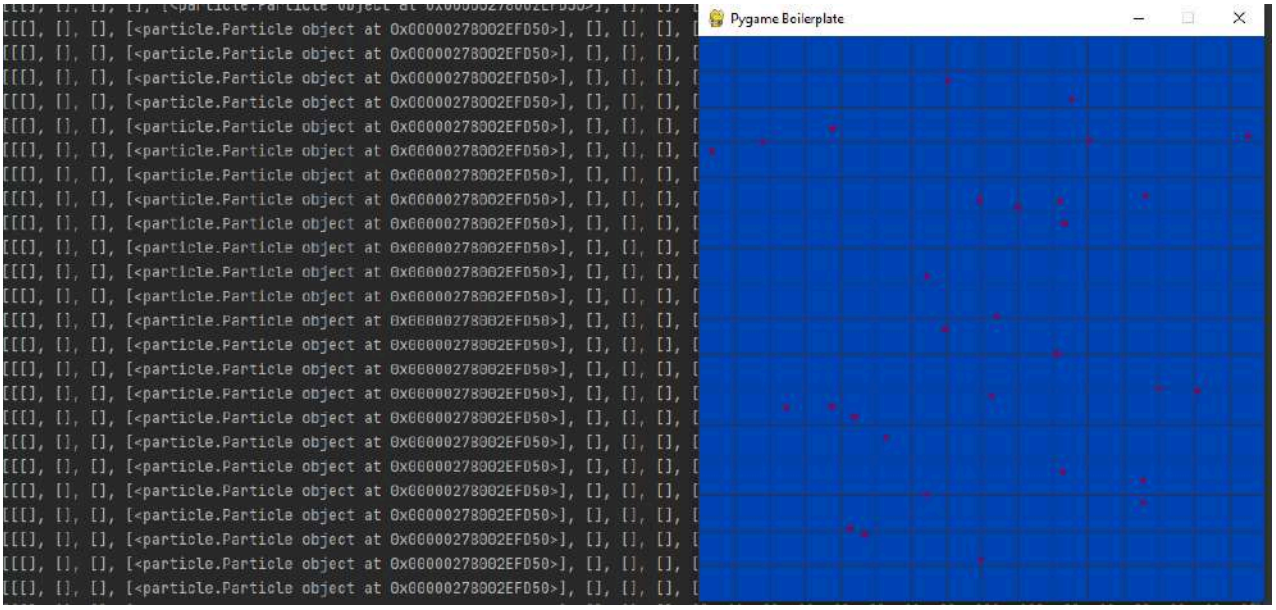
can set a fixed length for my spatial map (not necessarily related to the grid dimensions) and allocate particles to their respective cells based on the hash. This method is very useful for a grid with unspecified boundaries, which is not applicable to my project.
One issue that could arise with this is a hash collision. If this were to occur, it would mean a particle would be out of place. However, this does not render it useless, as the program would just have another particle which it will have to deal with (in this case by disregarding it upon checking if the distance between the two bodies are less than the smoothing radius. While this alternative method does minimise memory consumption, this issue was not of particular concern to me as speed has been the limiting factor thus far. My grid dimension and particle count is such that a large majority of cells are occupied anyway, meaning the downsides of using a spatial map with a list length based on the grid dimensions are insignificant.



The final method I will consider is a K-D tree. This is an efficient data structure that once again partitions the space into sections. However, here the partitions are unequal in area. The basic premise is that we keep splitting a box in alternating axes into two about a particle, thus creating a binary tree. First, we have to create the tree. Split the screen into two vertically about a preferably central particle(or the median). This particle will be the root node. Then, split each region, now horizontally, about a particle which will be the child of the particle from which the region was initially split from. Repeat these steps until each particle has been added to the tree, at which point the tree will be complete. This process has a time complexity of O(n log n). Now, we can calculate the neighbours, by navigating the tree and comparing distances between the point in question and the points in each leaf node. While unique and effective, I feel uneasy towards this method. Coding an implementation would be difficult and as such I would be compelled into using external modules. This would mean I have less control over the algorithm. My chosen solution may be applied to various simulations, and so I would like a greater degree of control over the algorithm.



In conclusion, I believe that using a cell list would be best for my project. Clearly, the naive approach is out of the question as with a time complexity of O(n^2), most machines would struggle to run the fluid flow. While K-d trees are an efficient data structure, I am concerned by the higher time complexity; the linear time complexity of the cell list would align well with my objectives as I would like to place an emphasis on the number of particles. While of course the spatial map's efficiency would degrade exponentially with an increase in dimension, I don't believe that my program will use a particularly large grid dimension as I intend to default to a relatively large radius.




```
def remove_particle(self, particle):
    cell = self.hash_position(particle.position)
    self.grid[cell].cellList.discard(particle)
    # np.delete(self.grid[int(cell[0]), int(cell[1])], particle)

def insert_particle(self, particle):
    # print(particle, particle.position, particle.next_position)
    new_cell = self.hash_position(particle.next_position)
    if new_cell == None:
        print("Error in insert_particle method of Particle") # logging purposes
        return
    elif not 0 <= new_cell < self.rows * self.cols: # particle is outside screen.
        # resolving particle position
        r = self.radius
        particle.next_position = np.clip(particle.next_position, (r, r), (screen_width-r,
screen_height-r))
        return self.insert_particle(particle) # return fresh instance

    self.grid[new_cell].cellList.add(particle)
```

Collisions

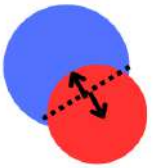
Implementing a form of collisions for particles is essential, whether that be to ensure balls interact with walls properly in the projectile motion simulation, maintaining a core principle of the ideal gas law with elastic collisions, or simply to create some variance in the vector field pathfinder.

To begin, you first have to recognise when there has actually been a collision. There were two ways that I could go about implementing this. The first and the more rudimentary option

was using the built-in pygame library. For example, given a circle and pygame rectangle object, you can use the provided colliderect method. However, this would restrict me to making all my obstacles and walls into pygame Rects. As such, I opted for a more mathematical approach. I have made separate functions for resolving collisions between particle on obstacle, and particle on particle.

```
def resolve_static_collision(self, next_particle):
    # finding magnitude of vector
    distance = self.vector_field.get_magnitude(next_particle.next_position -
self.next_position)
    overlap = 0.5 * (distance - (self.radius + next_particle.radius))

    # finding direction and applying final vector
    self.next_position -= overlap * (self.next_position - next_particle.next_position) /
distance
    next_particle.next_position += overlap * (self.next_position -
next_particle.next_position) / distance

    if np.isclose(self.velocity, np.zeros_like(self.velocity), atol=1).all():
        self.velocity = np.zeros_like(self.velocity)
```

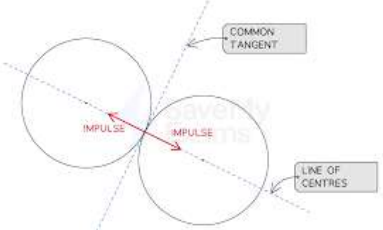
I found the overlap by getting half of the difference between the radii and the distance between particle centres. This gives the distance that each particle has to travel.



A vector has both a direction and magnitude; we have to find the directions in which they have to move. This was done by getting the direction vector between the particles' centres. Moving along the line of collision is the shortest distance the particles have to take to resolve, and so is the desired option.

Next, multiply the direction vector by the calculated magnitude to get the final vector. Apply this to each particle in opposing directions and the positions are successfully resolved

```
def resolve_dynamic_collision(self, next_particle):
    distance = self.vector_field.get_magnitude(next_particle.next_position -
self.next_position)
    normal = self.vector_field.normalise_vector(next_particle.next_position -
self.next_position)
    tangent = np.array([-normal[1], normal[0]])

    tangential_vel_i = tangent * np.dot(self.velocity, tangent)
    tangential_vel_j = tangent * np.dot(next_particle.velocity, tangent)

    normal_vel_i = normal * ((np.dot(self.velocity, normal) * (self.mass -
next_particle.mass) + 2 * next_particle.mass * np.dot(next_particle.velocity, normal)) /
(self.mass + next_particle.mass))
    normal_vel_j = normal * ((np.dot(next_particle.velocity, normal) * (next_particle.mass
- self.mass) + 2 * self.mass * np.dot(self.velocity, normal)) / (self.mass +
next_particle.mass))

    self.velocity = tangential_vel_i + normal_vel_i
    next_particle.velocity = tangential_vel_j + normal_vel_j
```

Who two particles collide, they bounce off each other to moving in opposite directions. I have been able to apply my knowledge of oblique collisions in further maths. After resolving the positions of the particles, I have to then resolve the velocities of both particles. This was covered in my maths lessons. Essentially, there are two components to consider: the velocities parallel (or tangential) to the line of centres, and perpendicular (or normal) to the line of centres.
The component of the velocities parallel to the line of contact remains the same. However, the other axis can lose energy, which is dictated by a coefficient of restitution. Earlier iterations of the collision system had a parameter for this, where I would pass the factor of which the normal velocity would lose energy, though I have since gotten rid of this.

I had many different attempts at implementing wall collisions. Past attempts failed due to having graphical inaccuracies, or being overly computationally complex. Ultimately, I decided upon a similar technique as resolving collisions between particles.

```
def collision_event_obstacles(self):  # Used for collision detection with Obstacle
Object
    for obstacle in self.container.obstacles:
        if self.check_obstacle_collision(obstacle.position, obstacle.width,
obstacle.height):  # if collision
            self.resolve_obstacle_collision(obstacle)  # Put particle into a valid
position
            return True
    return False
```

This draws many parallels to the particle collision implementation. The above handles the collision step for obstacles. It loops through each obstacle object in the obstacles list and checks for a collision with a particle and itself.

```
def check_obstacle_collision(self, obstacle_pos, width, height, custom_radius=None):

    closest_x = max(obstacle_pos[0], min(self.next_position[0], obstacle_pos[0] + width))
    closest_y = max(obstacle_pos[1], min(self.next_position[1], obstacle_pos[1] +
height))

    square_distance = (self.next_position[0] - closest_x) ** 2 + (self.next_position[1] -
closest_y) ** 2

    if not custom_radius:
        return square_distance < self.radius ** 2
    return square_distance < custom_radius ** 2
```

A key difference with the particle collision system is that the line of collision is always acting parallel to either the x or y axis, unlike earlier where the line of centres could occur in any direction. This simplified the velocity resolution significantly. One caveat here was determining which axis the collision happened, or in other words, whether to push the bull the left/right or the top/bottom edge of the wall. This was done by adding penetration values from which it could resolve velocities correctly.

This implementation was successful in the projectile motion simulation. However, I opted for a more crude approach in the vector field pathfinder in an attempt to find a better performing algorithm.

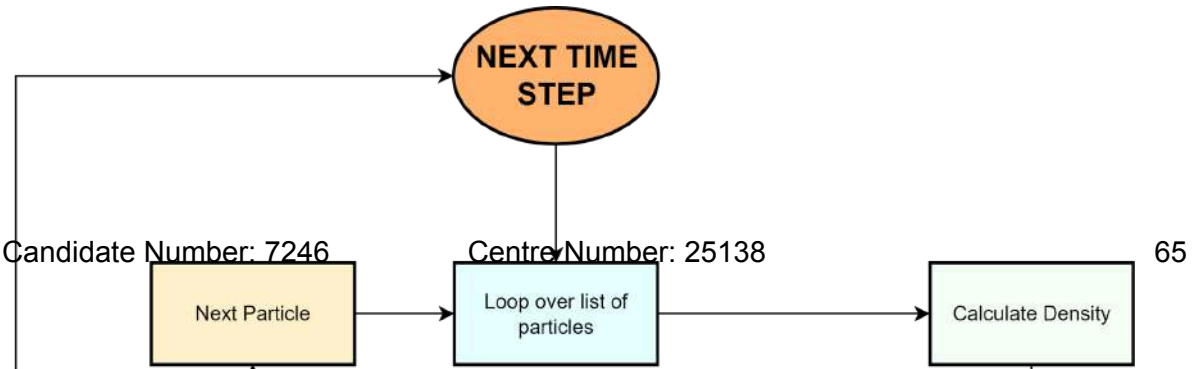
```
def resolve_obstacle_collision(self, obstacle):
    # calculating displacement vector from the rectangle to the circle
    displacement = self.position - np.array([max(obstacle.position[0],
min(self.position[0], obstacle.position[0] + obstacle.width)),
                                                    max(obstacle.position[1],
min(self.position[1], obstacle.position[1] + obstacle.height))])

    # needed to determine how ball got into the rectangle
    penetration_x = max(0, self.radius - abs(displacement[0]))
    penetration_y = max(0, self.radius - abs(displacement[1]))

    # direction of displacement
    direction_x = 1 if displacement[0] > 0 else -1
    direction_y = 1 if displacement[1] > 0 else -1

    # if platform, have greater damping
    if obstacle.is_platform:
        damping = 0.4
    else:
        damping = self.damping

    # determines if ball moves horizontally or vertically
    if penetration_x < penetration_y:
        # resolving position
        self.next_position[0] += penetration_x * direction_x
        # reversing velocity
        self.velocity[0] *= -1 * damping
        if obstacle.is_platform:
            self.velocity[1] *= damping
    else:
        self.next_position[1] += penetration_y * direction_y
        self.velocity[1] *= -1 * damping
        if obstacle.is_platform:
            self.velocity[0] *= damping

    if np.isclose(self.velocity[1], 0, atol=2):
        self.velocity[1] = 0
```

Fluid Flow

As discussed in the analysis, there are only three forces behind a fluid flow simulation.

1. Pressure forces
2. Viscosity forces
3. External forces (such as gravity)

By far the most challenging are the pressure and viscosity forces as they require complex logic and arithmetic



Density Calculation

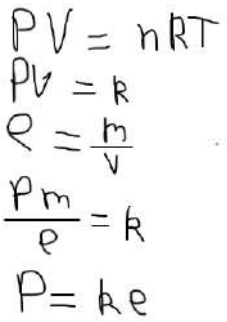
$$A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h),$$

The paper outlines a method to find property A of a given particle. This equation can be used eventually to find the pressure and viscosity forces. SPH relies on analysing the particles within the given particle's vicinity, called the smoothing radius, and taking into account the effect these particles have on the given particle. Firstly, I will try to digest the meaning of this equation. Firstly, A is the property we are trying to find of a given particle. The sigma function states that we have to go through each pass with the next j (just the next neighbouring particle which meets the criteria) and find the sum of these values. M_j is the mass of the neighbouring particle, rho_j is the density of the neighbouring particle, A_j is the property of the neighbouring particle which is the property of what we are trying to find initially. The function W calls the smoothing kernel which I will focus on in greater detail separately. Here, we are calculating the total effect the neighbouring particle will have. We pass the distance between the two particles and the smoothing radius, and the kernel will produce a value based on these values. When calculating a property using this equation, the density of the neighbouring particles are needed. I was confused therefore how to begin finding this density. Fortunately, when substituting A for density, the fraction means that the density over density reduces to one, eliminating this variable from the equation. I am now able to calculate any property needed in the future (in my case, the pressure and viscosity forces), given that I can calculate the density for all particles. Below is my python code for calculating the density. Note that here I am utilising the spatial map to vaguely retrieve the neighbouring particles, and then asserting that they are actually within the smoothing radius.

```
def calculate_density(self):
    density = 0
    neighbouring_particles =
self.spatial_map.get_neighbouring_particles(self)
    for neighbour_particle in neighbouring_particles:
        if neighbour_particle != self:  # don't include self in density
            distance =
self.spatial_map.get_magnitude(neighbour_particle.position - self.position)
            influence =
self.spatial_map.kernel.calculate_density_contribution(distance)
            density += influence * neighbour_particle.mass  # scale by
particle's mass
    self.density = density
```

Pressure Calculation

The first force in the equation is the pressure force. This is where the greater depth water will want to flow to the lower depth water, in reference to the sloshing water discussed earlier. The ideal gas law which I have studied as part of physics states that PV = nRT, or pressure x volume equals the gas constant multiplied by the temperature, scaled by the number of moles in the gas. This can be applied here to find a pressure value for the particles. Initially, I sought to implement a method where the temperature and gas constant could be adjusted. However, I found that to be needlessly complex; I already have mass as an adjustable parameter, which would at face value give a similar effect as temperature and gas constant parameters. Therefore, in an attempt to reduce the complexity of the simulation to the user as well as the recommendation from the paper, I reduced the formula significantly into p = k.rho, or pressure equals density multiplied by a constant. This was derived from the actual ideal gas law: replacing nRT with a single constant gives PV = k. Volume here can be replaced with mass over density, where mass can again be reduced to a constant. Rearranging for pressure gives the outlined equation. This is a more efficient equation as the pressure of a particle can be attained through just one multiplication. Given I intend to potentially have a thousand particles, I was keen to ensure that the simulation runs smoothly. Moreover, this is much easier to implement and understand, both for myself and the user. The paper suggests adding a rest density, which will affect the gradient of the smoothing curve to make it more stable.




```
def calculate_pressure(self):  # ideal gas law
   self.pressure = stiffness_constant * (self.density - self.spatial_map.rest_density)
```

In order to calculate the pressure force, we can use the general equation, replacing A with pressure. This largely follows the same process as calculating the density, whereby I utilise the spatial map to find neighbouring particles and use the smoothing kernel to calculate their respective influences.

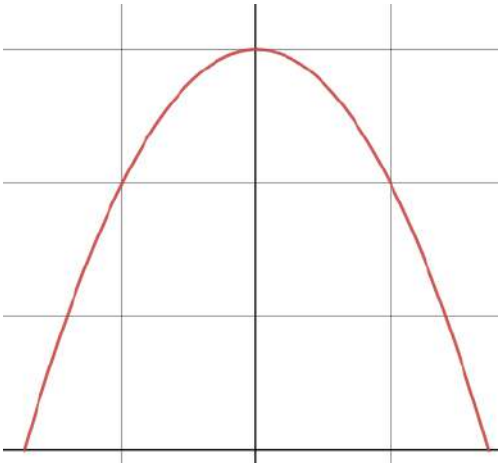
```
def calculate_pressure_contribution(self, particle, neighbour_particle, dist):
   direction_vector = self.spatial_map.normalise_vector(particle.position -
neighbour_particle.position)
   if particle.density == 0 or neighbour_particle.density == 0:
       return np.array([0,0])
   pressure = particle.pressure / (particle.density ** 2)
   neighbour_pressure = neighbour_particle.pressure / (neighbour_particle.density ** 2)

   smoothing_kernel_gradient = self.spatial_map.kernel.cubic_spline_kernel_gradient(dist)

   pressure_force = -neighbour_particle.mass * 0.5 * (pressure + neighbour_pressure) *
smoothing_kernel_gradient * direction_vector
   return pressure_force
```

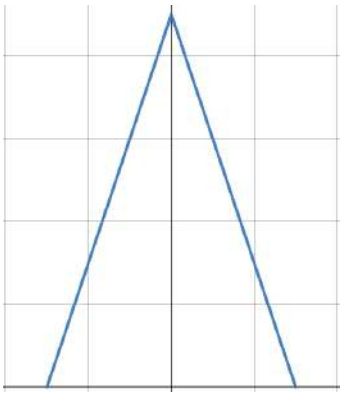
In order to translate this pressure value into actual movement, I created a new function which combined the above concepts into a force, which led to a change in velocity.

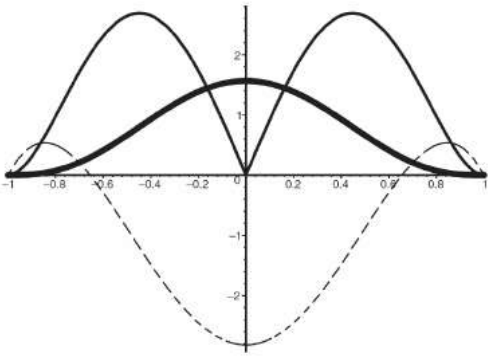
```
def calculate_pressure_force(self):
    self.pressure_force = np.zeros(2, dtype=float)
    neighbouring_particles = self.spatial_map.get_neighbouring_particles(self)
    for neighbour_particle in neighbouring_particles:  # Use pressure of neighbouring
particles
        if self != neighbour_particle:
            distance = self.spatial_map.get_magnitude(self.position -
neighbour_particle.position)
            self.pressure_force -= self.calculate_pressure_contribution(self,
neighbour_particle, distance)
    self.velocity += self.container.dt * self.pressure_force / self.mass
```

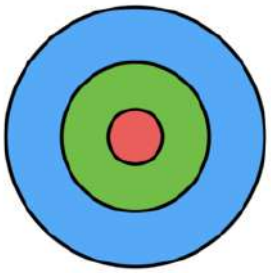
Smoothing Kernel

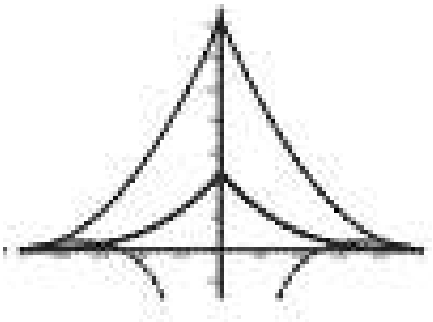
The smoothing kernel was a concept entirely new to me. However, after some research, it is simply a mathematical function to dictate how much influence a property has based on the distance. In a glass of water, the main force on a given section of water is mainly influenced by the molecules of water close to it. In a large container, the molecules on the outskirts might have for example a large pressure difference compared to the centre, but since the distance is large, this force is considered negligible. This idea is not restricted to fluid flow; in my projectile motion simulation, I have a target which is similar to a dart board. The user fires particles at this target and the closer to the centre it lands, the more points the user achieves. This example on the left is one of the basic smoothing functions which I will use to present the basic properties of a smoothing function. Take the x axis to be the distance of the given object to the goal. The cut off point is the smoothing radius, or h. Beyond this distance, any objects will not be considered as they are too far to have an influence. As such, the function returns 0 to any distance beyond this smoothing radius. Symmetry is a key property of these functions, as what matters is the absolute distance, rather than the direction. In the linear example, the value returned is indirectly proportional to the distance. This is found in a beginner dart board, consisting of concentric circles. In my projectile motion simulation, I sought to smooth the top of the function, such that there is a high initial gradient which steadily tapers out. This makes it so that the user can get a high











amount of points for getting it vaguely near to the centre. I did this through the function f(x) = 1-x^2.

When it comes to applying a smoothing kernel to fluid flow, the functions tend to look more complex. Ideally for density but also most other properties, values near the ends should be low with an equally low gradient. This would ensure a seamless transition between a zero distance and a very small distance. Values around the centre should be similar to the negative quadratic as they should have the majority effect. This results in a hill shape, whereby values with a short distance have a much stronger influence compared to those with a greater distance. Included on the graph on the right are the first and second derivatives which demonstrate how the gradient changes with distance. This is called a poly 6 kernel and its function is (h^2-r^2)^3. However, as the paper outlines, this hill shape where the gradient tapers out towards the centre can cause problems in regards to the pressure force. If the distance - or radius - is small, then so is the gradient and the repulsion force, leading to particles clustering together. To solve this, it proposes Debrun's kernel where the function is 'spiky' in the middle to provide the necessary repulsion force to stop particles clustering together. Its function is (h-r)^3. Lastly, we also need an alternative for the viscosity force. At some distances, the Laplacian can end up being negative, leading to the force being the wrong way and increasing the relative velocity instead of applying the intended damping effect

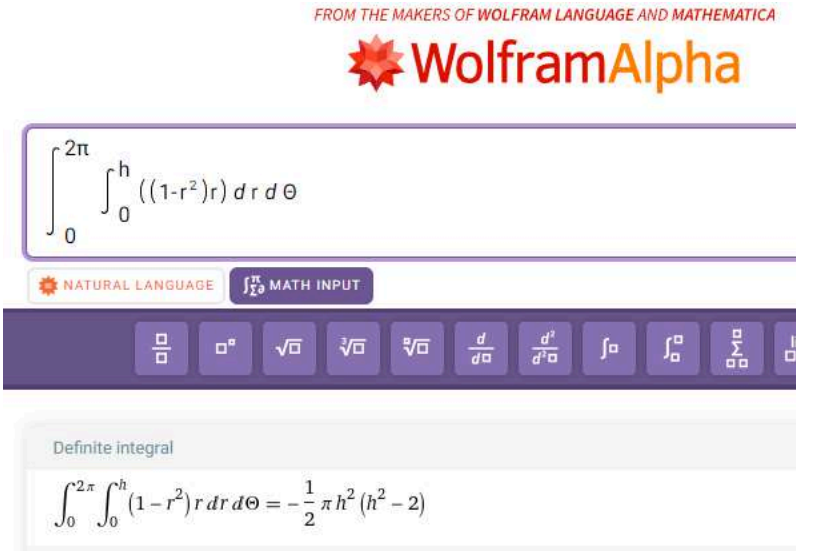
```
def __init__(self, smoothing_length, poly_6=False, gaussian=False, cubic_spline=False,
spiky=False, test=True):
    self.h = smoothing_length / 2 if cubic_spline else smoothing_length  # the radius
within which a neighbouring particle will be considered

    self.cubic_spline = cubic_spline
    self.poly_6 = poly_6
    self.gaussian = gaussian
    self.spiky = spiky
    self.test = test
    if poly_6:
        self.normalisation_constant = 315 / (64 * np.pi * self.h ** 2)  # ** 9
    elif gaussian:
        self.normalisation_constant = 1 / (np.sqrt(2 * np.pi) * self.h)
    elif cubic_spline:
        self.normalisation_constant = 10 / (7 * np.pi * self.h ** 2)
    elif spiky:
        self.normalisation_constant = 15 / (np.pi * (self.h ** 6))  # ** 6
    elif test:
        self.normalisation_constant = 1
```

The above code shows my implementation of normalising the output. Depending on parameters taken when initialising the object, an appropriate value is assigned to self.normalisation_constant. This value can then be multiplied with the value from the smoothing function to give a normalised answer.

```
def test_kernel(self, particle_radius):
    return ((max(0, particle_radius ** 2 - self.h ** 2)) ** 3) / (np.pi * self.h ** 2
/ 4)

def cubic_spline_kernel_gradient(self, particle_radius):
    ratio = particle_radius / self.h
    if 0 <= ratio < 1:
        return -3 * ratio
    elif 1 <= ratio < 2:
        return -0.75 * (2 - ratio) ** 2
    else:
        return 0

def poly_6_kernel(self, particle_radius):
    if particle_radius <= self.h: # if particle is within smoothing radius
        return self.normalisation_constant * ((self.h ** 2 - particle_radius ** 2) **
3)
    return 0

def gaussian_kernel(self, particle_radius):
    return 0  # :(

def spikey_kernel(self, particle_radius):
    if particle_radius <= self.h:
        return self.normalisation_constant * (self.h - particle_radius) ** 3

    return 0
```

For all of the kernels which have been applied to fluid flow, it is necessary to normalise the final sum by a constant, so that the integral of a given kernel function equals 1. I found this confusing at first as I hadn't understood the effect the number of particles could have. When adjusting the smoothing radius, we also adjust the volume of the smoothing function, leading to disproportionate density, pressure etc values. Therefore, to normalise this and ensure consistency, we have to divide by this volume. To calculate the normalisation constant, you first find the integral of the function with limits of 0 and h, with respect to r, where h is smoothing radius and r is the distance between the particles, to find the contribution from the radial distances, i.e. from the centre to the circumference. Then, we once again find an integral to take into account the circular domain: find the integral of the previously attained function with limits of 0 and 2(pi) with respect to

$$W_{\text{viscosity}}(\mathbf{r},h) = \frac{15}{2\pi h^3}\begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & \text{otherwise.} \end{cases}$$



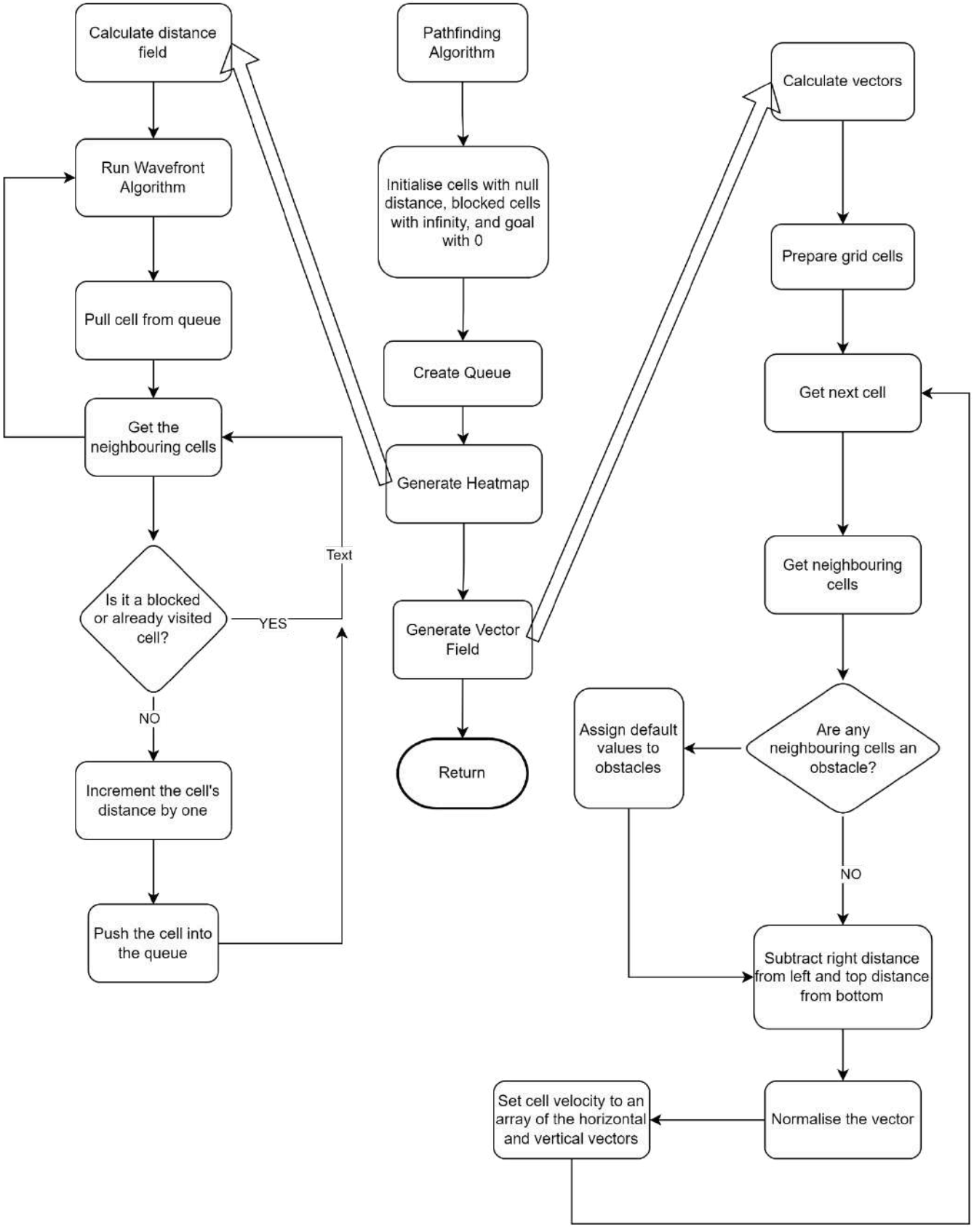
theta (the angle). This gives the normalisation constant. Let's say I was trying to find the normalisation constant for 1-r^2 from my projectile motion example. The result on the right gives a normalisation constant of (pi h^2) - 0.5(pi h^4). We can then use this to normalise the final value when calculating a given property. With the provided smoothing kernels for fluid flow, the functions are more difficult but the normalisation constants are fortunately provided by the paper.

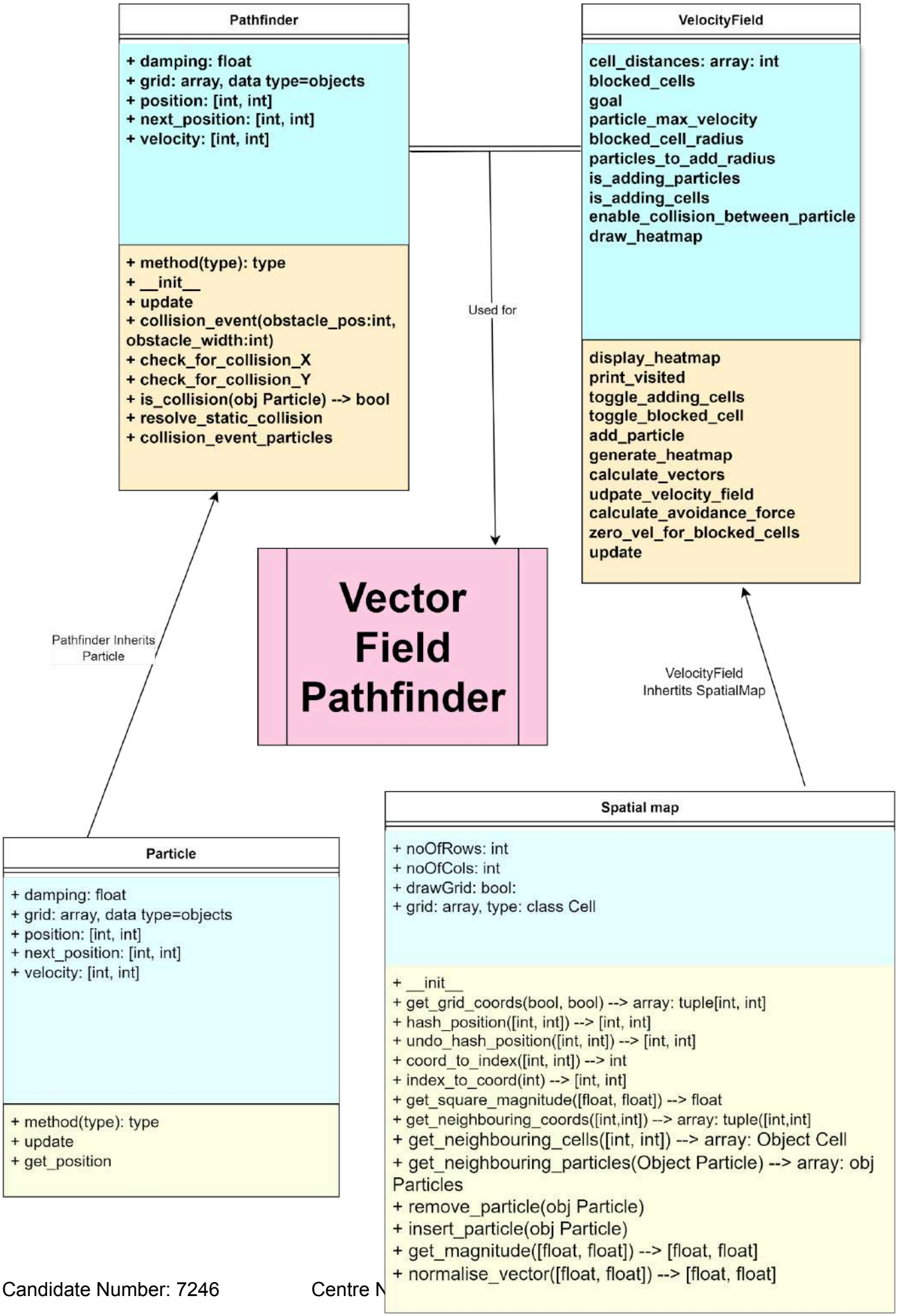
Post-Project Note

After having invested an excessive amount of time in attempting to implement this, I have decided to divert my time and efforts onto the other simulations in order to be able to deliver a more finished product and have more to show. This topic proved much more complex than I initially anticipated which led to very slow progress. I believe my key downfall was the lack of deep understanding I initially had when trying to dissect the paper. For example, only after having gone a fair way into the programming of the simulation, I realised a few key mistakes which required major restructuring. This in turn led to more issues which proved very-time consuming to fix.

Although I did not finish the simulation, I believe that I nonetheless have enough on it to warrant a section on the paper. As it is unfinished, it will not be included in the final program.

Vector Field Pathfinder




Pathfinder
+ damping: float
+ grid: array, data type=objects
+ position: [int, int]
+ next_position: [int, int]
+ velocity: [int, int]
+ method(type): type
+ __init__
+ update
+ collision_event(obstacle_pos:int, obstacle_width:int)
+ check_for_collision_X
+ check_for_collision_Y
+ is_collision(obj Particle) --> bool
+ resolve_static_collision
+ collision_event_particles
VelocityField
cell_distances: array: int
blocked_cells
goal
particle_max_velocity
blocked_cell_radius
particles_to_add_radius
is_adding_particles
is_adding_cells
enable_collision_between_particle
draw_heatmap
display_heatmap
print_visited
toggle_adding_cells
toggle_blocked_cell
add_particle
generate_heatmap
calculate_vectors
udpate_velocity_field
calculate_avoidance_force
zero_vel_for_blocked_cells
update
Used for
Vector Field Pathfinder
Pathfinder Inherits Particle
VelocityField Inhertits SpatialMap
Particle
+ damping: float
+ grid: array, data type=objects
+ position: [int, int]
+ next_position: [int, int]
+ velocity: [int, int]
+ method(type): type
+ update
+ get_position
Spatial map
+ noOfRows: int
+ noOfCols: int
+ drawGrid: bool:
+ grid: array, type: class Cell
+ __init__
+ get_grid_coords(bool, bool) --> array: tuple[int, int]
+ hash_position([int, int]) --> [int, int]
+ undo_hash_position([int, int]) --> [int, int]
+ coord_to_index([int, int]) --> int
+ index_to_coord(int) --> [int, int]
+ get_square_magnitude([float, float]) --> float
+ get_neighbouring_coords([int,int]) --> array: tuple([int,int]
+ get_neighbouring_cells([int, int]) --> array: Object Cell
+ get_neighbouring_particles(Object Particle) --> array: obj Particles
+ remove_particle(obj Particle)
+ insert_particle(obj Particle)
+ get_magnitude([float, float]) --> [float, float]
+ normalise_vector([float, float]) --> [float, float]

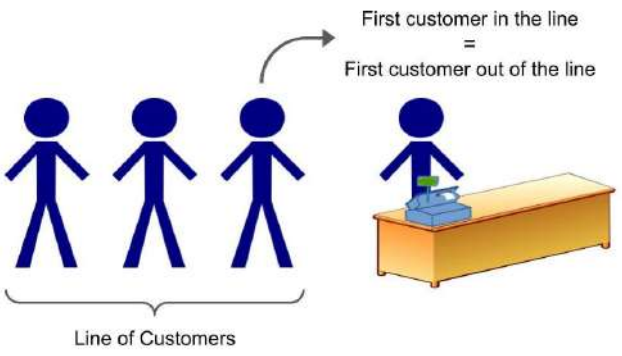
With my vector field pathfinder, there are a series of steps which outline the entire pathfinding algorithm. They can be summarised into

1. **Generating the distance field.**
2. **Calculating the velocity vector of each cell**
3. **Imposing the vector onto the particles to influence their movements**

Breadth-First Search

In order to calculate the vector field, I would first have to calculate the distance from each cell to the goal. There were two main considerations with this: depth-first search or breadth-first search. Depth first search is a tree traversal algorithm that explores as deep as it can into the tree, before backtracking and exploring the alternate path again to its leaf node. This repeats until the goal is reached, or in my case the entire tree has been searched. It is where we fully explore a given path, before backtracking and exploring the alternative path, hence the depth-first, and as such utilises a stack. Some popular algorithms include Dijkstra's algorithm, or A* algorithm which uses a heuristic. Conversely, breadth-first search explores all the nodes at a given depth, and then increases the depth and again searches all the nodes until completion. For my simulation, I will be assigning a distance value to every cell, and so I have decided to implement a breadth-first search technique, in particular the wavefront algorithm. *Below is my current implementation of the wavefront algorithm*. In regards to the stack implementation, I pull the from the front of the queue and search for its neighbours, which are then pushed onto the back of the queue - provided it's not been visited. This ensures that cells which have been in the queue for the longest are seen first.

A queue is the data structure needed for my implementation of the wavefront algorithm. It is similar to a real life queue, where people might form a queue on their way to the checkout. The person at the front, or the person who arrived earliest, is given priority to the checkout. Similarly, the cell which entered the queue first is also handled first. In this way, a queue operates on a 'first in, first out' basis, as opposed to the 'last in, first out' of a stack. The queue in my program will store the which have been discovered but have yet to be processed. There are three main functions to a queue. The first is enqueueing data. This is where a piece of data is pushed to the queue, i.e. the data is appended to the end of the queue list. When a neighbouring cell is discovered, it will be enqueued, providing that it has not already been processed. Secondly, you can also dequeue data. This is where the data at the front of the queue or at the first index of the list is pulled for handling. Here, I do this through queue.pop(0), which takes the data at index 0 (the start of the list). The final operation I considered was peeking. This is where I would be able to process or observe the piece of data at the start of the queue without actually dequeueing it. For my implementation, I will be



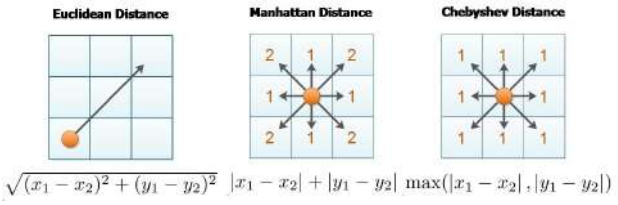
handling a cell in its entirety in one pass, so I will not be utilising this. I have made the following pseudocode as an in depth overview for how the queue will be utilised. The final implementation will be found in the next chapters.

```
FUNCTION BreadthFirstSearch
    Queue --> array(goal_reference)
    # Create the queue with the goal coordinates

    WHILE Queue IS NOT empty:
        current_coord --> DEQUEUE Queue
        # As per the FIFO property, we dequeue the first element of the queue

        MARK current_coord AS visited
        # Mark the current coordinate as visited to avoid revisiting

        neighbouring_coords --> FIND neighbouring coordinates OF current_coord
        FOR EACH neighbour IN neighbouring_coords:
            IF neighbour IS NOT blocked AND NOT visited:
                Handle distance calculations OF neighbour

                ENQUEUE neighbour
                # Enqueue the neighbour into back of the queue
```

There are a few drawbacks to the breadth-first search algorithm however. Firstly, the program becomes much more computationally and memory expensive as the size of the grid increases. This is because the number of cells increases at a large rate, which means more cells have to be searched and more data has to be stored. I have decided to limit the maximum dimensions to 50x50 as I found that to be the point where my home pc would not stay at the desired frame rate at the initial configuration. However, each user will have a different experience due to differences in hardware.
Factors that determine performance:

- ☐ A CPU with greater clock speed. This would allow for instructions to be executed at a faster rate, and so for cells to be processed quicker.
- ☐ A suitable amount of main memory. As mentioned, a greater size means that more memory is needed. If the program were to rely on virtual memory as it did on my laptop with next to no free memory, it would slow down the performance significantly.
- ☐ *Having too many obstacles or particles. Each time step, the program checks for obstacle collisions by looping each particle through each obstacle. More obstacles/particles therefore lead to more checks and a slower performance. The user can also enable collisions between particles.*
- ☐ *Displaying the distance heatmap, discussed in more detail in the next chapter.*

Distance Field



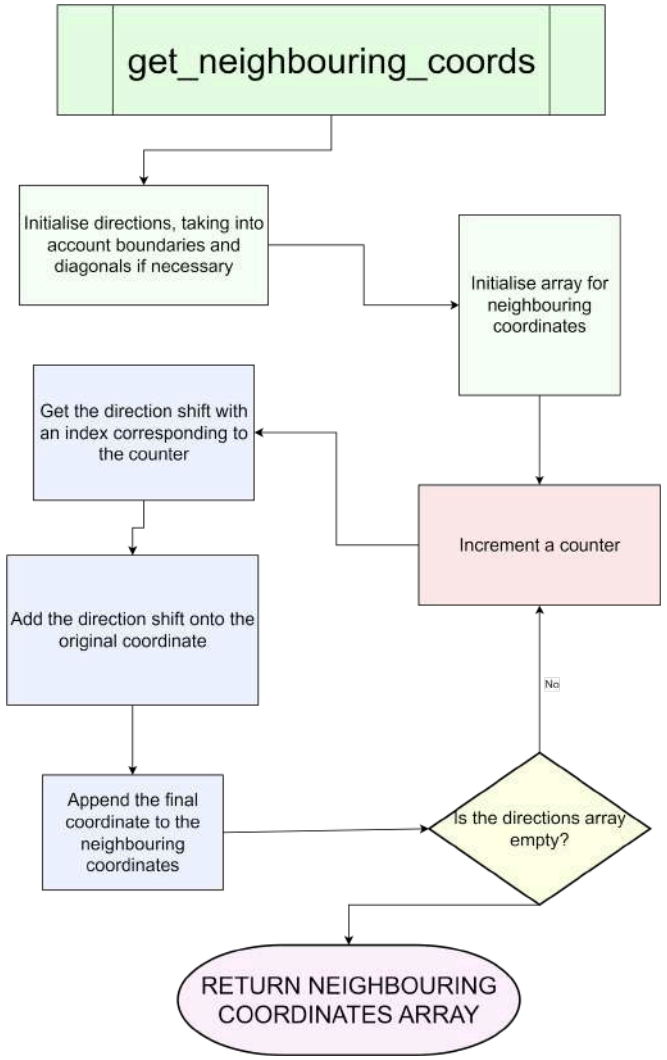
Referring to the flowchart shown under the 'Vector Field Pathfinder' subheading, I first initialise the cells as detailed later in the initialisation step. Then begins the bulk of the wavefront algorithm. First I set the distance of the cell corresponding to the goal coordinates to 0. As all cells aside from the goals will hopefully have a larger distance, they should ideally lead back to the goal cell. This goal is the starting point of the algorithm, so I initialise a queue with the only piece of data in it being the goal coordinates. Note the while loop here - while queue: - which ensures that the indented code will continue running for as long as the queue contains data items or in other words continue running for as long as a enqueued cell has unvisited neighbours.

Pseudocode for method coord_to_index:

```
FUNCTION coord_to_index(co-ordinates)
X ⟶ co-ordinates[0]
Y ⟶ co-ordinates[1]
RETURN X + Y * GRID_WIDTH
```

```
neighbouring_coords =
self.get_neighbouring_coords(current_coord,
include_diagonal=False)
```

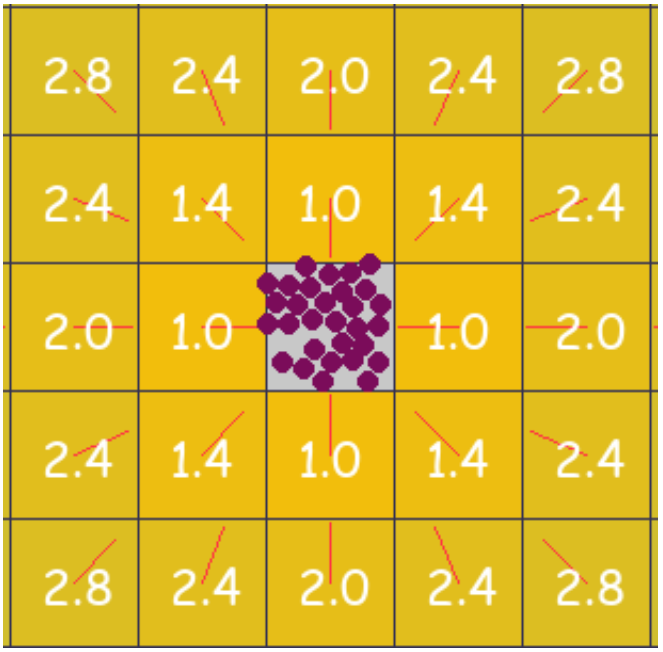
To find the neighbouring cells, I use the below function from the SpatialMap class in the baseClasses module. I can then access the cell sets which are stored as attributes to the cells.



```
    def get_neighbouring_coords(self, coord, include_diagonal=False,
include_self=False, placeholder_for_boundary=False):
        row, col = coord
        directions = [[1, 0], [-1, 0], [0, -1], [0, 1]] # right, left, up, down
        neighbouring_coords = []
        if include_diagonal:
            directions.extend([[-1, -1], [1, -1], [-1, 1], [1, 1]]) # need to
change if using euclidean distance

        if include_self:
            directions.append([0, 0])

        for dir in directions:
            neighbour_row, neighbour_col = row + dir[0], col + dir[1]
            if 0 <= neighbour_row < self.noOfRows and 0 <= neighbour_col <
self.noOfCols:
                neighbouring_coords.append((neighbour_row, neighbour_col))
            elif placeholder_for_boundary:
                neighbouring_coords.append(None)
        return neighbouring_coords
```

```
current_coord = queue.pop(0) # first cell dequeued
current_index = self.coord_to_index(current_coord)
current_distance = self.cell_distances[current_index]
```

First, I pop the first cell and find its index value through doing the method coord_to_index. To calculate the distance of a given cell, we have to take the preceding cell for which we have the distance and increment the distance by a value. In general implementations, it seems that Manhattan distance is the most popular, where a move up or right corresponds to a path cost of 1. Whilst I tried this, I was dissatisfied with the movement of the particles, so I implemented a more accurate distance field where a cell on the diagonal would have a path cost of sqrt(2). This worked out to be much more accurate than the initial Manhattan distance implementation. In real life, particles can move at any angle. However, in order to create a distance field, you must make a compromise. To increase accuracy, you could decrease the size of each cell, but then that leads to a slower performance.



```
for next_coord in neighbouring_coords:
    try:
        next_index = self.coord_to_index(next_coord)
        if next_coord not in self.blocked_cells and
self.cell_distances[next_index] == float('inf'):
            # update distance if  cell is not blocked and not visited
            change_x = abs(next_coord[0] - current_coord[0])
            change_y = abs(next_coord[1] - current_coord[1])
            if change_x == 1 and change_y == 1: # diagonal movement
                path_cost = np.sqrt(2)
            else:  # orthogonal movement
                path_cost = 1
            self.cell_distances[next_index] = current_distance + path_cost
            queue.append(next_coord)
    except IndexError:
        pass
```

I find the current distance of the cell by finding the appropriate distance value through the cell_distances list, which is a one-to-one of the cells in self.grid.

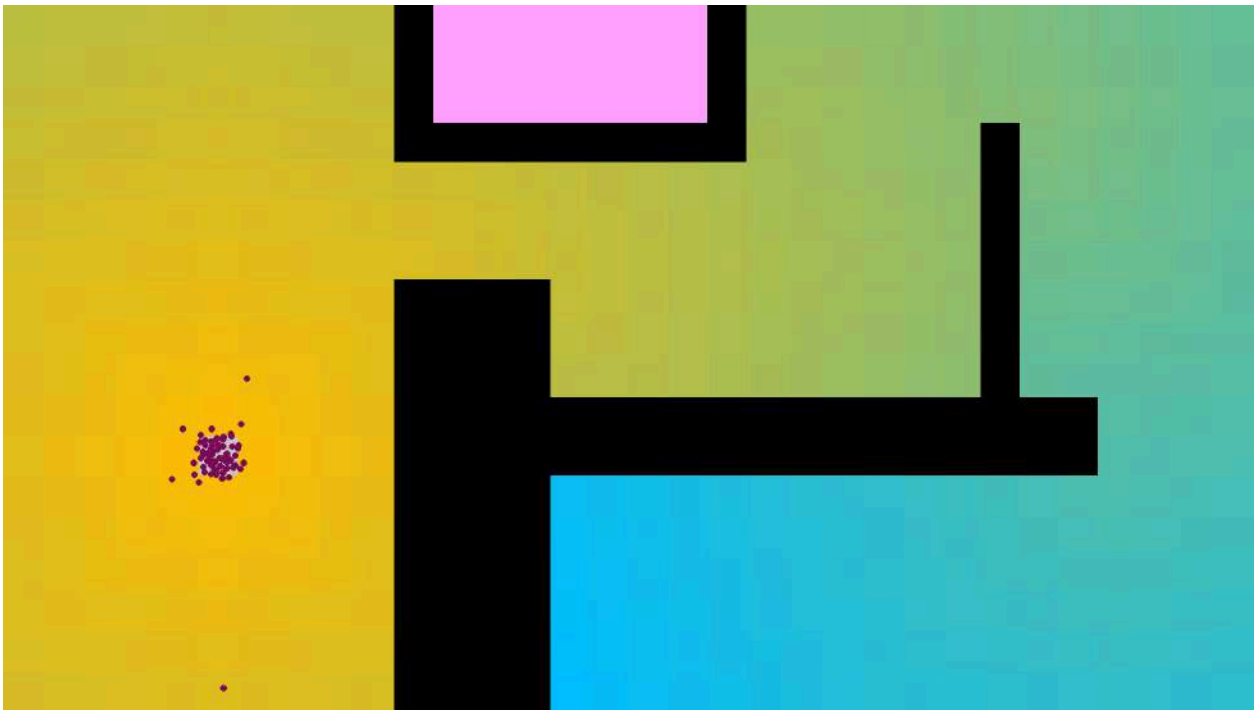
```
self.cell_distances[next_index] = current_distance + 1
```

Looping through the neighbouring coordinates, I set the distance of the neighbouring cells to the current cell's distance plus one, provided that the cell has not already been visited. Upon dealing with the cell, its corresponding state in the visited array is toggled. If I did not have the array for visited cells, the queue would never empty as cells would be continually added to the queue ad infinitum.

```
def generate_heatmap(self, goal_coords):
    self.cell_distances = np.empty_like(self.grid)
    # initialise cell distances with infinity, obstacles with -1
    for cell_index, cell in enumerate(self.cell_distances):
        cell_coord = self.index_to_coord(cell_index)
        if cell_coord in self.blocked_cells:
            self.cell_distances[cell_index] = -1
        else:
            self.cell_distances[cell_index] = float('inf')


    # set distance to the goal cell to 0
    goal_index = self.coord_to_index(goal_coords)
    self.cell_distances[goal_index] = 0

    self.cell_distances[goal_index] = 0

    # queue needed for breadth-first search
```

```
    queue = [goal_coords]

    while queue:
        current_coord = queue.pop(0) # first cell dequeued
        current_index = self.coord_to_index(current_coord)
        current_distance = self.cell_distances[current_index]

        neighbouring_coords = self.get_neighbouring_coords(current_coord,
include_diagonal=False)
        for next_coord in neighbouring_coords:
            next_index = self.coord_to_index(next_coord)
            if next_coord not in self.blocked_cells and
self.cell_distances[next_index] == float('inf'):
                # update distance if  cell is not blocked and not visited
                self.cell_distances[next_index] = current_distance + 1
                queue.append(next_coord) # enqueue the cell
```

To display this information to the user, I created a function to show a distance heatmap and grid distances when the user toggles buttons. To get a colour gradient, I would have to first find the minimum and maximum distances, as this would represent the two extremes. The minimum distance is obviously 0 for the goal itself, and for the maximum I used numpy's max function on cell_distances, after first filtering out any infinity distances for inaccessible cells. I was then able to assign a new value between 0 and 1 determined by where its distance fell relative to the maximum and minimum.
I opted for a light and smooth colour gradient as I wanted the heatmap to remain a background feature.



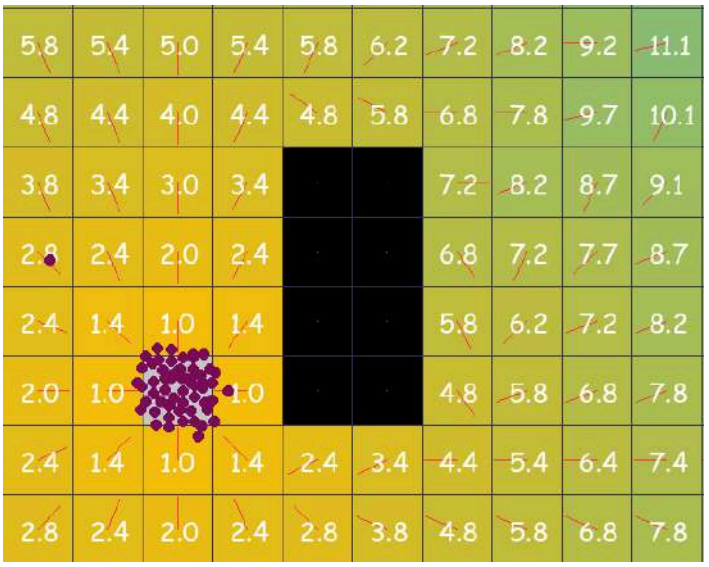
```
def display_heatmap(self, screen): # drawing a colour gradient depending
on the distance
   max_distance = np.max(list(filter(lambda x: np.isfinite(x),
self.cell_distances)))

   for i in range(self.cols):
       for j in range(self.rows):
           distance = self.cell_distances[self.coord_to_index((i, j))]
           if distance > 0:
               if np.isinf(distance): # give inf cell a specific colour
                   pygame.draw.rect(screen, (255,160,255), (i *
self.box_width, j * self.box_height, self.box_width, self.box_height))
               else:
                   norm_distance = distance / max_distance
                   colour = np.array([255 * (1 - norm_distance), 190, 255 *
(norm_distance)], dtype=int)
                   pygame.draw.rect(screen, colour,
                                    (i * self.box_width, j *
self.box_height, self.box_width, self.box_height))
```

Velocity Field

The second step as outlined earlier to implementing the vector field pathfinding algorithm is generating a vector field from the earlier created distance field.

As a precautionary measure, I apply a distance of -1 to all cells in the blocked cells array. These cells are not accessible to particles, so by applying this redundant value, it allows me to later handle this separately.

Fundamentally, calculating the vector field is straightforward. To explain my thinking when I came to approach this problem, I thought of it as navigating a town. Let's say that I am trying to reach the town centre. To the left of me is a path that takes 8 minutes, and to the right is a path that takes 10 minutes. If a person were to be in that position, they should travel left. In my calculate_vectors function, the same principles apply. By doing the left time minus the right time, we get a value of -2. This indicates that a given person should travel left, or in regards to the simulation, a particle should ideally move left to get to the goal as quickly as possible. Often I would get overwhelmed by the direction of the project but it is at its core a shortest pathfinding algorithm.ld get overwhelmed by the direction of the project but it is at its core a shortest pathfinding algorithm.



```
x_vector = dist_copy[1] - dist_copy[0]
```

Before the algorithm can calculate the final velocity vector, it needs to first find the distance values of the surrounding cells.

```
for index, (values) in enumerate(zip(self.cell_distances, self.grid)):
    distance, cell = values
```

The above shows the code for connecting the distance values with the corresponding cell from self.grid. Here, I use the zip function. This returns a zip object, which takes in the two lists and creates a tuple from each value at a given index. The values variable here in each iteration of the for loop is a tuple containing the cell distance and the cell at the given index. In the next line, I unpack the tuple into the distance and cell.

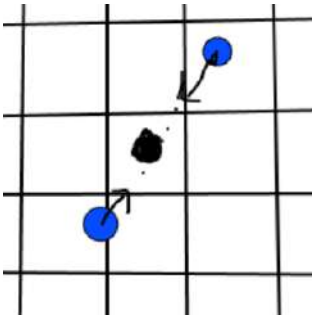
```
        coords = self.get_neighbouring_coords(self.index_to_coord(index),
placeholder_for_boundary=True)
        distances = [0, 0, 0, 0] # right, left, up, down
        for index, eachcoord in enumerate(coords):
            if eachcoord and eachcoord not in self.blocked_cells:
                distances[index] =
self.cell_distances[self.coord_to_index(eachcoord)]
            else:
                distances[index] = -1 # set blocked cell distance to -1 to
```

After setting the velocity of blocked cells to zero as a prerequisite, I look for the neighbouring cells through the get_neighbouring_cells method. An important thing I did here was create an auxiliary distance array. This allowed me to catch the distances of blocked cells (which was earlier set as -1). The way I handled these exceptions is discussed later in the ‘Obstacle Influence on Vector Field’ section.

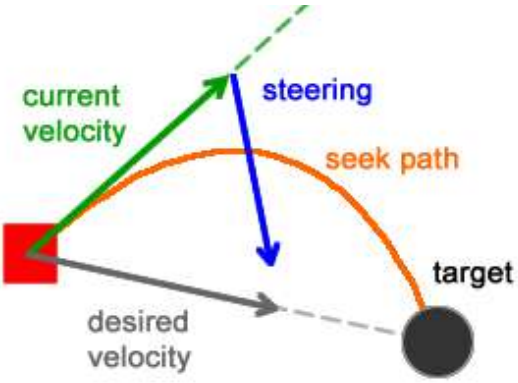
```
        x_vector = dist_copy[1] - dist_copy[0]
        y_vector = dist_copy[2] - dist_copy[3]

        cell.velocity = self.normalise_vector(np.array([x_vector, y_vector]))
```

Lastly, I ensure that the cell’s velocity is normalised. Firstly, the vector assigned to each cell leads to a force on the diagram, which then impacts the particle’s velocity component. As you can observe, when looking at a given axis, the distance between the left and right (or top and bottom) cell could be 2. However, the alternate axis could have a difference of 0 distance, in which case the given velocity component would also be 0. This means that the magnitude of the velocity of different cells will usually vary. This situation is even more present by the way I have implemented my blocked cells. Seeing as all cells should have equal weighting, it is important for all the cells to have an equal impact on the particles.

```
def calculate_vectors(self):
   for cell_coord in self.blocked_cells: # todo
       self.cell_distances[self.coord_to_index(cell_coord)] = -1 # initialise
distances


   for index, (values) in enumerate(zip(self.cell_distances, self.grid)):
       distance, cell = values
       if self.index_to_coord(index) in self.blocked_cells:
           cell.velocity = np.array([0,0]) # reset blocked cells velocity
           continue
       coords = self.get_neighbouring_coords(self.index_to_coord(index),
placeholder_for_boundary=True)

       distances = [0, 0, 0, 0] # right, left, up, down
       for index, eachcoord in enumerate(coords):
           if eachcoord and eachcoord not in self.blocked_cells:
               distances[index] =
self.cell_distances[self.coord_to_index(eachcoord)]
           else:

               distances[index] = -1 # set blocked cell distance to -1 to
stand out
       dist_copy = distances.copy()
       for index, distance in enumerate(distances):
           if distance == -1:
               if index == 0 and distances[1] != -1:
                   dist_copy[0] = distances[1] + 2
               elif index == 1 and distances[0] != -1:
                   dist_copy[1] = distances[0] + 2
               if index == 2 and distances[3] != -1:
                   dist_copy[2] = distances[3] + 2
               elif index == 3 and distances[2] != -1:
                   dist_copy[3] = distances[2] + 2

       x_vector = dist_copy[1] - dist_copy[0]
       y_vector = dist_copy[2] - dist_copy[3]

       cell.velocity = self.normalise_vector(np.array([x_vector, y_vector]))
```



One of the vital components to this simulation and the user is the particle movement. At first, I set the particle velocity as the cell velocity. This led to janky and robot-like movement.. I moved onto my initially planned method, which was applying a fraction of the current cell velocity onto the particle's own velocity every time step. This began to resemble a particle with inertia and displayed some, although limited, fluid-like visuals. However, a key problem was the lack of arrival behaviour. As shown on the right, the particles would oscillate about a point indefinitely. The amplitude of the oscillation was directly proportional to the speed upon reaching the goal and as there were no friction behaviours,, it would retain this energy indefinitely. This created unsatisfying and unrealistic behaviours. To remedy this, I believed that yet another steering force in the arrival force would solve this. My implementation completed this, but I was still dissatisfied with an underlying problem where the particles at a reasonable speed would 'swing' too far around a corner. Plus, this behaviour was more computationally expensive, meaning I had to limit my particle count. I reviewed this behaviour with my peers and they too agreed this was not an ideal reflection of particles. Therefore, I scrapped the above and implemented a seek steering behaviour which offered much more desirable results while maintaining speed.



- The variable 'desired_velocity' calculates the ideal velocity for the particle to reach the target.
- The variable 'steering_force' calculates just that (labelled in blue)
- Finally, we apply the steering force, which leads to the example seek path

In line 4, you can see that the magnitude of the desired velocity is given by a self.particle_max_velocity (the velocity of each cell is already normalised). As the name tells, this gives the maximum speed the particle can reach.

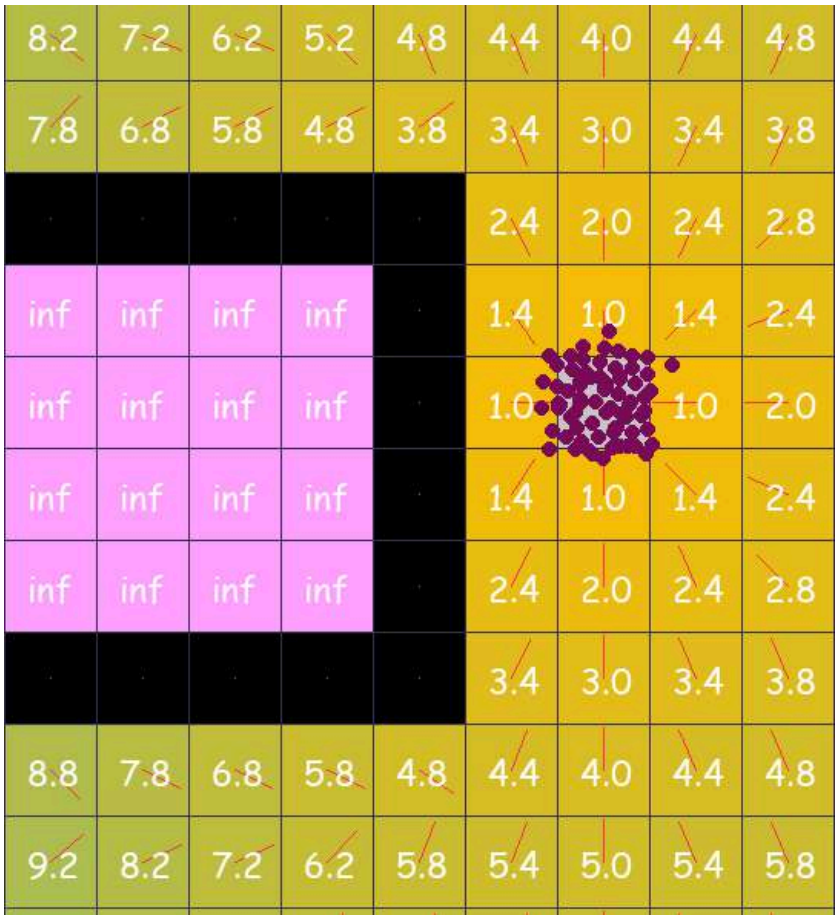
```
def update(self):
    field_strength = 0.02
    for eachCell in self.grid:
        desired_velocity = eachCell.velocity *
self.particle_max_velocity
        if any(np.isnan(desired_velocity)) or
any(np.isinf(desired_velocity)):
            continue

        for eachParticle in eachCell.cellList:
            steering_force = desired_velocity - eachParticle.velocity
            eachParticle.velocity += (steering_force * field_strength)
```

Initialisation Step

```
def generate_heatmap(self, goal_coords):
    self.cell_distances = np.empty_like(self.grid)
    # initialise cell distances with infinity, obstacles with -1
    for cell_index, cell in enumerate(self.cell_distances):
        cell_coord = self.index_to_coord(cell_index)
        print(cell_coord)
        if cell_coord in self.blocked_cells:
            self.cell_distances[cell_index] = -1
        else:
            self.cell_distances[cell_index] = float('inf')
```

etc

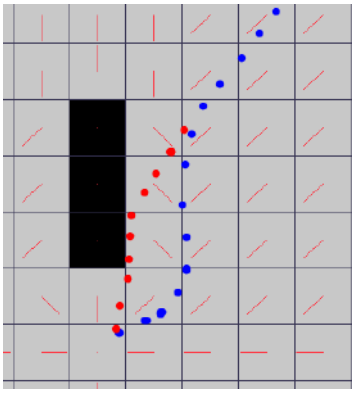
One key oversight which I initially missed was initialising the cell properties before running the bulk of the algorithm. For context, imagine the user has run the algorithm once and each cell has been assigned a distance and velocity vector. The user has since placed some walls which has made a region inaccessible to the particles. Now, the user decides to create a new goal which runs the algorithm again. However, the wavefront algorithm can't reach the inaccessible cells, and so those are not even registered to be handled in the calculate vectors step. These inaccessible cells keep their velocity vector and distance from the previous iteration. As such, particles which happen to be located in the inaccessible region are influenced towards a goal which no longer exists. This was very unsightly as it appears that the out of bounds particles follow a specific yet completely irrelevant path. My solution to this was to add an initialisation step before handling the main processes. I looped through all cells and assigned the free cells and blocked cells with a null distance and an infinity distance respectively. This allowed me to handle any unresolved cells afterwards thanks to being able to identify their state from their distance value. As a result, any cells in an inaccessible region would be left with a zero velocity vector, thus fixing the appearance of a mystery goal.



Obstacle Influence on the Vector Field

Another key decision was in dictating the role obstacles played on the pathfinding algorithm. The first that came to mind was to have the obstacles have no effect and simply be a static wall which the particles would collide with. This led to an excessive number of collisions with the wall and the particles seemed to stick to the obstacles. As such, I decided that the obstacles ought to have a repulsive force to provide a more fluid-like response. As such, I attempted to implement two different steering behaviours (*self.calculate_avoidance_force and self.calculate_collision_avoidance*)

This gave a much better response. However, a relatively large number of particles or more than just a few obstacles - both of which I deemed to be non-negotiable - would cause immense slow downs. As such, I decided to tamper with the vector field, so that the number of particles would not disproportionately slow down the entire program. Below is my current implementation. Here, I artificially included the blocked cells in the distance field temporarily to assign it a distance of +2 against its counterpart. This meant that, ignoring the other axis, the vector would point away from the obstacle. On the left image, the vectors show my latest implementation. The blue dots represent a particle following my new method, though slightly exaggerated. The red represents the path of the particle before the current implementation. To achieve the red solution, I simply replaced the +2 with a +1, so that the blocked cell had a neutral effect when calculating vector. However, after receiving feedback when testing, I decided that the blue solution where the obstacles push the particles away was more suitable.

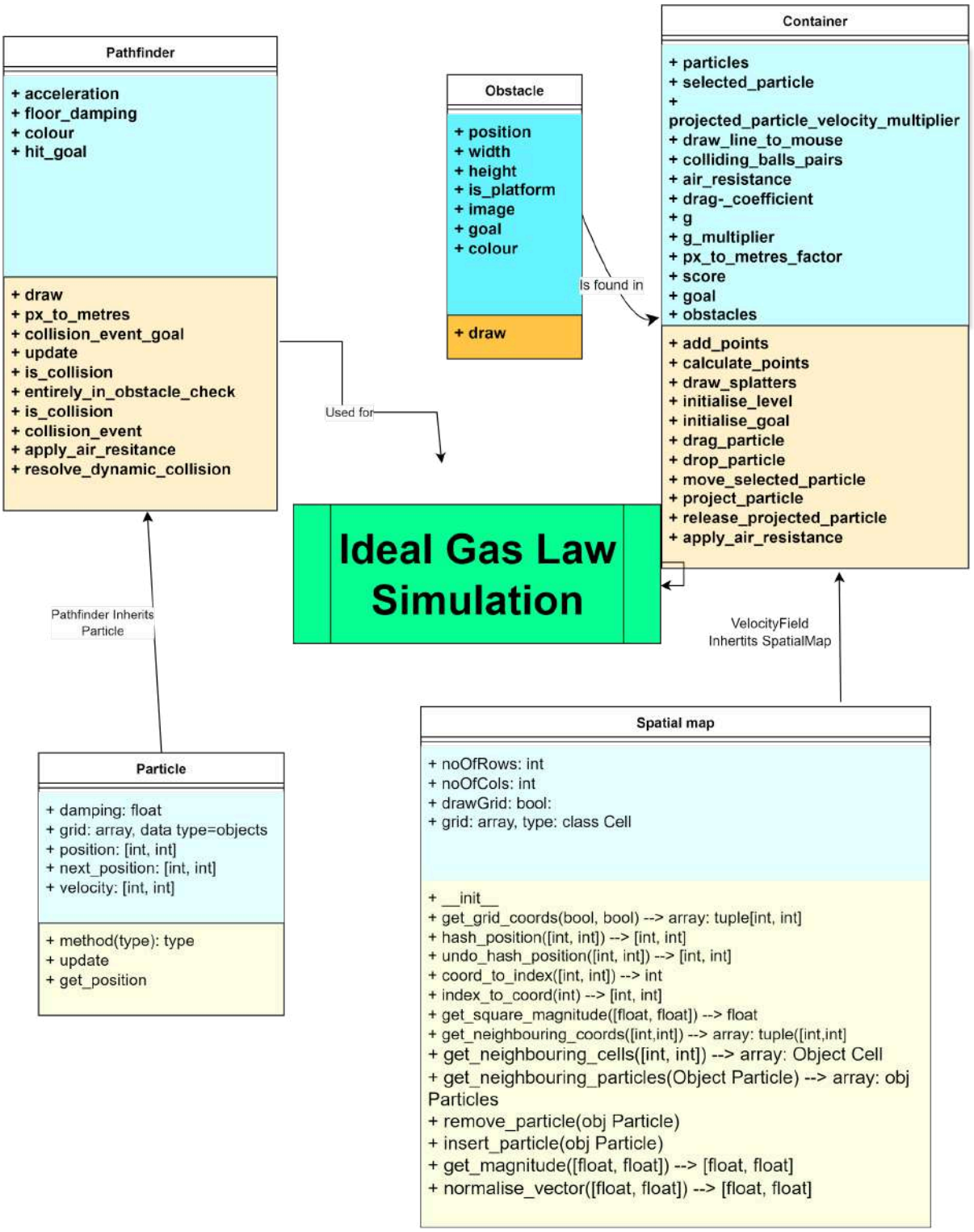


```
if distance == -1:
    if index == 0 and distances[1] != -1:
        dist_copy[0] = distances[1] + 2
    elif index == 1 and distances[0] != -1:
        dist_copy[1] = distances[0] + 2
    if index == 2 and distances[3] != -1:
        dist_copy[2] = distances[3] + 2
    elif index == 3 and distances[2] != -1:
        dist_copy[3] = distances[2] + 2
```

```
    def calculate_avoidance_force(self, position): # now disabled
        radius = 1 * box_width
        avoidance_strength = 150 # Adjust avoidance strength to avoid the obstacle to
        # choose strength of the steering force to desired taste
        steering_force = np.zeros(2)

        for obstacle in self.blocked_cells:
            distance = np.linalg.norm(position - obstacle)
            if distance < radius:

                steering_force += avoidance_strength * (position - obstacle) / distance
        return steering_force
```

```
def calculate_collision_avoidance(self, particle):  # Steering behaviour,
now deprecated
   magnitude = 1000
   ahead = particle.position + self.normalise_vector(particle.velocity) *
self.box_width
   box_centre_add = self.box_width / 2
   for coord in self.obstacles:
       distance_vector = ahead - (self.undo_hash_position(coord) +
box_centre_add)
       if self.get_magnitude(distance_vector) < self.box_width:
           return self.normalise_vector(distance_vector) * magnitude
   return 0
```

Ideal Gas Law

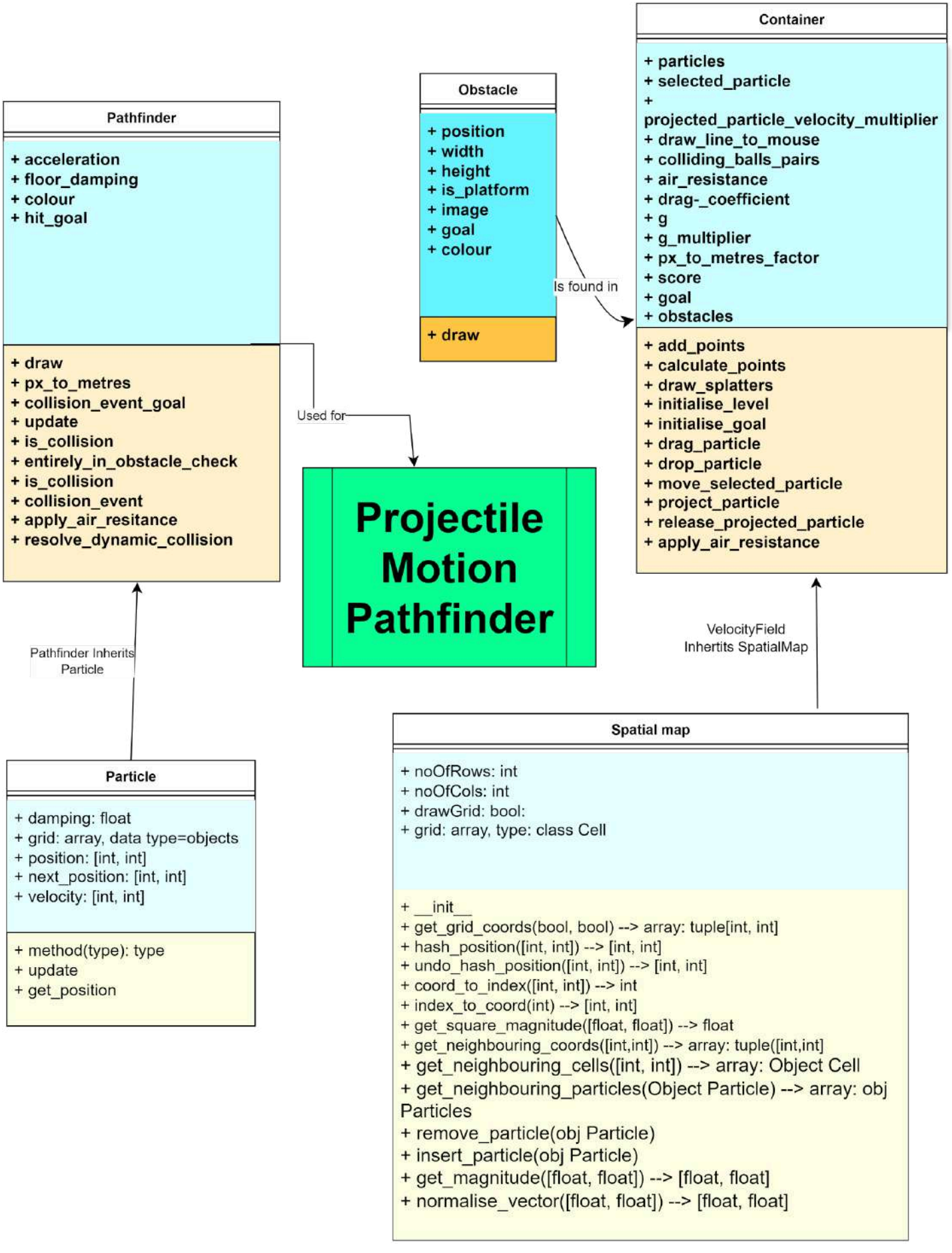


Root Mean Square Velocity

The rms velocity is essentially the average speed of the particle. In the below algorithm, I use the Pythagorean theorem to get the magnitude of the velocity. To make the movement of the particle meaningful, I have included a multiplier so that the pixels the particle has travelled over on the screen could correspond to a more realistic value.

```
def calculate_rms_velocity(self):
    total_square_vel = 0.0
    for particle in self.particles:
        squared_velocity_magnitude = np.sum((particle.velocity /
self.px_to_metres) ** 2)
        total_square_vel += 0.5 * particle.mass * squared_velocity_magnitude

    rms = np.sqrt(total_square_vel / len(self.particles))
    return rms
```

Projectile Motion


Container
+ particles
+ selected_particle
+ projected_particle_velocity_multiplier
+ draw_line_to_mouse
+ colliding_balls_pairs
+ air_resistance
+ drag-_coefficient
+ g
+ g_multiplier
+ px_to_metres_factor
+ score
+ goal
+ obstacles
+ add_points
+ calculate_points
+ draw_splatters
+ initialise_level
+ initialise_goal
+ drag_particle
+ drop_particle
+ move_selected_particle
+ project_particle
+ release_projected_particle
+ apply_air_resistance
Obstacle
+ position
+ width
+ height
+ is_platform
+ image
+ goal
+ colour
+ draw
Is found in
Pathfinder
+ acceleration
+ floor_damping
+ colour
+ hit_goal
+ draw
+ px_to_metres
+ collision_event_goal
+ update
+ is_collision
+ entirely_in_obstacle_check
+ is_collision
+ collision_event
+ apply_air_resitance
+ resolve_dynamic_collision
Used for
Projectile Motion Pathfinder
Pathfinder Inherits Particle
VelocityField Inhertits SpatialMap
Particle
+ damping: float
+ grid: array, data type=objects
+ position: [int, int]
+ next_position: [int, int]
+ velocity: [int, int]
+ method(type): type
+ update
+ get_position
Spatial map
+ noOfRows: int
+ noOfCols: int
+ drawGrid: bool:
+ grid: array, type: class Cell
+ __init__
+ get_grid_coords(bool, bool) --> array: tuple[int, int]
+ hash_position([int, int]) --> [int, int]
+ undo_hash_position([int, int]) --> [int, int]
+ coord_to_index([int, int]) --> int
+ index_to_coord(int) --> [int, int]
+ get_square_magnitude([float, float]) --> float
+ get_neighbouring_coords([int,int]) --> array: tuple([int,int]
+ get_neighbouring_cells([int, int]) --> array: Object Cell
+ get_neighbouring_particles(Object Particle) --> array: obj Particles
+ remove_particle(obj Particle)
+ insert_particle(obj Particle)
+ get_magnitude([float, float]) --> [float, float]
+ normalise_vector([float, float]) --> [float, float]

Levels

```
769,640,101,101
448,555,156,520,0
129,748,132,24,1
175,597,60,25,1
177,463,45,18,1
192,310,45,25,1
```

Here is an example level stored in the levels file. The first two columns signify the coordinates of the obstacle. The next two columns determine the height and width. Note that for the target, the width and height are the same as this represents the radius of the circular target. All obstacles bar the target have a boolean value. A True value means that the specified obstacle is a platform. Referring back to the collisions explanation in the universal algorithms, platforms have a smaller damping factor, so balls lose an excessive amount of energy upon collision, making it ideal to fire the balls off of.

In my program, I have an intialise_level function which takes a filename and initialises the contents into the program, as well as other essentials. Below, I create a container for the balls to sit in for easy access.

```
# creating ball box
box_dimensions = [
    ((100,1055),150,25),
    ((100,970),25,110),
    ((225,970),25,110)]
for row in box_dimensions:
    plank = Obstacle(*row, ball_box_image)
    plank.is_platform = True
    self.obstacles.append(plank)
```

Here, I initialise the splat pictures.

```
splatters = ["splat1.png", "splat2.png", "splat3.png"]
splat_width = 100
self.collision_splatters = []
self.splattered_particles = []
for splat in splatters:
    img = pygame.image.load("./Simulations/SimulationFiles/Assets/images/" + splat)
    img.convert_alpha()
    img = pygame.transform.scale(img, (splat_width, splat_width * img.get_height() // img.get_width()))
    self.collision_splatters.append(img)
```

The below code is for parsing the file contents. The goal is handled separately as there are special considerations. All other rows are passed into an Obstacle object

```
try: # parsing level
    with open(file_name, "r") as file:
        goal = file.readline()
        self.initialise_goal(goal.split(",")) # handling goal separately
        for line in file:
            line = line.split(",")
            object = Obstacle((int(line[0]), int(line[1])), int(line[2]), int(line[3]), wall_image)
            self.obstacles.append(object)
            if int(line[4]):
                object.is_platform = True
                object.colour = (37,41,74) # temporary
    return True

except FileNotFoundError:
    print("File not found")
    print(file_name)
    return False # and run level 1

except Exception as e:
    print(e)
    return False
```

Users with the appropriate access rights have access to a draw mode. This is used to design a new level and save it. The new file is saved under ProjectileLevels and can then be accessed by all other users. The first obstacle the user creates is the target. Then, the user creates platforms or walls. All levels require a target, and so the level may only be saved if a target has been drawn.

```
elif event.key == pygame.K_s:
   if obstacles:
       with open(("./Simulations/SimulationFiles/Assets/ProjectileLevels/lvl" +
str(level_no)), "w") as file:

file.write(f"{obstacles[0].position[0]},{obstacles[0].position[1]},{obstacles[0].width},
{obstacles[0].height}")
           for obstacle in obstacles[1:]:

file.write(f"\n{obstacle.position[0]},{obstacle.position[1]},{obstacle.width},{obstacle
.height},{int(obstacle.is_platform)}")
       print("Level saved")
   Else: # target not drawn
       print("Add a goal to save the level")
```

Scoring Points

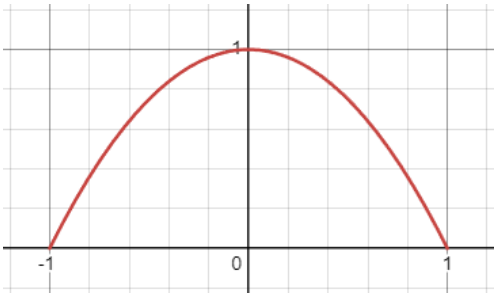
Points are scored by hitting the target with a ball. After achieving a high enough score, the next level then becomes unlocked in the main UI.
To handle target collisions, I first have to check if the particle is within the target each time step. If the distance from the particle to the target centre is smaller than its radius, then the particle is deemed to be within the target. Note that here I am using the square distance as the square root function is a costly calculation.

```
def entirely_in_obstacle_check2(self, pos, radius): # circle
    square_distance = self.vector_field.get_square_magnitude(pos - self.position)
    if square_distance < radius ** 2:
        return True
    return False
```

There were many different techniques I tried for hitting the target. However, I eventually settled on the following implementation. A ball is only considered to have finally 'hit' the target when it is stationary and in the actual target. When the ball is within the target, it will no longer be affected by external forces and will begin to slow down, according to a given penetration factor. Each time step that the ball remains in the target, it will slow down.

```
goal = self.vector_field.goal
if self.entirely_in_obstacle_check2(goal.position, goal.width):
    self.velocity = self.velocity * (1 - self.vector_field.penetration_factor)
    self.acceleration *= 0
    self.hit_goal = True
    self.colour = (25,125,195)
```

If the velocity then becomes zero (with a tolerance of 2 since theoretically a number will never reach 0 through just division), a sequence of steps begins. First, the appropriate score is added to the score. Generally with a target, the points increase linearly as distance from the centre decreases. However, as referenced in the Smoothing Kernel chapter in fluid flow, I opted for a different approach here. Instead, I use 1 - r^2 where r is the radius to determine the number of points scored. This reduces the loss in points with the larger radii, as shown on the diagram. Next, I remove the particle from the system, and replace it with a splat image.



$$y = 1 - x^2 \{y > 0\}$$

```
    if np.allclose(self.velocity, np.zeros_like(self.velocity), atol=2):
        self.vector_field.selected_particle = None
        try:
            self.vector_field.calculate_points(self)
            print(self.vector_field.score)
            self.vector_field.particles.remove(self)
            self.vector_field.splattered_particles.append(self)
        except:
            return

. . . . . .

def calculate_points(self, particle):
    multiplier = abs((self.get_magnitude(self.goal.position - particle.position) /
self.goal.width))
    points = int(100 * (1 - multiplier ** 2))
    self.add_points(points)
```

When the user closes the program, the score is returned to the main UI, which then determines if a high enough score was reached to allow entry into the next level.

Data Structures

Structure	Description	Intended Use
List	A list is used to store collections of data. In python, it is a dynamic data structure, meaning it has a variable length and each value can be altered. Each value is indexed, so this allows the same values to be distinguished within this structure.	Lists will be used in my program when storing a collection of data which have different data types, thanks to its flexibility in storing these. A key property of lists is that the values are indexed, meaning that I would be able to quickly access a value. I will also predominantly use them when storing non-numeric values which will not require much mathematical calculation. Where this is not the case, I will use a numpy array
	`self.particles = []` *In baseClasses.py → 194*	
Tuple	A tuple is another ordered collection of items similar to a list. However, these are immutable, meaning that its values cannot be altered nor can a value be added to the tuple upon creation	I will use them for occasions where the values of the array would not have to be altered nor added to regularly. For example, tuples will be used throughout for storing RGB colour values.

<table>
<tr><td></td><td colspan="2">

```
self.colour = (184, 146, 255)
```

projectileMotionSimulation.py → 223</td></tr>
<tr><td rowspan="2">Set</td><td>A set is also a collection of data. Like a list, it is mutable, so you can add values to the set. They are not indexed, so a given value will not have an index associated with it. The key different is that a set doesn't allow for duplicate values.</td><td>I will use sets where uniqueness is a necessity. For example, when I dealt with resolving collisions, I used a set when going through the collisions list. I would then add the main ball into the set. If the other ball is in the set, then we should not resolve again because it would use the position values from before the main ball was resolved, thus causing inconsistencies. The set here increases efficiency as it quickens lookup time.</td></tr>
<tr><td colspan="2">

```
# example of a set
completed = set()
for ball_i, ball_j in vector_field.colliding_balls_pairs: # loop over
all collision
   completed.add(ball_i)
   if ball_j not in completed: # ensure that the particle in question
hasn't already been resolved
       ball_i.resolve_dynamic_collision(ball_j)
       pygame.draw.line(screen, (0, 255, 0), ball_i.position,
ball_j.position)
vector_field.colliding_balls_pairs.clear() # reset the list for the
next time step
```

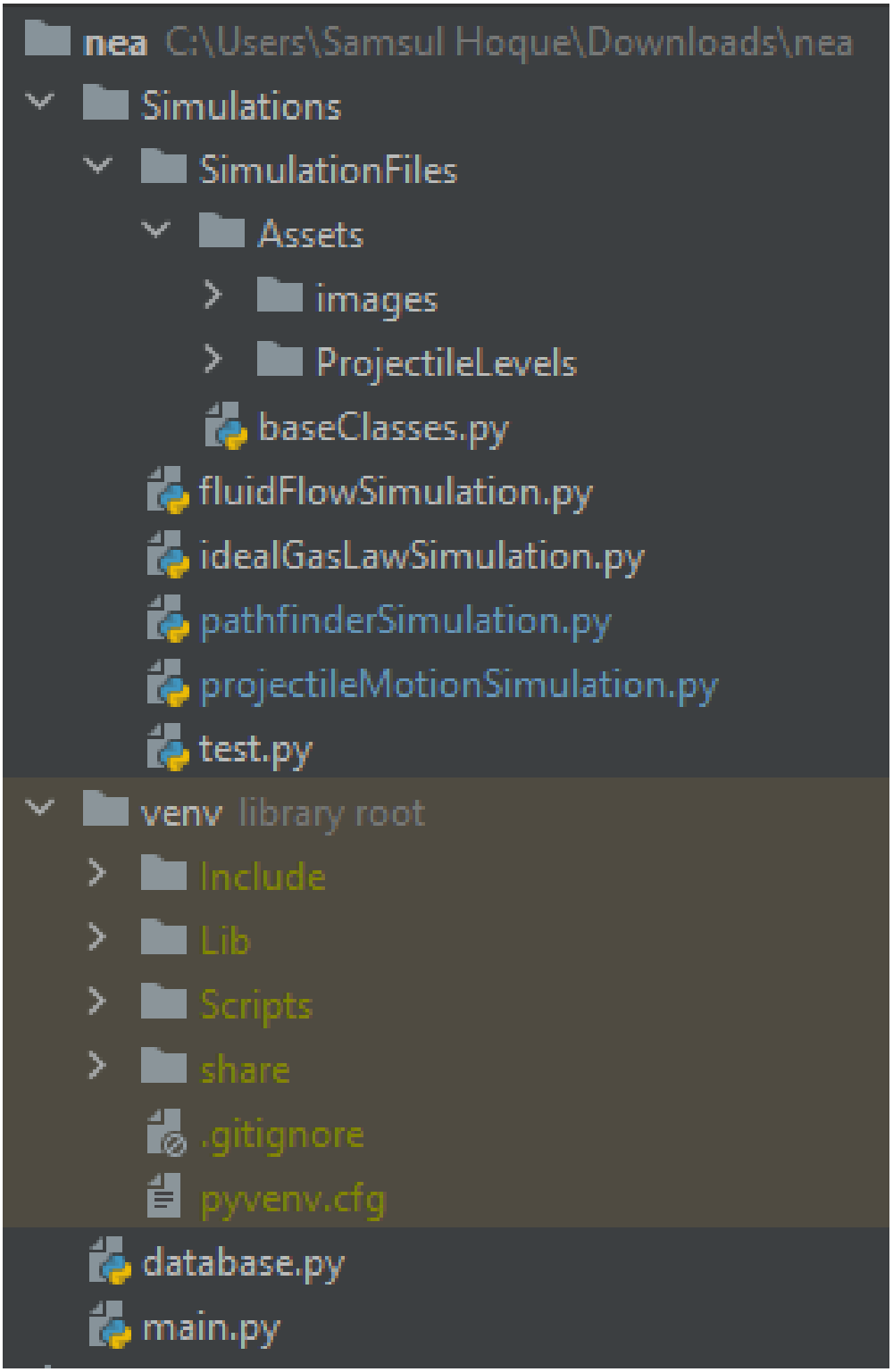
</td></tr>
<tr><td rowspan="2">Numpy Array</td><td>This is fairly similar to a list. One key difference is that all values must be of the same data type. You can also perform a variety of vector arithmetic with them such as combining two vectors with ease or performing the dot product. The homogeneity in the data type means that operations are much more efficient than if I were to use lists</td><td>Numpy arrays are the most important data structure in my program. They are used extensively in all of the simulations, in particular for storing position and velocity vectors. Here, it allows me to easily manipulate the path of the particles; for example, if I want to dampen the velocity by a factor of 0.9, I can simply multiply the vector by 0.9. Plus, they allow for more complex operations such as the dot product, which is necessary for calculating the tangential and normal velocities when resolving the dynamic collisions.</td></tr>
<tr><td colspan="2">

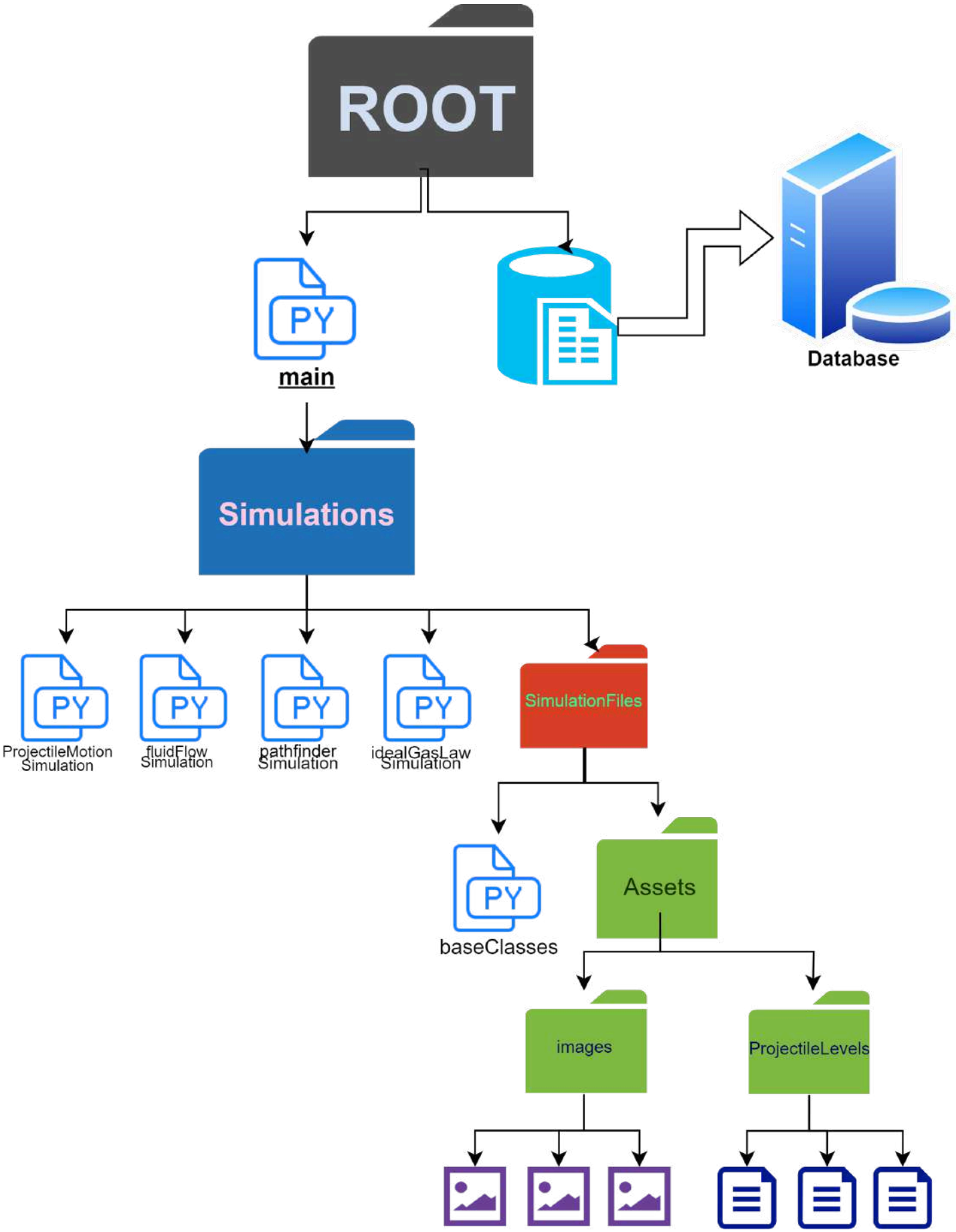
```
tangential_vel_i = tangent * np.dot(self.velocity, tangent)
```

baseClasses.py → 122</td></tr>
</table>

File Organisation

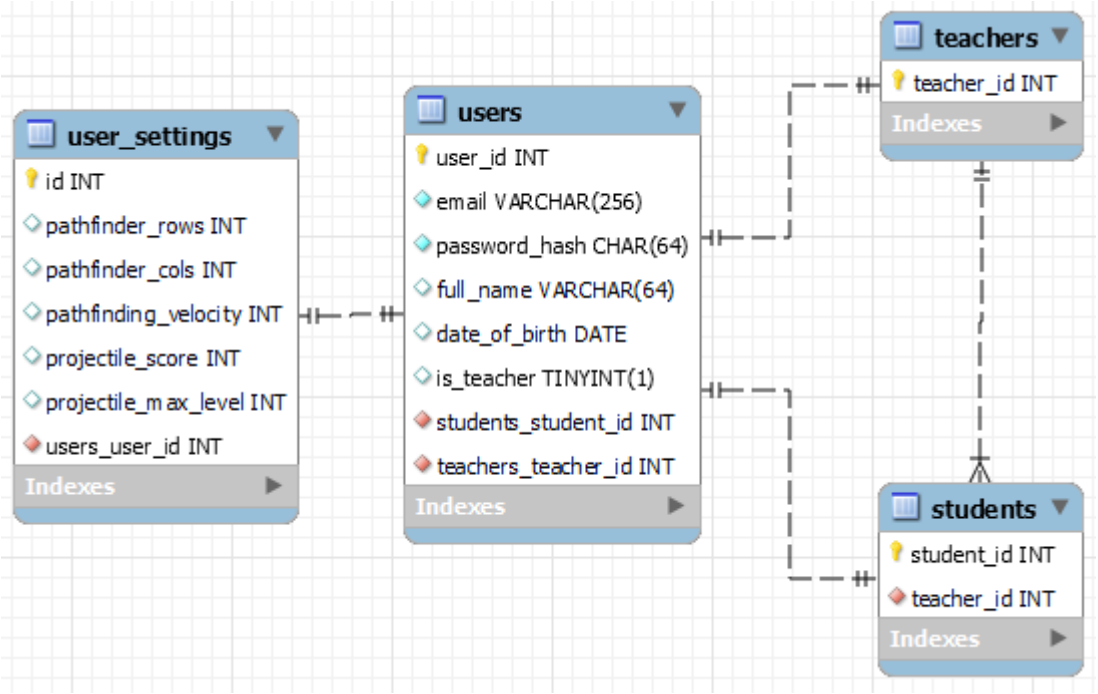
The program begins by executing the main program. The file for accessing the database is stored in the same directory. All simulations are stored in Simulations. In the same directory, I have a folder called SimulationFiles where I placed any files which might be needed to run the simulations. For example, the base classes can be found there, as well as images and projectile motion levels.




ROOT
PY
main
Database
Simulations
PY
ProjectileMotion Simulation
PY
fluidFlow Simulation
PY
pathfinder Simulation
PY
idealGasLaw Simulation
SimulationFiles
PY
baseClasses
Assets
images
ProjectileLevels

Data Mapping

Overview



The above outlines the database structure for the program. I have normalised where possible to eliminate data redundancy and inconsistency. For example, as shown in the table creation SQL code, I have included 'DELETE ON CASCADE', which ensures that if an account is deleted, all associated records are deleted along with it. A user can either be a student or a teacher and so has one-to-one relationships with each of these tables. Many students can have one teacher, so students is a many-to-one with the teachers.

My database has certain rules that the data must meet in order to be stored in the database, such as datatype and length. This ensures the database's integrity. It also makes searching through the data simpler. For example, I could write SQL code to find all students who are 18 or older.

Database Restriction	Description	Example	Valid Data	Invalid Data
Data type	A data value must be the datatype specified during the table creation	**DATE** *for the user's date of birth*	01-03-2024	123456
Length	To store a data value, its length shouldn't exceed the storage size allocated	CHAR(**64**)	A 64 long string	A 65 long string

Range	The data value must be between a certain range.	The data type INT requires numbers to be between -2^31 and 2^31, or in other words be 32 bits.	123456	1x10^10
Presence	The data field might require a data value to be given	In my simulation, I use NOT NULL to specify that it can not be left blank, for example the email field.	abc@abc.com	*"NULL"*

System Security

The security of the system and integrity of the data is a near-universal problem. Hackers are always searching for new ways to break into systems to get unauthorised access to data, which becomes increasingly valuable. The Cambridge Analytica scandal demonstrates the role data has in the world. A recent example of a data breach is from AT&T.

AT&T* has determined that AT&T data-specific fields were contained in a data set released on the dark web approximately two weeks ago. While AT&T has made this determination, it is not yet known whether the data in those fields originated from AT&T or one of its vendors. With respect to the balance of the data set, which includes personal information such as social security numbers, the source of the data is still being assessed.

AT&T has launched a robust investigation supported by internal and external cybersecurity experts. Based on our preliminary analysis, the data set appears to be from 2019 or earlier, impacting approximately 7.6 million current AT&T account holders and approximately 65.4 million former account holders.

Currently, AT&T does not have evidence of unauthorized access to its systems resulting in exfiltration of the data set. The company is communicating proactively with those impacted and will be offering credit monitoring at our expense where applicable. We encourage current and former customers with questions to visit www.att.com/accountsafety for more information.

As of today, this incident has not had a material impact on AT&T's operations.

https://about.att.com/story/2024/addressing-data-set-released-on-dark-web.html

Whilst the information I am storing is probably not as valuable as the data leaked in the above breach, it is still sensitive. For example, each user creates a password for their account. This prevents any unauthorised people from using their account. What more, a poll by Google showed that 1 in 8 Americans use the same password for all accounts, whilst another 52% used the same password for some accounts. If a hacker were to gain access to my users' passwords, it is likely they could go on to access the user's other accounts with potentially more disastrous consequences. As such, I have implemented hashing into the database.

A hash function is a one-way function which maps a piece of data into a new value. The key feature is that it is usually impossible to obtain the plaintext from a hash, which is why these are often used to store passwords. Note that the program never prints out a plaintext password. Firstly, the password field has an input mask. Secondly, to verify the password, the hash is sent to the database rather than the plaintext. If the two hashes match, only then will the database return the requested information. There are many different hash functions in use, each with their pros and cons. The primary considerations are speed and memory. In the end, I settled on the sha256 hash which is a popular choice for passwords. This produces a 32 byte string, and so I reserved 64 characters for this in the database (32 bytes = 64 hex digits). The large size means that hash collisions are very uncommon. Plus, it remains very fast and efficient. This makes it an obvious choice and it's standardised for good reason. I also considered password salting for added security to reduce the vulnerability to dictionary attacks, but I thought the benefits didn't outweigh the losses in speed..

I used the hashlib module from the python built-in library. This provided me with the ability to quickly hash the password. For convenience, I converted the string into hex for storage in the database.

```
email, password = self.email.text().strip(),
sha256(self.password.text().encode()).hexdigest()
user_info = self.database.verify_login(email, password)
```

A well-known vulnerability is SQL injections. To learn more about this attack, I used the following video by computerphile: https://www.youtube.com/watch?v=ciNHn38EyRc.
This enabled me to perform some penetration testing on my database. In a nutshell, a SQL injection is when the attacker sends their own malicious code in a specific format when the program asks for a user input, such that the server runs that malicious code. As an example, let's say I asked the user for an email and sent it over to the database so that I could print out their full name, I managed to perform a SQL injection to instead drop the entire database, thereby breaking the entire program. An well-prepared attacker might instead choose to get the full names or emails of the users. To protect against this, I use prepared statements so that the program escapes any of the queries that might otherwise change the logic of the query

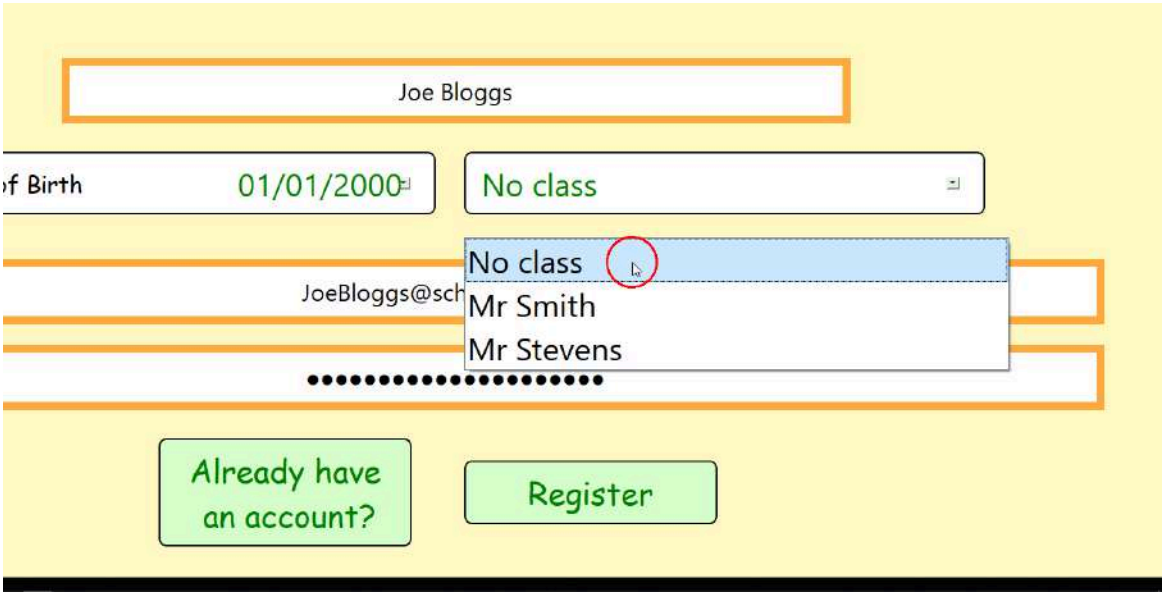
```
self.__conn.execute("""SELECT email FROM users WHERE full_name = %s;""", (name,))
```

My SQL Code

The users table is the main table in my program and is what is used when verifying the user log in. The primary key is the user_id, which is an auto_incrementing and unique integer. The table also stores the email address with a variable character length of up to 256

characters, which is in line with common practice. The password_hash stores the hash of the user password. The table also stores the user's full name and date of birth, with the latter being stored in the DATE datatype. I have declared some of the key fields to be 'NOT NULL' so that each account is guaranteed to have these values.The is_teacher field has a boolean datatype which dictates whether the given user is a teacher or a student. This is needed to add a new user to the designated table.

```
CREATE TABLE users (
    user_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    email VARCHAR(256) UNIQUE NOT NULL,
    password_hash CHAR(64) NOT NULL,
    full_name VARCHAR(64),
    date_of_birth DATE,
    is_teacher BOOLEAN DEFAULT FALSE
);
```



Students and teachers each have their own tables.. Teachers will have a teacher_id as a primary key, and students a student_id. Each table also holds a foreign key in user_id, as both students and teachers are fundamentally users. Students also have a designated teacher. In my program, I determine the access rights of the user by the teacher_id. All students have teacher_id's associated with them. If a student selects No class, they will be assigned a teacher_id of 0 and be grouped up with other students who also selected this option. One example is if the teacher_id is None, I adjusted the grid to add a spinbox widget.

```
def initialise_program(self):
    self.penetration_factor_button = QSpinBox()
    self.penetration_factor_button.setAlignment(Qt.AlignmentFlag.AlignRight)
    self.penetration_factor_button.setMaximumWidth(180)
    self.penetration_factor_button.setSuffix("%")
    self.penetration_factor_button.setRange(1, 100)
    self.penetration_factor_button.setValue(15)


    if self.teacher_id is None: # no teacher_id means user is a teacher
        self.projectile_instruction.setText(
            self.projectile_instruction.text() + "\nTeachers can use the spinbox to control the penetration factor!")
        self.weeklyButton.setMaximumWidth(350)
        self.projectile_widget_layout.addWidget(self.weeklyButton, 2, 2, 1, 1)
        max_level = len(self.projectile_sim_buttons)
        self.projectile_widget_layout.addWidget(self.penetration_factor_button, 2, 1, 1, 1)
    else:
        self.weeklyButton.setFixedWidth(750)
        self.projectile_widget_layout.addWidget(self.weeklyButton, 2, 1, 1, 2)
        max_level = self.user_settings[-1]

    print(self.user_settings)
    self.pathfinding_rows.setValue(self.user_settings[2])
```

```
CREATE TABLE teachers (
   teacher_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
   user_id INT NOT NULL,
   FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);

CREATE TABLE students (
   student_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
   user_id INT NOT NULL,
   teacher_id INT NOT NULL,
   FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
   FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id) ON DELETE
CASCADE
);
```

Each user has a user_settings. The two tables are linked by their user_id. The fields stores variables for the various simulations; for example, pathfinder_rows stores the last used pathfinder_rows so that the user's options are remembered from the last login. A key concept here is the 'ON DELETE CASCADE'. This means that if a record is removed from users, the associated record bound by user_id in user_settings is also deleted. This helps to reduce data inconsistencies.

```
CREATE TABLE user_settings (
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    user_id INT NOT NULL,
    pathfinder_rows INT DEFAULT 18,
    pathfinder_cols INT DEFAULT 32,
    wall_collision_damping FLOAT DEFAULT 0.8,
```

```
    pathfinding_velocity INT DEFAULT 500,
    projectile_max_level INT DEFAULT 1,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);
```

In my program, I have created many functions which can be executed to run SQL code, some of which I have included below

```
def create_user(self, email, password_hash, full_name, date_of_birth,
is_teacher=False, teacher_email=None):
   try:
       if not is_teacher:
           # if student, get their teacher identifier
           teacher_id = self.get_teacher_id(teacher_email)


       self.__conn.execute("""
       INSERT INTO users (email, password_hash, full_name, date_of_birth,
is_teacher) VALUES (%s, %s, %s, %s, %s);
       """, (email, password_hash, full_name, date_of_birth, is_teacher))
       self.__conn.fetchall()

       user_id = self.__conn.lastrowid

       # create user settings for the new user
       self.__conn.execute("""
       INSERT INTO user_settings (user_id) VALUES (%s);
       """, (user_id,))
       self.__conn.fetchall()

       if is_teacher:
           # add the teacher to the teachers table
           self.__conn.execute("""
           INSERT INTO teachers (user_id) VALUES (%s);
           """, (user_id,))

       else: # is a student
           self.__conn.execute("""
           INSERT INTO students (user_id, teacher_id) VALUES (%s, %s);
           """, (user_id, teacher_id))
       print("\n\nUser Created.", email, password_hash) # debugging
       return True
   except Exception as e:
       # printing debugging information
       print(email, password_hash, full_name, date_of_birth)
       print("Could not create user")
       print(e)
       return False
```

The above shows the database-side process for creating a new user. The process is largely the same for both students and teachers, at least initially. A new user is added to the table with the relevant information. Then, if the user is a teacher, they are added to the teachers table. Otherwise, they must be a student, so their teacher's teacher_id is retrieved using the get_teacher_id method, which needs an email address. Then, we can perform the

appropriate insertion into the students table. The exception here is not typically called unless there has been a clash with one of the parameters with what's in the table, for example a user trying to create a new account with an already used email address.

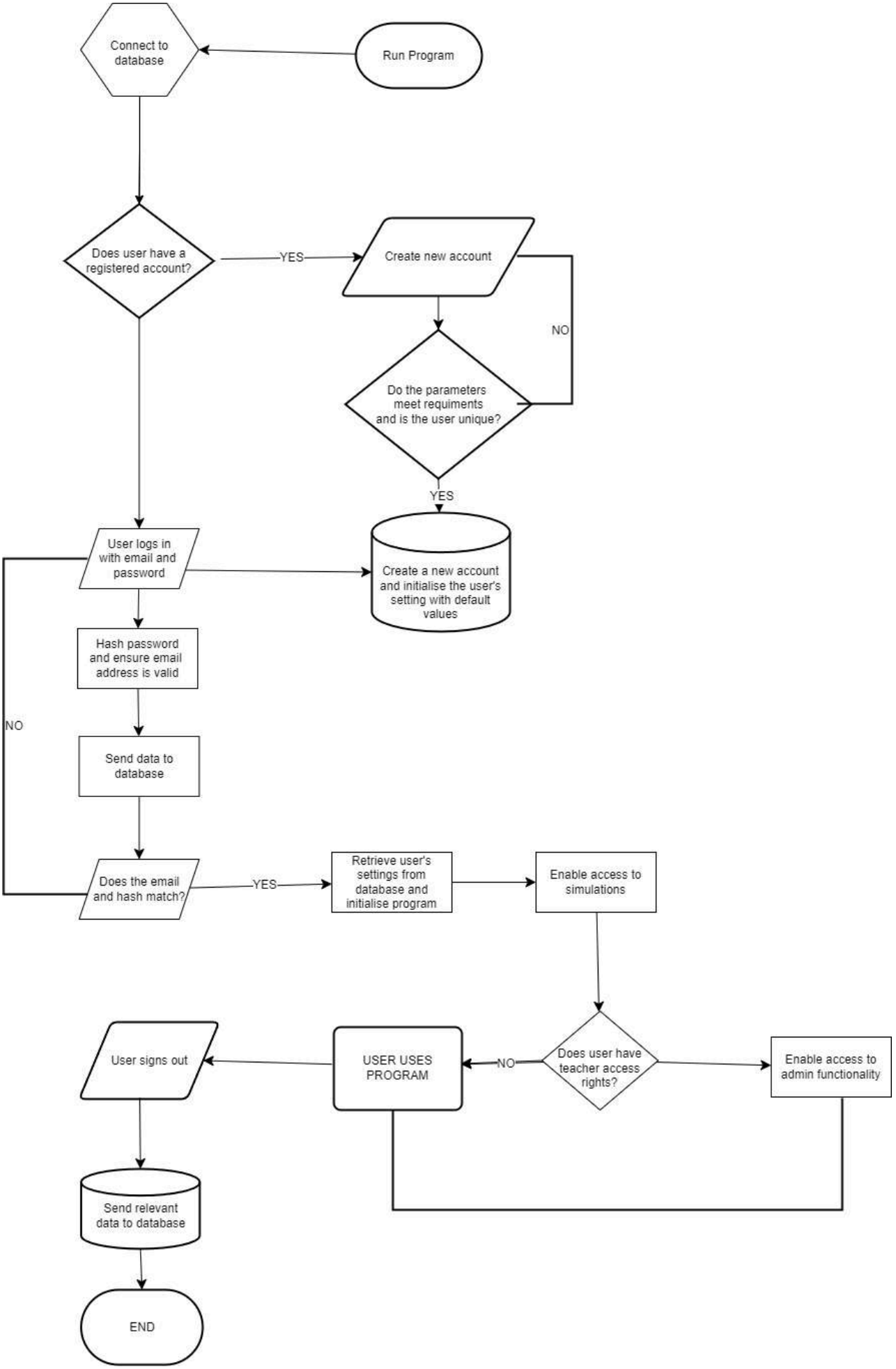
```
def save_and_shut_down(self, user_settings):
    settings = user_settings.copy()
    settings.append(settings[0])
    print(settings)
    print(settings[2:])
    self.__conn.execute("""
    UPDATE user_settings SET pathfinder_rows = %s, pathfinder_cols = %s,
wall_collision_damping = %s, pathfinding_velocity = %s,
projectile_max_level = %s WHERE user_id = %s;
    """, settings[2:])
    self.__conn.fetchall()
```

The above method triggers when the user logs out of the program and is run just before the PyQt6 window is terminated. This saves the data from the simulation into the database. Here, I use the UPDATE sql command to set the user_settings record into the new values, as determined during the running of the program.

```
def get_projectile_rankings(self, teacher_id):  # Used for projectile
motion leaderboard
    self.__conn.execute("""
    SELECT full_name, projectile_score
    FROM users, user_settings, students
    WHERE users.user_id = students.user_id
    AND users.user_id = user_settings.user_id
    AND students.teacher_id = %s
    ORDER BY projectile_score DESC;
    """, (teacher_id,))
    result = self.__conn.fetchall()
    return result
```

Here, I fetch the leaderboard rankings

=


Connect to database
Run Program
Does user have a registered account?
YES
Create new account
NO
Do the parameters meet requiments and is the user unique?
YES
User logs in with email and password
Create a new account and initialise the user's setting with default values
Hash password and ensure email address is valid
NO
Send data to database
Does the email and hash match?
YES
Retrieve user's settings from database and initialise program
Enable access to simulations
User signs out
USER USES PROGRAM
NO
Does user have teacher access rights?
Enable access to admin functionality
Send relevant data to database
END

Validation

Validation is essential in this program to ensure that all user inputs have been correctly entered.

Login Screen

The most validation takes place during the account creation stage. One small error means that invalid information could permanently sit in the database or cause errors (providing it meets the valid format – I expand on this in the Data Mapping chapter). As such, the program performs some routine checks.
FIrstly, the password must be at least 6 characters long.

```
if len(self.password.text()) < 6:
    detailed_text += "Password must be at least 6 characters.\n"
    valid = False
```

A name must also be given.

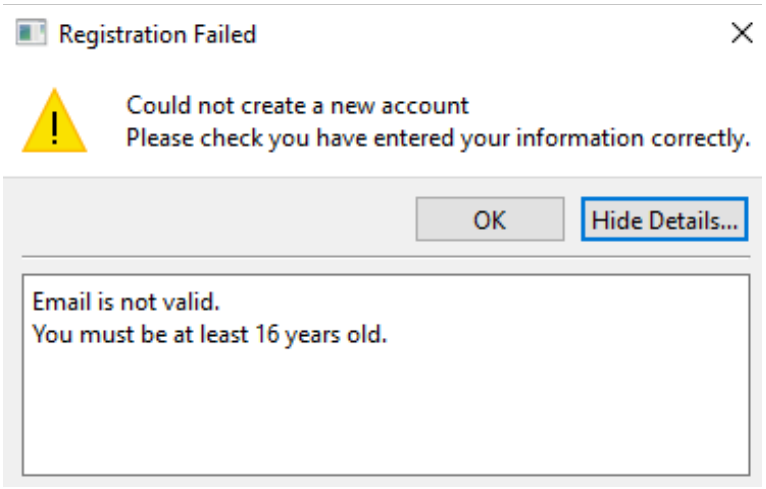
```
if not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
self.email.text()):
    detailed_text += "Email is not valid.\n"
    valid = False
```

The user must be above the age of 16.

```
if self.date_of_birth.date().addYears(16) > QDate.currentDate():
    detailed_text += "You must be at least 16 years old.\n"
    valid = False
```

And finally, I used the regex (regular expressions) module to ensure that the email met the standard format.

```
if not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
self.email.text()):
    detailed_text += "Email is not valid.\n"
    valid = False
```



If all fields do not meet the requirements, a new window is shown explaining the details.

Field	Description	Valid Data	Invalid Data
Full Name	The name of the user. It must not be empty	Joe Bloggs	*EMPTY*
Date of Birth	The user's date of birth. The QDateWidget handles the validation to ensure only valid dates are used. I have also added a check to ensure the user is older than 16	28/02/2001	29/02/2001
Email	The email of the user in the correct format	abc@email.com	abc@.com
Password	A private bit of text longer than 6 characters	abcde	abcdef

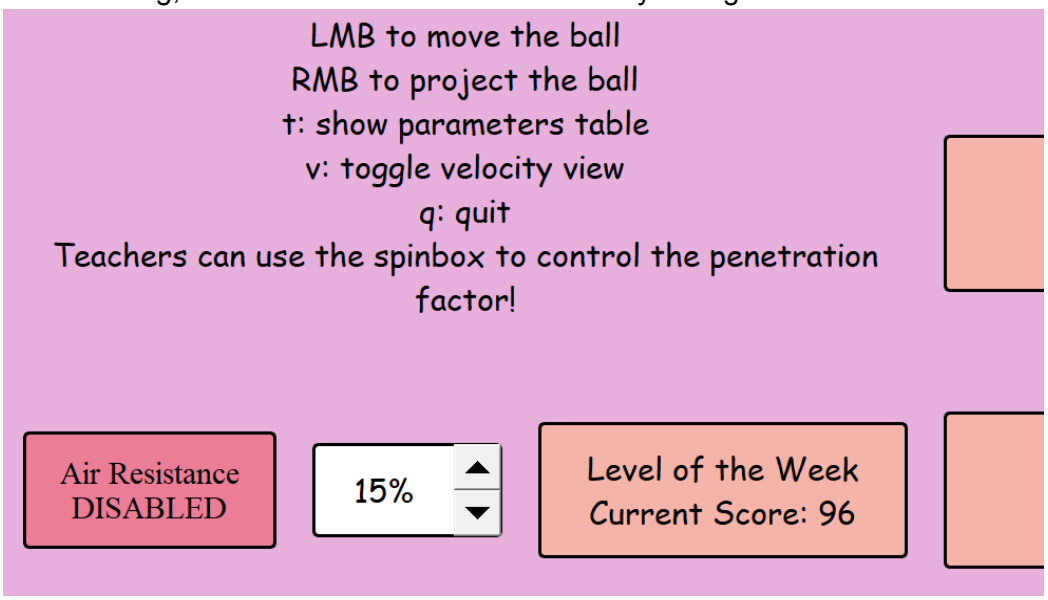
Full function (Has since changed, see testing):

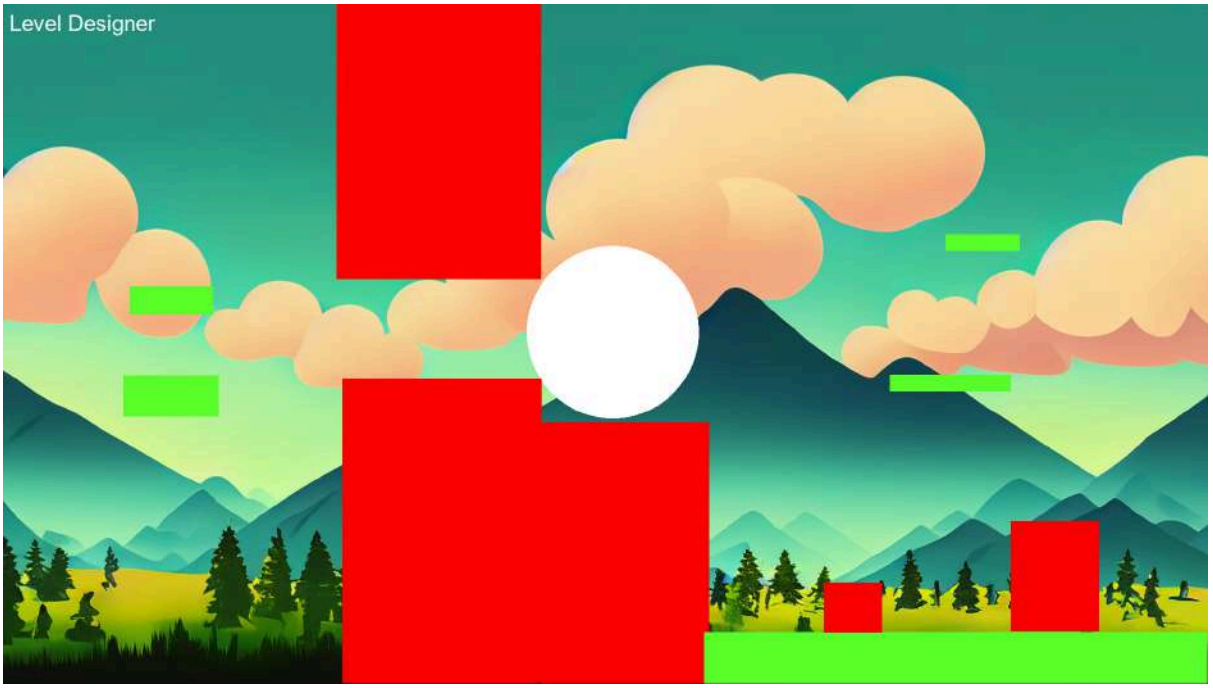
```
def create_new_db_user(self):
    valid = True
    detailed_text = ""
    if len(self.password.text()) < 6:
        detailed_text += "Password must be at least 6 characters.\n"
        valid = False
    if not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
self.email.text()):
        detailed_text += "Email is not valid.\n"
        valid = False
    if self.full_name.text() == "":
        detailed_text += "Full name is not given.\n"
        valid = False
    if self.date_of_birth.date().addYears(16) > QDate.currentDate():
        detailed_text += "You must be at least 16 years old.\n"
        valid = False
    if valid: # send to database to create user
        teacher =
self.database.get_teacher_email_by_name(self.teacher_dropdown.currentText())
        self.database.create_user(self.email.text(),
sha256(self.password.text().encode()).hexdigest(),
                                  self.full_name.text(),
self.date_of_birth.date().toString("yyyy-MM-dd"), teacher_email=teacher)
        self.toggle_login_register() # prompt user to login
        return True
    else:
        box = QMessageBox()
        box.setText("Could not create a new account\nPlease check you have entered
your information correctly.")
        box.setDetailedText(detailed_text)
        box.setIcon(QMessageBox.Icon.Warning)
        box.setWindowTitle("Registration Failed")
        box.exec()
        return False
```

Access Rights

To successfully align with my goal, it is important for the teachers to have a good understanding of the program, so that they might then incorporate it into the class's learning. As such, I have given teachers extra privileges.

1. With the projectile motion mode, students have to complete the preceding level to access a new level. Teachers however have immediate access to all levels so that they may quickly navigate to a level. One use case may be if a student asks for help; the teacher can quickly play the level on their own account to help, rather than the teacher having to use the student's device/account.
2. Again with projectile motion, teachers have a level designer feature, where they may edit the levels for all other students. Although I have enabled this for all levels for the time being, the intention is that the teacher only changes the 'level of the week





3. With the projectile motion weekly leaderboard, students are assigned to a dataset consisting of only their class, so other classes are not included in the leaderboard. When the teacher of the class uses the system, they will still be able to view the class leaderboard. Their own personal score will not be added to the public leaderboard rankings and instead their score will only be visible to themselves

1	2	3
1	Sam Smith	98
2	student	0
3	student	0
4	student	0
5	student	0
6	student	0
7	student	0
8	student	0
9	student	0
10	asdffsaddfasdf	0
-	Mr Stevens	475

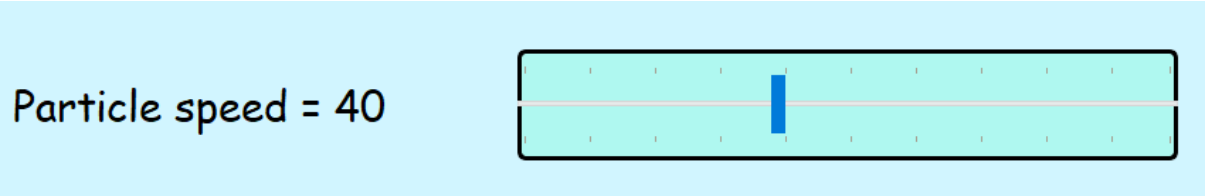
Simulations Validation

Vector Field Pathfinder

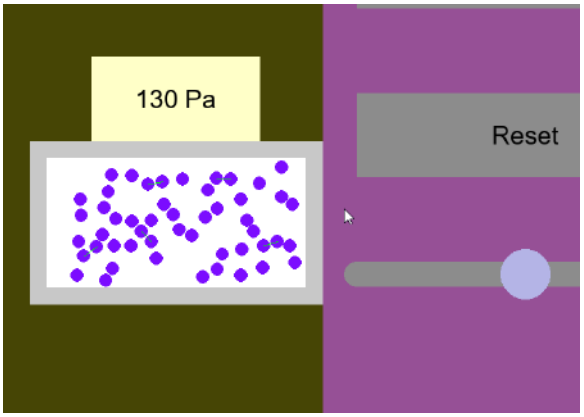
I have left it up to the user to decide the number of rows and columns in the cells. While technically not ideal, I have decided against ensuring that each cell must be square as I found the program gave me equally desirable results with rectangular cells. I found that the grid dimensions had a great effect on performance. As such, the rows and columns have been capped to 100. I have also set the minimum to 4 as to ensure that the program will still function as intended.

```
self.pathfinding_rows = QSpinBox() # no of rows in simulation
self.pathfinding_rows.setRange(4, 50)
self.pathfinding_layout.addWidget(self.pathfinding_rows, 0, 0, 1, 1)
self.pathfinding_rows.setValue(self.user_settings[2]) # adding saved initial value

self.pathfinding_cols = QSpinBox() # no of cols
self.pathfinding_cols.setRange(4, 50)
self.pathfinding_layout.addWidget(self.pathfinding_cols, 0, 2, 1, 1)
self.pathfinding_cols.setValue(self.user_settings[3])
```

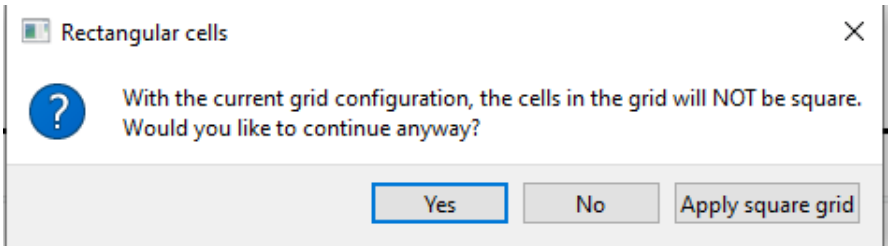
I have also set a range on the magnitude of the desired velocity with the steering behaviours. For ease of understanding, I present this as speed to the user. I have a slider widget for the user to select their desired speed.



In my vector field pathfinder, the user also has a range of inputs which dictate the flow of the system. In the main menu, I have two spinbox widgets which dictate the number of rows and columns in the grid. While technically the cells should all be square, user feedback showed that this didn't necessarily affect the user experience. In fact, some users claimed that they preferred to have an equal number of rows and columns. Therefore, I did not restrict the ratio of rows to columns, though I did have a 32 by 18 grid set as default. The user can change the values of the spin box and the values determine the dimensions of the grid. Both parameters have a set range as my analysis showed that too many cells would be unnecessarily inefficient. There is also a slider widget for a particle speed, which is actually the magnitude of the desired velocity. This is used in calculating the steering force, such that a greater magnitude of desired velocity means a greater force on the particle. As a rough estimate, I limited the range of this value from 1 to 100. My justification for this is that 50 was the ideal speed from my testing. I then went a bit below and a bit above to accommodate for the variance in other people's opinions.




```
self.pathfinding_rows = QSpinBox()
self.pathfinding_rows.setRange(4, 50)
self.pathfinding_layout.addWidget(self.pathfinding_rows, 0, 0, 1, 1)
self.pathfinding_rows.setValue(self.user_settings[2])

self.pathfinding_cols = QSpinBox()
self.pathfinding_cols.setRange(4, 50)
self.pathfinding_layout.addWidget(self.pathfinding_cols, 0, 2, 1, 1)
self.pathfinding_cols.setValue(self.user_settings[3])

self.pathfinding_speed = QSlider()
self.pathfinding_speed.setOrientation(Qt.Orientation.Horizontal)
self.pathfinding_speed.setRange(100, 1100)
self.pathfinding_speed.setValue(self.user_settings[5])
self.pathfinding_speed.setTickPosition(QSlider.TickPosition.TicksBothSides)
self.pathfinding_layout.addWidget(self.pathfinding_speed, 2, 1, 1, 3)
```



If the user does try to use an odd grid arrangement, they will be met with a message box warning them of the potential issue. The user has the option to continue anyway, to cancel, or to apply the necessary grid size for the cells to be square

```
def run_pathfinder(self):
    row = self.pathfinding_rows.value()
    col = self.pathfinding_cols.value()
    if row / col != self.height() / self.width():  # If cells are
rectangular
        warning_box = QMessageBox()
        warning_box.setWindowTitle("Rectangular cells")
        warning_box.setIcon(QMessageBox.Icon.Question)
        warning_box.addButton(QPushButton("Apply square grid"),
QMessageBox.ButtonRole.ApplyRole)
        warning_box.setStandardButtons(QMessageBox.StandardButton.No |
QMessageBox.StandardButton.Yes)
        warning_box.setText(
            "With the current grid configuration, the cells in the grid
will NOT be square.\nWould you like to continue anyway?")
        user_input = warning_box.exec()  # Either Yes for continue, No, or
No and implement square grid
        if user_input == QMessageBox.StandardButton.No:  # Don't run sim
            return
        if user_input != QMessageBox.StandardButton.Yes:  # Apply square
grid and don't run sim
            self.correct_grid_ratio()
            return
    speed = self.pathfinding_speed.value() * 20
    pathfinderSimulation.run(row, col, speed)
```

The algorithm to make the cells square is the correct_grid_ratio

```
def correct_grid_ratio(self):
    width, height = self.width(), self.height()
    gcd = np.gcd(width, height) # greatest common denominator
    cols, rows = width // gcd, height // gcd
    while rows < 15 or cols < 15: # i.e. if the grid is too small as it is
        rows *= 2
        cols *= 2
    if rows > 100 or cols > 100: # If the screen size is such that the
smallest grid size is impractical
        rows, cols = 18, 32  # Simply apply a default grid size of 18x32
    self.pathfinding_rows.setValue(rows)
    self.pathfinding_cols.setValue(cols)
```

Projectile Motion Simulation

The projectile motion simulation is largely straightforward. The main property is the level system. Users only have access to a level if they achieved the necessary score in the previous level. Otherwise, the cursor would switch to a forbidden cursor and the colour of the button would change to signify the button is inactive.
For the ideal gas simulation, I have had to implement many restrictions to ensure that a user input can not cause the program into some unexpected behaviours.

1. The changing of the container dimensions in particular has several nuances which I had not initially anticipated. I prevented the user from reducing the widths and heights of the container to less than 20 pixels. This ensured that balls would always remain within the confines of the wall as intended, rather than teleporting between two positions every time step. It also ensured that the wall was always selectable, as before the walls wouldn't be if the user had set the right wall to be further left than the left wall.
2. I had to ensure that the simulation region does not cross into the menu region. The right section is designated to the widgets. Initially, the user was able to drag the right wall over the menu, thus stopping the user from accessing the variables. Therefore, I ensured that the right wall would remain in its assigned region regardless of user inputs. In the image, I am attempting to drag the wall into the menu region, but the wall remains in its designated space. The user also has a slider which changes the temperature. As expected with a real gas, there is no real limit to the maximum temperature of the container. However, the temperature must be positive, as 0 kelvin is absolute zero. I placed a minimum value of 1 K on the temperature. At this speed, there would still be some small particle speed in the container; this is intended as if I were to go to 0 K, the particles should be completely still. However, with my implementation of reducing the particle speed, some inconsistencies might occur. As such, I decided to limit the minimum temperature to 1K

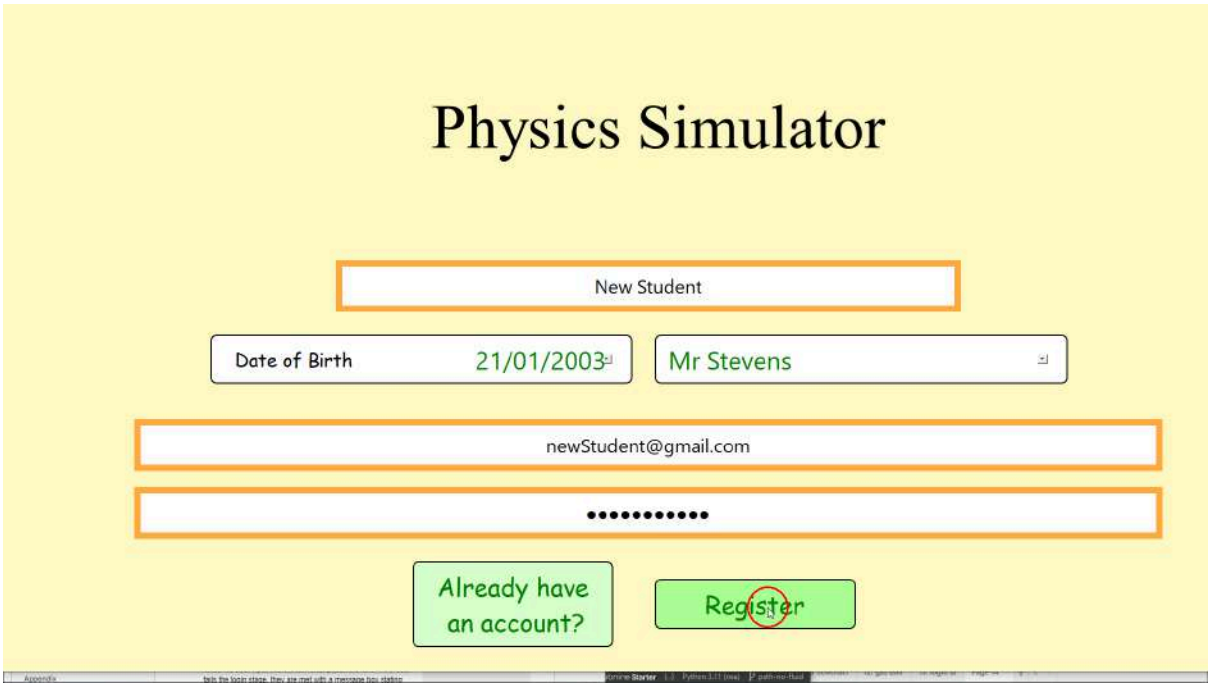
```
def update(self):
   if self.is_clicked:
       if self.slider:
           self.knob[0] = max(min(pygame.mouse.get_pos()[0],
self.knob_rest_pos[0] + 0.5 * self.size[0]), self.position[0])
           self.knob_value += 0.005 * (self.knob_rest_pos[0] - self.knob[0])
           self.knob_value = max(1, self.knob_value)# > mustn't reach 0 K
           self.parent.temperature_change(self.knob_value)
```

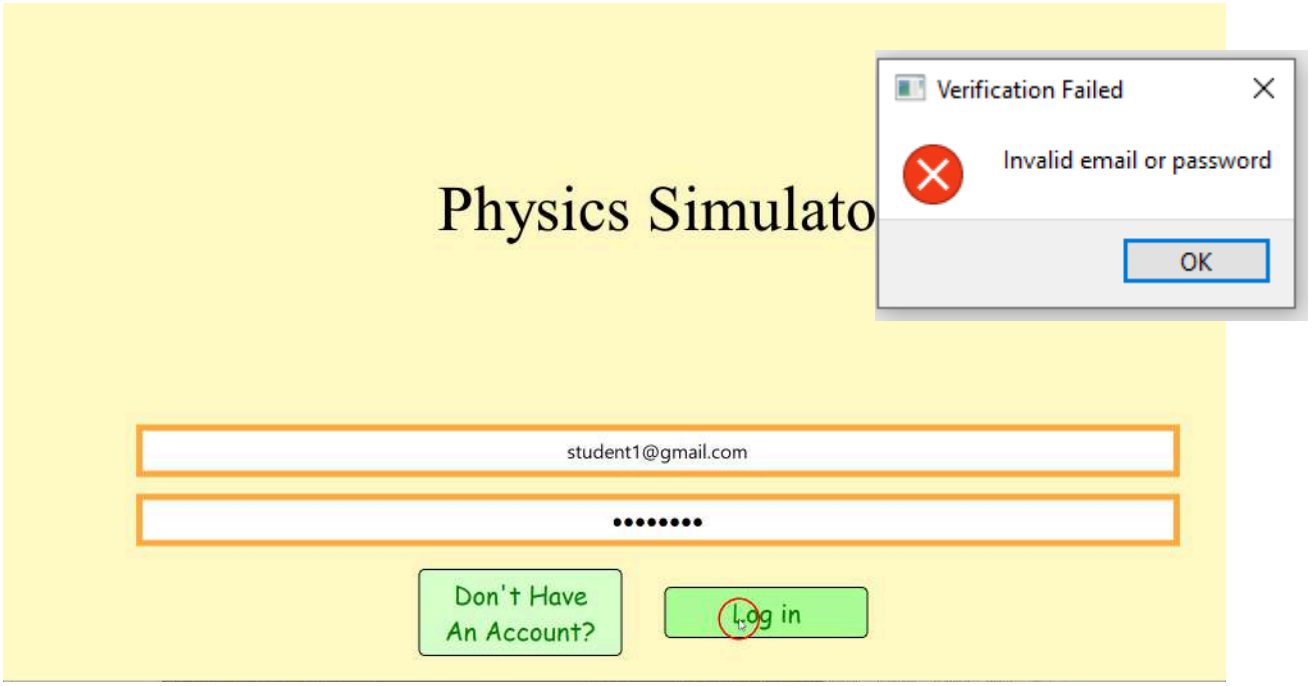
.

UI Explanation

In the Current System Research chapter where I reviewed the University of Texas's ideal gas law simulation, I was very critical of the clunky interface overloading the user. One of my objectives was to deliver a simple and straightforward UI. The below

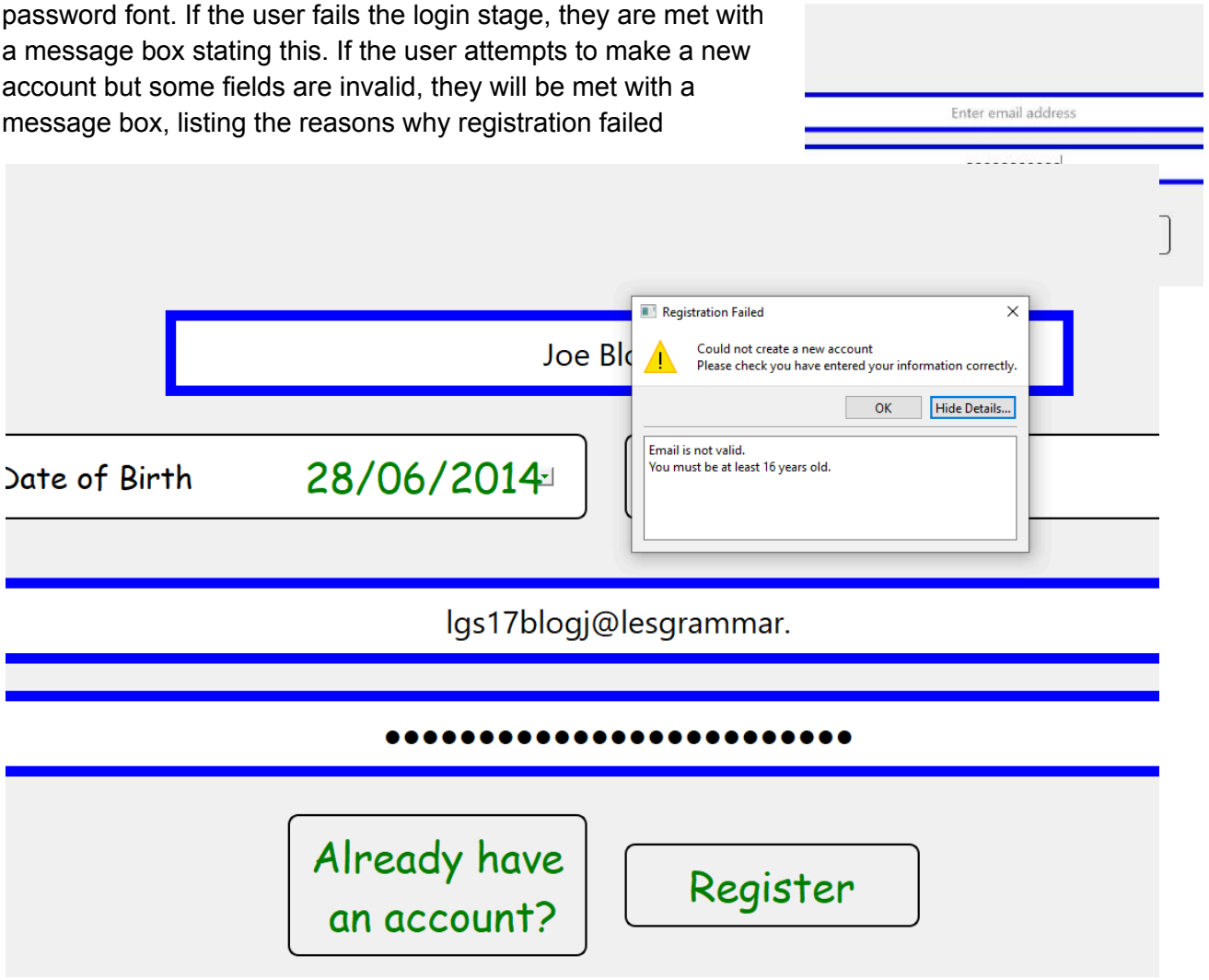
Login Screen

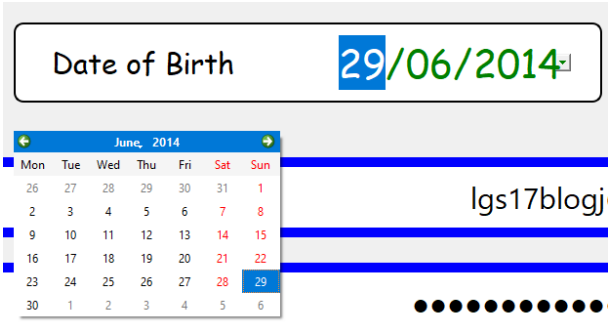




When the user first runs the program, they are met with a login system. At the centre of the page, there are two text fields: one for the email address and one for the password. When the user selects that they do not have a registered account, more text fields appear as required for the database. The uppermost text field asks for your full name, which will be used for teachers to identify the student. The next is the date of birth field with a calendar popup option. There is a dropdown menu for the user to select which class he belongs to.

The password field hides the user input into the universal password font. If the user fails the login stage, they are met with a message box stating this. If the user attempts to make a new account but some fields are invalid, they will be met with a message box, listing the reasons why registration failed



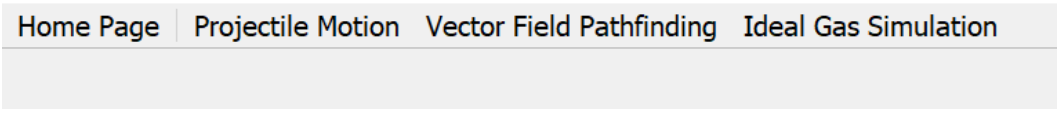


This login system is isolated from the rest of the program so that the user's full focus is on remembering their login details. Students, both from personal and my peers' experiences, easily fall prey to distractions or overloading. My login UI aims to be minimalist with high contrast to limit these problems. My date of birth has a calendar popup for ease of access to those who prefer a more graphical input option. Each text field has a placeholder text which indicates the desired input. The password text field has an input mask which protects the user from shouldering, for example. A student with bad intent might see the user's password and abuse their privacy, or the user might leave their password as an input and leave the desktop. In the latter case, the perpetrator would be unable to access a plaintext password. The dropdown menu allows the user to quickly select his/her teacher. As you can see from the above image, the program gives a suitable response if the user fails registration, detailing exactly why they were unable to make an account upon selecting the 'More Details' option.

```
def create_new_db_user(self):
    valid = True
    detailed_text = ""
    try:
        if len(self.password.text()) < 6:
            detailed_text += "Password must be at least 6 characters.\n"
            valid = False
        if self.email.text() == "":
            detailed_text += "No email given.\n"
            valid = False

        elif not
re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
self.email.text()):
            detailed_text += "Email is not valid.\n"
            valid = False

        if len(self.email.text()) > 256:
            detailed_text += "Email is too long.\n"
            valid = False

        if self.full_name.text() == "":
            detailed_text += "Full name is not given.\n"
            valid = False
        if self.date_of_birth.date().addYears(16) > QDate.currentDate():
            detailed_text += "You must be at least 16 years old.\n"
            valid = False

        assert valid
        # send to database to create user
        teacher =
self.database.get_teacher_email_by_name(self.teacher_dropdown.currentText()
)

         # etc.
```

Home Screen

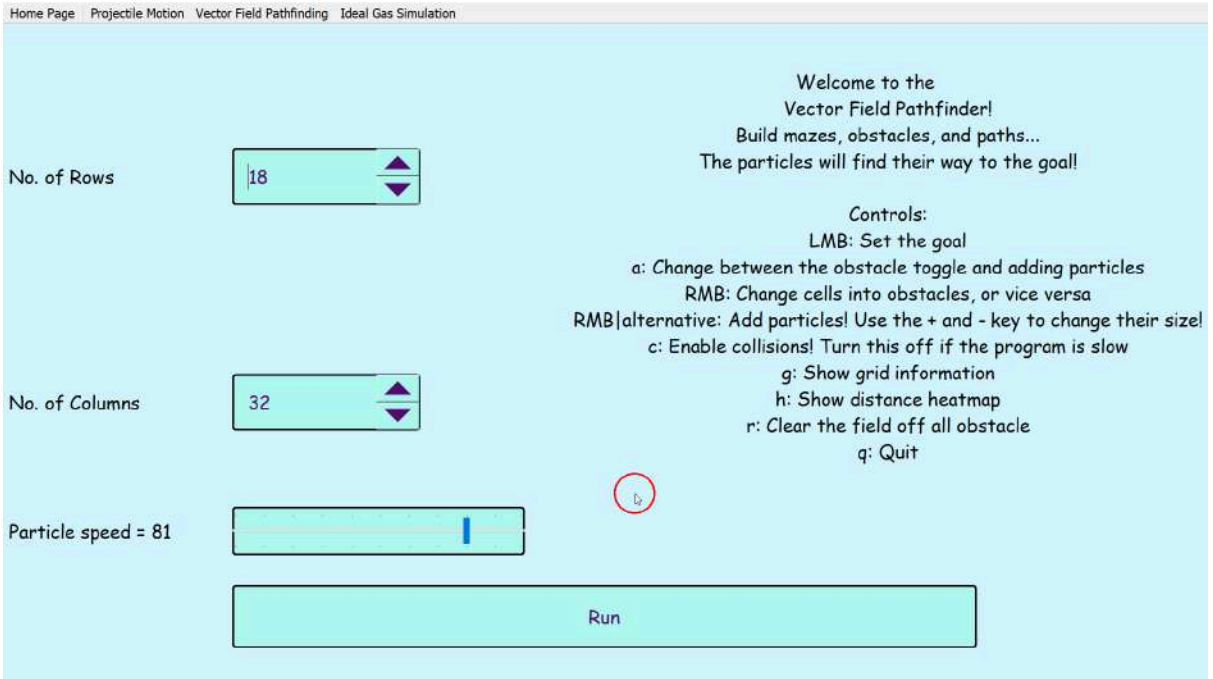
When the user successfully completes the login stage, the screen switches to a home page.



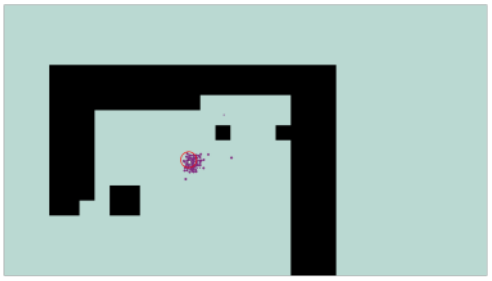
A toolbar also becomes accessible on the top of the screen. The first button directs to the home page itself. Separated by a separator are buttons for the simulations. The home page gives an option to sign out and exit the program.

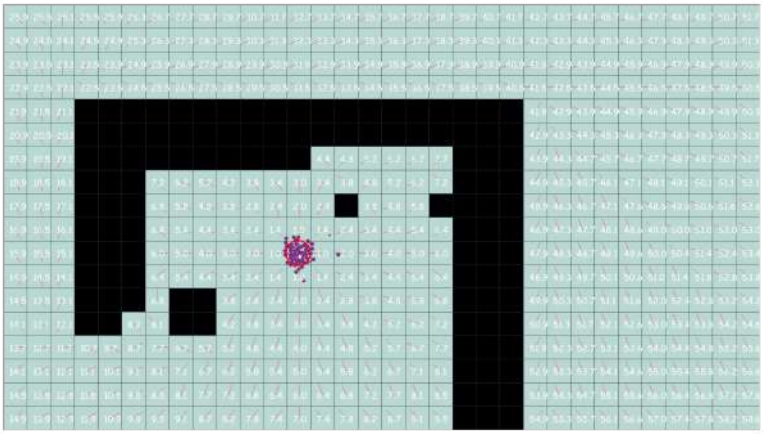
I had many considerations in how I would make the user navigate the program to their desired simulation. As you can see from the sketch on the right, I initially planned to have each simulation section designated its own quadrant. In the centre, I would have a variety of options. For example, I could have a leaderboard based on the students in their class, or a welcome message for the user. The user would sign out from a button on the top right. However, after much consideration, I decided against this. Firstly, I believed that with the projectile motion simulation for example, the allocated quadrant would not be sufficient for all the choices that I want. Moreover, I was fearful that some of the features would have limited visibility amongst the rest of the program. As such, I opted for a toolbar approach instead. Each simulation is given its own space clear of any clutter, once again in an aim to prevent overloading.

Vector Field Pathfinding

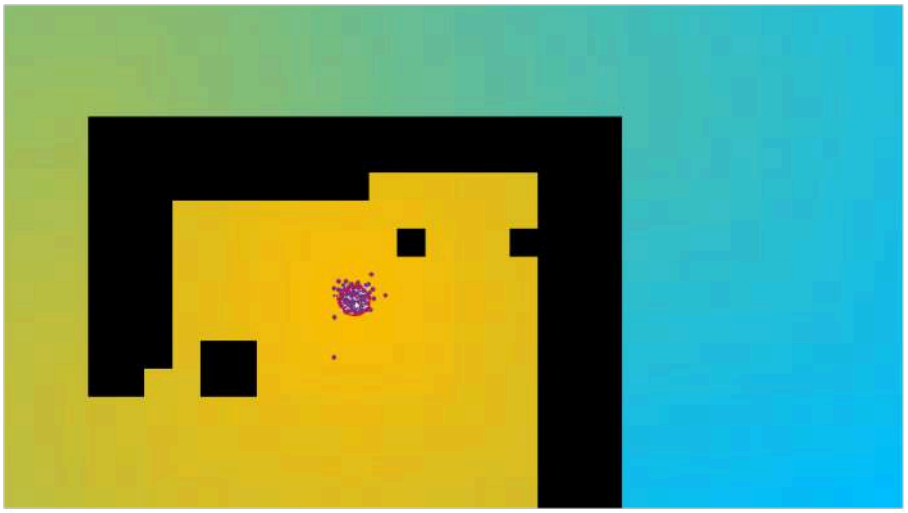


When the user first runs the program, they are met with a randomly generated vector field. This simulates a set of directionless particles. Once the user sets a goal, the particles will find a path to the goal. The user can toggle between showing a heatmap. The user also has the option to view the metadata, such as the distance field and the velocity vector of the cell. The user can add or remove obstacles which are shown as black, and the user can add particles of different radii.





The right image shows the simulation with the grid enabled. This provides the direction vector and distance of each cell

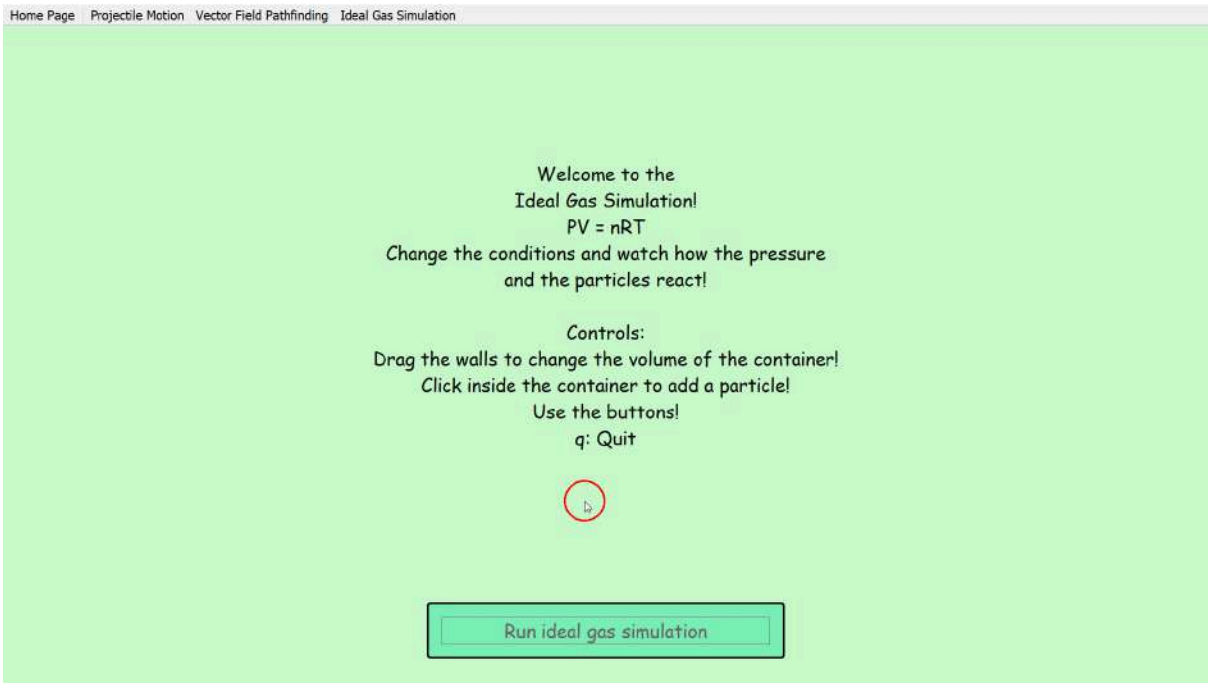


Here is the same program, this time with the distance heatmap enabled. Regions which are from the goal are shown in blue, and closer cells are shown in an increasingly yellow colour.

You can also combine both view modes and view the grid information on top of the distance heatmap.

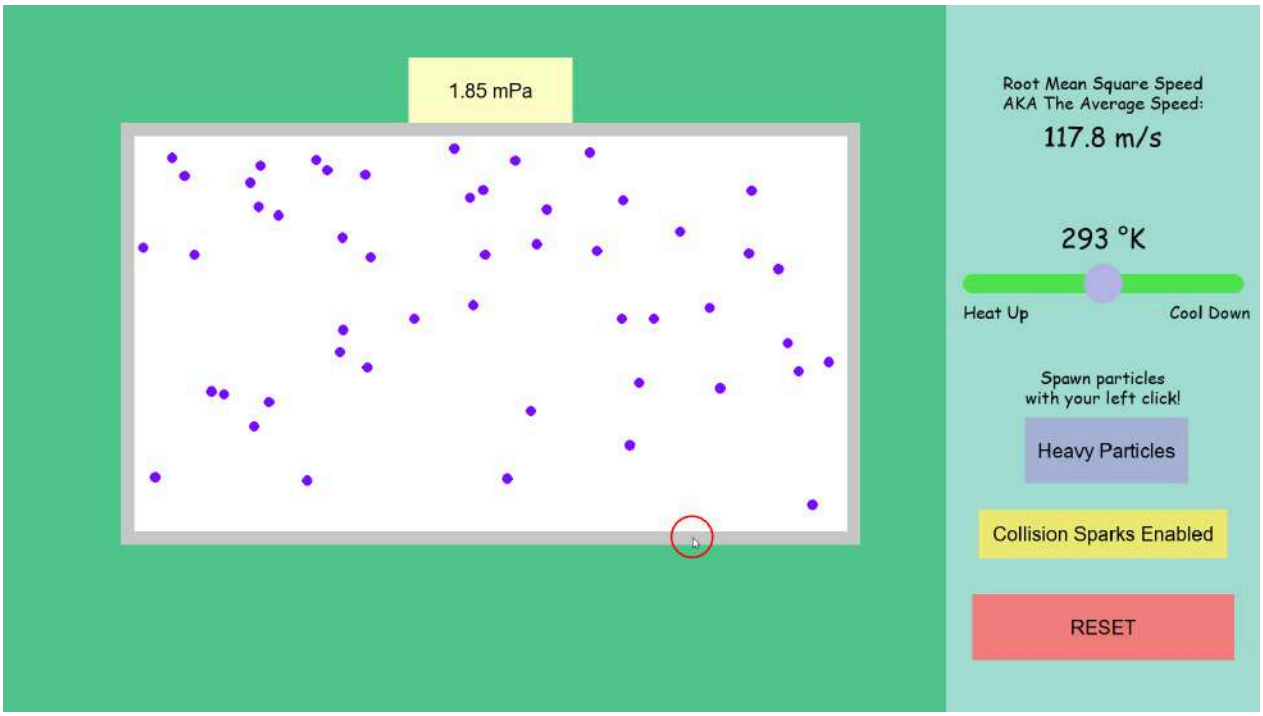
For this simulation, I decided against using widgets and instead opted for the user to dictate the simulation properties by mouse clicks. The two primary inputs which impact the simulation are the left and mouse buttons. Each mouse click has a strong effect on the system rather than a mere behind the scenes impact. For example, the right click will always toggle an obstacle or add a particle and the left click will change the cell of the goal. This non-widget based approach is more suitable for this type of simulation in my opinion as the user could actively see the particles making their way to the current mouse position. This leads to a more engaging experience. User testing has shown that the grid layout has been vital to the user experience. Without it, the functionality of the simulation can be obscured. Seeing as this is a learning tool, the user has the option to show the divisions between cells and observe the velocity vectors of each cell.

Ideal Gas Law Interface



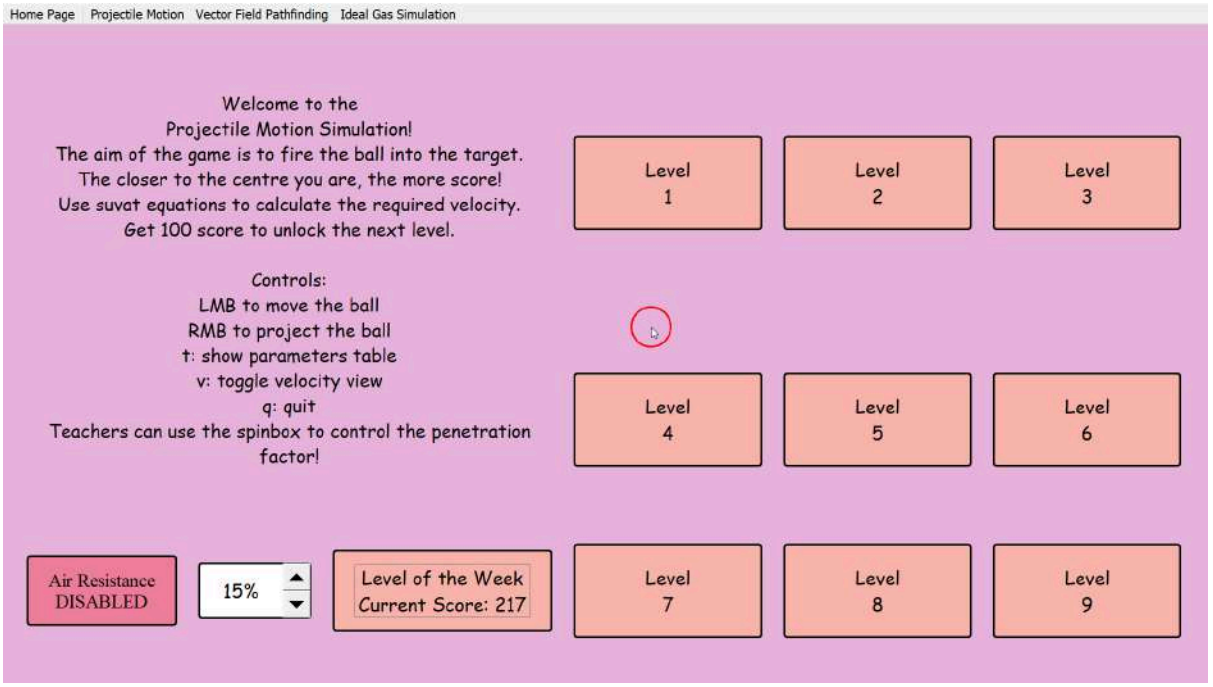
This key aspect of this simulation is the container. The user can control the dimensions of the container by selecting one of the walls and dragging it to their desired position. At the top of the container is a display for the pressure. This updates dynamically to ensure that the pressure value is consistent with any of the user's changes.

The user has a menu bar on the right. There contains some key information about the container such as the temperature or the root mean square speed. There are also options to add light or heavy particles, change the temperature using a slider, etc. If the user decides that they would like to reset the particles and the temperature, they can do so with the reset button, which returns the container to its original state except for the walls..



The ideal gas law revolves heavily around a set of parameters. As such, I wanted to take a more widget-based approach. This way, the user can perform their own experiments by adjusting certain parameters and observing certain characteristics. For example, they could notice how the temperature is directly proportional to the pressure. As with the previous simulations, I have chosen a vibrant colour scheme that is loosely related to the colour scheme of the simulation page on the PyQt6 window

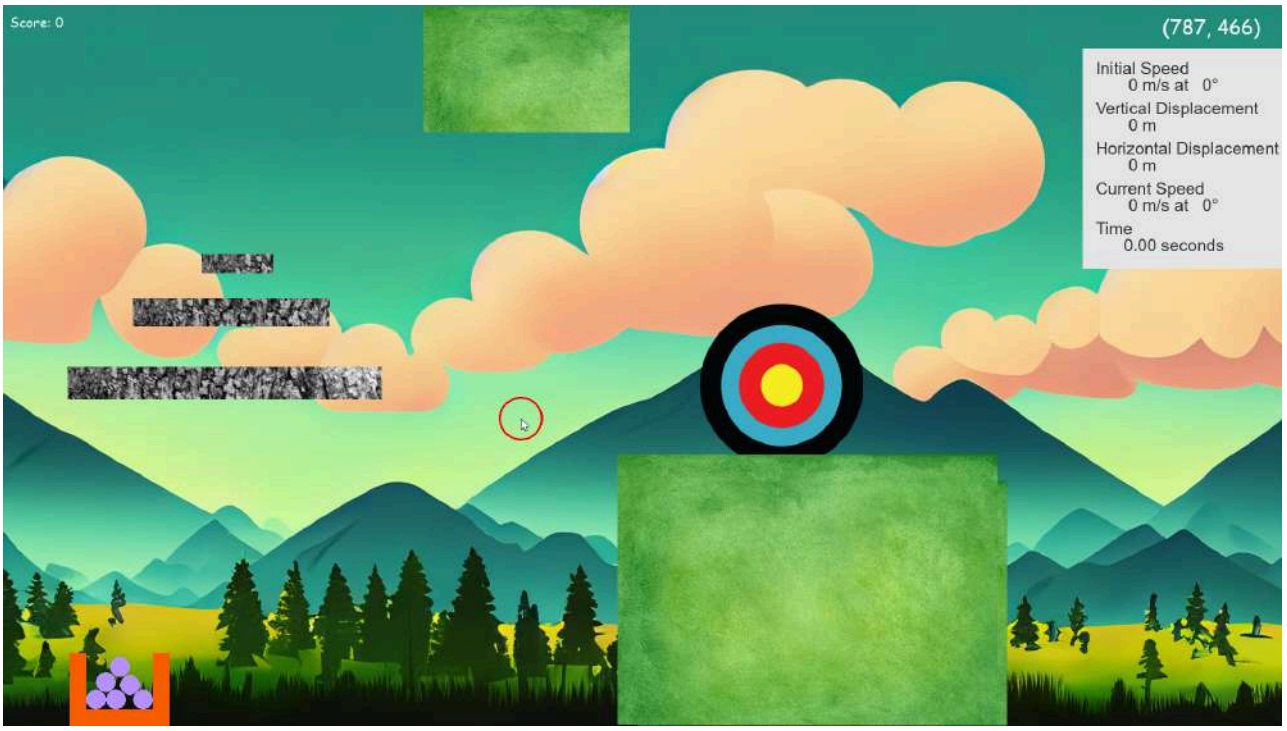
Projectile Motion Interface

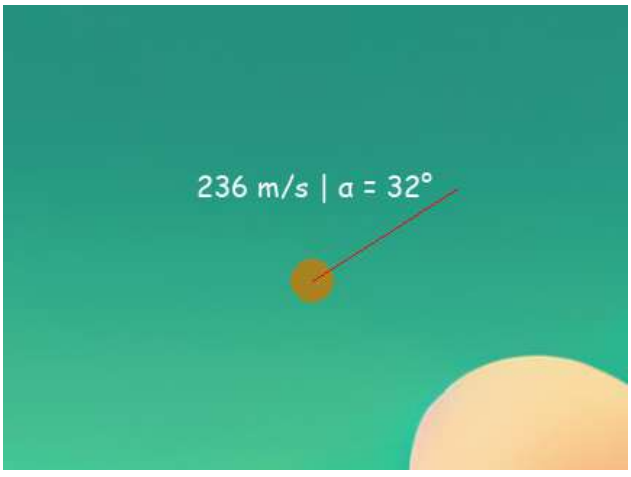


In the PyQt6 window, the user is met with 9 levels. Users with the given access rights can create levels using the level designer. In my case, this will be any teacher. This creates text files which can be read by the program to produce a level.

```
1047,617,138,138
1518,763,165,183,1
1124,938,391,138,1
786,370,108,303,1
1051,287,573,42,0
1383,425,264,109,0
770,609,394,146,0
674,795,64,171,0
```

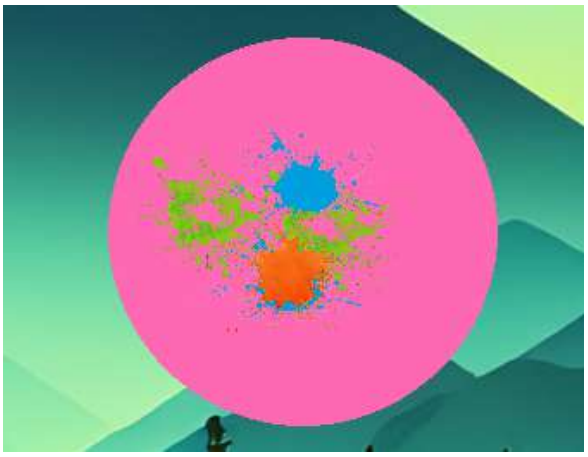
The user will only be able to complete a level if they have completed the one prior. Their progress is saved so that they do not have to restart at each login. For example, when I try to select level 4, the button turns green to indicate I can access this level. However, if I try to access level 5, the button turns orange to indicate otherwise and the cursor changes to a forbidden symbol. When the user selects level 5, no response which in turn tells the user they can not access this level.



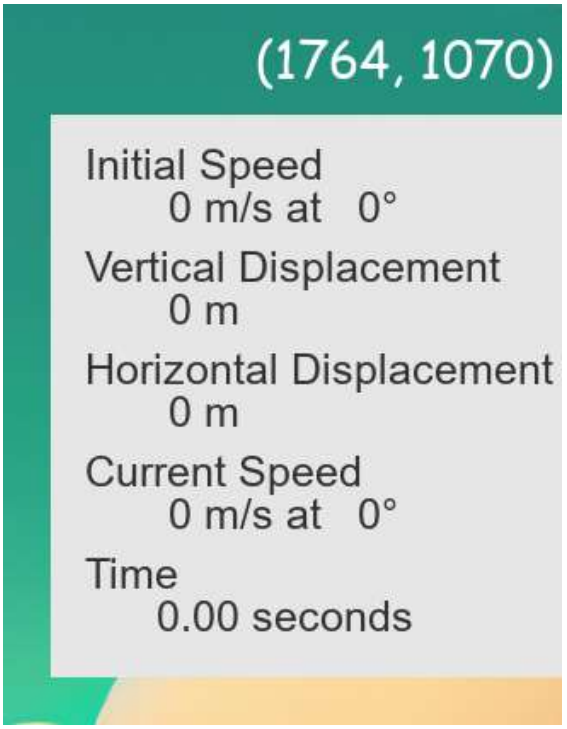


The user can toggle how they want to view the projected velocity. For example, in the left image, the program provides the speed and the angle of elevation. On the left, the program provides a velocity in vector form, using the unit vectors i and j.

When the user successfully fires a projectile into the target, the ball will transform into a splat image.



The user can drag the ball around, for example onto one of the platforms, using the left mouse button.
The user's current score is stored on the top left of the screen.
The top right shows the cartesian coordinates of the current mouse position.



A table detailing the properties of the currently selected particle is given, which gives essential values like current velocity or the time until the first collision

Technical Solution

Additional Explanation for Code (See Appendix)

baseClasses.py

This is a file containing the fundamental building blocks for all the simulations and is imported into all the other simulation files.

6	Class Particle. All simulations will inherit this class and build upon it. It holds the majority of the general particle physics properties, like mass, velocity, position, etc. This is a fundamental object and forms the basis of every simulation
22	__init__: Insert the particle into the spatial map if it exists
24 31 77	Collision_event_obstacles, resolve_obstacle_collision, check_obstacle_collision These are the main obstacle handling functions. The collision_event_obstacles loops through all particles, checking for collisions and then resolving if that returns True
88 93 104	Collision_event, is_collision_resolve_static_collision Pretty much the exact same concept as the obstacle collision handler, but with particle collisions. Collision_event is the main function. It loops through each particle and cheks for a collision, at which point it will resolve their position if necessary
117	Resolve_dynamic_collision. This is an extension of the particle collision system above. These particles ensure the conservation of momentum when two particles collide. In real life, when two balls hit, they will bounce off each other. The resolve_static_collision undoes the overlap in the direction of the contact. This function
137	Update: This is responsible for updating the particle properties. It will apply the acceleration, handle boundary conditions, and deal with the spatial map interaction (if necessary)
162	Apply_air_resistance: This calculate the drag force through a formula and applies it in resistance to the direction of velocity
173	Class Cell: This is a little object used for the spatial map, which has a particle list and a velocity attribute
179	class SpatialMap: This is the spatial map or a general container, whichever the program needs. All particles are generally bound to this object. This class holds many of the general functions
196	Get_grid_coords: This is a helper function which just return all the available coordinates in the grid
209 215	Hash_position, undo_hash_position: The first hashes a pygame coordinate of a particle into the corresponding cell in the spatial map, and the second does the inverse.
218 221	Coord_to_index, index_to_coord: As it sounds, it just converts between the two

232 235 241	Get_magnitude, get_square_magnitude: Gets the magnitudes of a given vector. Note the @staticmethods decorator here to indicate it as such. Also normalise_vector, which does as it sounds
247 267 276	get_neighbouring_(thing): These functions help find various things. The first is the fundamental one. This gets the neighbouring coordinates. Lines 267+ access the cell_lists from those coordinates, and lines 276+ collects the particles within those cell_lists, so they link together well.
289 305	Remove and insert particles into the spatial map by checking the positions. Typically run every time step.
285 END	All the remaining functions deal with the mouse's interaction with the particle in question. Although these are currently only used in the projectile motion simulation, I still regard it as a fundamental function and can be applied to a variety of projects

pathfinderSimulation.py

2	Import the baseClasses
5	This function is the main one. This is called when you are trying to run the simulation
21	The event loop
23-60	This handles all user inputs such as adding obstacles or showing the grid.
64-96	This section of code is responsible for updating the vector field and particle objects, and also displaying things to the screen.
102	Class Pathfinder. Uses polymorphism (inherits the Particle class)
107-135	The same concepts as the collision functions in the base class. However, here I use an alternative collision detection algorithm which is much more efficient. As such, these functions override the parent methods.
138	Class VelocityField: This inherits the SpatialMap class
154-168	These functions are responsible for displaying the heatmap. To display the heatmap, we need the maximum distance in the field (to normalise off of). This is costly, so the latter function is used to only calculate it once.
171	Print_visited: A function used for testing when the program was in development

17-7 191 194	These functions handle adding and removing obstacles from the grid, and adding particles to the grid.
200- 285	These functions are part of the pathfinding algorithm. They calculate the distance field and velocity field. The update_velocity_field wraps the functions together into one concise function
207- 315	These are behaviours regarding obstacles which are all now deprecated. The first two are steering behaviours that were inefficient, and the last was simply ineffective
317	The update for the object, which applies the velocity field onto the particles using a steering behaviour

projectileMotionSimulation.py

3	Import the baseClasses
6	The level designer function is called draw_mode. This simply tracks the mouse position and creates circles and rectangles, which correspond to a target and obstacles. It then writes to a file to save it.
100	The main function to run an actual level
119	Event loop
127- 148	This section is responsible for updating the simulation every time step. It handles anything that needs updating, i.e. the particles, the obstacles, and the kinematics table.
157	This block of code is responsible for drawing the line and velocity when the user is preparing to fire a ball
178	This section handles all the user inputs, such as firing and moving the particles
219	Class ProjectileParticle. This inherits the Particle class
233- 254	These are additional collision handling functions, this time for the target. The collision_event_goal slows the particle until it stops, at which point awards points. The entirely_in_obstacle_check is used to check if the particle is inside of the target
256	Update: overriding the parent method to produce a more specific function
275	Class Container: This inherits the SpatialMap object from baseClasses
295	Here, I am initialising the variables which will be used for the kinematics table.
313- 350	These functions are responsible for the kinematics table. The first two are used to keep track of time, that is to start and stop. The last two update the variables each time step, and output the table onto the screen

352 355	These calculate the points scored, and adds them to the user's score
366 413	These initialise the level and goal respectively. The main purpose is to parse a level file's contents into a set of obstacles and a target.
420	Class Obstacle: A simple object I made to streamline the process. These hold generic attributes like position and width

idealGasLawSimulation.py

5	The function which runs the simulation. I feel the comments here explain the code in enough detail
35	Note here I am now doing the resolve_dynamic_collision function, to ensure that all collisions are elastic
85	Class GasParticle: inherits the Particle object from baseClasses.
91	Update: overrides the parent class. Here, it calls the parent method with super().update() but adds its own specific parameters.
95	Class Widget: A helper class I made to make the input/output aspect easier to develop. As the name suggests, this class is responsible for the widgets in the simulation
171	Class Container: inherits the SpatialMap from baseClasses.py
173	A key addition to this class is the handling of widgets. These widgets are used to interact with the container and change various parameters
205	Initialise_container: Executed in the class initialisation and also whenever the user resets the container. This applies the default or initial values to the container
218 224	These function calculate the pressure and the rms velocity respectively, which are the primary outputs of the simulation
256 273	These are responsible for the drawing side of the program, that is drawing the widgets and container walls.
293- END	These combine to enable the user to change the dimensions of the container's walls. This includes updating the dimensions arrays, tracking the selected wall, and validation to ensure the desired dimensions do not break the system.

fluidFlowSimulation.py

Please note again that this has not been included in the final program as it was unfinished.

2	Import the baseClasses
6	The main function to run the simulation
8+	Initialising the spatial_map, and then initialising FluidParticle objects to add to the spatial map
26	This block of code draws the background and the grid lines
36	This contains the simulation logic, such as updating the densities, pressures, and general properties
46	Class FluidParticle: inherits from class Particle
56	__init__: Here I am initialising some variables that are to be updated and assigned a value
64+	Calculate_density: This does as it sounds. It is run every time step and is needed to start finding the pressure and viscosity.
83+	Various functions I created with the aim to calculate the pressure force.
110	A formula given by the paper for finding a given property of the particle
129	Class Smoothing Kernel: An object used to calculate the contributions of neighbouring particles. Please refer to the Smoothing Kernel chapter in the Key Algorithms chapter
152	Using the booleans given in the initialisation stage, it returns the influence of a particle using the formula specified in the corresponding function.
208	Class FluidSpatialMap: Inherits the SpatialMap object from the base classes
215	__init__: Here, I am creating two SmoothingKernel objects with different parameters. These represent different kernels which are used to calculate pressure influence and viscosity influence.
218	A crude way in which I sought to calculate the initial average density at the start of the simulation, where particles were randomly distributed

database.py

This is a file which connects to a database server and interacts with it using SQL queries. It is used by main.py

4	Class Database
5	__init__: Here, I establish a connection with the server with the passed parameters
16	This is used to reset the database, maybe to clear out all students or to implement a change in structure. Within, I drop all tables, and then create new ones along with some placeholder users
77	This is used to create a user.

102-155	These are various functions required by main.py. They all fetch and return data to the main file.
157	This verifies a user login attempt by fetching any records with the given email and password hash. If the fetched results come to nothing, an AssertionError is raised, at which point the function returns None. If it does find one however, then it will return the result. Note that only record will every be found as the email address is a unique field
172	This fetches the user's settings from the user_id. Every user has a user settings record
179	This is used to save an updated array into the user_settings table.

Coding Styles Table

Coding Style Techniques	
Subroutines with appropriate interfaces	
Loosely coupled subroutines - module code interacts with other parts of the program through its interface only	
Cohesive modules - module code does just one thing	
Modules - subroutines with common purpose grouped	See file structure. A specific example is database.py
Defensive programming.	main.py: create_new_db_user() idealGasLaw.py: change_wall_dimensions → Line 325
Good exception handling	projectileMotionSimulation.py: intialise_level → 394 main.py: create_new_db_user()
Well-designed user-interface	Refer to User Interface chapter.

Modularisation of code	Done throughout code. One specific example are the collision functions, eg: baseClasses.py: (Collision_event → 88 Is_collision → 93 Resolve_static_collision →104)
Good use of local variables	pathfinderSimulation.py: update → 317 pathfinderSimulation: calculate_vectors → 240 idealGasLawSimulation: calculate_pressure → 218
Minimal use of global variables	None used
Managed casting of types	See appendix
Use of constants	idealGasLawSimulation: __init__ → 176 (self.R = 8.3145) fluidFlowSimulation: __init__ → 142 (self.normalisation_constant = 315 / (64 * np.pi * self.h ** 2)
Appropriate of indentation	ProjectileMotionLevel + “level_no”
Self-documenting code	See appendix for full code
Consistent style throughout	See appendix for full code
File Paths parameterised	projectileMotionSimulation: __init__ → 307 (`if not self.initialise_level("./Simulations/SimulationFiles/Assets/ProjectileLevels/lvl" + str(level_no)):`)
Meaningful identifier names	See throughout
Annotation used effectively where required	See throughout

Technical Skills Table

SKILLS	
Complex data model	See Data Mapping chapter / database.py
Queue in breadth first algorithm	patnfinderSimulation.py: generate_heatmap → 216
Spatial map	pathfinderSimulation.py { Collision_event → 128, Update → 325}

	baseClasses: { get_neighbouring_particles → 276, Remove_particle → 285, Insert_particle → 289 }	
Complex mathematical functions	baseClasses: resolve_dynamic_collision → 117 baseClasses: apply_air_resistance → 162 fluidFlowSimulation.py: calculate_density → 70 (Navier-Stokes equations)	
navier-stokes equations	FluidFlowSimulation	
Matrix operations	baseClasses.py: resolve_dynamic_collision → 122, 126	
Object Oriented Programming	Inheritance	Used for every simulation eg pathfinderSimulation, Pathfinder class
	Polymorphism	idealGasSimulation: update → 92
	Overriding	pathfinderSimulation: collision_event_particles →113
Data structures	See Data Structures chapter	
File Handling	projectileMotionSimulation.py: draw_mode → 53	
Files organised for sequential access	See ProjectileMotionLevels folder	
Records	See Data Mapping chapter	
Data Types	idealGasLaw: __init__ → 97 `self.position = np.array(position, dtype=float)`	

Steering Behaviours	`pathfinderSimulation.py:`
	`Calculate_avoidance_force → 287`
	`Calculate_collision_avoidance → 299`
	Update → 326	

Decorators	baseClasses.py with @staticmethod:
	`get_square_magnitude`
	`get_magnitude`	

Hashing	main.py: `user_created =`
	`self.database.create_user(self.email.text(),`
	
	`sha256(self.password.text().encode()).hexdigest(),`
	
	`self.full_name.text(),`	

	`self.date_of_birth.date().toString("yyyy-MM-dd"),` `teacher_email=teacher)`

Testing

Small Test Table Per Objective

- Screenshot evidence of tests
- Normal, boundary & erroneous data where possible
- Destructive tests where possible
- Show fixes for failed tests if possible

Category 1 - Main Window

1) Provide a user interface for the user

1.1) The UI should be simple and straightforward being careful to minimise overloading the user

1.2) The UI should be full screen to prevent the student becoming distracted by other programs

1.3) The program display correctly for all resolutions

1.4) The user should be able to create an account with the program.

1.5) The user should be able to log in to a previously made account

1.6) All user inputs such as email address should be fully validated

` 1.7) Any parameters the user might select through using the simulations should be stored for future use, and the program should pre-load these parameters the next time they l og in to the program

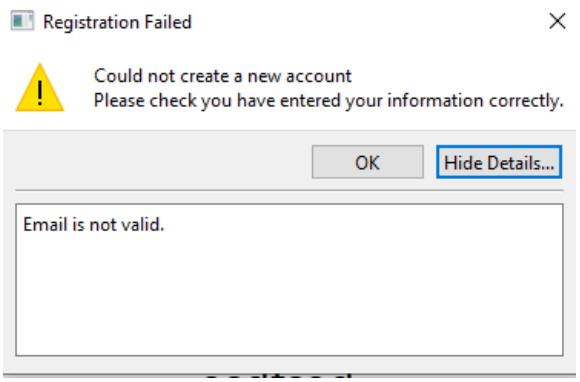
1.8) A teacher should be able to have extra access rights, such as view a student's progress within their class or edit levels
1.9) The user interface should provide engaging and helpful responses, for example i if the user fails an email login.

```
if len(self.password.text()) < 6:
   detailed_text += "Password must be at least 6 characters.\n"
   valid = False

if self.email.text() == "":
   detailed_text += "No email given.\n"
   valid = False

elif not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$", self.email.text()):
   detailed_text += "Email is not valid.\n"
   valid = False

if self.full_name.text() == "":
   detailed_text += "Full name is not given.\n"
   valid = False

if self.date_of_birth.date().addYears(16) > QDate.currentDate():
   detailed_text += "You must be at least 16 years old.\n"
   valid = False
```

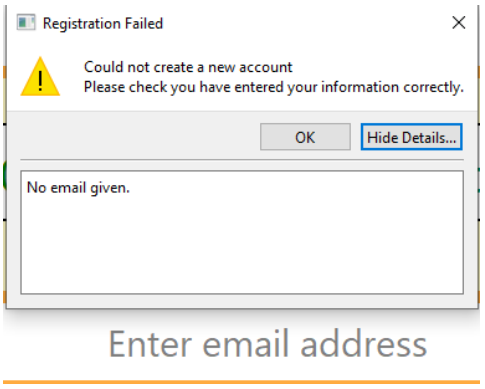
#	Test Type	Test Data	Reason	Outcome	Pass/Fail?
1	Valid	*Email* = a@b.c	Email meets valid format	Registered	Pass
2	Valid	*Email* = 203zZ@3.asdfrr	Email meets valid format	Registered	Pass
3	Invalid	*Email* = @sd.co.uk	No characters before @ symbol	Invalid	Pass
4	Invalid	*Email* = em@gmail.	No characters after final .	Invalid	Pass
5	Null	*Email* =	No email provided	Invalid	Pass
6	Boundary	*Email* = **400 char long**	Emails can't be this long and database cannot store it	Valid	Fail
7	Valid	*Password* = Pass34**	Password is at least 6 long	Valid	Pass
8	Boundary	*Password* = 123456	Password is at least 6 long	Valid	Pass
9	Invalid	*Password* = 1234	Password is not long enough	Invalid	Pass
10	Valid	*Password* = **400 chars long**	No cap on password length as using a hash.	Valid	Pass
11	Valid	*Full name* = a^^^^££	Few restrictions on name	Valid	Pass

12	Invalid	*Full name* =	Name must be given	Invalid	Pass
13	Valid	*DOB* = 01/01/1999	User is at least 16 years old	Valid	Pass
14	Invalid	*DOB* = 01/01/2010	User is younger than 16	Invalid	Pass
15	Boundary	*DOB* = 08/04/2008	User is exactly 16	Valid	Pass
16	Boundary	*DOB = 07/04/2008*	User is 1 day away from 16	Invalid	Pass

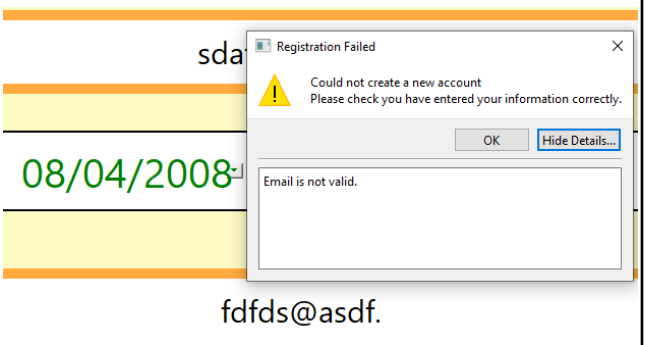
Tests 2 and 3:



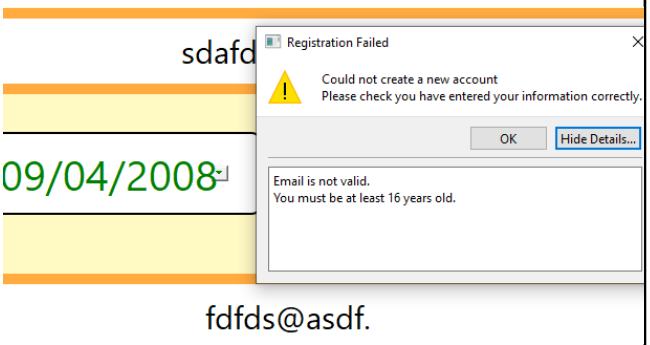
Test 5



Test 15



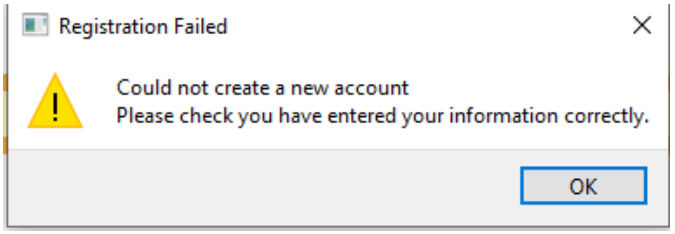
Test 16



Test 6

The testing here revealed some flaws with the email validation in particular. Currently, if the inputs get past the validation checks, it will automatically send the user to the login step. However, there can be some unforeseen errors. With test 6, it was just a case of adding another check to ensure the email is not too long. However, this does not deal with the underlying issue as there are no checks for server-side errors. I believe it to be impractical to validate everything on the program side as I will undoubtedly miss something. For example, the email might already have a registered account associated with it, which is impossible to validate without going to the database beforehand.

My fix was to simply attempt the account registration anyway, and catch any problems in a try except statement. This ensures that every user account will be valid. However, it does mean that if there is an error server-side, the program is unable to state why in the ‘More details’ section.




```
def create_new_db_user(self):
    valid = True
    detailed_text = ""
    try:
        if len(self.password.text()) < 6:
            detailed_text += "Password must be at least 6 characters.\n"
            valid = False
        if self.email.text() == "":
            detailed_text += "No email given.\n"
            valid = False

        elif not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
self.email.text()):
            detailed_text += "Email is not valid.\n"
            valid = False

        if len(self.email.text()) > 256:
            detailed_text += "Email is too long.\n"
            valid = False

        if self.full_name.text() == "":
            detailed_text += "Full name is not given.\n"
            valid = False
        if self.date_of_birth.date().addYears(16) > QDate.currentDate():
            detailed_text += "You must be at least 16 years old.\n"
            valid = False

        assert valid # send to database to create user
        teacher =
self.database.get_teacher_email_by_name(self.teacher_dropdown.currentText())
        user_created = self.database.create_user(self.email.text(),
sha256(self.password.text().encode()).hexdigest(),
                                 self.full_name.text(),
self.date_of_birth.date().toString("yyyy-MM-dd"), teacher_email=teacher)
        if not user_created:
            raise AssertionError
        self.toggle_login_register() # prompt user to login
        return True
    except AssertionError:
        box = QMessageBox()
        box.setText("Could not create a new account\nPlease check you have
entered your information correctly.")
        box.setDetailedText(detailed_text)
        box.setIcon(QMessageBox.Icon.Warning)
        box.setWindowTitle("Registration Failed")
        box.exec()

        return False
```

Category 3 - Vector Field Pathfinder

For reference, here are the pathfinder objectives::
3) Create a vector field pathfinder simulation

3.1) Particles should exhibit fluid-like properties

3.2) The program should generate a distance field.

3.3) The program should generate an accurate velocity field.

3.4) The user should be able to dynamically change cells into obstacles which block particles.

3.5) The user should be able to add new particles with different properties such as mass and density which react accordingly

s 3.6) The program should have a strong and intuitive UI.

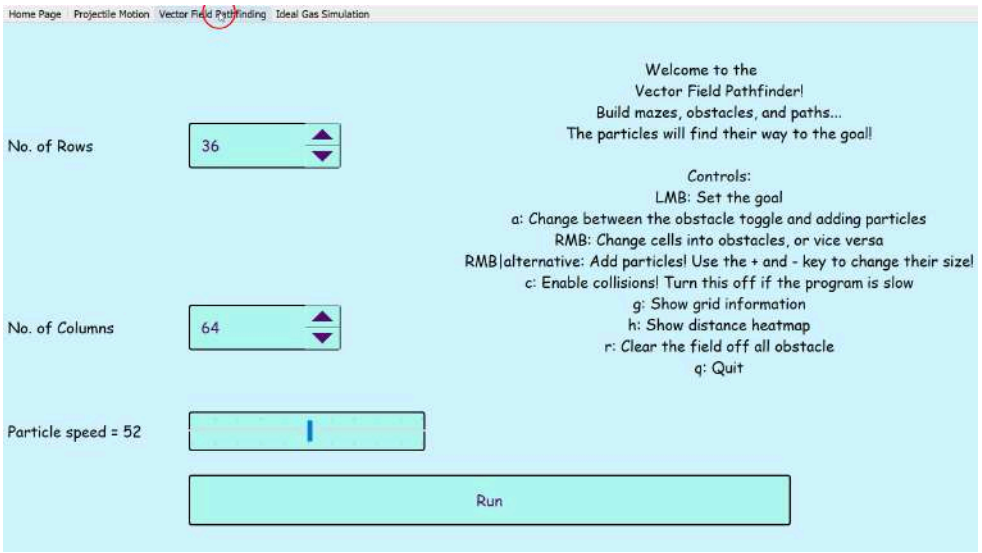
3.7) The user should have access to helpful information for the running of the program, such as the controls

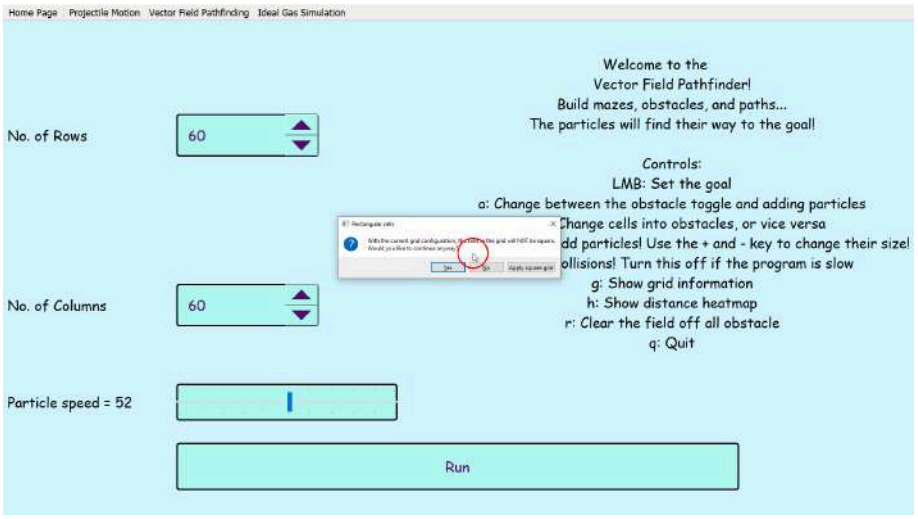
3.8) The program should be able to provide an abstraction of the particles and appear to flow.

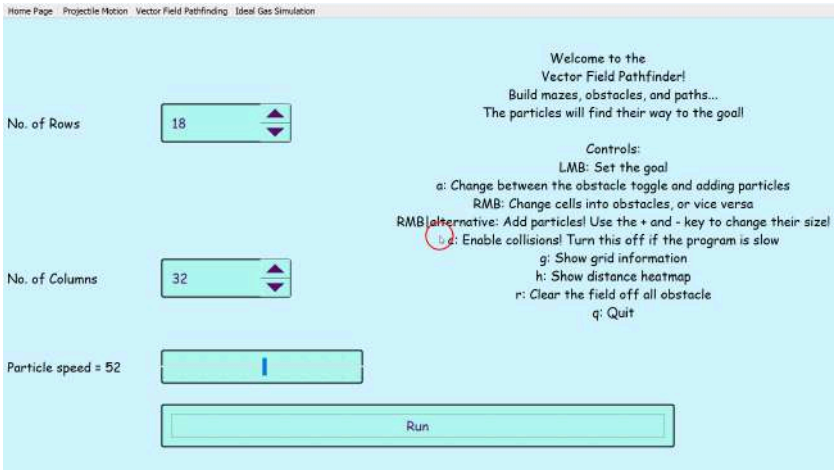
3.9) The particles should have some natural variation to prevent a blank uniformity, such as different masses.

3.10) The program should run efficiently with a reasonable set of parameters

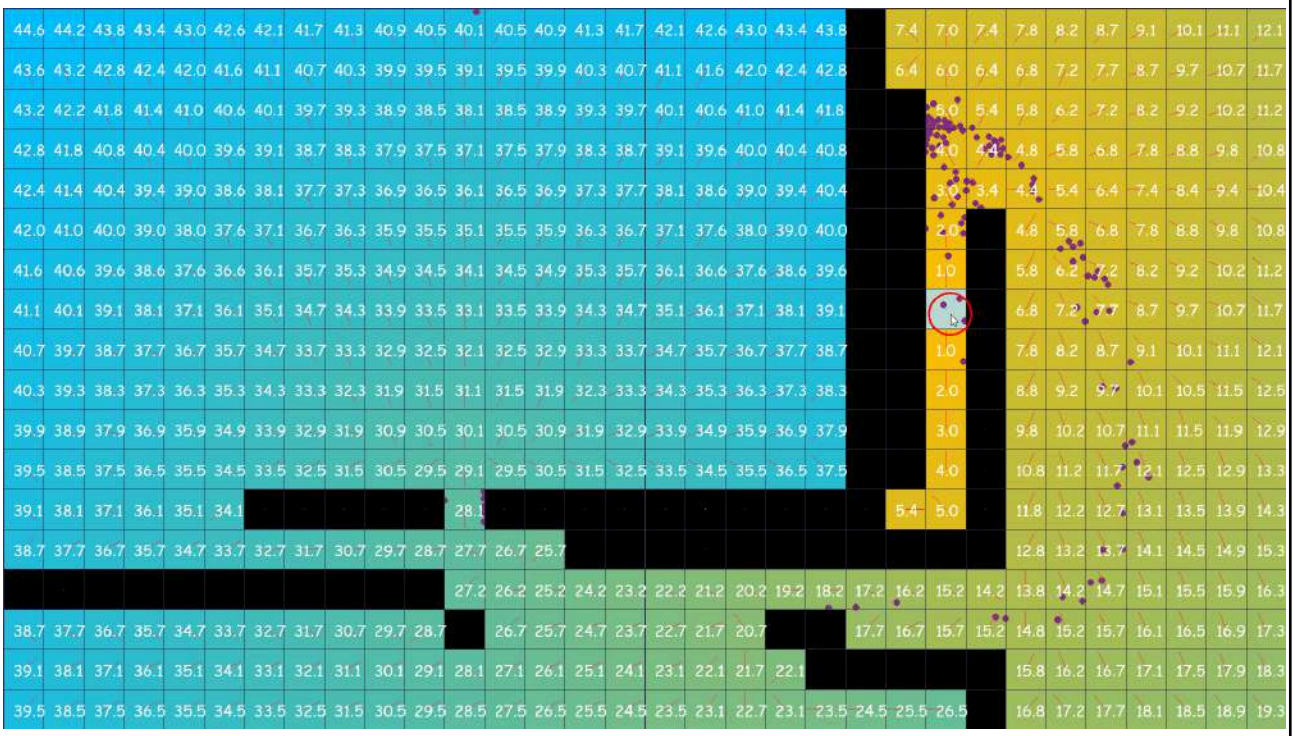
#	Description	Test Data/Type	Outcome	Additional Comments	Objectives Met
1	Selecting Vector Field Pathfinder tab	Normal	The program remembers the settings I put in from the last session	The widgets in the UI are clear and serve a purpose	



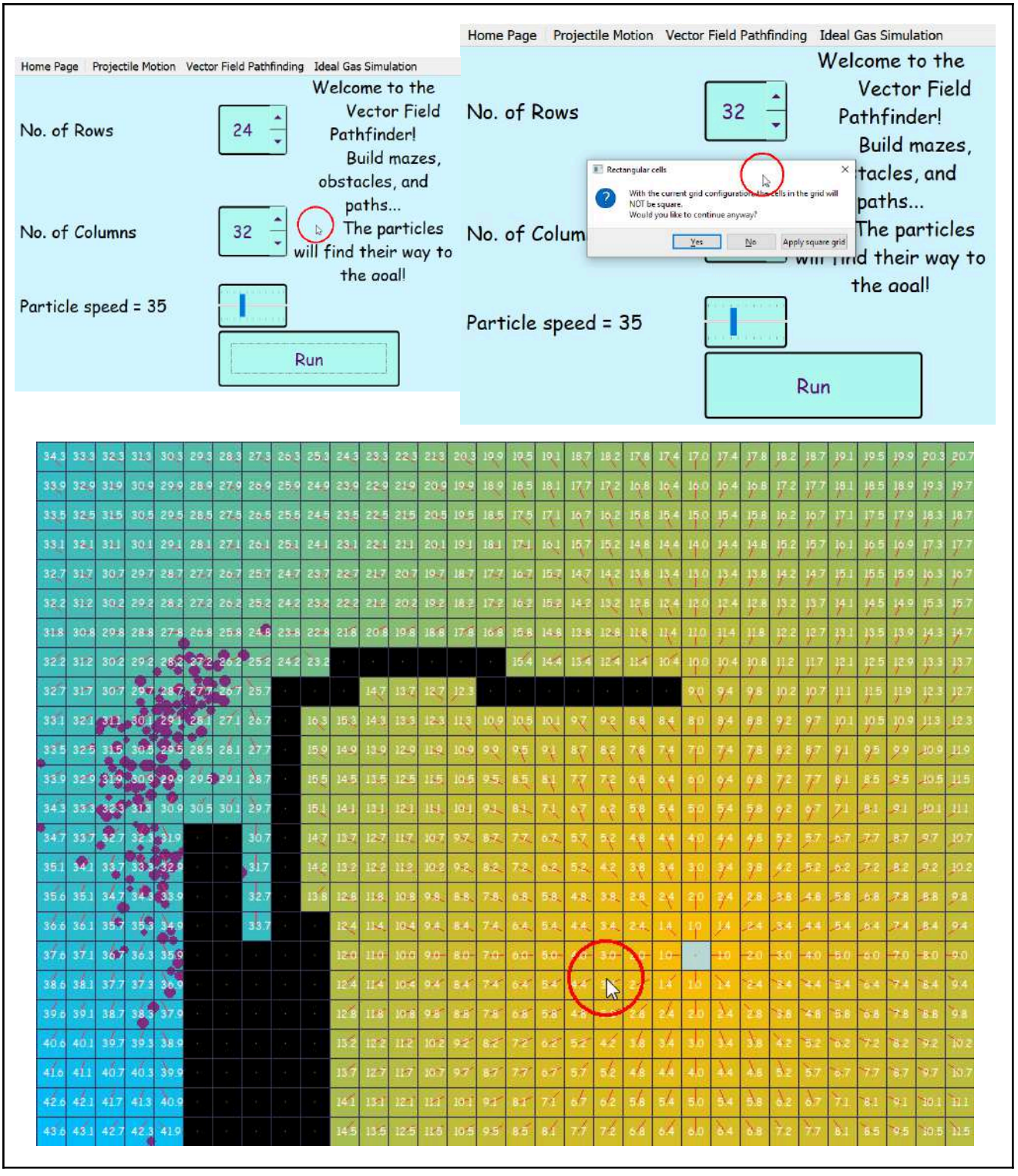
2	Attempting to run pathfinder with rectangular cells	Erroneous	A message box is shown		1.3 3.7
					
3	Apply Square Grid option of the message with 1920x1080 display	Normal	The program calculates the necessary dimensions for the grid to be square	Works as intended	1.3 3.7

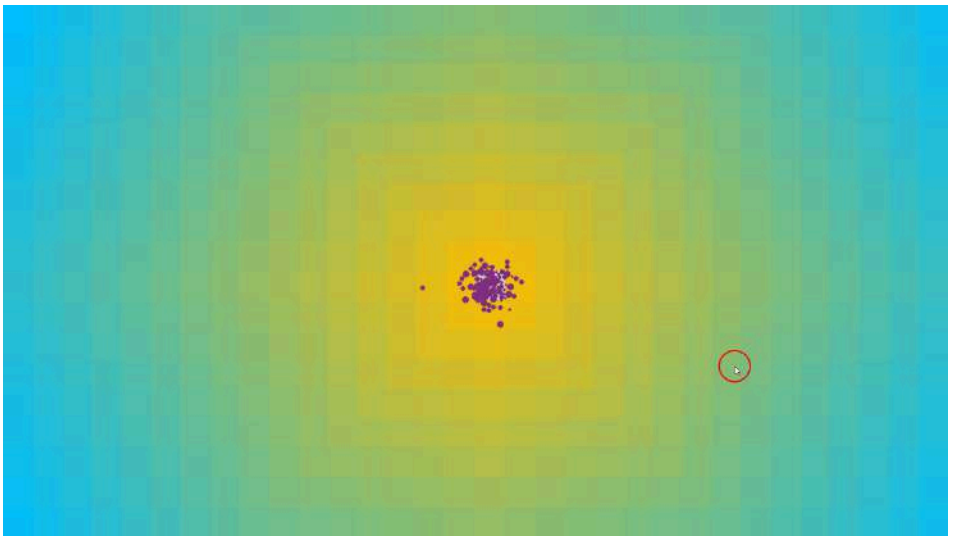
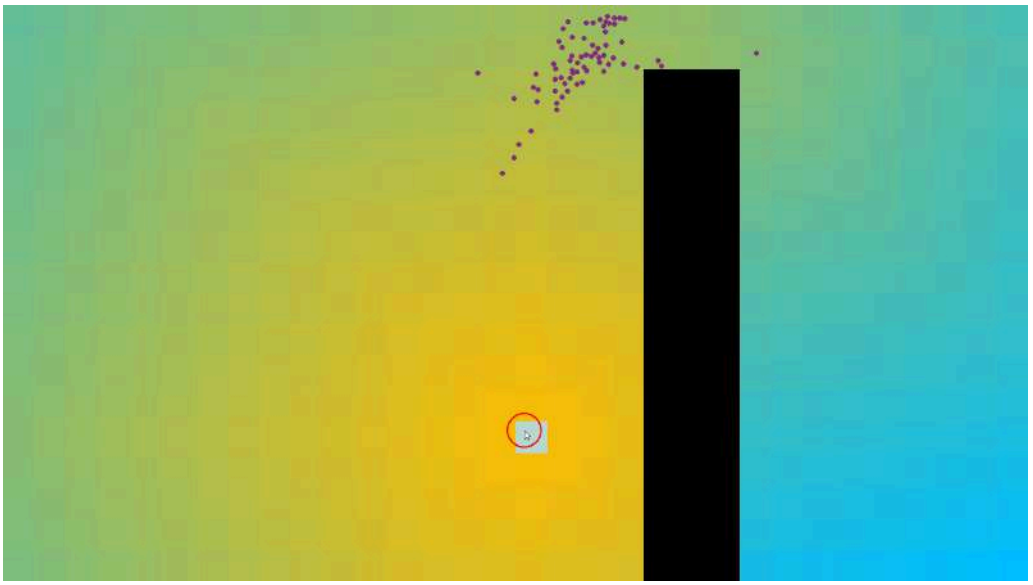


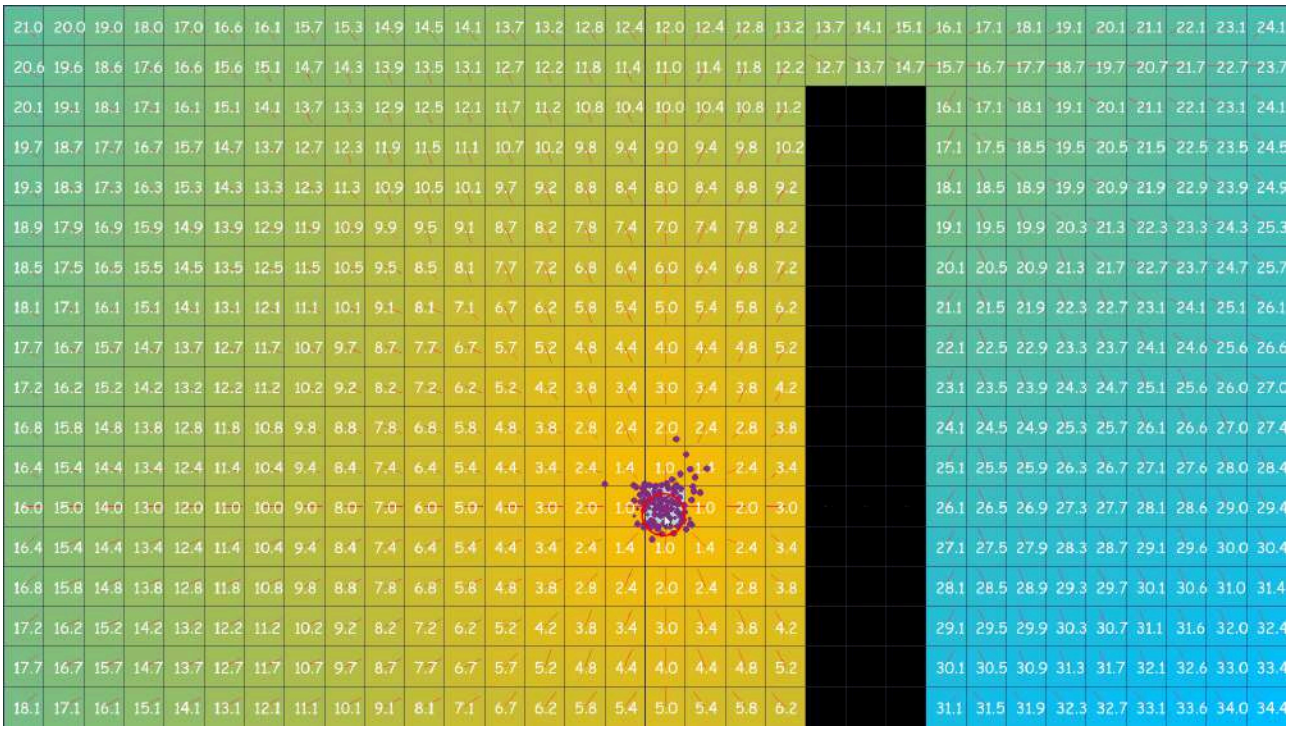
1920x1080 display correctly calculates that a 18x32 grid is necessary

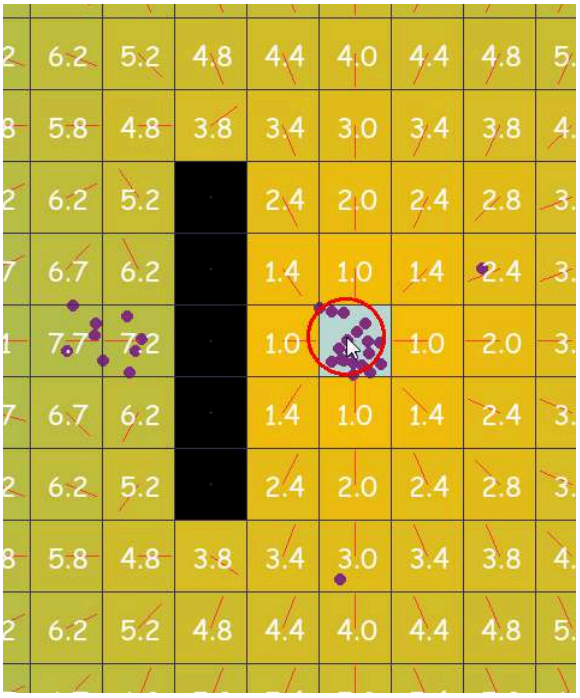


4	Apply Square Grid option of the message with 800x600 display	Normal	The program calculates the necessary dimensions for the grid to be square	I changed my resolution to test this properly	1.3 3.7

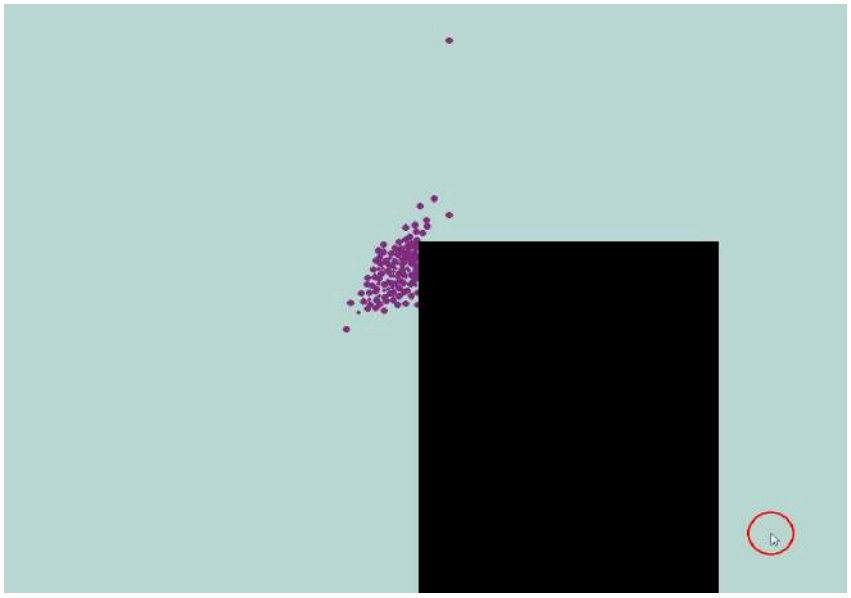

Home Page
Projectile Motion
Vector Field Pathfinding
Ideal Gas Simulation
No. of Rows
24
No. of Columns
32
Particle speed = 35
Run
Welcome to the Vector Field Pathfinder!
Build mazes, obstacles, and paths...
The particles will find their way to the goal!
Home Page
Projectile Motion
Vector Field Pathfinding
Ideal Gas Simulation
No. of Rows
32
No. of Colum
Particle speed = 35
Run
Welcome to the Vector Field Pathfinder!
Build mazes,
tacles, and
paths...
The particles
d their way to
the goal!
Rectangular cells
With the current grid configuration the cells in the grid will NOT be square.
Would you like to continue anyway?
Yes
No
Apply square grid

#	Description	Test Data/Type	Outcome	Additional Comments	Objectives Met
5	When packed together, particles will attempt to move away from each other due to collisions.	Normal	I left the simulation for a long time and the particles had lots of movements	This will only occur if collisions have been enabled by the user	3.1 3.9
					
6	The particles should flow around corners	Normal	The particles ‘drift’ or flow around the corners.	Particle speed plays a factor in how the particle responds.	3.1
					

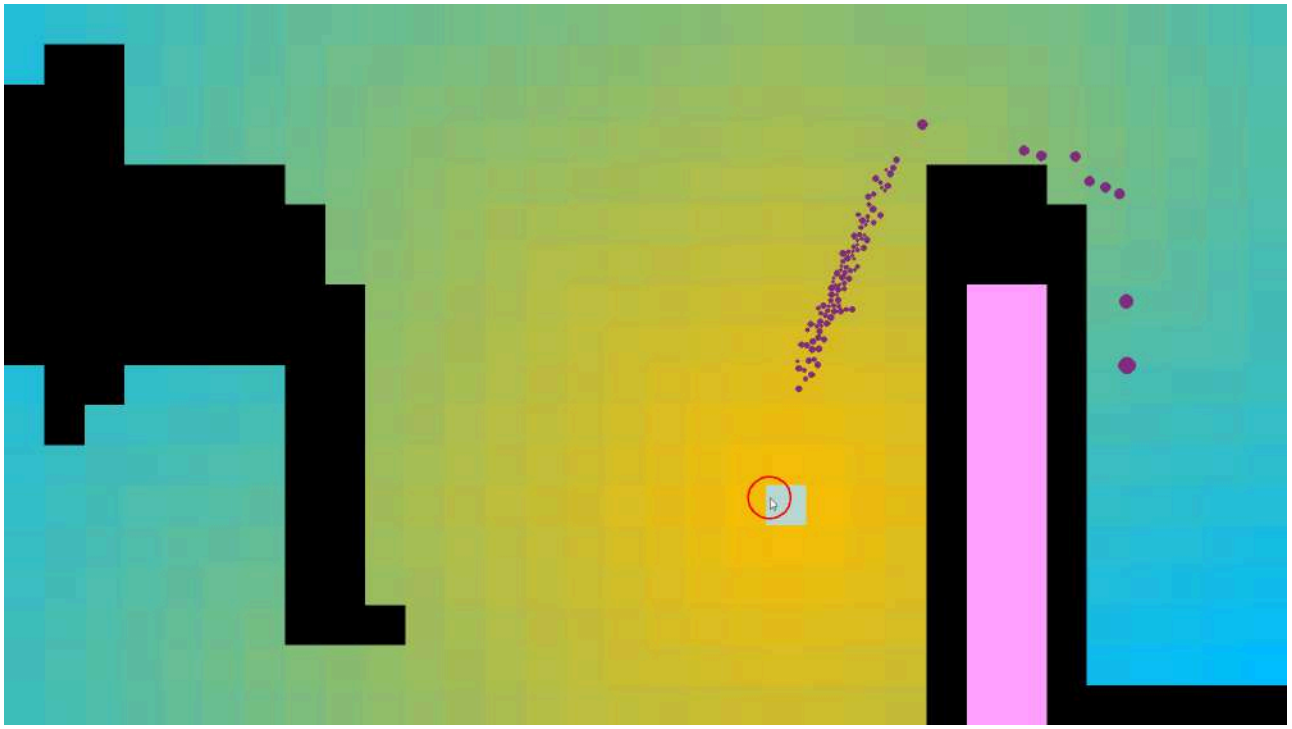
7	When the user selects a new goal, the program will generate a distance and velocity field	Normal	A grid is shown with the cell's metadata		3.2 3.3
					
8	**Adding particles in specific cells.**	Erroneous	Cells with no distance change either side will have no velocity in that direction. As such, particles will not reach the goal.	From my research, this appears to be a local optima problem.	3.3 3.5

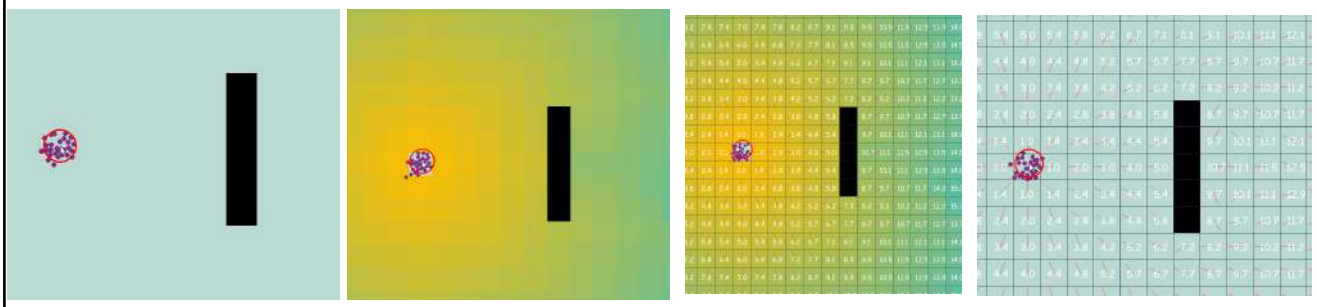
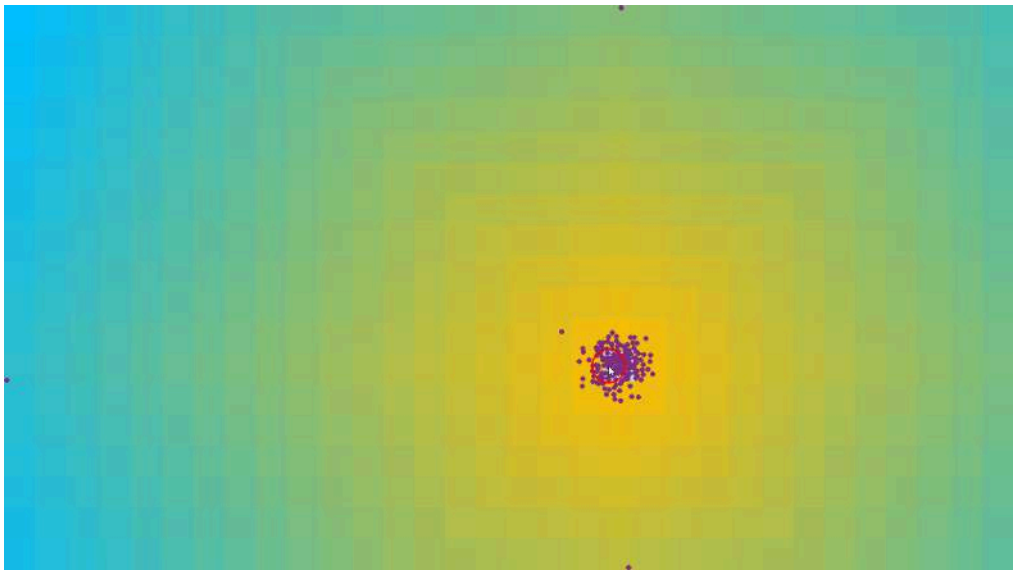


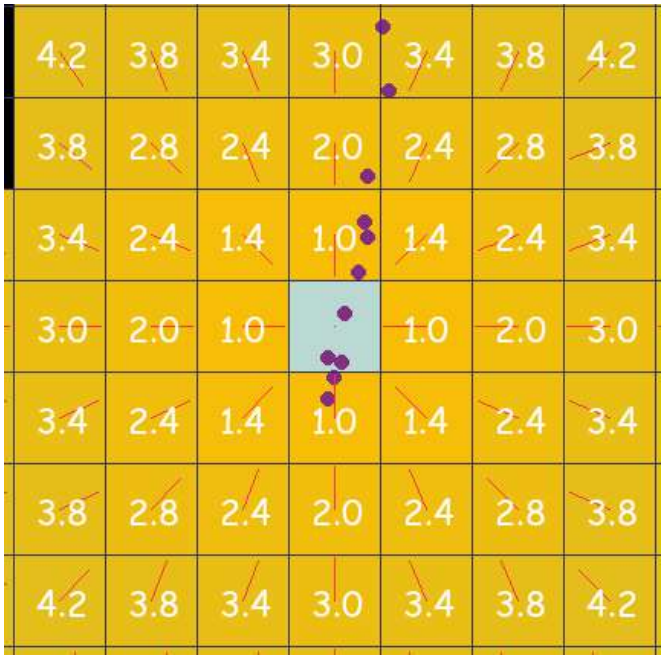
9	Particles collide with the obstacles	Normal	Particles bounce off the obstacles with a reduced velocity	Works very efficiently but particles will very rarely bypass the collision system and get stuck in an obstacle. However, the efficiency far outweighs this con.	3.4



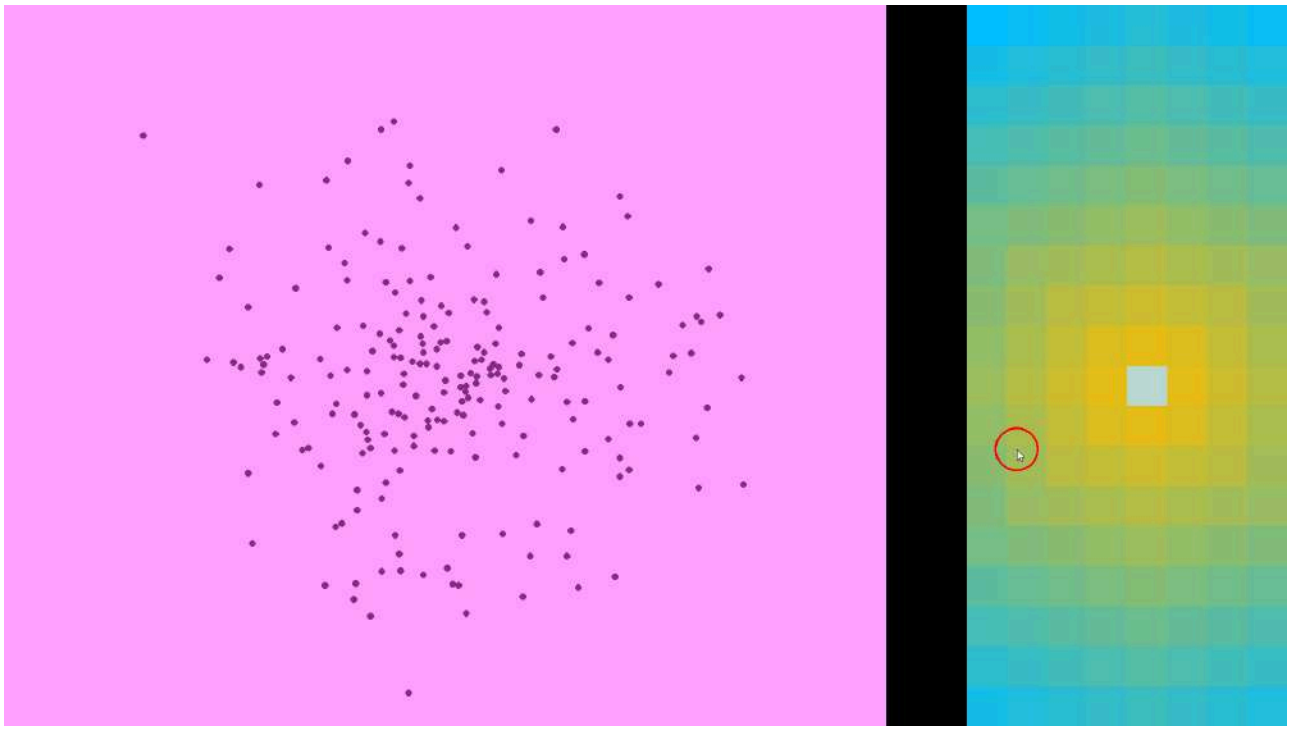
10	Creating obstacles	Normal	You can hold and drag the mouse and the program automatically draws obstacles underneath. If the first selected cell is an obstacles, dragging the	This feature runs very seamlessly and felt like second nature to use.	3.4

			mouse will remove obstacles instead of adding them		
					
11	Different size particles can be added	Normal	Different sized particle are shown	Note that the larger particles are slower as they have a greater mass	3.1 3.5 3.8
					
12	Clearing all obstacles with the reset button	Normal	All obstacles have been cleared.	I pressed the R key which stands for ‘reset’	3.4
13	Cycling	Normal	Two extra view modes,	Works as intended, though	3.6

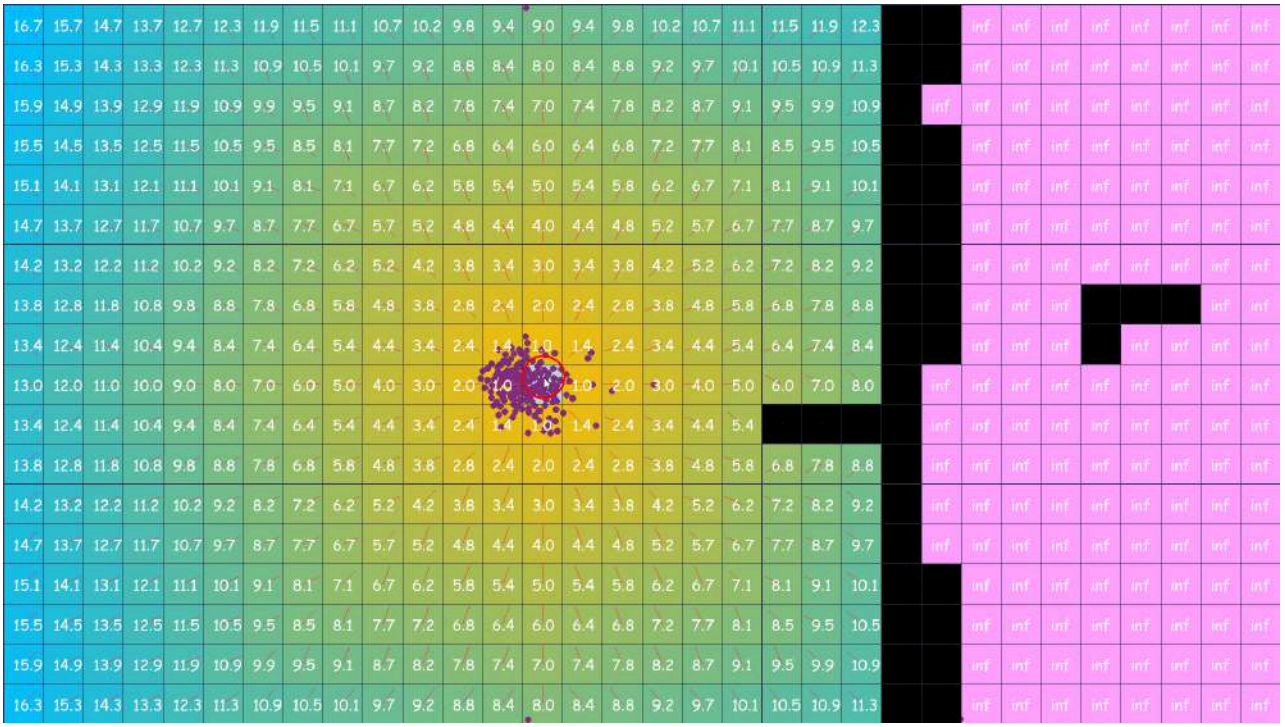
	between different view modes		which can also be combined. This gives 4 total view modes	some slight concerns with the visibility of the text.	
					
14	Selecting the same goal	Erroneous	No change	The simulation runs a check to only proceed with the change if the selected cell under the cursor is different to the current goal. This massively improved performance when a user would hold down the mouse (for the particles to actively follow the cursor).	3.9
					
15	Testing if distances with diagonal is implemented correctly	Normal	Distance from a cell to its diagonal is 1.4, which is sqrt(2) rounded.	This produced a much nicer effect than the orthogonal direction implementation I had before.	3.2



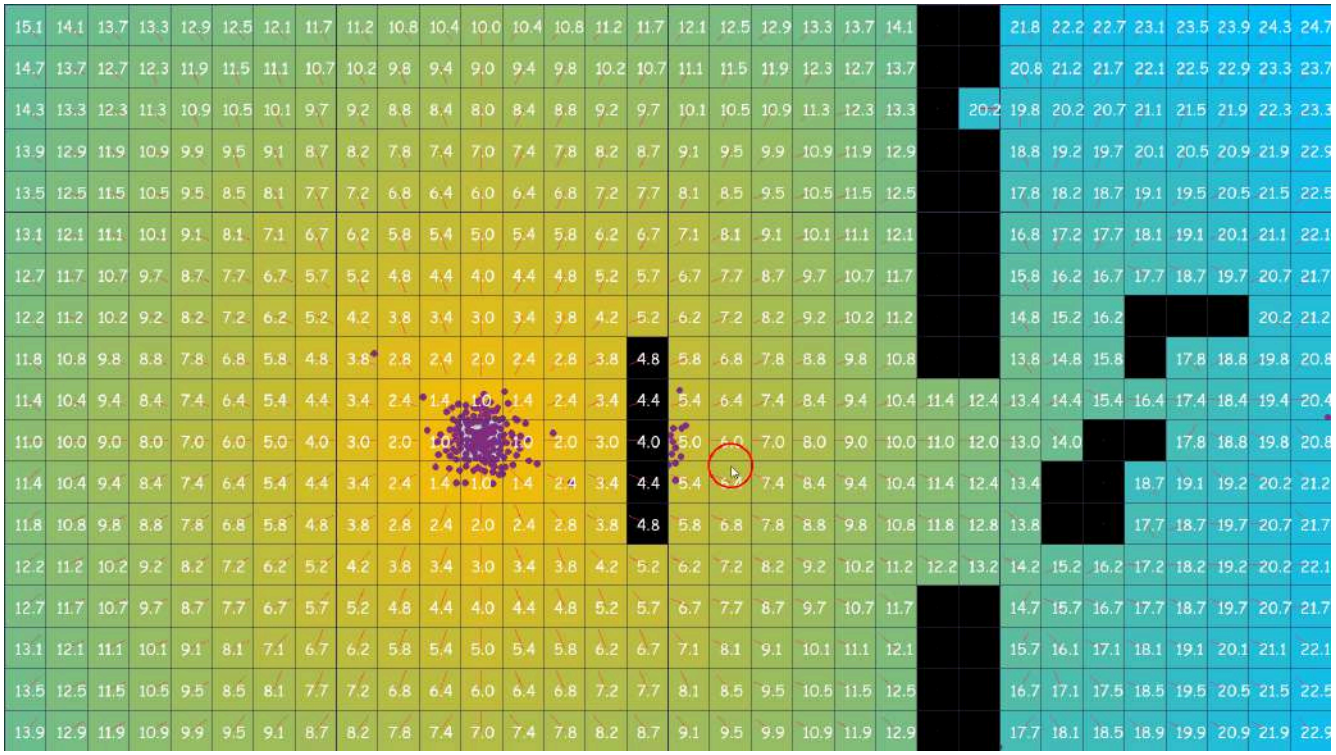
16	Selecting goal where particles can not go	Erroneous	Particle disperse appearingly randomly	The particles no longer have an external force on them to produce an acceleration on the particles	3.2 3.3



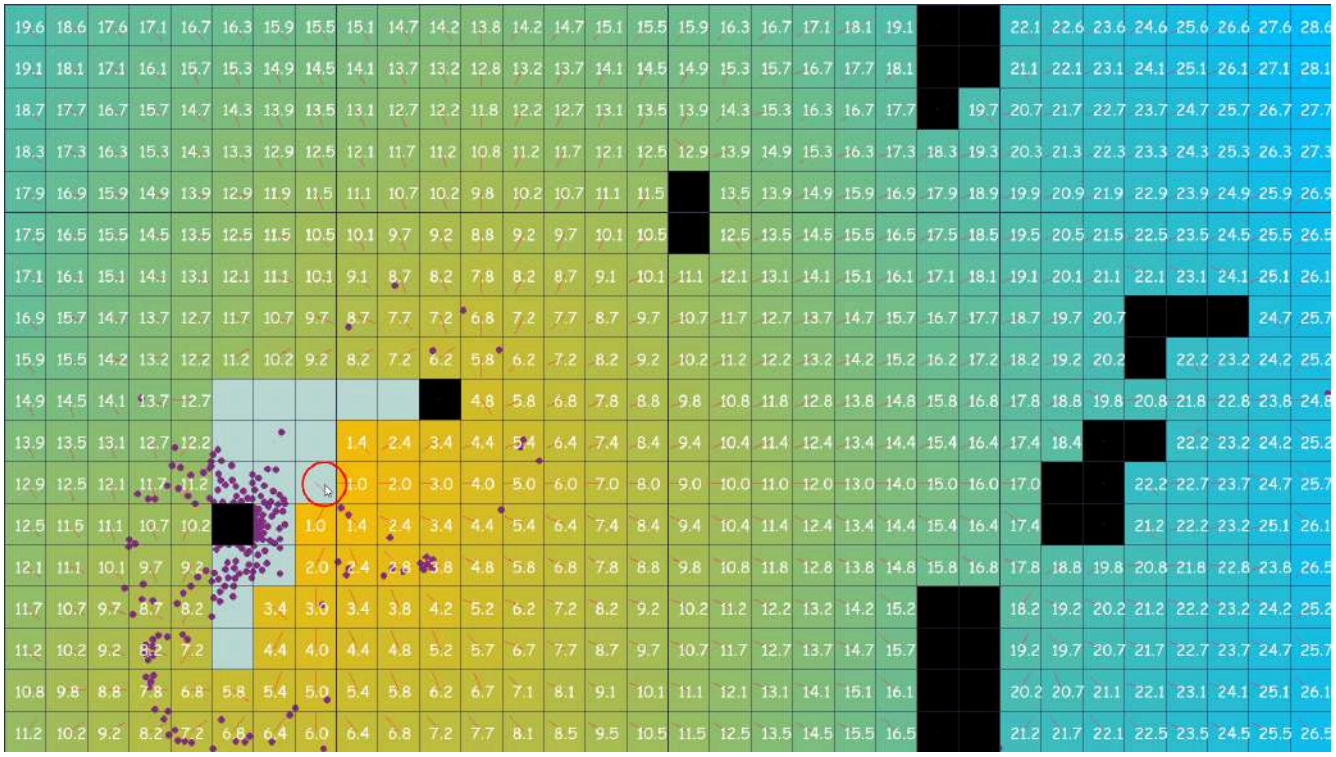
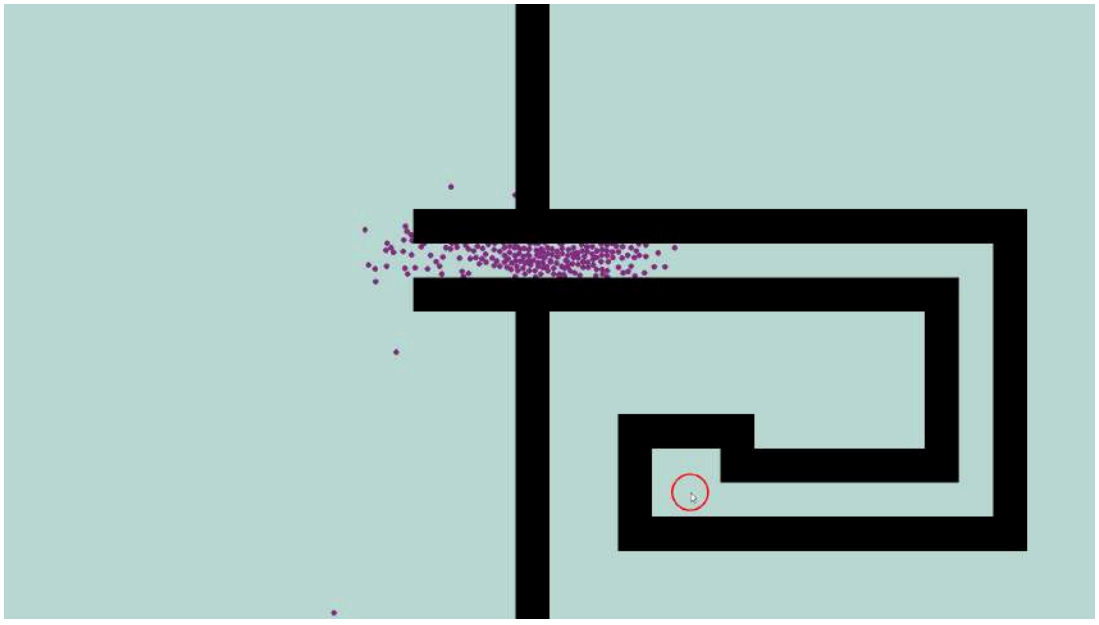
17	Selecting a goal where some cells are cut off	Erroneous	The isolated cells are shown in pink and have a distance of infinity.	Works as intended	3.2 3.3

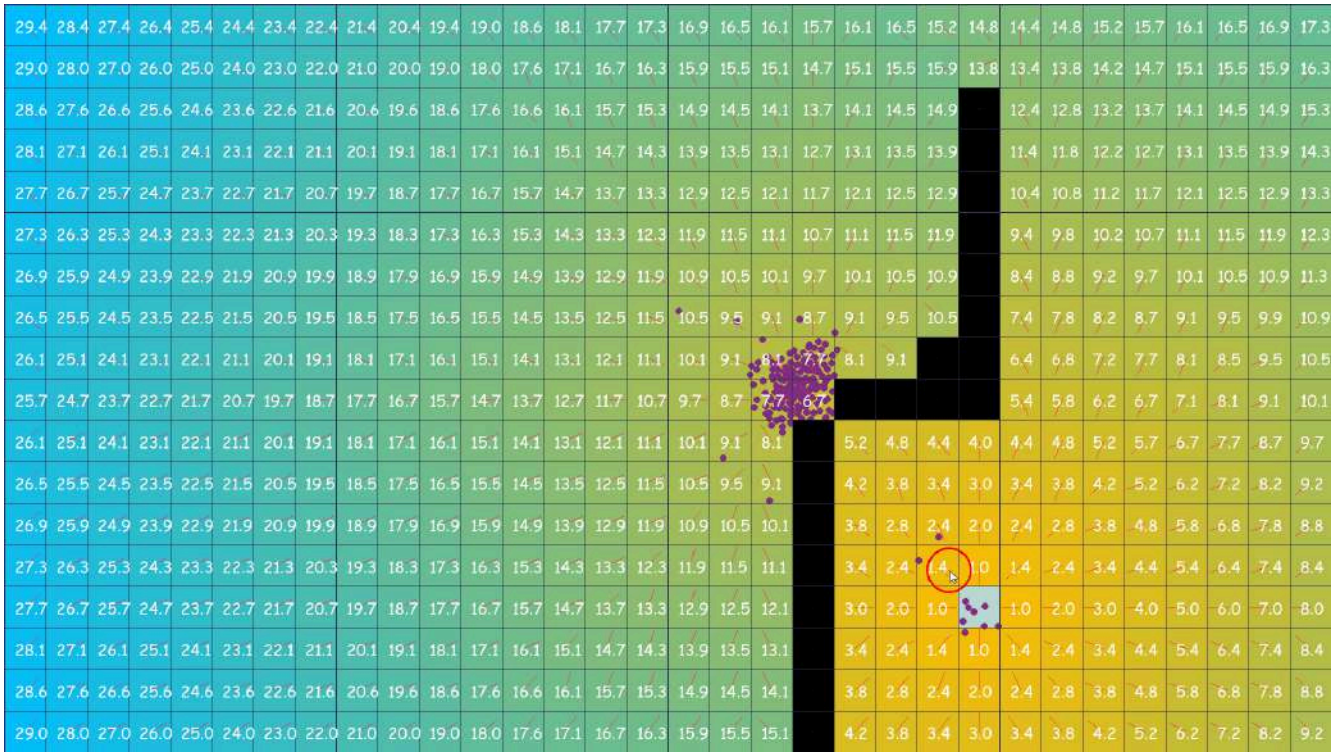


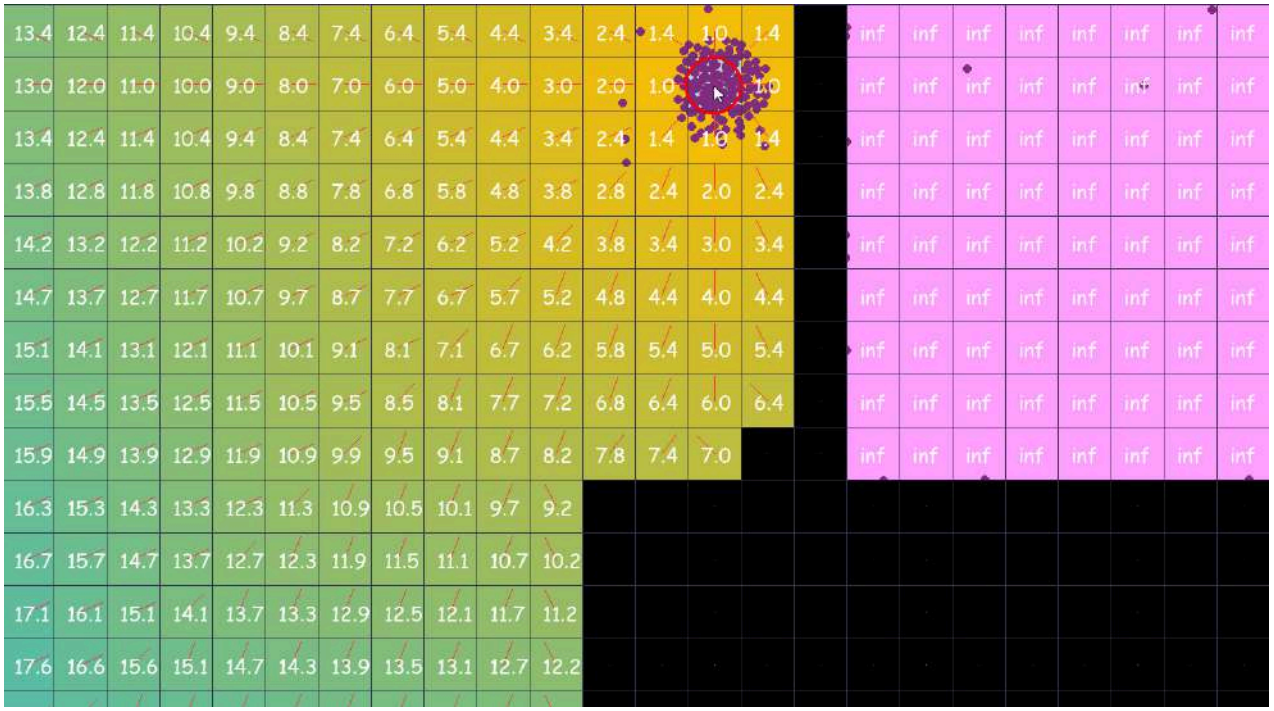
18	Adding new obstacles but not updating velocity field	erroneous	Particles repeatedly bump into the new obstacles	**Not ideal but as expected**. To fix this, I would have to compromise on performance and regenerate the velocity field every time an obstacle is added or removed.	3.2 3.3 3.4



19	Removing	erroneous	Particles can get stuck in	**Not ideal but as expected**.	3.4

	obstacles but not updating velocity field		the now free cells. Some particles move infinitely as they always overshoot the goal, and there is no force to send them back	Read the above.	3.5
					
20	Creating a one wide gap for the particles to travel through	Erroneous	Particles can successfully navigate the path	Works as intended	3.2 3.3
					

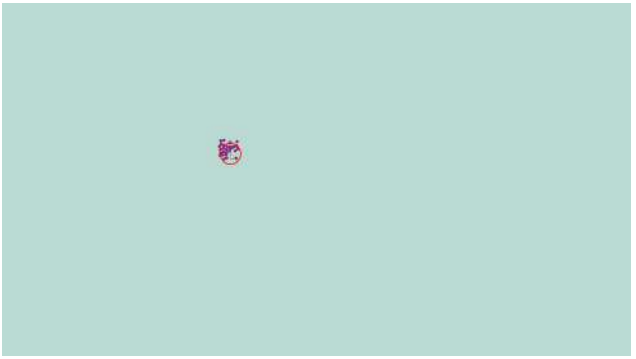
21	Creating a diagonal gap as below, where particles can't pass through	Erroneous	Particles will wrongly try to get to the goal through the diagonal instead of going around, thereby getting stuck	**Fail**. This happened because of my recent decision to implement diagonal movement for the wavefront algorithm. The algorithm incorrectly believes that the particles can travel diagonally.	3.2 3.3
					
22	Testing collision system	Normal	Some particles teleport to inaccessible regions	**Not ideal but expected.** This is due to the particle collision algorithm, where I have restricted a particle to one collision per time step. This massively saves on efficiency and creates a more random effect as seen earlier, as the overlap can get large. However, sometimes the below can occur. However, it is not enough to outweigh the benefits.	3.4



23	Updating Velocity Field	Timing	Efficiently updated.	18x32 grid	3.3 3.9

Updating Velocity Field
Generating distance field...
Distance field generated, now generating velocity field...
Velocity field generated
Time elapsed: 0.009008407592773438

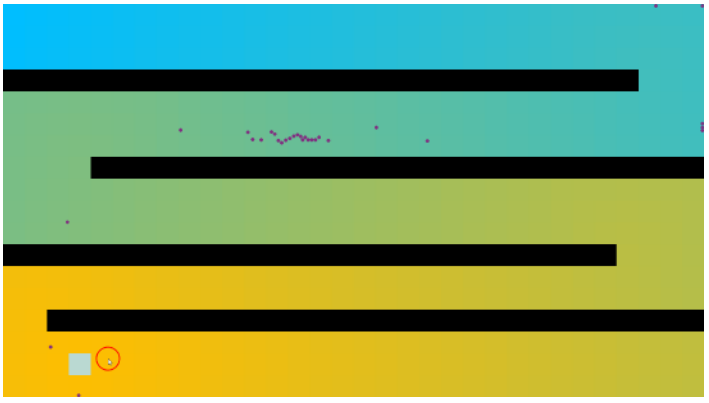
Displaying heatmap
Time elapsed: 0.008005857467651367



24	Updating Velocity Field	Timing	More efficient thanks to the more linear path	18x32 with obstacles	3.3 3.9

Updating Velocity Field
Generating distance field...
Distance field generated, now generating velocity field...
Velocity field generated
Time elapsed: 0.0070073604583740234

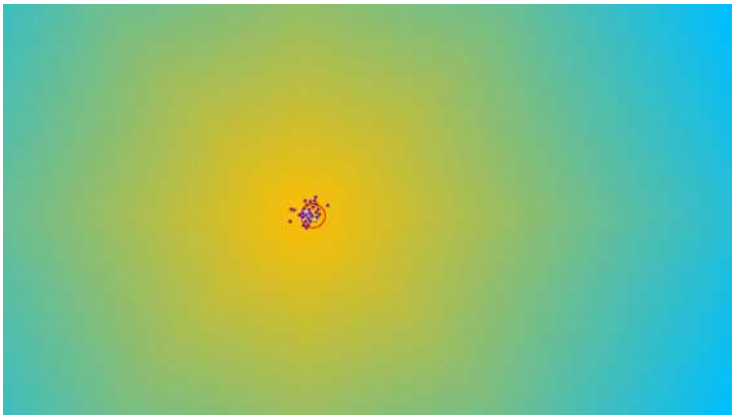
Displaying heatmap
Time elapsed: 0.007006645202636719



25	Updating Velocity Field	Timing	Larger grid has increased algorithm completion time	32x64 grid	3.3 3.9

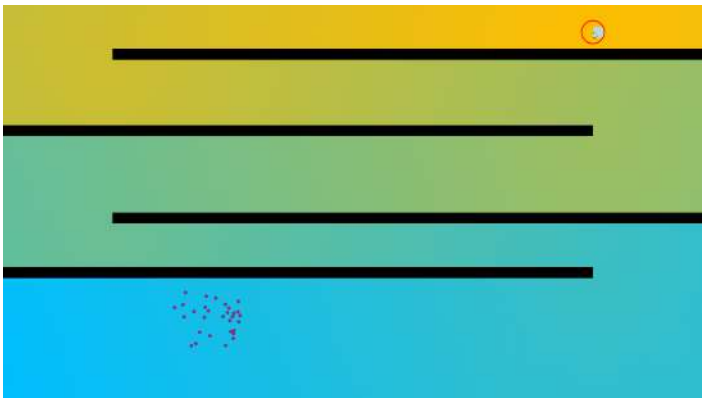
Updating Velocity Field
Generating distance field...
Distance field generated, now generating velocity field...
Velocity field generated
Time elapsed: 0.03603363037109375

Displaying heatmap
Time elapsed: 0.020018339157104492



26	Updating Velocity Field	Timing	Same as above	36x64 grid 216 obstacles	3.3 3.9

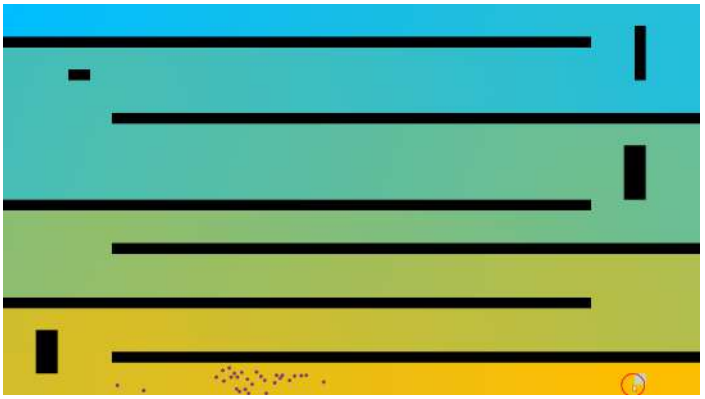
Updating Velocity Field
Generating distance field...
Distance field generated, now generating velocity field...
Velocity field generated
Time elapsed: 0.03303050994873047



27	Updating Velocity Field	Timing	It seems obstacles do not have much of a negative effect on the algorithm.	36x64 grid 364 obstacles	3.3 3.9

Updating Velocity Field
Generating distance field...
Distance field generated, now generating velocity field...
Velocity field generated
Time elapsed: 0.03102850914001465

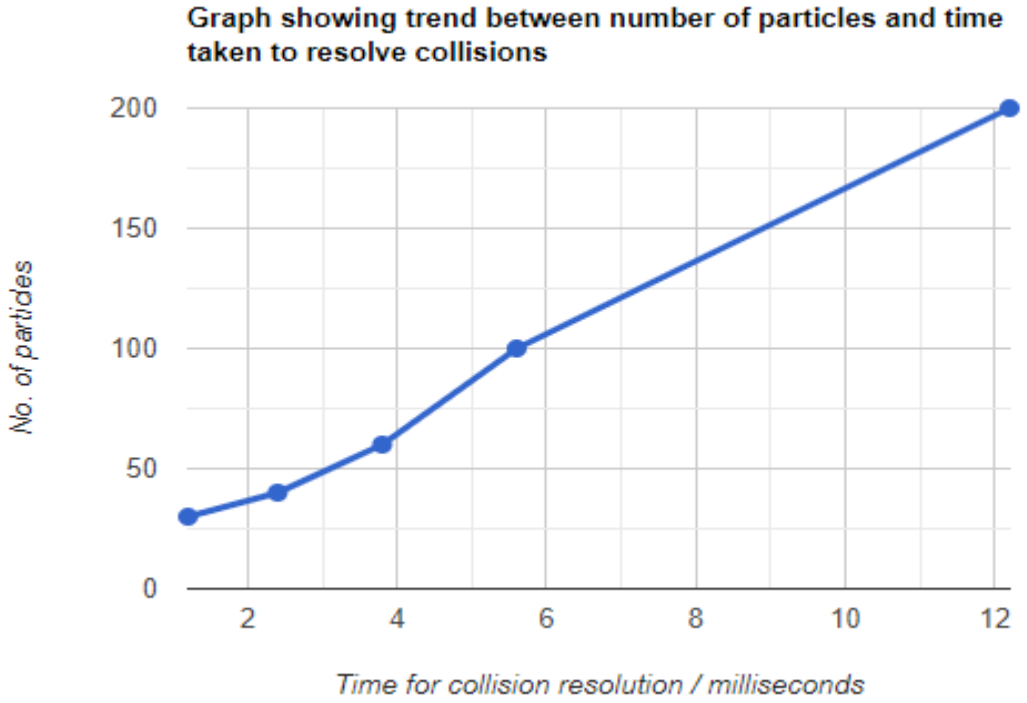
Displaying heatmap
Time elapsed: 0.017014741897583008



28	Time for collisions to be resolved	Timing	N/A	18x32 grid 30 particles - default	3.9

Time elapsed for collision resolution: 0.0020623207092285156 Time elapsed for collision resolution: 0.0010020732879638672 Time elapsed for collision resolution: 0.0010013580322265625 Time elapsed for collision resolution: 0.0010020732879638672 Time elapsed for collision resolution: 0.001001119613647461					
29	Time for collisions to be resolved		N/A	18x32 grid 40 particles	3.9
Time elapsed for collision resolution: 0.0020012855529785156 Time elapsed for collision resolution: 0.0030028820037841797 Time elapsed for collision resolution: 0.002002716064453125 Time elapsed for collision resolution: 0.003003358840942383 Time elapsed for collision resolution: 0.002000570297241211					
30	Time for collisions to be resolved	Timing	N/A	18x32 grid 60 particles	3.9
Time elapsed for collision resolution: 0.003003835678100586 Time elapsed for collision resolution: 0.0030035972595214844 Time elapsed for collision resolution: 0.0030040740966796875 Time elapsed for collision resolution: 0.004004478454589844 Time elapsed for collision resolution: 0.003003358840942383					
31	Time for collisions to be resolved	Timing	N/A	18x32 grid 100 particles	3.9
Time elapsed for collision resolution: 0.0060062408447265625 Time elapsed for collision resolution: 0.0060956478118896484 Time elapsed for collision resolution: 0.0050048828125 Time elapsed for collision resolution: 0.006006002426147461 Time elapsed for collision resolution: 0.0050029754638671875					
32	Time for collisions to be resolved	Timing	N/A	18x32 grid 200 particles	3.9
Time elapsed for collision resolution: 0.01301121711730957 Time elapsed for collision resolution: 0.012011289596557617 Time elapsed for collision resolution: 0.012011051177978516 Time elapsed for collision resolution: 0.012011051177978516 Time elapsed for collision resolution: 0.012010335922241211					

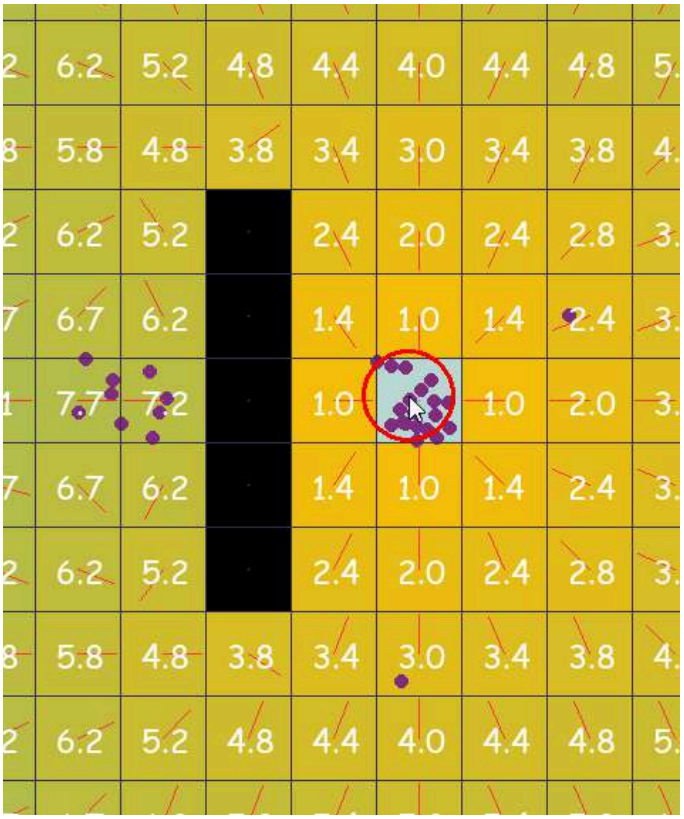
Reflection



Initially, I believed that the time taken to resolve collisions with the number of particles would be exponential. However, the testing shows that it follows a linear relationship. Earlier, I mentioned I changed my code to make it so that one particle may only have one collision per time step. This created a pleasing effect where particles could be constantly moving due to the inaccuracies in calculating the overlap. This also evidently reduced it to a linear relationship, as an extra particle means at most one extra collision per time step.

A large chunk of testing was done on the performance of the simulation. I believed that the user experience dropped slightly due to latency, particularly when with an excessive number of particles. To test my theory, I used the program and timed various things. The slowest component was comfortably the pathfinding algorithm. Next was the distance heatmap, which I hadn't noticed before. Observe test 24 with a time of about 0.02 seconds: this means that even if every other component worked instantaneously, the program would be capped at 50 fps. I then ran the simulation with the heatmap turned off, which then showed to be much quicker. Interestingly, the particle collisions had little impact on performance, which now makes me reconsider whether I should've enforced collisions by default.

A failed test which took me by surprise was the local optima problem in test 7. Here, the distance above the cells above and below and equal, and so the resultant vector has no vertical component. Eventually, I found an article by code.tutsplus.com which offered their own fix:



The most elegant way (I've found) to fix the problem is to subdivide both the heatmap and the vector field once. Every single heatmap and vector field tile has now been split into four smaller tiles. The problem remains the same with a subdivided grid; it has only been slightly minimized.

The real trick that solves the local optima problem is to initially add four goal nodes, instead of just one. To do this we simply have to modify the first step of the heatmap generation algorithm. When we used to only add one goal with a path distance of 0, we now add the four tiles that are closest to the goal.

There are several ways to choose the four tiles, but how they are chosen is largely irrelevant - as long as the four tiles are adjacent (and traversable), this technique should work.

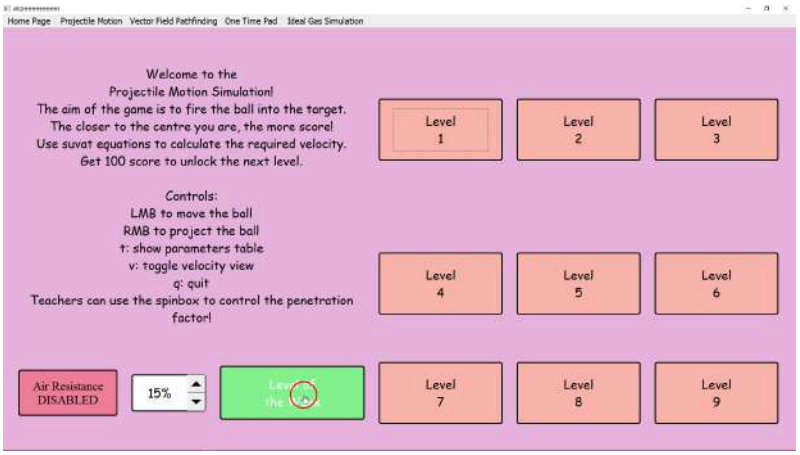
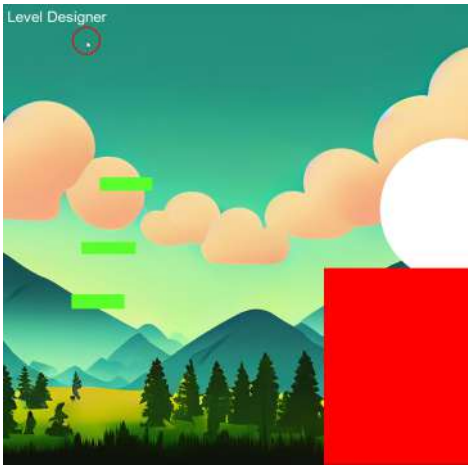
Here is the altered pseudocode for the heatmap generation:

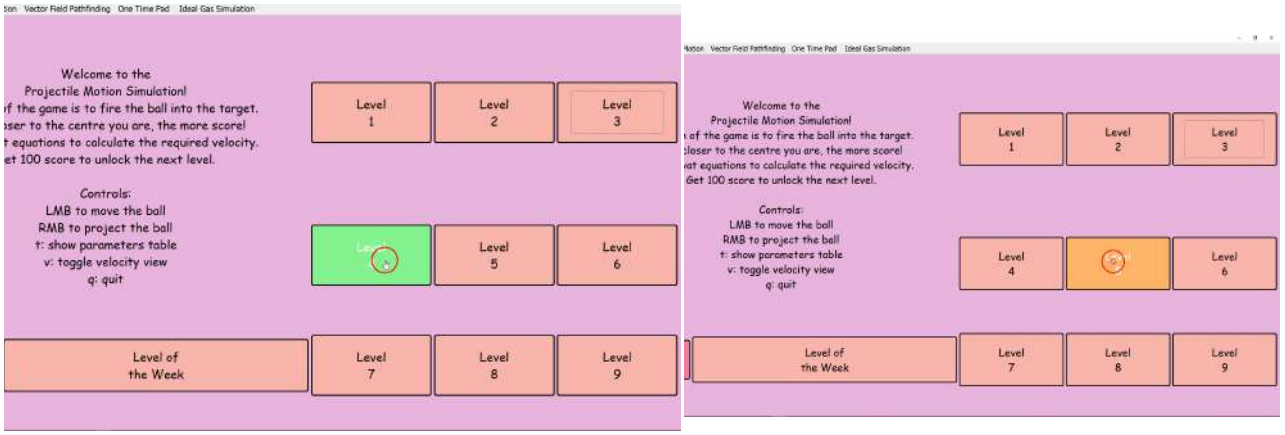
1. First, the algorithm begins at the *four goal tiles*, and marks *all four goal tiles* with a path distance of `0`.
2. Then, it gets each marked tile's unmarked neighbors, and marks them with `the previous tile's path distance + 1`.
3. This continues until the entire reachable map has been marked.

While this does fix my problem, it does mean that the algorithm takes 4x as long. Users of my simulation often hold down the mouse to set the goal to be wherever the cursor is currently. This increased time would correlate to much more noticeable delays that would be exaggerated when quickly switching the goals. As such, I have for the time being decided against implementing a fix.

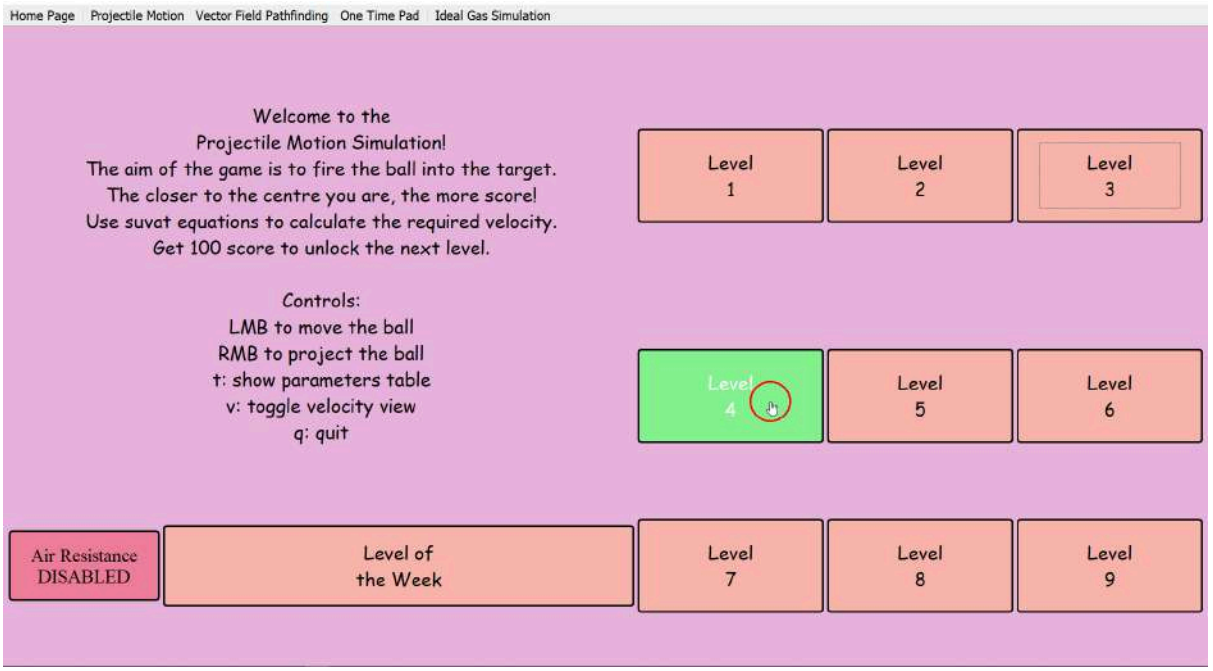
Category 4 - Projectile Motion Simulation

Note that some aesthetics may change but will functionally remain the same.

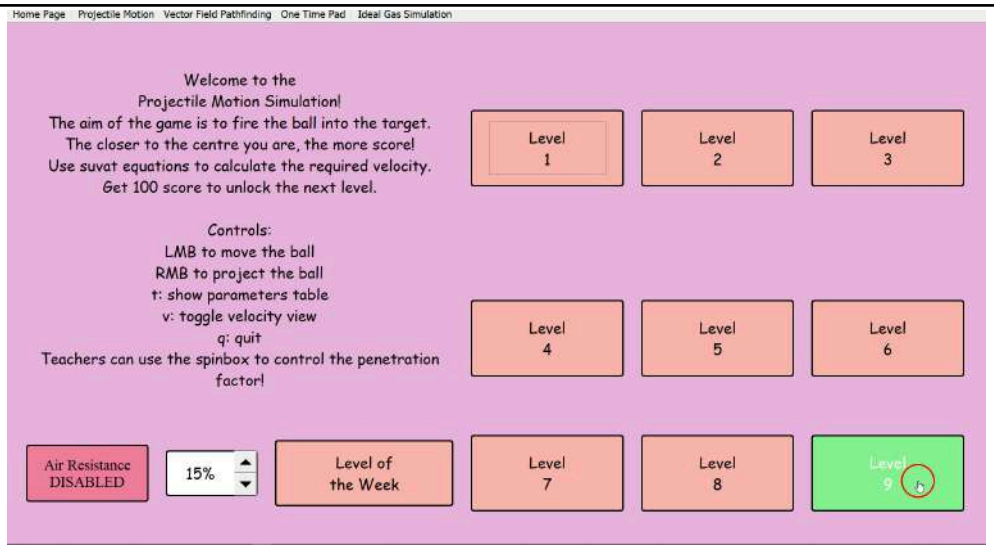
#	Description	Test Data/Type	Outcome	Additional Comments	Objectives Met
1	Teachers can create the weekly level for the projectile simulation	Normal	Level editor opened	As intended	1.8
	 				
2	Students do not have the option to edit the weekly level	Normal	Level Opened	The ‘teachers can use the spinbox’ text is not visible to students	1.8
3	A student tries to access the next level when they have not achieved a high enough score in the previous one	Normal	Level 5 has an orange background when the cursor hovers over it. The cursor changes to a forbidden symbol. The button does run anything.	Works as intended	4.6



4	The user settings from a previous login should be remembered by the program.	Normal	The program successfully remembers the student's progress	I used the same account as one used in the above tests	1.7



5	Teachers have automatic access to all levels	Normal	A new teacher account can access level 9.	Works as intended	1.8



6	Created a new account and viewed the projectile motion scoreboard	Normal	The column headings are not displayed, with numbers being shown instead. The scores and names are all correct	The functionality is working as intended. The leaderboard takes scores from the weekly level	4.6

1	2	3
1	Sam Hoskins	582
2	New User	554
3	Joe Cole	462
4	John Travolta	324
5	dfdsafsd	277
6	Nam Nam	154
7	Joe Bloggs	73
8	ghgigigjjjg	0
9	asdffadssfafsad	0
10	gyig	0
-	Test Account	0

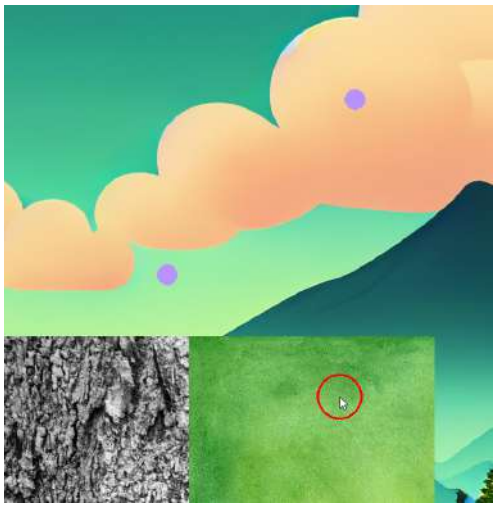
```
self.projectile_leaderboard = QTableWidget()
self.projectile_leaderboard.setColumnCount(3)
self.projectile_leaderboard.setHorizontalHeaderLabels(["Ranking", "Full Name", "Score"])
self.projectile_leaderboard.verticalHeader().setVisible(False)
self.projectile_leaderboard.setSizePolicy(QSizePolicy.Policy.Preferred, QSizePolic
```

7	Achieving a score in the weekly level in the same session	Normal	It should say ‘Test Account’ instead of ‘asdfasd’.as the name.	The new score has correctly been added, but strangely the name is wrong.	4.6

1	2	3
1	Sam Hoskins	582
2	asdfasd	562
3	New User	554
4	Joe Cole	462
5	John Travolta	324
6	dfdsafsd	277
7	Nam Nam	154
8	Joe Bloggs	73
9	ghgigigjjjg	0
10	asdffadssfafsad	0
-	Test Account	562

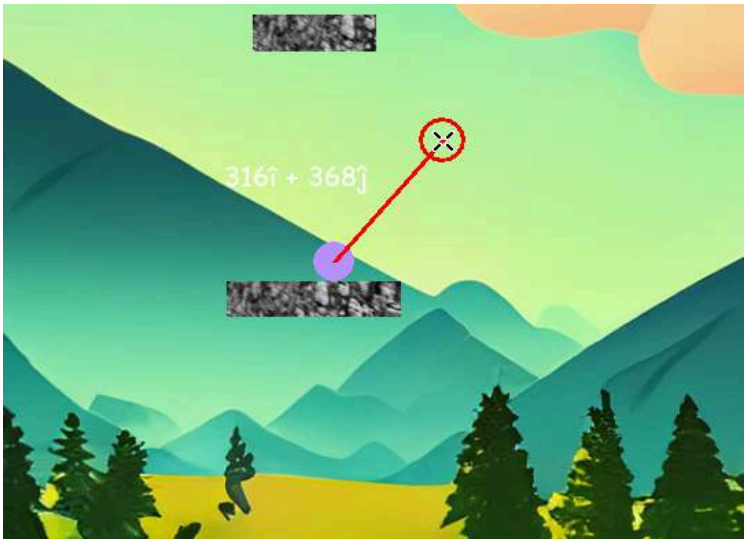
#	Description	Test Data/Type	Outcome	Additional Comments	Objectives Met
9	Checking level loads correctly	Normal	Balls correctly spawn and fall into the box.	Sometimes the ball may bounce out of the box. Fixing this is just a case of adjusting initial spawn positions.	4.3

10	Particles react differently when hitting a wall vs a platform.	Normal	Ball loses lots of energy when hitting platform, but only loses some when hitting wall	Platforms can be used to fire balls off of, or simply as a rough surface	4.3



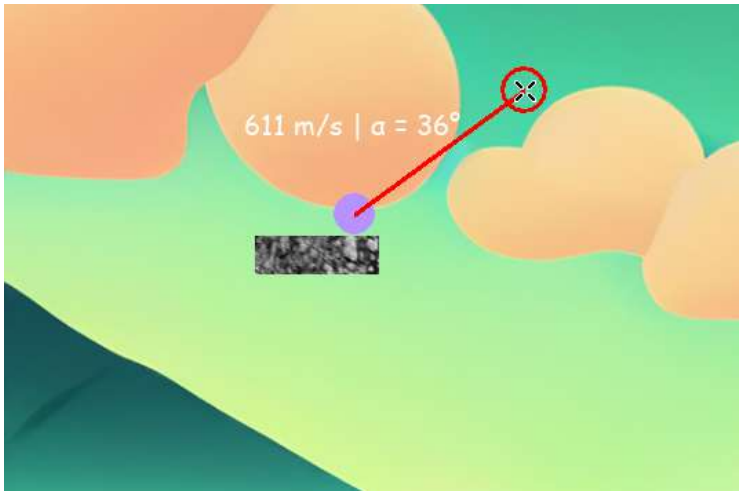
11	Dragging the ball around the screen	Normal	The ball follows the mouse position when holding it down with left click	Works as intended	4.8

12	Line draw	Normal	A line from the ball's centre to the mouse position. The projected velocity is also shown above the ball		4.2 4.8

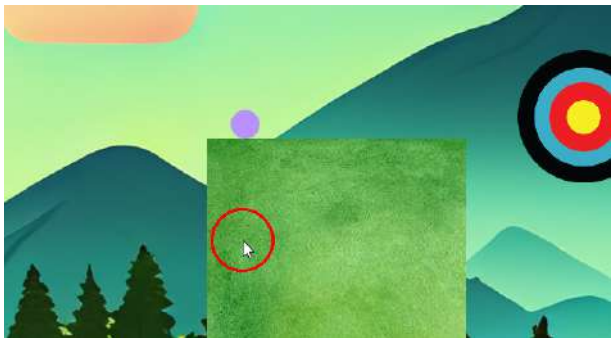


Please note that the red circle around the cursor is not from the program. It is included in the screenshotting software I used.

13	Toggling between velocity view modes	Normal	The projected velocity is now shown in vector form, as opposed to the previous speed and angle form.	This is triggered by selecting v whilst the user is projecting the ball. This setting is also saved for the rest of the simulation runtime.	4.2



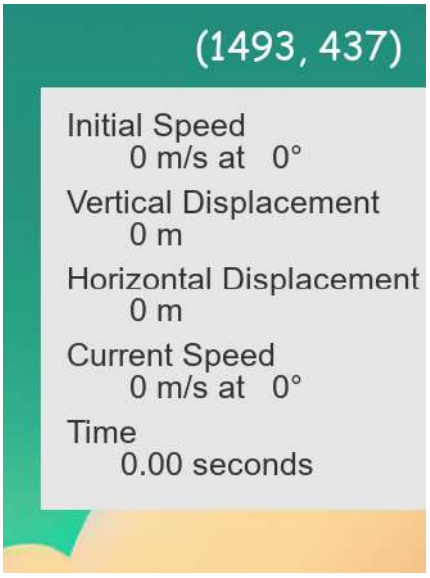
14	User drags ball inside an obstacle	Erroneous	Ball appears above the obstacle when released.	The ball will navigate to the top of the obstacle thanks to the collision algorithm.	4.3

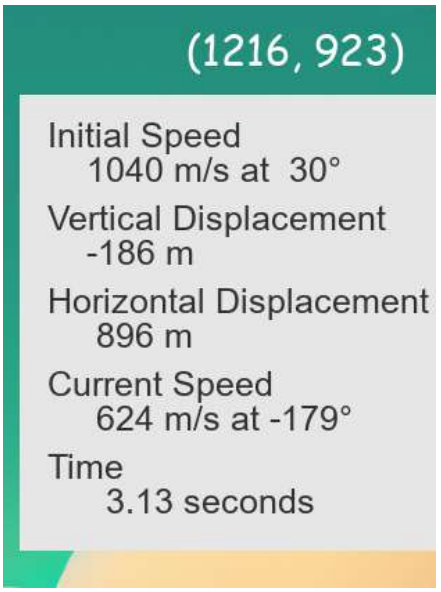


15	Moving balls with other balls	Normal	Balls react appropriately and can be pushed with each other.	Works as intended	4.1

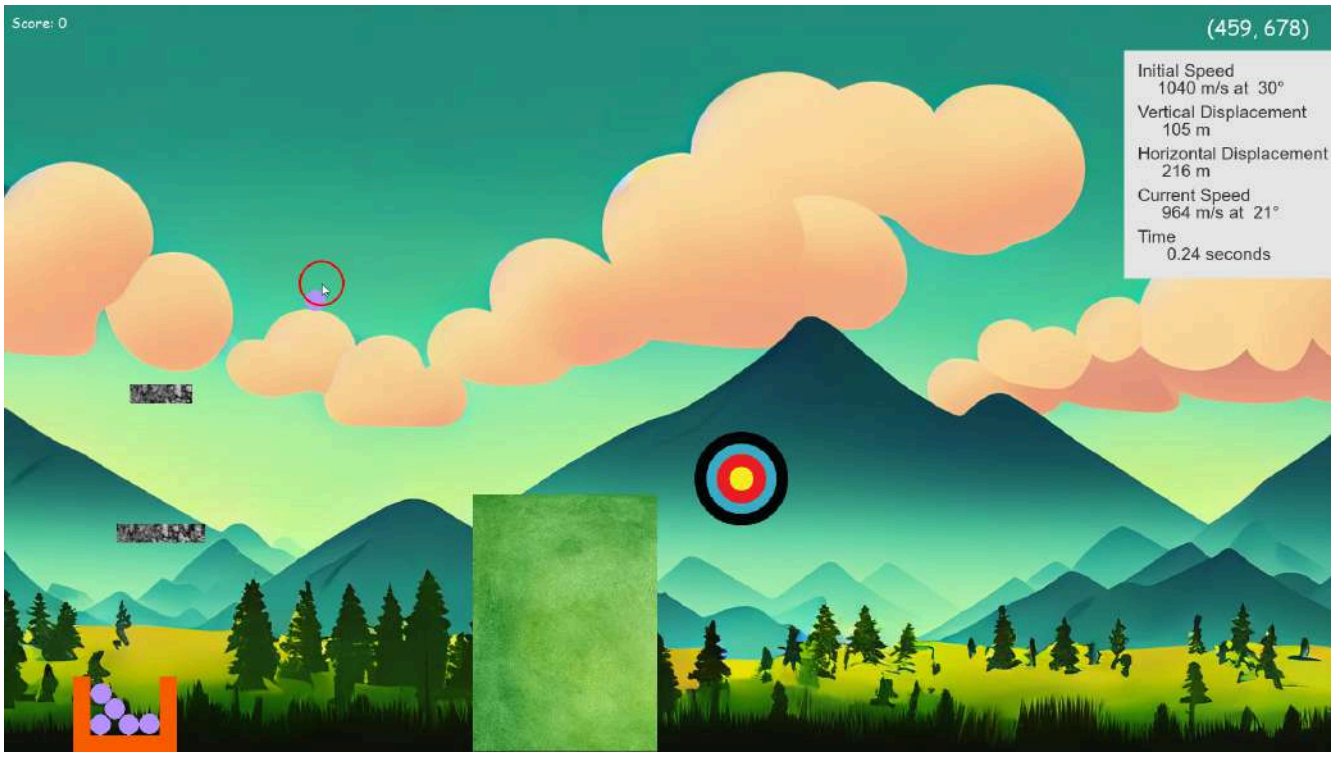
16	Data table loads correctly and visibly	Normal	The initial speed of the ball is	Currently, the timer does not stop upon a collision with	4.4

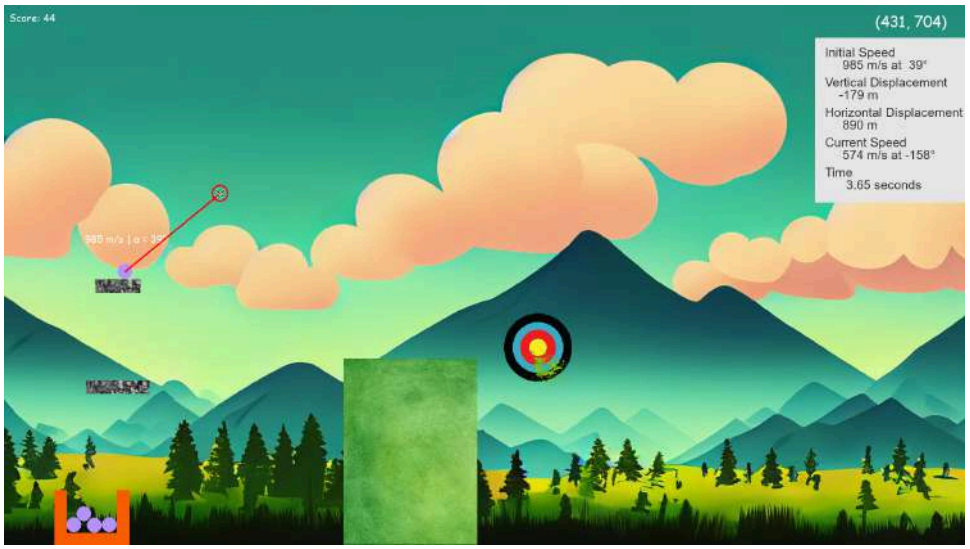
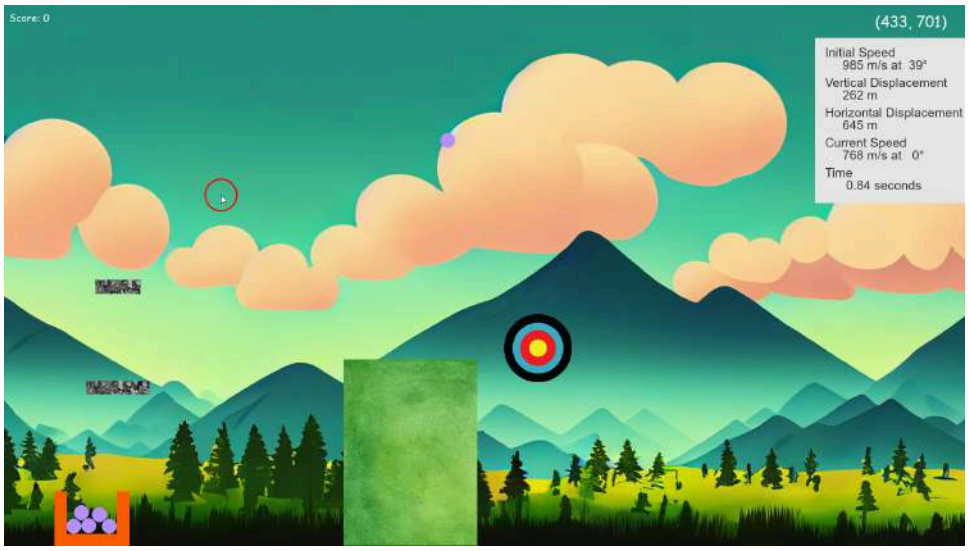
			stored. Various properties are recorded until the ball collides with a wall.	another particle, which can sometimes not be the preferred effect.	

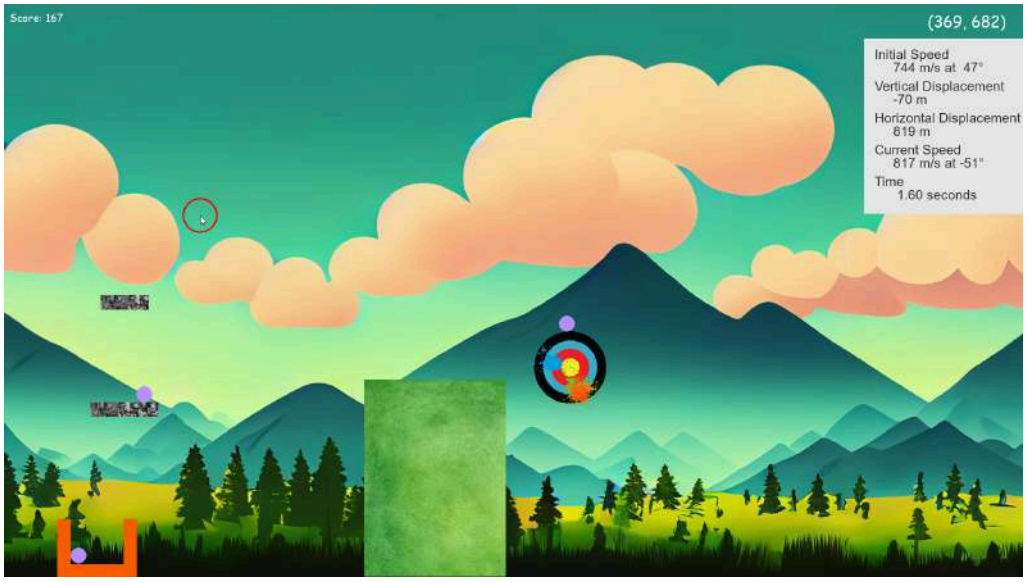




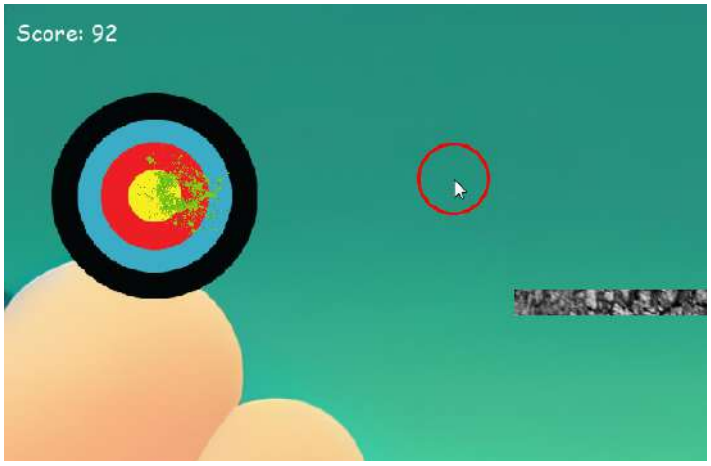
17	Testing air resistance	Normal	At higher speeds, air resistance has a greater effect than at lower speeds.	Works as intended.	4.5



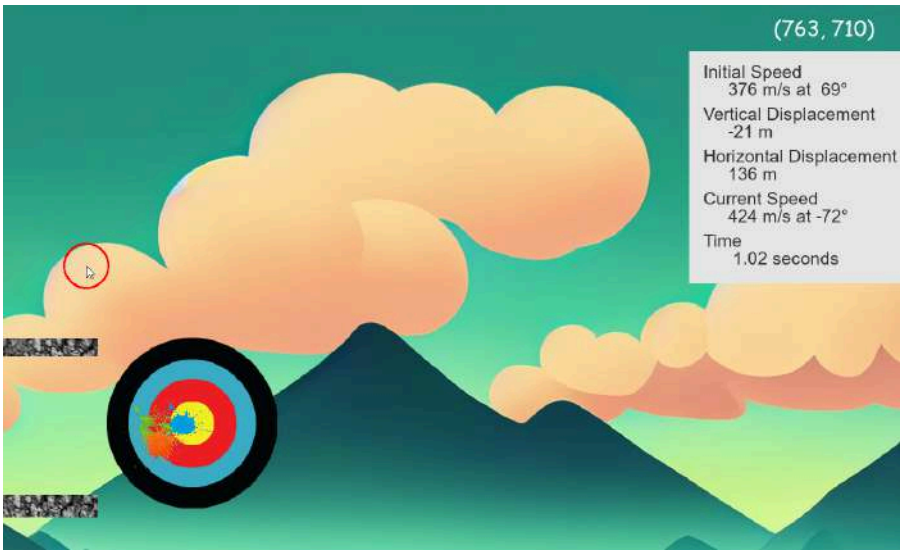
18	Testing if ball fires correctly	Normal	Ball's properties accurately corresponds to the table properties and follows the expected parabola shape	This was done with air resistance disabled.	4.2
 					
19	Testing if ball's properties are accurately tracked	Normal	The information table updates correctly	See above photos for image	4.4
20	Ball hits target or obstacle	Normal	The time is paused and the continuous variables are stopped to indicate this.	The user is now able to reflect on the projectile motion by observing the information saved in the table.	4.5



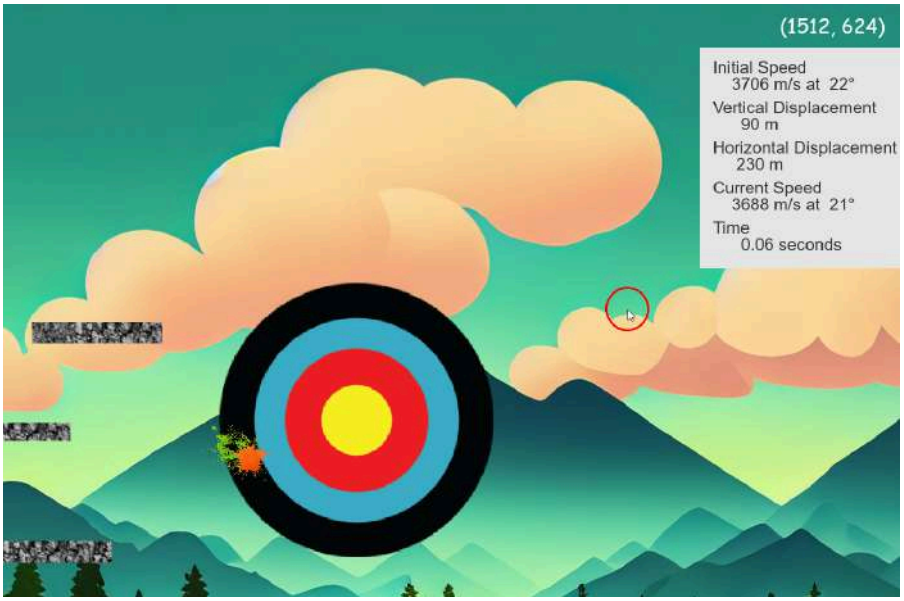
21	Getting ball to land in target	Normal	Ball correctly slows and turns into a splat upon landing on target. Points corresponding to how close to the centre the ball landed is correctly added to the score	Score is calculated according to 1-r^2, as discussed in previous chapters.	4.5



22	Testing penetration factor effects hitting the target	Normal	Penetration effect successfully changes how deep the ball is able to travel.	As intended, the ball can fly past the target if it is fired with too high of a velocity, as the speed does not reduce to zero fast enough.	4.5



Observe the initial speed and angle in the info table. I applied a small velocity and thanks to the low penetration factor, it made it to the centre of the target.



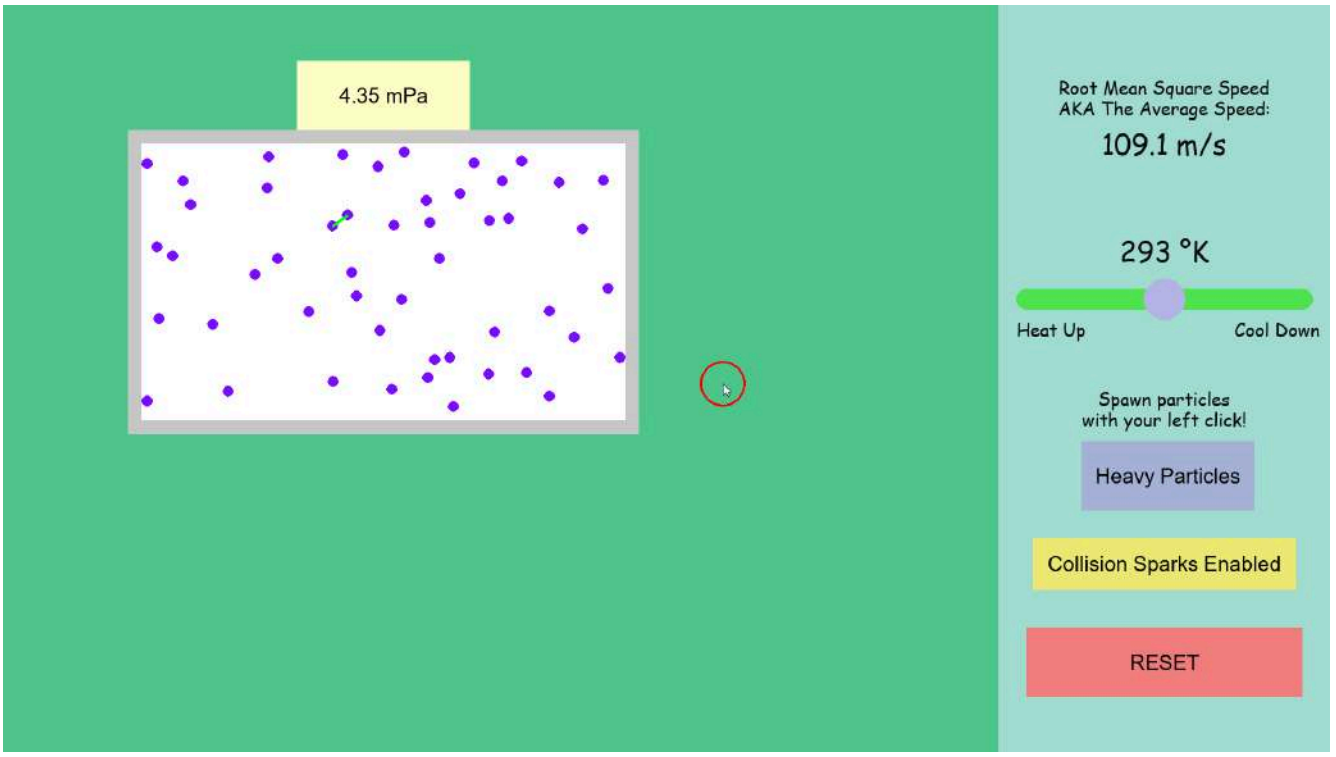
Here, the penetration factor is wrongly very high when the target is very big. As such, the ball hardly makes it into the target despite the very high speed (again see 'Initial Speed')

It is up to the teacher to choose the appropriate penetration factor for the level they have created.

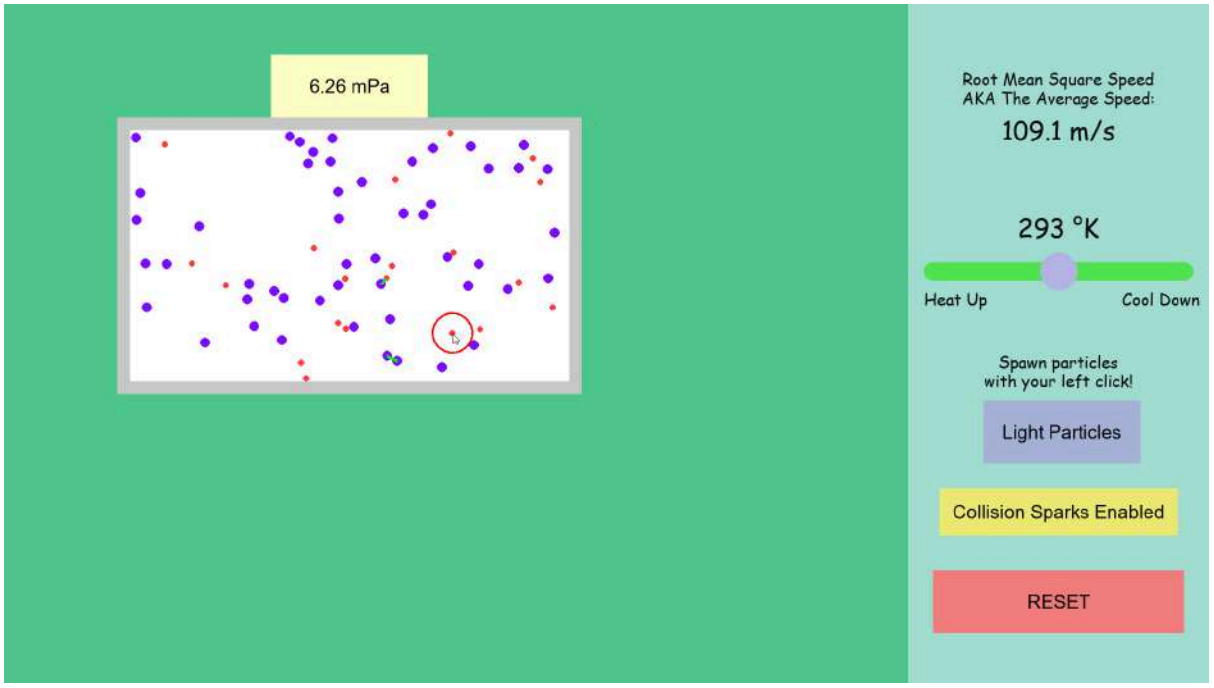
23	Dragging the ball directly into the target.	Erroneous	The ball immediately transforms into a splat and very few points are awarded	This is something that I overlooked. If the user drags the ball into the target, it is likely that they will have a moment where the cursor stays in the same place for two time steps. This registers as a hit. However, often very few points are awarded as I struggled to make it far into the target.	4.6

Category 5 - Ideal Gas Law

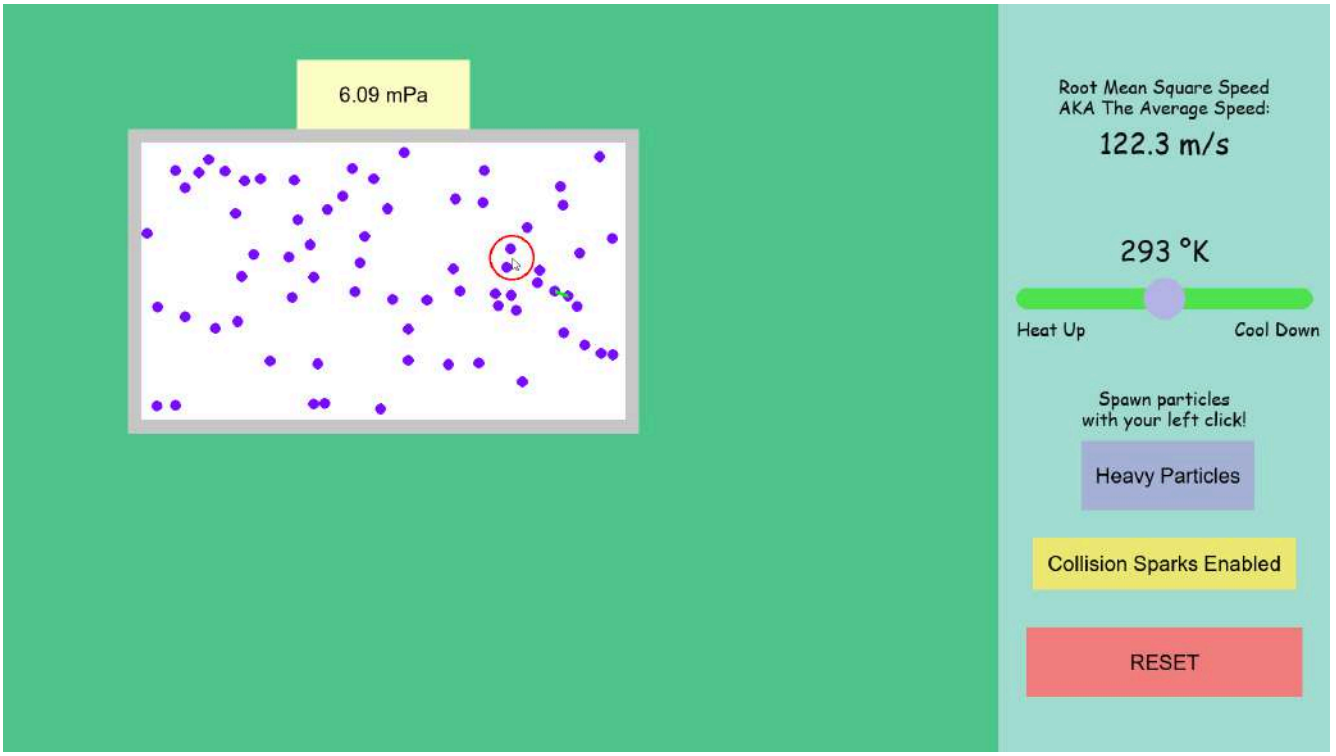
#	Description	Test Data/Type	Outcome	Additional Comments	Objectives Met
1	Starting the simulation	Normal	Container loads with default values	Loads correctly.	5.2 5.7



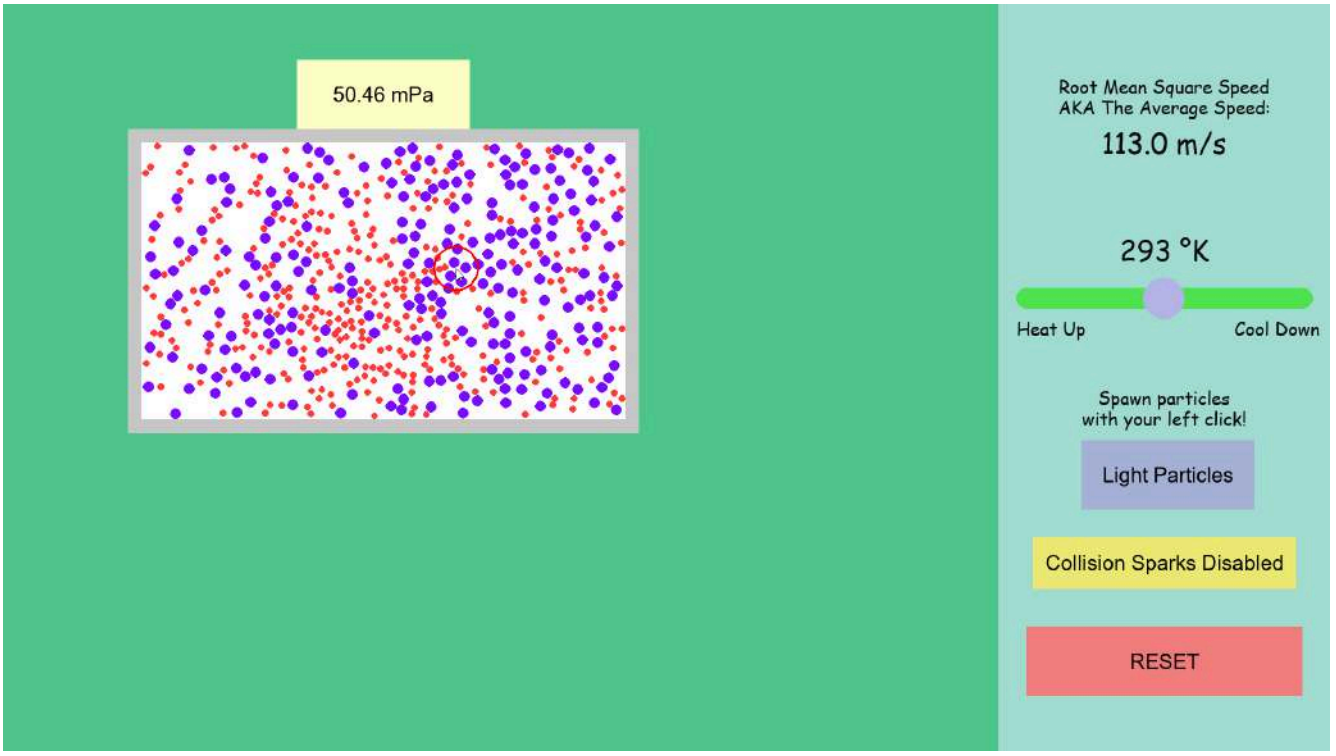
2	Adding light particles	Normal	Light particles with a small mass and radius spawn at the cursor after a mouse click	Works as expected	5.1 5.6



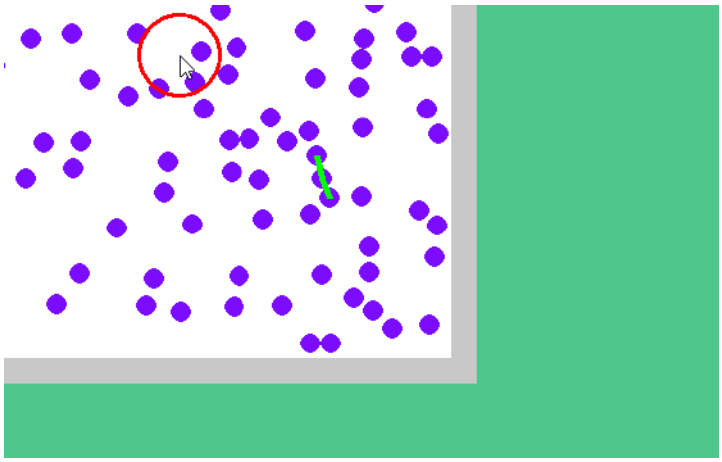
3	Adding Heavy Particles	Normal	Heavy particles with a larger mass and radius spawn at the cursor after a mouse click	Works as expected	5.1 5.6



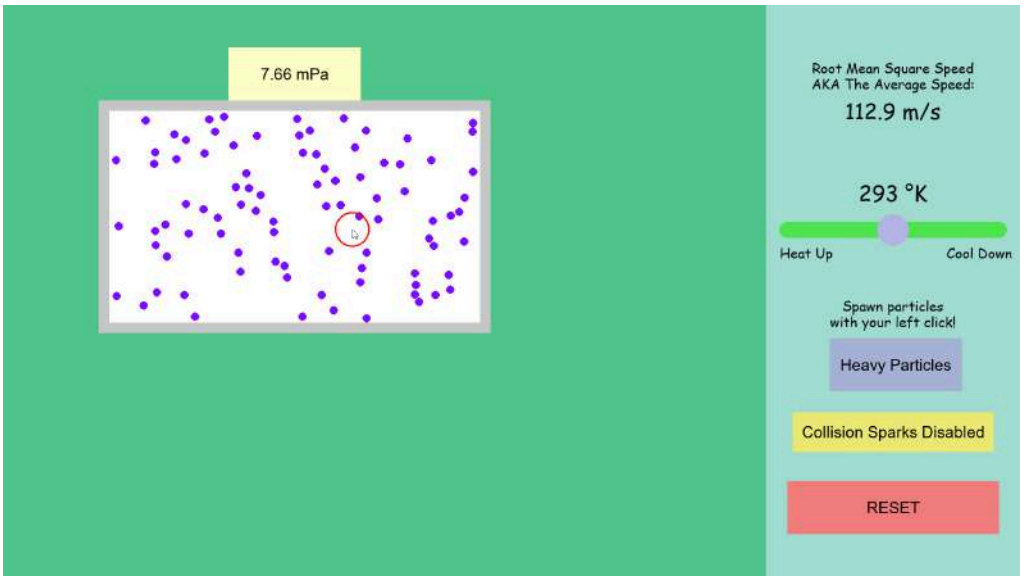
4	Adding particles outside of container	Erroneous	Nothing happens	The function will only trigger if the mouse cursor is within the wall dimensions	
5	Adding an excessive number of particles		The simulations reduces to a very small framerate	Unlike with the pathfinder, every single collision gets resolved with each time step (provided there is enough space). As such, the number of particles has a great effect on efficiency	5.6



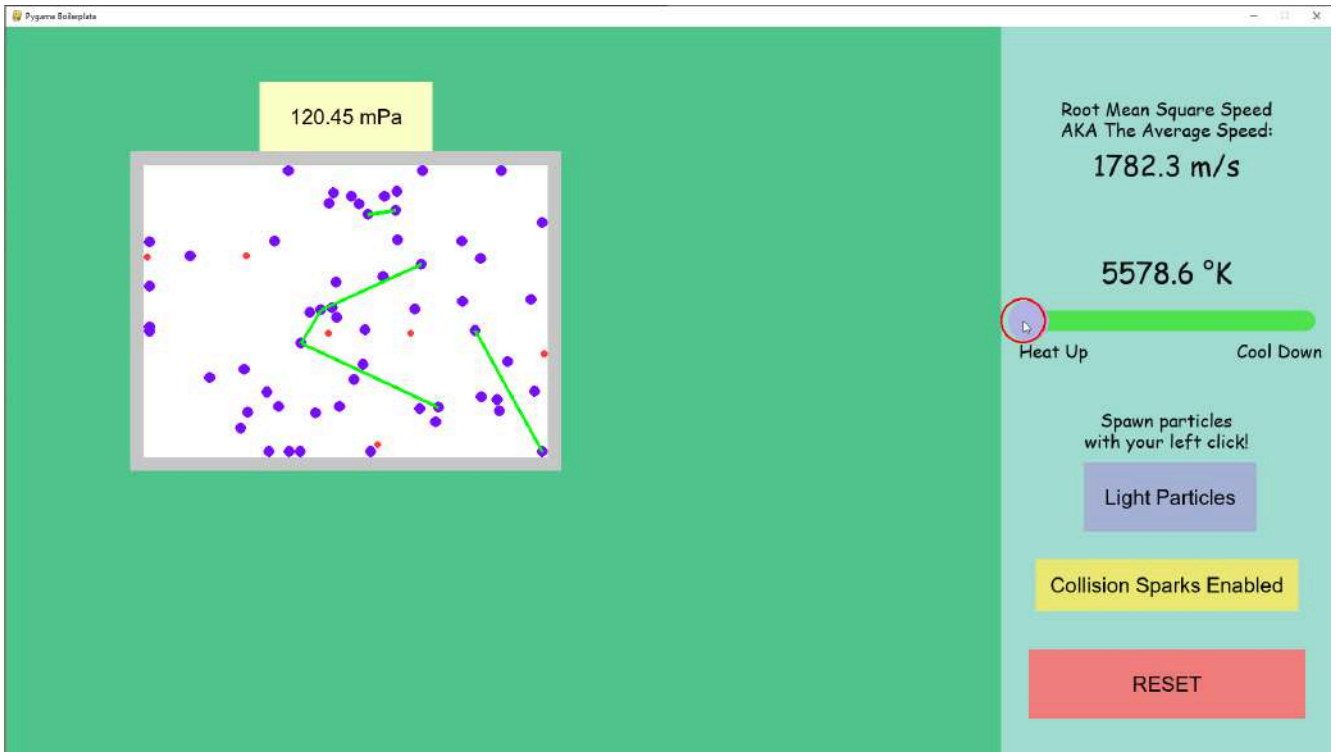
6	Enabling Collision Sparks	Normal	Little green sparks are shown for a single time step between currently colliding particles.	This helps to highlight the collisions to the user	5.7



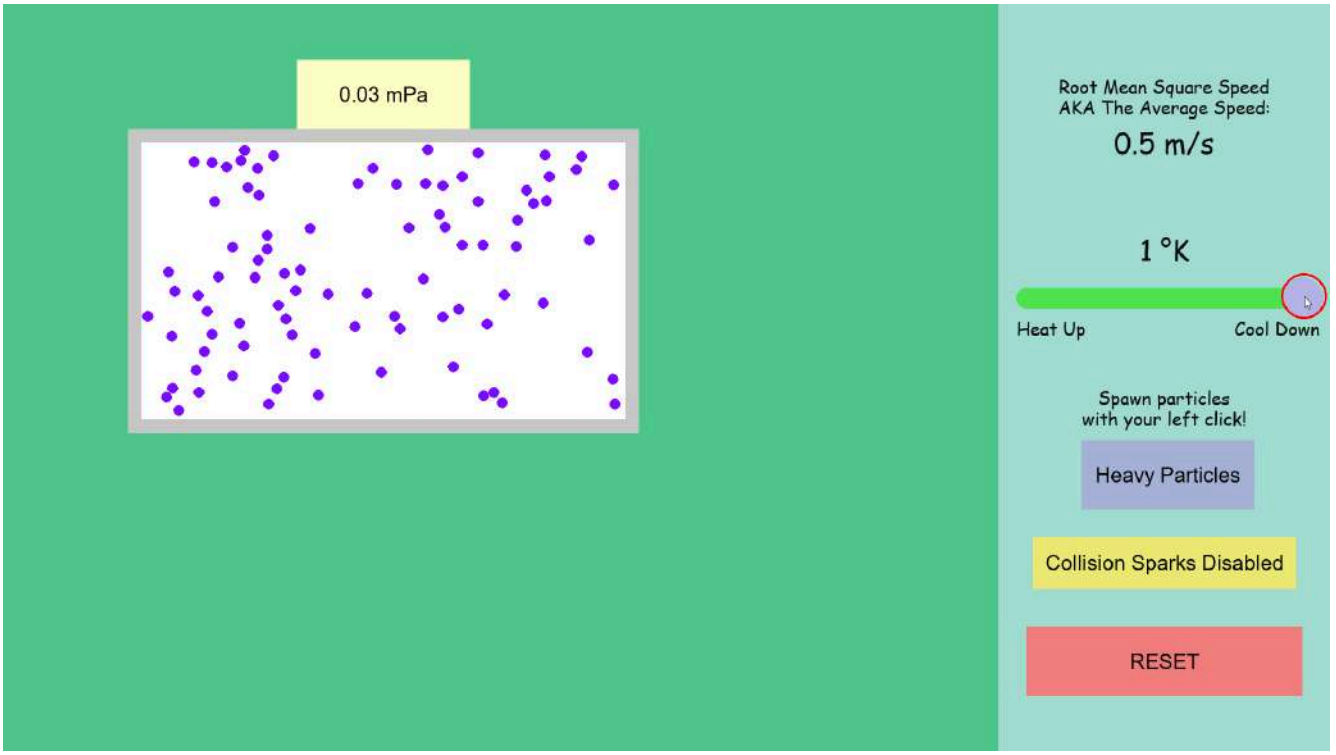
7	Disabling Collision Sparks	Normal	No visual embellishments are shown for a collision.	Works as intended	5.7



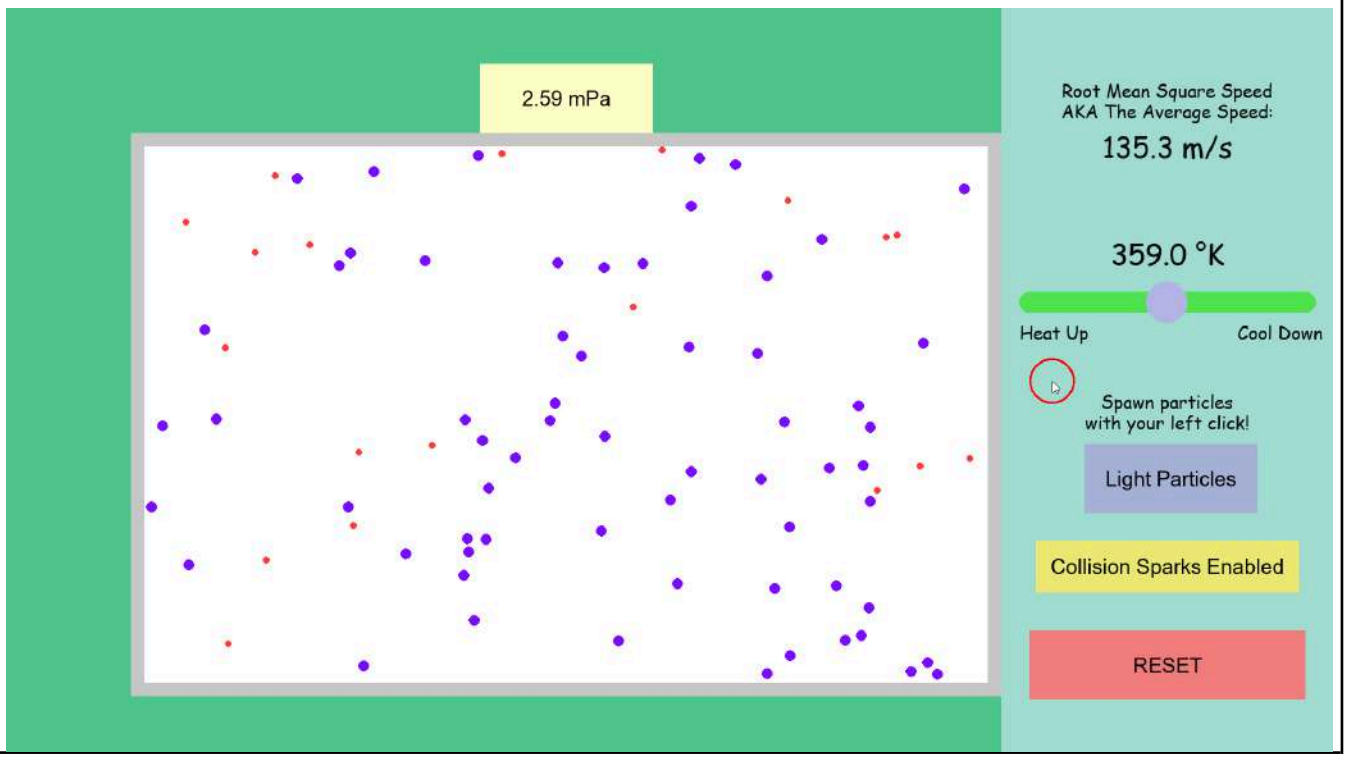
8	Increasing temperature to very high temperatures	Erroneous	Particles move very fast and appear to teleport from one end of the container to the other side. Graphical inaccuracies relating to the collision sparks are very common	It takes a very, very long time to reach the temperatures at which the simulation breaks down. Note that if the user decides to reduce the temperature back down to a more reasonable number, it will be completely fine.	5.5 5.2



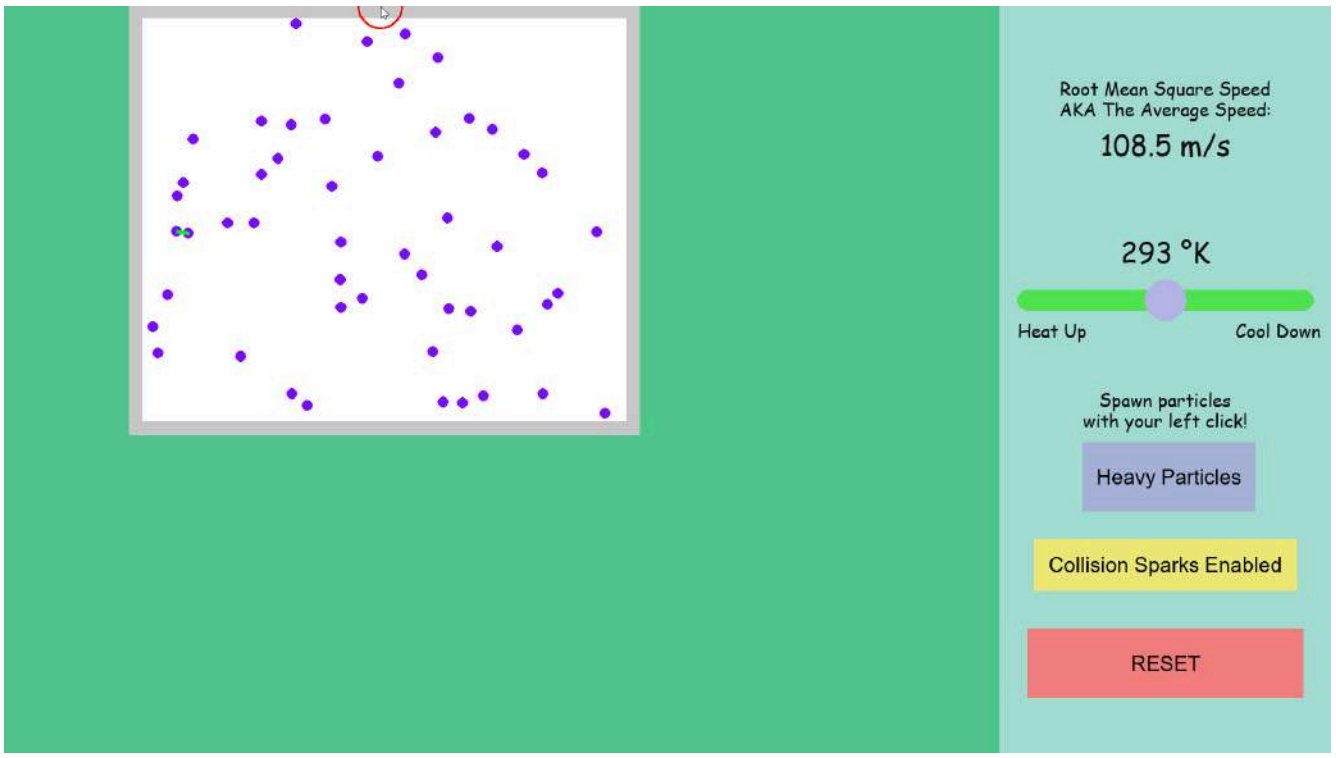
9	Attempting to reduce temperature to and below 0 K	Erroneous	The temperature is restricted to a minimum of 1 K	In real life, the temperature must be above zero as kelvin is an absolute scale.	5.5 5.2



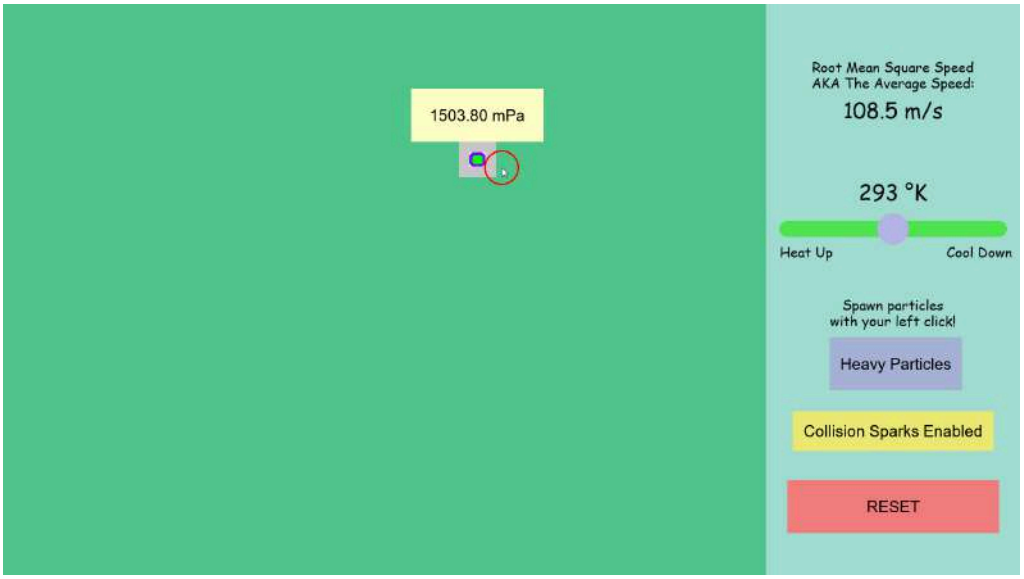
10	Trying to expand container into menu region	Erroneous	The right wall does not follow the cursor into the menu region	I used some validation here to prevent issues.	5.2 5.4



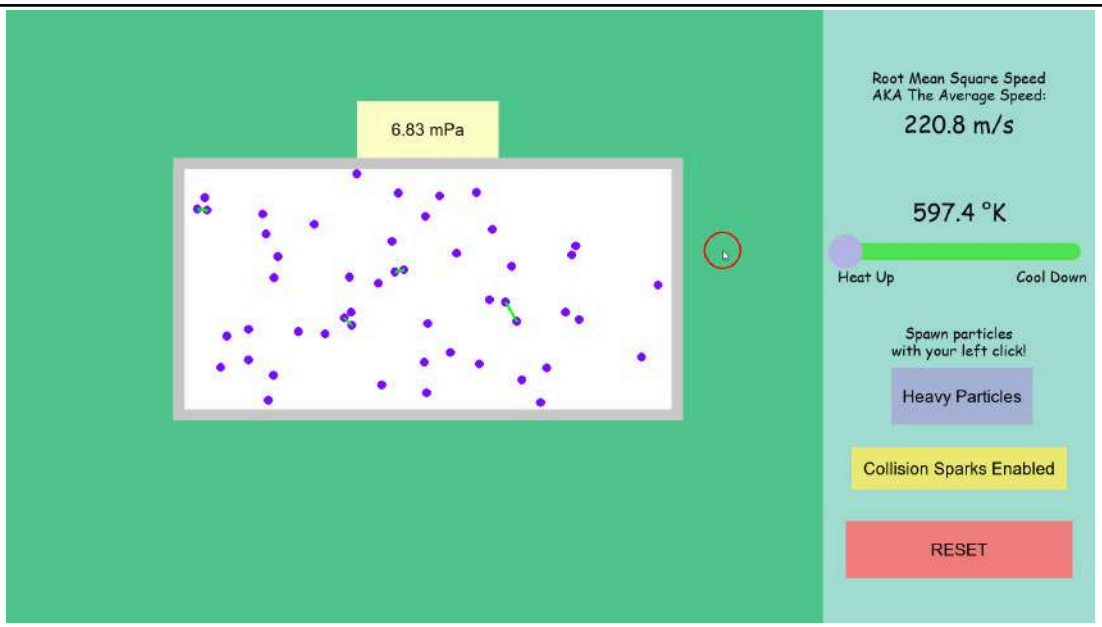
11	Attempting to push right wall into left wall	Erroneous	The right wall stops just before and will not go any further left	Works as intended. There is a gap between the walls, albeit very small. The program still runs smoothly, though the visuals are questionable at this size. For example, the particles are overlapping and thus always colliding, meaning that constant collision sparks are shown.	5.2 5.4
					
12	Pushing upper wall above screen	Erroneous	Pressure display is not visible	Fail. The pressure display is no longer visible as it is above the display. Moving the upper wall down makes the display visible again.	5.4 5.7



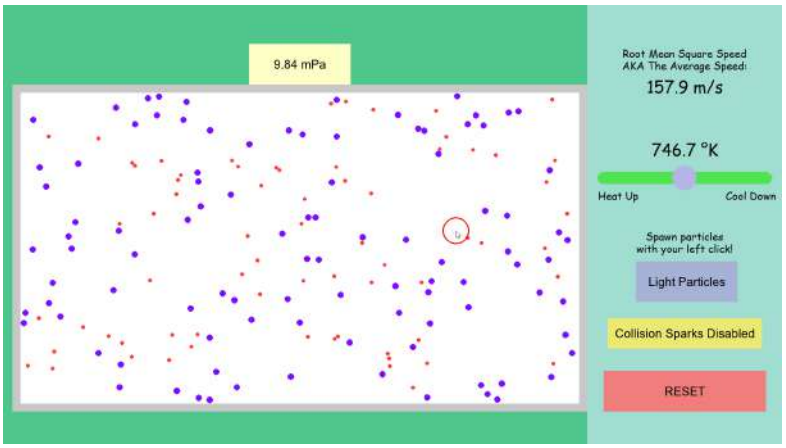
13	Making container very small	Erroneous	The program does not error. The container does not shrink anymore than the below. The particles become indistinguishable	Whilst the size of the container means that there is not much to see, the pressure value still remain accurate	5.4



14	Trying to pull slider knob away from its allowed region	Erroneous	The knob does not come off the bar.	Validated correctly	5.2 5.5

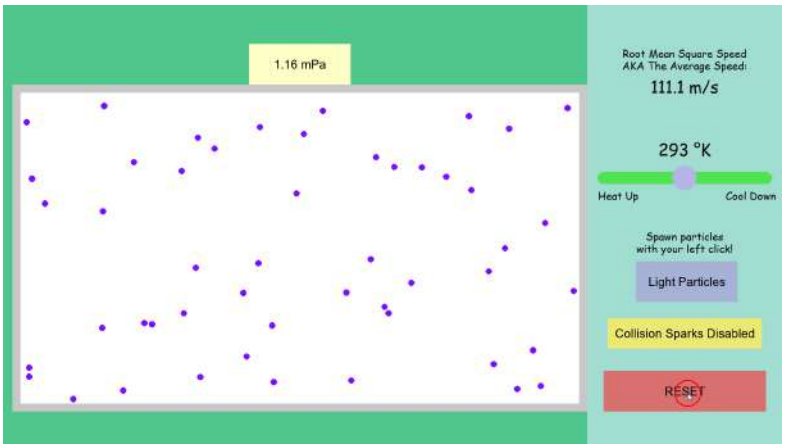


15	Resetting container		The number of particles and the temperature reset to the initial values.	Note that the container dimensions remain the same for continuity.	



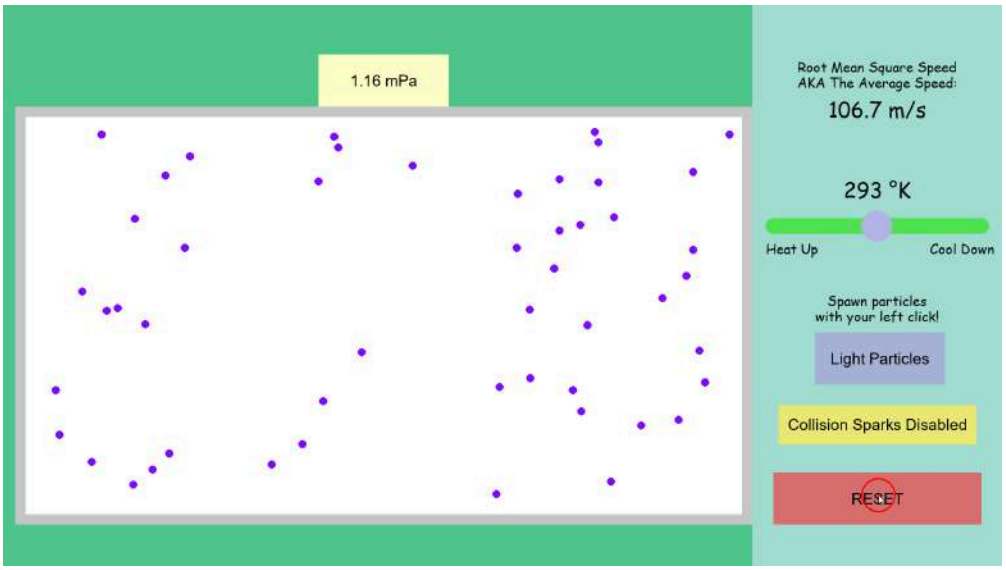
BEFORE

AFTER



16	Resetting an already reseted container	Erroneous	The root mean square velocity	Currently, particles spawn with a random velocity between a	5.2

			changes and a new set of particles spawn.	specified range. This means that the rms speed does not remain constant.	




```
base_v = 60
self.temp_slider.knob_value = self.initial_temperature
self.temperature = self.initial_temperature

self.particles.clear()
self.particles.extend([GasParticle(0.1, 8, self, position=np.array(
    [randint(dim[0], dim[2]), randint(dim[1], dim[3])]), velocity=np.array(
    [randint(-base_v, base_v), randint(-base_v, base_v)], dtype=float)) for _ in
range(50)])
```

Screen recording video of system

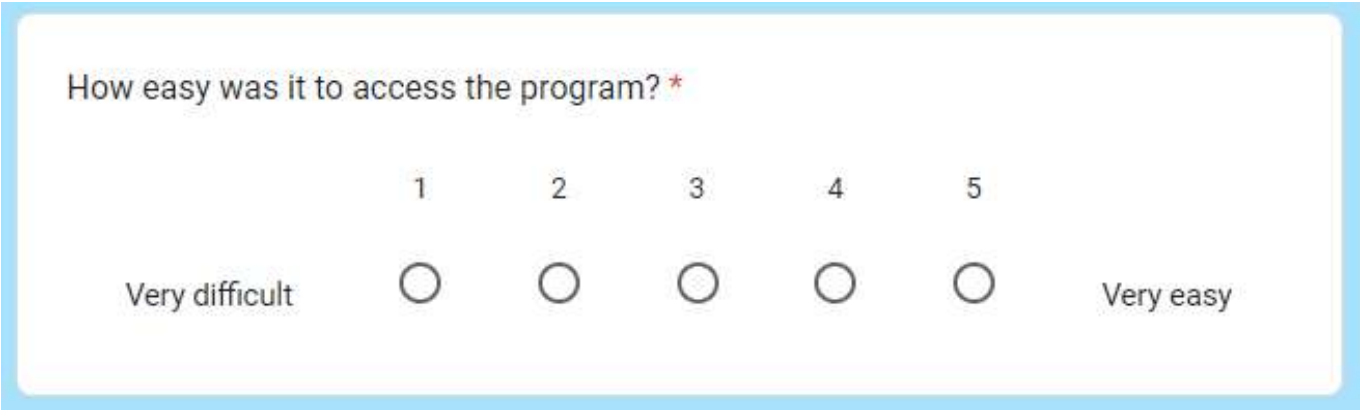
Evaluation

User feedback

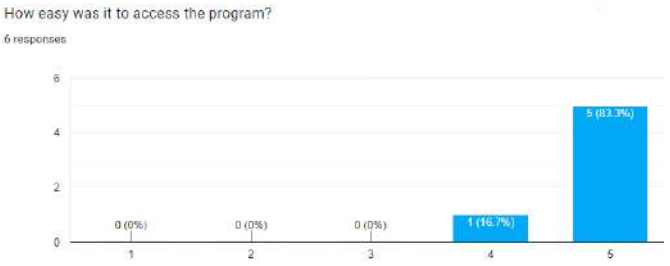
Survey Data

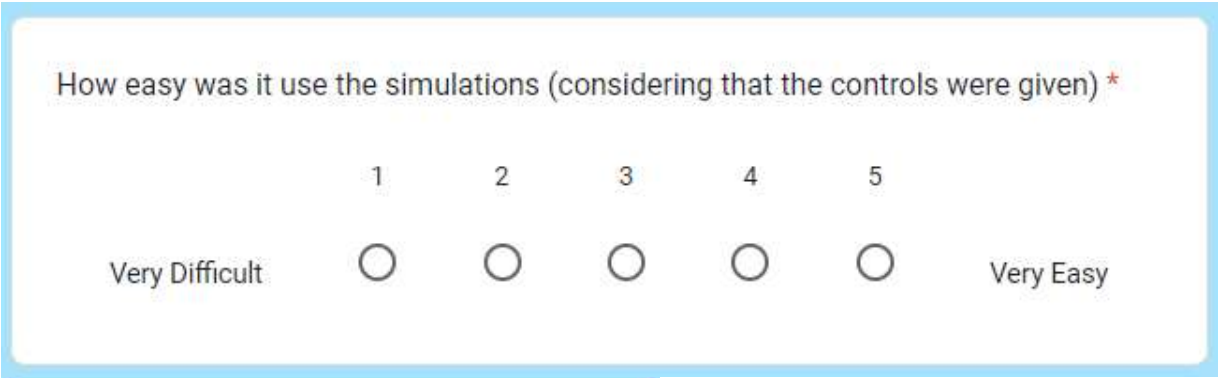
As with the requirements gathering data, I again opted to use a google form to gather feedback from those who used the system. I distributed the program to my class (of which 90% take physics, computer science, and maths) and received X responses. Although I sat with them as they used the program and got some good feedback, none of this was formally

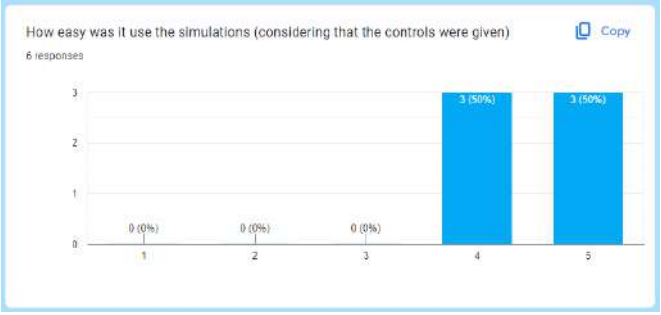
written down. As such, I decided to create a form for the users to complete after using the program. The results are detailed below



I began with asking the user how they got on with accessing the program. This referred to the login system in particular, as I was keen to find out if the account creation was straightforward enough. This received reassuringly high scores



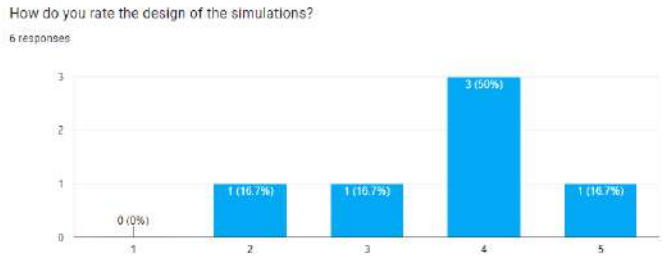




This also received good scores. However, one tester chose a low rating. When followed up on, he said that it was difficult to pick up a ball in motion. To fix this, I could increase the selection radius. Seeing as the response was generally very positive, I feel as though this was a more isolated case

How do you rate the design of the simulations? *

	1	2	3	4	5	
Very Poor	○	○	○	○	○	Very Good



Though still high, it was the lowest scoring question with an average of 3.7. Tester said that they would have preferred more images in the simulations to create a more punchy theme. A user with a particularly high resolution display said that the projectile motion simulation did not expand accordingly with the display. However, both the ideal gas law simulation and in particular the pathfinder functioned perfectly. The majority of the criticism came from the poorly styled and largely unfinished log off page.

How was your experience using the vector field pathfinder? *

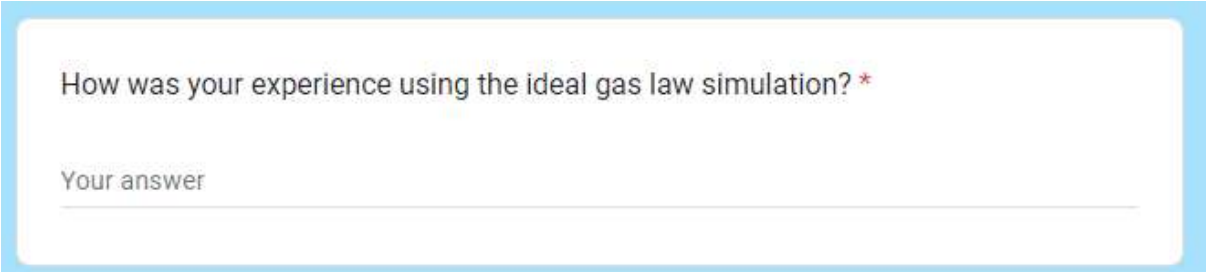
Your answer

To summarise the responses, testers said that this was the most refined simulation. The interface was very smooth and the output was near-perfect. One tester who attempted to find issues with the program commented that obstacles could be placed directly over particles.

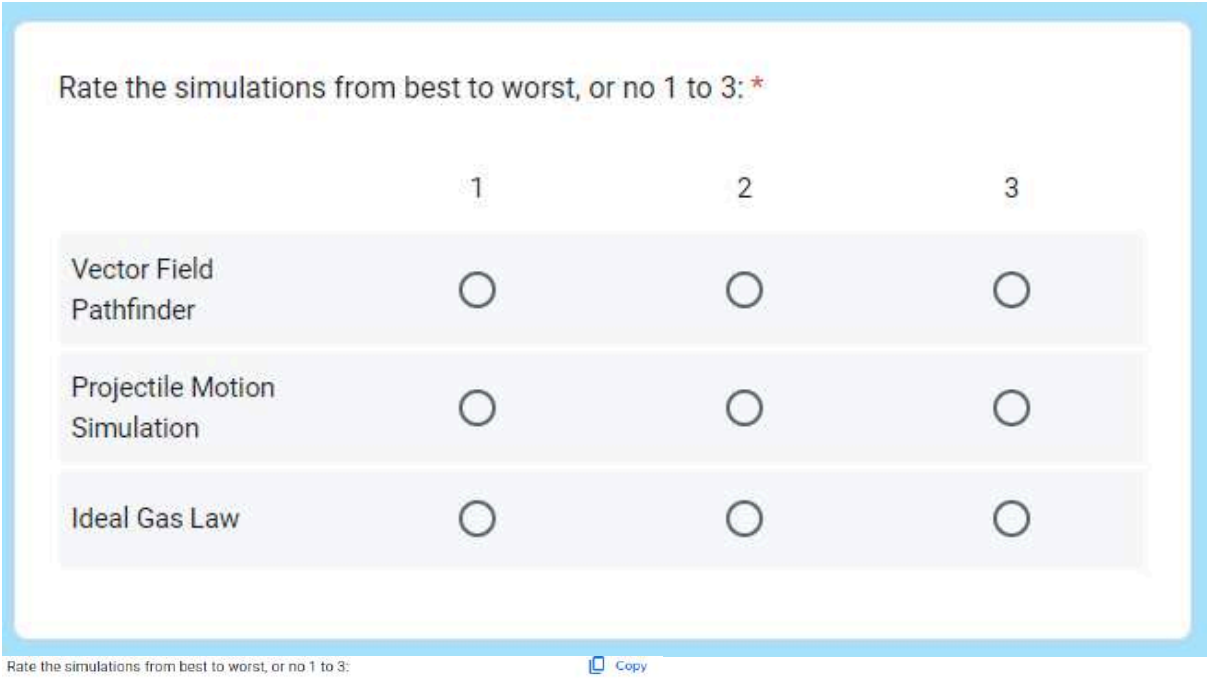
How was your experience using the projectile motion simulation? *

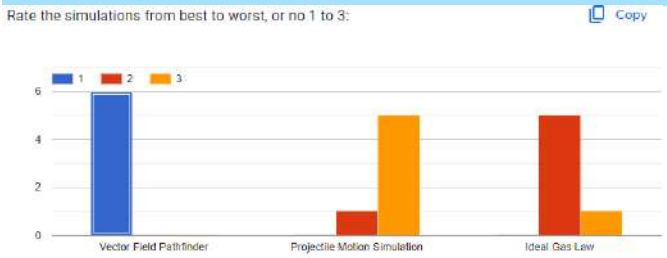
Your answer

Users had concerns over the aesthetics of the simulation, claiming that the images for the walls and platforms were disappointing. Some users commented positively on the sandbox aspect, whereby the user could move the balls around and fire in whatever direction they desired.



Responses praised the interface of the simulation, claiming that the button menu made it feel like ‘a proper school simulation’. One user claimed that they tried hard to cause an error, but was unable to.





All but two testers put the pathfinder in first, the gas simulation in second, and the projectile motion simulation in third. I contribute this to the more interactive and engaging user experience.

A disappointing number of testers did not leave a response. The few who did had largely positive words to say. One user commented that they felt the log out page containing the projectile motion leaderboard was the ‘the only thing (I) haven’t finished’.

Evaluation of how well/not well each objective has been met

Main Window

Objective Review

Objective	Testing	User Feedback
The UI should be simple and straightforward being careful to minimise overloading the user	The UI appears as such	Some users said it was perhaps too simple. Pictures could be added
The UI should be full screen to prevent the student becoming distracted by other programs	The screen appears full screen	It was full screen for all the testers.
The program display correctly for all resolutions	For the few resolutions I tested, the program was at least functioning	One tester said the projectile motion simulation displayed poorly on his device
The user should be able to create an account with the program	Success.	Testers all successfully made accounts
The user should be able to log in to a previously made account	Success.	Testers only had one session so N/A
All user inputs such as email address should be fully validated	Success. See Main Window testing. Testing revealed some flaws as discussed which have been fixed.	Users could not manage to access the system with invalid details
Any parameters the user might select through using the simulations should be stored for future use, and the program should pre-load	Success. See test 4.4	Testers only had one session so N/A

these parameters the next time they log in to the program		
A teacher should be able to have extra access rights, such as view a student's progress within their class or edit levels	Success. SeeTest 4.3	I gave a couple testers access to the admin account and everything went as expected
The user interface should provide engaging and helpful responses, for example if the user fails an email login	Success. See test 1.2, 1.15	The

Vector Field Pathfinder

Objective	Testing	User Feedback
Particles should exhibit fluid-like properties	Success. Test 3.6	One said that this was the 'coolest' simulation and that it moves like water.
The program should generate a distance field.	Majority success. Tests 3.8 and 3.15 revealed some minor issues	No user experienced the local optima problem – I only discovered it myself after many uses. A few users experienced the same issue as in test 3.15.
The program should generate an accurate velocity field	Success	Users were able to see the workings of the algorithm by changing the view mode
The user should be able to dynamically change cells into obstacles which block particles	Success. See test 3.10	Users were particularly impressed at the speed and ease of use.
The user should be able to add new particles with different properties such as mass and density which react accordingly	Success. See test 3.11	Some users felt like an indicator was needed for the current particle radius and whether the add particle mode was enabled
The program should have a strong and intuitive UI	Success, though I have lots of experience with the simulation	Users quickly picked up the controls very quickly and were manipulating the simulation with ease.

The simulation should function as expected regardless of the user's settings	Partial success. See test 3,24, 3,26, 3.4	Users experimented in depth with this simulation. Some observed graphical inaccuracies when using very obscure grid dimensions, but that is practically unavoidable
The particles should have some natural variation to prevent a blank uniformity	Success.	Users were impressed by the pseudo-fluid flow, which this feature contributed to.
The program should run efficiently with a reasonable set of parameters	Success. One unexpected bottleneck was the heatmap calculation. See tests 23 onwards + reflection	Some users saw a noticeable slow down when changing parameters like the grid size and particles. I advised them to disable the heatmap, at which point the performance dramatically improved.

Projectile Motion Simulation

Objective	Testing	User Feedback
Create balls which follow the laws of motion accurately	Success. See test 4.18	Not much to comment on.
The user should be able to project a ball with a inputted velocity	Success. See test 4.13	Some users initially missed the velocity text until I pointed it out, as it almost blended into the skies of the background.
The simulation should have walls or obstacles	Success. See test 4.10	Users criticised the aesthetics of the walls, but liked the addition of a platform from which they could fire the ball from.
The simulation should provide live data for the currently in motion projectile	Success. See test 4.16. Note that I had missed a bug which a user found (see on the right).	Users vigorously tested this system. One user found a bug whereby the timer would not stop if the ball in motion hit a particle. For example, if a ball rested on top of two particles, the timer would wrongly continue
The user should have the option for extra features such as air resistance	Success. See test 4.17	Some users fired the ball very fast and were disappointed. I then suggest they enable air resistance, where this issue was partially resolved. One other option is that I could reduce the particle_max_value value.

The user should have a reward system	Partial success. See test 4.3, where levels were locked until a suitable score was reached in the previous level. However, some issues were found with the leaderboard.	Users showed excitement at the idea of progressing through levels and even the leaderboard. This is encouraging as this simulation is meant to be fun, rather than a chore.
The simulation should be easy to use	N/A	The majority of users found the controls straightforward. Note that many users had not seen a simulation like this before, where the user was given full control over the ball.

Ideal Gas Law

Objective	Testing	User Feedback
The particles should follow the same rules as outlined in the ideal gas law	Success.	Not much to comment on.
The simulation should have an organised and validated interface	Success.	Users liked the design with the menu bar, with one stating it felt like a ‘proper school simulation’.
A pressure value should be given to the user in a sensible format	Success. Though the pressure display could be pushed up outside the screen.	One user commented how it was ‘cool’ that the pressure display moved with the container.
The user should be able to change the volume of the box	Success. See test 10	I noticed that users were thorough in playing with the volume. The only issue encountered was met in test 12.
The user should be able to change the temperature of the box	Success. See test 9. Only criticism is that there is no upper limit.	Users did not encounter any issue with the slider. No student attempted the same test as test 9, which suggests that it is an extreme case that only a very unkeen student would attempt.
The user should be able to add more particles to the box	Success. See test 2, 3	Users were able to successfully add particles. One user noticed the disparity between this and the

		pathfinder. In the pathfinder, you can hold down the cursor to continuously add particles, whereas here it is one at the time. To ensure consistency, I will look to adapt this in the future.
The properties of the box should be shown to the user in an easy to read format	Subjective but I believe in success.	One user asked to test the simulation's mathematics, but was unable to do so without first working out the volume units through the ideal gas law. Therefore, I could straight up give the total volume to the user.

Possible Improvements

Due to the nature of the project with the variety of simulations, there is a mix of potential improvement that can be made. To keep it organised, I have detailed potential improvements according to the simulation.

Miscellaneous

1. The most glaring failure is the fluid flow simulation. I believe that I was not too far off in the pieces clicking together but I had encountered countless errors which meant progress was painfully slow. As such, I felt I had to prioritise other simulations so that I could actually provide something of value. Firstly, I would have to fix a logic error I think with the spatial map – it was not arranged correctly meaning that when particles had to be searched, some cells and so particles were not considered. From there, I could go about implementing the viscosity force and perhaps gravity as well. Beyond that, I could then enable the user to adjust parameters like the number of particles, or allow the user to place an outward force from the cursor so that they might observe the effect. However, this was my very first venture and a few major refactors have taken place. I believe that with considerably more time, I would be able to reintegrate it into the program and begin to produce a physically pure output.

Projectile Motion Leaderboard

1	2	3
1	Sam Hoskins	582
2	Joe Cole	462
3	John Travolta	324
4	dfdsafsd	277
5	Nam Nam	154
6	Joe Bloggs	73
7	ghgigigjjig	0
8	asdffadssfafsad	0
9	gvjg	0
10	asdffsad	0
-	New User	0

2. While I am for the most part pleased with the functionality of the user interface, I believe that it has room for improvement. Currently, there is no option for the user to log out without the program closing. A tester who was a teacher stated that they made a student account but then was unable to switch into their teacher account without restarting the program.
3. I believe that there is a lot more potential for the user settings side of the program. I only had time to implement one leaderboard for the projectile motion simulation. Thanks to the database structure, it is fairly simple to create more tables like these, like for the last

times users have logged in. I could also make it so that only users with access rights (i.e. teachers) would be able to view these tables. This could enable teachers to track a student's usage and encourage the students as needed

4. As shown in the user feedback, the aesthetics received disappointing scores. The design of the user interface seems to have been too unrefined. To improve on this, I could add images to create a more punchy tone and work on the styling.

Vector Field Pathfinder

I believe this to be the most refined simulation; it is after all the simulation I managed to complete first. Everything that I sought to achieve was done so, apart from a few little bugs mentioned in testing. In my opinion, the next step to further the simulation experience is to render the particles to generate a similar output to the right. Here, the output is not a set of discrete particles, rather it's more like a fluid.

Projectile Motion

1. In A-Level questions, a common talking point is the maximum height an object has reached. This along with other variables could be added to the information table which could enhance learning.

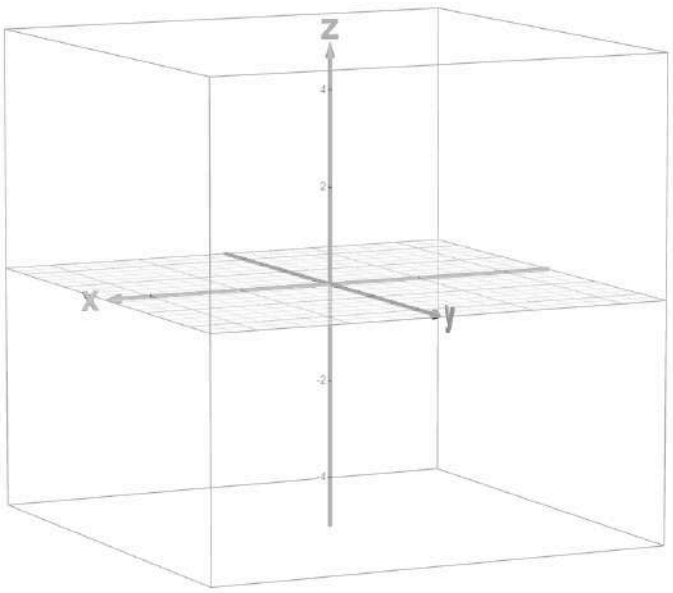
2. Changes could be made to the level to make it feel more dynamic. Currently, the user can see the entire map including the target. This means they can use trial and error to work out the general velocity needed to reach the target without using the mathematical concepts. One suggestion to fix this is to have a wide map in which the camera would pan side-to-side to track the ball's current position, similar to the mobile game 'Angry Birds'. This would increase the immersion of the game and ensure that the student does the necessary working out using kinematics. Alternatively, taking inspiration from 'Angry Birds' again, I could implement a projection line which dictates the path of that ball. The user could observe the effect of the inputted velocity without actually firing the ball, meaning they could fine-tune to ensure a perfect fire.

$$V_t = \sqrt{\frac{2mg}{\rho A C_d}}$$

3. When considering exam questions, terminal velocity is often not considered. However, in line with the real world, I could implement this. In real life, a particle will not accelerate downwards to an infinite speed from the force of gravity. Instead, it will reach an equilibrium point where the force due to gravity is equal to air resistance, and will therefore stop accelerating. To implement this, I would have to calculate the terminal velocity according to the formula and set a maximum speed for the ball.

Ideal Gas Law

1. The appearance of the particles should be adjusted according to their individual properties. For example, the user could have the option for faster particles to be shown in red and slower particles in blue. This would provide more clarity to the simulation. A common misconception with the ideal gas law is that the particles all travel at a uniform speed. However, the speed suggested is merely the average speed of the particles.



2. A third dimension could be added to the container to properly simulate the real world. Currently, the simulation is restricted to 2 dimensions. This can lead a student into another misconception. Given more time, I could implement a 3D solution where the same concepts would apply, but with particles also being able to move in the z-axis. One 3D view model that I would be keen to replicate is the desmos 3D graphing tool showing on the right. Here, I can accurately zoom and rotate with just the mouse to view a specific region. Moreover, the system defaults to automatic rotation to ensure that the user is aware of its 3D nature. Provided I am able to replicate this model, I would have to add another component to the position and velocity vectors, and then upgrade the collision system.

References

[1] Navier-Stokes equations video (See Modelling → Fluid Flow Research)
https://www.youtube.com/watch?v=ERBVFcutI3M

[2] Key papers used for fluid flow (Used extensively throughout fluid flow research
https://matthias-research.github.io/pages/publications/sca03.pdf
https://sph-tutorial.physics-simulation.org/pdf/SPH_Tutorial.pdf

[3] Inspiration and resources for fluid flow thanks to Sebastian Lague
https://www.youtube.com/watch?v=rSKMYc1CQHE&t=409s

[4] Steering behaviours game logic (See vector field pathfinder)
https://code.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732

[5] Particle collision theory (Used for 'Collisions' in 'Key Algorithms')
https://www.youtube.com/watch?v=LPzyNOHY3A4

[6] Understanding goal based vector field pathfinding
https://code.tutsplus.com/understanding-goal-based-vector-field-pathfinding--gamedev-9007t

[7] Drag Explanation (See projectile motion simulation)
https://en.wikipedia.org/wiki/Drag_(physics)

[8] Article used to justify end user's technical proficiency in Analysis
https://www.southernphone.com.au/blog/tech-savvy-toddlers-why-are-young-children-so-good-with-technology

[9] University of Texas Ideal Gas Law Simulation discussed in Current Systems Research
https://ch301.cm.utexas.edu/simulations/js/idealgaslaw/

[10] Terminal velocity equation (pg 184)
https://www.facebook.com/ScientificEquationsDaily/

[11] All graphs produced were using desmos.com

Video music in Testing

Appendix

Requirements Gathering Survey – Raw Data

Please enter your age group	For data quality purposes, please select option 2	What are you currently studying	**Are you familiar with or at least encountered any of the following:** Fields Fluid flow Mechanics Ocean currents Projectile Motion Pathfinding	Which techniques do your school employ when teaching any of the aforementioned topics	I am creating a physics simulation program. Would you say learning topics through visual simulations increase your efficiency and productivity?
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Practical Demonstrations	Yes
Sixth Form	Option 2	Any STEM subjects, Any Non-STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Visual Simulations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects, Any Non-STEM subjects	Yes	Powerpoint Presentations, Practical Demonstrations	Yes
Sixth Form	Option 2	Any Non-STEM subjects	Yes	Powerpoint Presentations, Visual Simulations, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Visual Simulations, Practical Demonstrations	Yes
Undergraduate	Option 2	Any STEM subjects	Yes	Textbook Questions, Visual Simulations, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics	Yes	Powerpoint Presentations, Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions	Yes
Junior	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Any STEM subjects	Yes	Textbook Questions, Visual Simulations, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Any STEM subjects, Any Non-STEM subjects	Yes	Powerpoint Presentations, Textbook Questions	No
Junior	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions	Yes
Sixth Form	Option 2	Physics, Any STEM subjects, Any Non-STEM subjects	Yes	Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions	Yes
Sixth Form	Option 2	Any STEM subjects	Yes	Textbook Questions, Visual Simulations, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics	Yes	Powerpoint Presentations, Visual Simulations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Visual Simulations, Practical Demonstrations	No
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions, Visual Simulations, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions, Visual Simulations	Yes
Sixth Form	Option 2	Any STEM subjects	Yes	Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Visual Simulations, Practical Demo	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions, Visual Simulations, Practical Demonstrations	Yes
Undergraduate	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions, Visual Simulations, Practical Demonstrations	Yes
Sixth Form	Option 2	Physics, Any STEM subjects	Yes	Textbook Questions, Practical Demonstrations	Yes
Sixth Form	Option 2	Any STEM subjects, Any Non-STEM subjects	Yes	Powerpoint Presentations, Textbook Questions, Visual Simulations	Yes
Sixth Form	Option 2	Any STEM subjects	Yes	Textbook Questions, Practical Demonstrations	Yes

I am creating a physics simulation program. Would you say learning topics through visual simulations increase your efficiency and productivity?	What topics do you struggle with that you feel a visual simulation may help with?	And now rank the most helpful teaching methods from 1 - 4, with 1 being the most helpful [Powerpoint Presentations]	And now rank the most helpful teaching methods from 1 - 4, with 1 being the most helpful [Textbook Questions]	And now rank the most helpful teaching methods from 1 - 4, with 1 being the most helpful [Visual Simulations]	And now rank the most helpful teaching methods from 1 - 4, with 1 being the most helpful [Practical Demonstrations]	Does your school offer regular practical demonstrations alongside teaching for each topic?
Yes	Vector Field Pathfinder, Gravitational Orbits, Ideal Gas Law, Projectile Motion	4	2	3	1	No
Yes	Vector Field Pathfinder, Fluid Flow, Projectile Motion	3	4	2	1	Yes
Yes	Vector Field Pathfinder, Gravitational Orbits, Projectile Motion, Interactive Electrical Circuits	4	1	2	3	No
Yes	Vector Field Pathfinder, Fluid Flow	3	4	2	1	No
Yes	Vector Field Pathfinder, Ideal Gas Law, Fluid Flow, Projectile Motion	4	1	3	2	No
Yes	Vector Field Pathfinder, Ideal Gas Law, Fluid Flow	4	3	1	2	Yes
Yes	Vector Field Pathfinder, Gravitational Orbits, Ideal Gas Law, Fluid Flow, Projectile Motion, Interactive Electrical Circuits	4	3	2	1	No
Yes	Ideal Gas Law, Fluid Flow, Projectile Motion	1	4	2	3	No
Yes	Vector Field Pathfinder, Ideal Gas Law, Fluid Flow, Projectile Motion	2	4	3	1	No
Yes	Ideal Gas Law, Projectile Motion	4	2	1	3	Yes
Yes	Vector Field Pathfinder, Ideal Gas Law	3	4	2	1	No
Yes	Gravitational Orbits, Ideal Gas Law	4	2	1	3	No
Yes	Vector Field Pathfinder, Ideal Gas Law	4	3	2	1	No
Yes	Vector Field Pathfinder, Ideal Gas Law, Fluid Flow	4	2	3	1	No
No	Vector Field Pathfinder	4	3	1	2	No
Yes	Vector Field Pathfinder	4	3	2	1	No
Yes	Vector Field Pathfinder	3	2	4	1	No
Yes	Vector Field Pathfinder, Projectile Motion	4	2	1	3	No
Yes	Vector Field Pathfinder, Ideal Gas Law	1	4	2	3	No
Yes	Vector Field Pathfinder, Gravitational Orbits, Ideal Gas Law, Fluid Flow, Projectile Motion, Interactive Electrical Circuits	4	2	1	3	Yes
Yes	Gravitational Orbits, Ideal Gas Law, Fluid Flow, Projectile Motion, Interactive Electrical Circuits	4	3	2	1	No
No	Gravitational Orbits, Ideal Gas Law, Fluid Flow, Projectile Motion	2	3	1	4	Yes
Yes	Gravitational Orbits, Ideal Gas Law, Fluid Flow, Projectile Motion	4	3	1	2	No
Yes	Gravitational Orbits, Ideal Gas Law, Projectile Motion	3	4	2	1	No
Yes	Ideal Gas Law, Fluid Flow, Projectile Motion	4	1	3	2	No
Yes	Ideal Gas Law, Fluid Flow, Projectile Motion	4	1	3	2	No
Yes	Vector Field Pathfinder, Ideal Gas Law, Projectile Motion	3	4	1	2	No
Yes	Vector Field Pathfinder, Projectile Motion	3	4	2	1	No
Yes	Vector Field Pathfinder, Ideal Gas Law, Fluid Flow	3	4	2	1	No
Yes	Vector Field Pathfinder, Interactive Electrical Circuits	4	3	1	2	No
Yes	Vector Field Pathfinder, Gravitational Orbits, Ideal Gas Law, Fluid Flow, Projectile Motion, Interactive Electrical Circuits	4	3	2	1	No

Summary Table

	User Ranking Total - Smaller scores are preferred	No. of respondants who are actively using this technique
Textbook Questions	88	25
Powerpoint Presentations	106	18
Visual Simulations	60	14
Practical Demonstrations	56	22

Desired Simulations - Greater score is preferred	
Vector Field Pathfinder	22
Gravitational Orbits	10
Ideal Gas Law	22
Fluid Flow	16
Projectile Motion	19
Interactive Electrical Circuits	6

	Yes	No
Does school offer regular practical demonstrations?	5	26
Would a visual simluation increase efficiency and productivity?	29	2
Is the user familiar with any of the related topics?	31	0

Survey Demographic	
Age Group	
Junior	2
Sixth Form	27
Undergraduate	2

Currently Studying	
Physics	22
Any STEM subjects	28
Any Non-STEM subjects	6

Full Code

baseClasses.py

```
# These classes form the foundations for all simulations
import numpy as np
from random import randint


class Particle:
    def __init__(self, mass, _radius, container, position=None, velocity=None):
        self.damping = 0.8  # energy loss factor after a collision
        self.radius = _radius  # size of the particle
        self.mass = mass  # particle mass
        self.container = container  # the container which the particle is bound to

        self.velocity = np.zeros(2, dtype=float) if velocity is None else velocity
        if position is not None:
            self.position = position
        else:
            width, height = self.container.screen_width, self.container.screen_height
            self.position = np.array(
                [randint(self.radius, width - self.radius), randint(self.radius, height -
self.radius)], dtype=float)
        self.next_position = self.position.copy()  # typically used for collision detection
        if self.container:
            self.container.insert_particle(self)  # into the spatial map

    def collision_event_obstacles(self):  # Used for collision detection with Obstacle Object
        for obstacle in self.container.obstacles:
            if self.check_obstacle_collision(obstacle.position, obstacle.width,
obstacle.height):  # if collision
                self.resolve_obstacle_collision(obstacle)  # Put particle into a valid
position
                return True
        return False

    def resolve_obstacle_collision(self, obstacle, is_object=True):  # resolve particle
position relative to obstacle
        if is_object:  # If obstacle is an object of class Obstacle
            position = obstacle.position
            width, height = obstacle.width, obstacle.height
            is_platform = obstacle.is_platform
        else:  # If obstacle is a list of values instead, parse it
            is_platform = False
            position = obstacle[0]
            width, height = obstacle[1], obstacle[2]

        # calculating displacement vector from the rectangle to the circle
```

```
        # unlike particle collisions, the vector must be either vertical or horizontal
        displacement = self.position - np.array([max(position[0], min(self.position[0],
position[0] + width)),
                                                 max(position[1], min(self.position[1],
position[1] + height))])

        # needed to determine how ball got into the rectangle
        penetration_x = max(0, self.radius - abs(displacement[0]))
        penetration_y = max(0, self.radius - abs(displacement[1]))

        # direction of displacement
        direction_x = 1 if displacement[0] > 0 else -1
        direction_y = 1 if displacement[1] > 0 else -1

        # if platform, have greater damping
        if is_platform:
            damping = 0.4
        else:
            damping = self.container.damping

        # determines if ball moves horizontally or vertically
        if penetration_x < penetration_y:
            # resolving position
            self.next_position[0] += penetration_x * direction_x
            # reversing velocity
            self.velocity[0] *= -1 * damping
            if is_platform:
                self.velocity[1] *= damping
        else:
            self.next_position[1] += penetration_y * direction_y
            self.velocity[1] *= -1 * damping
            if is_platform:
                self.velocity[0] *= damping

        if np.isclose(self.velocity[1], 0, atol=2):
            self.velocity[1] = 0

    def check_obstacle_collision(self, obstacle_pos, width, height, custom_radius=None):

        closest_x = max(obstacle_pos[0], min(self.next_position[0], obstacle_pos[0] + width))
        closest_y = max(obstacle_pos[1], min(self.next_position[1], obstacle_pos[1] + height))

        square_distance = (self.next_position[0] - closest_x) ** 2 + (self.next_position[1] -
closest_y) ** 2

        if not custom_radius:
            return square_distance < self.radius ** 2
        return square_distance < custom_radius ** 2

    def collision_event(self, save_collision=True):  # handles particle collisions
        for particle in self.container.particles:
            if self.is_collision(particle, save_collisions=save_collision):
                self.resolve_static_collision(particle)

    def is_collision(self, next_particle, save_collisions=True):  # checks for a particle
collision
        distance = self.container.get_square_magnitude(next_particle.next_position -
self.next_position)
        if self != next_particle:  # A particle should collide with itself
            # if distance between particles is less than the two radii, then there has been
```

```
a collision. Note that if
            # a > b, then a^2 > b^2. Sqrt is an expensive function, so we improve efficiency
by using square distances
            if 0 < distance <= (self.radius + next_particle.radius) ** 2:
                if save_collisions:
                    self.container.colliding_balls_pairs.append((self, next_particle))
                return True  # collision detected
        return False  # no collision detected

    def resolve_static_collision(self, next_particle):  # Resolve particle position in
collision event
        # Finding magnitude of vector
        distance = self.container.get_magnitude(next_particle.next_position -
self.next_position)
        overlap = 0.5 * (distance - (self.radius + next_particle.radius))

        # Finding direction vector and applying final vector
        self.next_position -= overlap * (self.next_position - next_particle.next_position) /
distance
        next_particle.next_position += overlap * (self.next_position -
next_particle.next_position) / distance

        # Sorting inaccuracies with dividing
        if np.isclose(self.velocity, np.zeros_like(self.velocity), atol=1).all():
            self.velocity = np.zeros_like(self.velocity)

    def resolve_dynamic_collision(self, next_particle):
        normal = self.container.normalise_vector(next_particle.next_position -
self.next_position)
        tangent = np.array([-normal[1], normal[0]])

        # oblique collisions in vector form
        tangential_vel_i = tangent * np.dot(self.velocity, tangent)
        tangential_vel_j = tangent * np.dot(next_particle.velocity, tangent)

        # using formula
        normal_vel_i = normal * ((np.dot(self.velocity, normal) * (
                self.mass - next_particle.mass) + 2 * next_particle.mass *
np.dot(next_particle.velocity,
                                                                                   normal)) / (
                                         self.mass + next_particle.mass))
        normal_vel_j = normal * ((np.dot(next_particle.velocity, normal) * (
                next_particle.mass - self.mass) + 2 * self.mass * np.dot(self.velocity,
normal)) / (
                                         self.mass + next_particle.mass))

        self.velocity = tangential_vel_i + normal_vel_i  # Combining the two parallel and
perpendicular components
        next_particle.velocity = tangential_vel_j + normal_vel_j

    def update(self, screen, custom_dimensions=None, vector_field=True):  # ran every time
step
        if custom_dimensions is None:  # by default, the particles are bound the display size
            dim = np.array([0, 0, screen.get_width(), screen.get_height()])
        else:
            dim = custom_dimensions

        # Checking position with the display dimensions
        # If the particle's x position is above the screen height, then reverse vertical
velocity
```

```
        # And vice versa with y position
        if self.next_position[0] > dim[2] - self.radius or self.next_position[0] < dim[0] +
self.radius:
            self.velocity[0] *= -1 * self.damping
        if self.next_position[1] > dim[3] - self.radius or self.next_position[1] < dim[1] +
self.radius:
            self.velocity[1] *= -1 * self.damping

        # Put the particle back in bounds
        self.next_position = np.clip(self.next_position, (dim[0:2] + self.radius),
                                     (dim[2:4] - self.radius))

        if vector_field:  # update the spatial map as needed
            self.container.remove_particle(self)
        self.position = self.next_position
        if vector_field:
            self.container.insert_particle(self)
        self.next_position = self.position + (self.velocity * self.container.dt / self.mass)

    def apply_air_resistance(self):
        vel = self.container.get_magnitude(self.velocity)
        vel_normalised = self.container.normalise_vector(self.velocity)  # just want the
direction vector
        # applying formula to find resistive force
        drag_force = -self.container.drag_coefficient * np.pi * self.radius ** 2 * vel ** 2
        self.velocity += (drag_force * vel_normalised) / self.mass

    def get_position(self):
        return int(self.position[0]), int(self.position[1])


class Cell:  # The cells in the spatial map
    def __init__(self):
        self.cell_list = set()  # list of particles currently within this cell
        self.velocity = np.array([randint(-1, 1), randint(-1, 1)], dtype=float)  # cell
velocity


class SpatialMap:
    def __init__(self, noOfRows, noOfCols, screen_size=(1920, 1080)):
        self.frame_rate = 48
        self.dt = 1 / self.frame_rate
        self.draw_line_to_mouse = None  # Used with firing particles
        self.projected_particle_velocity_multiplier = None  # Used with firing particles
        self.screen_width, self.screen_height = screen_size[0], screen_size[1]
        self.selected_particle = None
        self.draw_grid = True
        self.grid = np.array([Cell() for _ in range(noOfRows * noOfCols)])  # Spatial map;
used in pathfinder and fluid flow
        self.air_resistance = False  # dictates if air resistance should be considered
        self.drag_coefficient = 0.000000001  # arbitrary constant that gave good results for
air resistance
        self.rows, self.cols = noOfRows, noOfCols
        self.box_width, self.box_height = self.screen_width / self.cols, self.screen_height /
self.rows  # cell width and height
        self.damping = 0.8  # The factor of energy the particles lose after a collision
        self.particles = []  # Alternative to spatial map

    def get_grid_coords(self, x=False, y=False):  # get coordinates for spatial map
        x_coords = np.linspace(0, self.screen_width, self.cols, endpoint=False)
```

```
        if x:
            return x_coords

        y_coords = np.linspace(0, self.screen_height, self.rows, endpoint=False)
        if y:
            return y_coords

        x_values, y_values = np.array(np.meshgrid(x_coords, y_coords))
        coords = np.column_stack((x_values.ravel(), y_values.ravel()))
        return coords

    def hash_position(self, position):  # Find the cell the particle is on
        try:
            return self.coord_to_index((int(position[0] / self.box_width), int(position[1] /
self.box_height)))
        except ValueError:
            print(f"Error hashing position for position {position}")

    def undo_hash_position(self, position):  # get the cartesian coordinates of the cell
corner
        return np.array(position) // self.box_width

    def coord_to_index(self, coord):  # Convert a coordinate into an index suitable for 1d
arrays
        return coord[0] + coord[1] * self.cols

    def index_to_coord(self, index):  # Inverse operation of coord_to_index
        try:
            return index % self.cols, index // self.cols
        except TypeError:
            raise Exception("index_to_coord")  # logging

    @staticmethod
    def get_magnitude(vector):
        try:
            return np.sqrt(vector[0] ** 2 + vector[1] ** 2)
        except:  # In case of ANY errors (zero magnitude, incorrect datatype, etc.), return
zero vector
            return np.zeros_like(vector)

    @staticmethod
    def get_square_magnitude(vector):  #
        try:
            return vector[0] ** 2 + vector[1] ** 2
        except:  # usually vector's magnitude is 0 or type is wrong etc.
            return vector

    def normalise_vector(self, vector):
        if vector[0] == 0 and vector[1] == 0:  # If vector has zero magnitude
            return vector  # Return the zero vector
        return vector / self.get_magnitude(vector)


    def get_neighbouring_coords(self, coord, include_diagonal=False, include_self=False,
                                placeholder_for_boundary=False):
        # given a coord, find the neighbouring coords
        col, row = coord
        directions = [[1, 0], [-1, 0], [0, -1], [0, 1]]  # Right, left, up, down | Orthogonal
movement
        neighbouring_coords = []
```

```
        if include_diagonal:
            directions.extend([[-1, -1], [1, -1], [-1, 1], [1, 1]])

        if include_self:
            directions.append([0, 0])

        for offset in directions:
            neighbour_col, neighbour_row = col + offset[0], row + offset[1]
            if 0 <= neighbour_row < self.rows and 0 <= neighbour_col < self.cols:
                neighbouring_coords.append((neighbour_col, neighbour_row))
            elif placeholder_for_boundary:
                neighbouring_coords.append(None)  # simulation will deal with this separately
        return neighbouring_coords

    def get_neighbouring_cells(self, cell_row, cell_col, diagonal=False, use_self=False):
        neighbouring_cells = []

        for coord in self.get_neighbouring_coords((cell_row, cell_col),
include_diagonal=diagonal,
                                                  include_self=use_self):
            # Getting the cell objects at the requested coordinates
            neighbouring_cells.append(self.grid[self.coord_to_index(coord)])
        return neighbouring_cells

    def get_neighbouring_particles(self, particle):
        cell_row, cell_col = divmod(self.hash_position(particle.position), self.cols)
        neighbour_cells = self.get_neighbouring_cells(cell_row, cell_col, use_self=True,
diagonal=True)
        neighbouring_particles = []
        for cell in neighbour_cells:
            # Getting the particles which are currently with the cell according to the
spatial map
            neighbouring_particles.extend(cell.cell_list)
        return neighbouring_particles

    def remove_particle(self, particle):  # Remove a particle from the spatial map
        cell = self.hash_position(particle.position)  # find the cell it is in
        self.grid[cell].cell_list.discard(particle)  # remove it from the spatial map

    def insert_particle(self, particle):
        new_cell = self.hash_position(particle.next_position)  # Find the cell the particle is
in
        if new_cell is None:
            print("Error in insert_particle method of Particle")
            return
        elif not 0 <= new_cell < self.rows * self.cols:  # particle is outside screen
            # resolving particle position
            r = particle.radius
            particle.next_position = np.clip(particle.next_position, (r, r),
                                             (self.screen_width - r, self.screen_height - r))
            return self.insert_particle(particle)  # return new instance with the updated
position

        # add the particle to the cell's particle list
        self.grid[new_cell].cell_list.add(particle)


    def drag_particle(self, mouse_pos):  # moving a particle with the cursor
        for index, particle in enumerate(self.particles):
            rad = particle.radius
```

```
            if not rad < mouse_pos[0] < self.screen_width - rad and rad < mouse_pos[1] <
self.screen_height - rad:
                continue
            distance = particle.container.get_magnitude(np.array(mouse_pos) -
particle.position)
            if distance < rad:  # if cursor is within particle
                particle.velocity = particle.velocity * 0  # stop particle movement when
particle first clicked on
                self.selected_particle = index
                return  # as it has found the particle in question, no need to continue
searching

    def drop_particle(self):  # When user has released the cursor, release the particle
        self.particles[self.selected_particle].velocity *= 0
        self.selected_particle = None

    def move_selected_particle(self, mouse_position):  # Put particle onto cursor
        self.particles[self.selected_particle].position = mouse_position

    def project_particle(self, mouse_pos):  # now only used for firing projectiles in
projectile motion
        # Finding which particle the user is interacting with
        for index, particle in enumerate(self.particles):
            rad = particle.radius
            if not rad < mouse_pos[0] < self.screen_width - rad and rad < mouse_pos[1] <
self.screen_height - rad:
                continue
                # if the cursor is not within the radius of the particle, skip
            distance = particle.container.get_magnitude(particle.position -
np.array(mouse_pos))
            if distance < rad:  # if the cursor is within the radius of the particle
                particle.velocity = particle.velocity * 0  # stop particle movement
                self.draw_line_to_mouse = True
                self.selected_particle = index  # Used to track a given particle
                return

    def release_projected_particle(self, mouse_pos):
        particle = self.particles[self.selected_particle]
        # update particle velocity with the new direction vector
        particle.velocity = (np.array(mouse_pos) - particle.position) *
self.projected_particle_velocity_multiplier
        self.draw_line_to_mouse = False
        self.selected_particle = None  # Stop tracking particle
```

projectileMotionSimulation.py

```
import pygame
import time
from Simulations.SimulationFiles.baseClasses import *


def draw_mode(level_no, penetration_factor=0.15):  # Level designer: for teachers only
    pygame.init()
    frame_rate = 75
```

```
    screen_width, screen_height = pygame.display.Info().current_w,
pygame.display.Info().current_h
    screen = pygame.display.set_mode((screen_width, screen_height))

    obstacles = []

    pygame.display.set_caption("Create Level")
    background =
pygame.image.load("./Simulations/SimulationFiles/Assets/images/background1.jpg")
    background = pygame.transform.scale(background, (screen_width, screen_height))

    rect_origin = None
    rect_params = None
    mouse_hold = False

    clock = pygame.time.Clock()
    font = pygame.font.SysFont("Helvetica", 35)

    while True:
        screen.fill((169, 130, 40))
        screen.blit(background, (0, 0))

        if rect_origin is not None and rect_params is not None:  # If the user is currently
drawing something
            rect_x = rect_origin[0] - abs(rect_params[0]) if rect_params[0] < 0 else
rect_origin[0]
            rect_y = rect_origin[1] - abs(rect_params[1]) if rect_params[1] < 0 else
rect_origin[1]
            if obstacles:  # If obstacles isn't empty, i.e. has a goal already
                pygame.draw.rect(screen, (255, 0, 0), (rect_x, rect_y, abs(rect_params[0]),
abs(rect_params[1])))
            else:  # Add a goal to the level
                radius = int(np.sqrt(rect_params[0] ** 2 + rect_params[1] ** 2))
                pygame.draw.circle(screen, (255, 255, 255), rect_origin, radius)

        for obstacle in obstacles:
            obstacle.draw(screen)

        for event in pygame.event.get():  # Event handler
            if event.type == pygame.QUIT:
                pygame.quit()
                return

            if event.type == pygame.KEYUP:
                if event.key == pygame.K_q:
                    pygame.quit()
                    return

                elif event.key == pygame.K_s:  # Save the level to a text file
                    if obstacles:  # checks if the user has added a goal
                        with
open(("./Simulations/SimulationFiles/Assets/ProjectileLevels/lvl" + str(level_no)),
                                     "w") as file:
                            file.write(f"{penetration_factor}\n")
                            goal = obstacles[0]  # Write goal to file

file.write(f"{goal.position[0]},{goal.position[1]},{goal.width},{goal.height}")
                            for obstacle in obstacles[1:]:  # Write obstacles to file
                                file.write(
```

```
f"\n{obstacle.position[0]},{obstacle.position[1]},{obstacle.width},{obstacle.height},{int(o
bstacle.is_platform)}")
                        print("Level saved")
                    else:  # Target not drawn, can't save
                        print("No goal --> Not saved")

            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1 or event.button == 3:  # If user starts drawing
                    rect_origin = np.array(pygame.mouse.get_pos())
                    mouse_hold = True

            elif event.type == pygame.MOUSEBUTTONUP:  # User has released cursor --> add
obstacle to window
                if rect_origin is not None:
                    rect_params = np.array(pygame.mouse.get_pos()) - rect_origin

                    if not obstacles:  # The first item should be the target
                        obstacles.append(Obstacle(rect_origin, radius, radius, goal=True))
                        obstacles[0].colour = (255, 255, 255)
                        min_width = min(obstacles[0].width, obstacles[0].height)
                        obstacles[0].width = min_width
                        obstacles[0].height = min_width

                    else:
                        platform = True if event.button == 3 else False
                        obstacles.append(Obstacle(np.array([rect_x, rect_y]),
abs(rect_params[0]), abs(rect_params[1]),
                                                  is_platform=platform))

                    # Reset cursor tracking
                    rect_origin = None
                    rect_params = None
                    mouse_hold = False

            if mouse_hold:
                rect_params = np.array(pygame.mouse.get_pos()) - rect_origin

        text = font.render("Level Designer", True, (255, 255, 255))
        screen.blit(text, (10, 10))
        pygame.display.update()
        clock.tick(frame_rate)


def run(level_no, air_resistance=False):
    pygame.init()
    screen_width, screen_height = pygame.display.Info().current_w,
pygame.display.Info().current_h
    screen = pygame.display.set_mode((screen_width, screen_height))  # Ensure full screen
    pygame.display.set_caption("Projectile Motion Simulation")
    background =
pygame.image.load("./Simulations/SimulationFiles/Assets/images/background1.jpg")
    background = pygame.transform.scale(background, (screen_width, screen_height))
    rows, columns = 18, 32

    container = Container(rows, columns, level_no, air_resistance)
    frame_rate = container.frame_rate

    container.particles.extend([ProjectileParticle(1, 15, container) for _ in range(6)])  #
eccentricity
    for particle in container.particles:
```

```
        particle.position = np.array([randint(140, 210), randint(780, 980)])  # Drop
particles into box from a height
    font = pygame.font.SysFont("comicsans", 20)
    font_30 = pygame.font.SysFont("comicsans", 30)
    clock = pygame.time.Clock()

    while True:
        screen.fill((70, 69, 5))
        screen.blit(background, (0, 0))

        # Output score to screen
        text = font.render("Score: " + str(container.score), True, (255, 255, 255))
        screen.blit(text, (10, 10))

        for index, particle in enumerate(container.particles):
            particle.update(screen)  # Update particle properties
            if container.air_resistance:
                particle.apply_air_resistance()

        container.goal.draw(screen)

        for particle in container.particles:
            if particle.collision_event_obstacles() and container.moving_particle is
particle:
                container.initial_time = None  # Reset timer

            particle.collision_event()  # Handle particle collisions
            particle.collision_event_goal()  # Handle target collisions
            particle.draw(screen)

        for obstacle in container.obstacles:  # Draw obstacles to screen
            obstacle.draw(screen)
        container.draw_splatters(screen)

        # kinematic info
        container.update_kinematic_info()
        container.draw_kinematic_info(screen)

        completed = set()
        for ball_i, ball_j in container.colliding_balls_pairs:  # loop over all collision
            completed.add(ball_i)
            if ball_j not in completed:  # Ensure that the particle in question hasn't
already been resolved
                ball_i.resolve_dynamic_collision(ball_j)
        container.colliding_balls_pairs.clear()  # Reset the list for the next time step

        if container.draw_line_to_mouse and container.selected_particle is not None:
            particle = container.particles[container.selected_particle]
            pygame.draw.line(screen, (255, 0, 0), particle.position, pygame.mouse.get_pos(),
width=4)
            pygame.mouse.set_cursor(pygame.cursors.broken_x)
            projected_velocity = (np.array(
                pygame.mouse.get_pos()) - particle.position) *
container.projected_particle_velocity_multiplier
            if container.toggle_velocity_display:
                display_params = f"{int(projected_velocity[0])}i\u0302 +
{int(-projected_velocity[1])}j\u0302"
            else:
                display_params = f"{container.get_magnitude(projected_velocity).astype(int)}
m/s | \u03B1 = {int(np.arctan2(projected_velocity[1], projected_velocity[0]) * -180 /
```

```
np.pi)}\u00B0"
            text = font.render(display_params, True, (255, 255, 255))
            screen.blit(text, particle.get_position() - np.array([80, 80]))

        else:
            container.draw_line_to_mouse = False

        mouse_pos = pygame.mouse.get_pos()
        coordinates = np.array([mouse_pos[0], screen_height - mouse_pos[1]])
        text = font_30.render(f"({int(coordinates[0])}, {int(coordinates[1])})", True, (255,
255, 255))
        screen.blit(text, (screen_width - 180, 10))

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return
            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_q:
                    pygame.quit()
                    return container.score
                elif event.key == pygame.K_v:
                    container.toggle_velocity_display = not
container.toggle_velocity_display
                elif event.key == pygame.K_t:
                    container.show_kinematic_info = not container.show_kinematic_info

            if event.type == pygame.MOUSEBUTTONDOWN:
                particle_clicked = container.selected_particle
                if event.button == 1:
                    container.show_coordinates = not container.show_coordinates
                    if particle_clicked is None:
                        container.drag_particle(event.pos)

                elif event.button == 3 and particle_clicked is None:
                    container.project_particle(event.pos)

            elif event.type == pygame.MOUSEBUTTONUP:
                if event.button == 1 and container.selected_particle is not None:
                    container.drop_particle()  # User drops ball
                    pygame.mouse.set_cursor(pygame.cursors.Cursor())

                elif event.button == 3 and container.selected_particle is not None:
                    particle = container.particles[container.selected_particle]
                    container.release_projected_particle(event.pos)  # Fire projectile
                    pygame.mouse.set_cursor(pygame.cursors.Cursor())
                    container.start_timer(particle)

            if container.selected_particle is not None and not container.draw_line_to_mouse:
                container.move_selected_particle(event.pos)

        pygame.display.update()
        clock.tick(frame_rate)

#
class ProjectileParticle(Particle):
    def __init__(self, mass, particle_radius, container):
        super().__init__(mass, particle_radius, container)
        self.acceleration = np.array([0, self.container.g])
        self.colour = (184, 146, 255)
```

```
        self.hit_goal = False
        self.damping = 0.8

    def draw(self, screen):
        pygame.draw.circle(screen, self.colour, self.position, self.radius)

    def px_to_metres(self, pixel_val):  # Conversion factor
        return pixel_val / self.container.px_to_metres_factor

    def collision_event_goal(self):  # Ball within target handler
        goal = self.container.goal
        if self.entirely_in_obstacle_check(goal.position, goal.width):
            self.container.initial_time = None
            self.velocity = self.velocity * (1 - self.container.penetration_factor)
            self.acceleration *= 0  # Stop gravity whilst ball is in the target
            self.hit_goal = True
            if np.allclose(self.velocity, np.zeros_like(self.velocity), atol=2):
                self.container.selected_particle = None
                self.container.calculate_points(self)
                self.container.particles.remove(self)  # Remove particle from the container
                self.container.splattered_particles.append(self)  # Draw splat sprite

        else:
            self.hit_goal = False
            self.acceleration = np.array([0, self.container.g])

    def entirely_in_obstacle_check(self, pos, radius):  # Alternative algorithm
        square_distance = self.container.get_square_magnitude(pos - self.position)
        if square_distance < radius ** 2:  # a^2 < b^2 means that a < b
            return True
        return False

    def update(self, screen):  # Over riding parent method
        self.next_position = self.position + self.velocity * self.container.dt
        if self.next_position[0] > screen.get_width() - self.radius or self.next_position[
            0] < self.radius:  # or within blocked cell
            self.velocity[0] *= -1 * self.container.damping

        if self.next_position[1] > screen.get_height() - self.radius or
self.next_position[1] < self.radius:
            self.velocity[1] *= -1 * self.container.damping
            self.velocity[0] *= 0.99  # want to add a bit of energy loss. pretty much the
coefficient of restitution

        self.next_position = np.clip(self.next_position, (self.radius, self.radius),
                                     (screen.get_width() - self.radius, screen.get_height()
- self.radius))
        self.position = self.next_position

        # Only move balls that aren't being selected by the user
        if self.container.particles.index(self) != self.container.selected_particle:
            self.velocity = self.velocity + self.acceleration


class Container(SpatialMap):
    def __init__(self, rows, columns, level_no, air_resistance):
        screen_size = (pygame.display.Info().current_w, pygame.display.Info().current_h)
        super().__init__(rows, columns, screen_size=screen_size)
        self.projected_particle_velocity_multiplier = 5
        self.damping = 0.75
```

```
        self.draw_line_to_mouse = False
        self.colliding_balls_pairs = []
        self.drag_coefficient = 0.000000001
        self.air_resistance = air_resistance
        self.px_to_metres_factor = 2
        self.penetration_factor = 0.15
        self.toggle_velocity_display = False  # Switch between velocity and vector formats
        self.show_coordinates = False  # Shown in the top right of the screen
        self.score = 0
        self.goal = None

        self.obstacles = []
        self.g = 9.8

        # Kinematic parameters
        self.moving_particle = None
        self.current_time = 0
        self.initial_time = None
        self.initial_velocity = 0
        self.initial_angle = 0
        self.final_velocity = 0
        self.final_angle = 0
        self.initial_position = np.zeros(2)
        self.current_position = np.zeros(2)
        self.show_kinematic_info = True

        if not
self.initialise_level("./Simulations/SimulationFiles/Assets/ProjectileLevels/lvl" +
str(level_no)):
            self.obstacles = []

self.initialise_level("./Simulations/SimulationFiles/Assets/ProjectileLevels/lvl1")  # load
level one
        else:
            print("Level loaded successfully")

    def start_timer(self, particle):  # Kinematic infor
        self.stop_timer()
        self.initial_time = time.time()
        self.moving_particle = particle
        self.initial_position = particle.position
        self.initial_velocity = self.get_magnitude(particle.velocity)
        self.initial_angle = np.arctan2(particle.velocity[1], particle.velocity[0]) * -180 /
np.pi
        self.current_position = particle.position

    def stop_timer(self):  # Kinematic info
        self.initial_time = None

    def update_kinematic_info(self):
        if self.initial_time is not None:
            self.current_time = time.time() - self.initial_time
            self.final_velocity = self.get_magnitude(self.moving_particle.velocity)
            self.final_angle = np.arctan2(self.moving_particle.velocity[1],
                                          self.moving_particle.velocity[0]) * -180 / np.pi
            self.current_position = self.moving_particle.position

    def draw_kinematic_info(self, screen):  # Print out kinematic info to screen with a
table
        if self.show_kinematic_info:
```

```
            pygame.draw.rect(screen, (230, 230, 230, 127), (1620, 65, 300, 330))
            font = pygame.font.SysFont("Arial", 25)
            labels = ["Initial Speed",
                      "Vertical Displacement",
                      "Horizontal Displacement",
                      "Current Speed",
                      "Time"]
            values = [
                f"{round(self.initial_velocity):>8} m/s at
{round(self.initial_angle):>3}\u00B0",
                f"{round(self.initial_position[1] - self.current_position[1]):>8} m",
                f"{round(self.current_position[0] - self.initial_position[0]):>8} m",
                f"{round(self.final_velocity):>8} m/s at
{round(self.final_angle):>3}\u00B0",
                f"{self.current_time:>10.2f} seconds"]

            for index, text in enumerate(zip(labels, values)):
                screen.blit(font.render(text[0], True, (50, 50, 50)), (1640, 20 + 60 *
(index + 1)))
                screen.blit(font.render(text[1], True, (50, 50, 50)), (1640, 45 + 60 *
(index + 1)))

    def add_points(self, points):
        self.score += points

    def calculate_points(self, particle):  # Smoothing function to calculate points
        multiplier = abs((self.get_magnitude(self.goal.position - particle.position) /
self.goal.width))
        points = int(100 * (1 - multiplier ** 2))
        self.add_points(points)

    def draw_splatters(self, screen):
        length = len(self.collision_splatters)
        for index, particle in enumerate(self.splattered_particles):
            splat = self.collision_splatters[index % length]
            screen.blit(splat, particle.position - (splat.get_width() // 2,
splat.get_height() // 2))

    def initialise_level(self, file_name):
        ball_box_image = None

        # creating ball box
        box_dimensions = [
            ((100, 1055), 150, 25),
            ((100, 970), 25, 110),
            ((225, 970), 25, 110)]
        for row in box_dimensions:
            plank = Obstacle(*row, ball_box_image)
            plank.is_platform = True
            plank.colour = (248, 94, 0)
            self.obstacles.append(plank)

        splatters = ["splat1.png", "splat2.png", "splat3.png"]
        splat_width = 100
        self.collision_splatters = []
        self.splattered_particles = []

        for splat in splatters:
            img = pygame.image.load("./Simulations/SimulationFiles/Assets/images/" + splat)
            img.convert_alpha()
```

```
            img = pygame.transform.scale(img, (splat_width, splat_width * img.get_height()
// img.get_width()))
            self.collision_splatters.append(img)

        bounce_image =
pygame.image.load("./Simulations/SimulationFiles/Assets/images/wall.jpg")
        wall_image =
pygame.image.load("./Simulations/SimulationFiles/Assets/images/bouncy_wall.jpg")
        try:  # parsing level
            with open(file_name, "r") as file:
                self.penetration_factor = float(file.readline())
                goal = file.readline()
                self.initialise_goal(goal.split(","))  # handling goal separately
                for line in file:
                    line = line.split(",")
                    image = bounce_image.copy() if int(line[4]) else wall_image.copy()
                    new_obstacle = Obstacle((int(line[0]), int(line[1])), int(line[2]),
int(line[3]), image,
                                            is_platform=int(line[4]))
                    self.obstacles.append(new_obstacle)
            return True
        except FileNotFoundError:
            print("Invalid file name given")
            return False  # and run level 1 instead
        except Exception as e:
            print(e)
            return False

    def initialise_goal(self, goal):
        image = pygame.image.load("./Simulations/SimulationFiles/Assets/images/target.png")
        image = pygame.transform.scale(image, (2 * int(goal[2]), 2 * int(goal[2])))
        self.goal = Obstacle((int(goal[0]), int(goal[1])), int(goal[2]), int(goal[2]),
image, goal=True)
        self.goal.colour = (255, 105, 180)  # pink color


class Obstacle:
    def __init__(self, position, width, height, image=None, goal=False, is_platform=False):
        self.position = np.array(position)
        self.width, self.height = width, height
        self.colour = (255, 0, 0) if not is_platform else (90, 255, 43)
        self.goal = goal
        self.is_platform = is_platform
        self.image = image
        if image is not None and not goal:
            try:
                self.image = image.subsurface(pygame.Rect(0, 0, self.width, self.height))
            except:  # Subsurface rectangle outside surface area
                self.image = pygame.transform.scale(self.image, (self.width, self.height))

    def draw(self, screen):
        if self.image:
            if self.goal:
                screen.blit(self.image, self.position - self.width)
            else:
                screen.blit(self.image, self.position)
        elif self.goal:
            pygame.draw.circle(screen, self.colour, self.position, self.width)
        else:
            pygame.draw.rect(screen, self.colour, (self.position[0], self.position[1],
```

```
self.width, self.height))
```

pathfinderSimulation.py

```
import pygame
from Simulations.SimulationFiles.baseClasses import *


def run(rows, columns, max_velocity):
    pygame.init()
    screen_width, screen_height = pygame.display.Info().current_w,
pygame.display.Info().current_h
    screen = pygame.display.set_mode((screen_width, screen_height))  # Full screen
    pygame.display.set_caption("Pygame Boilerplate")

    vector_field = VelocityField(rows, columns, max_velocity)
    frame_rate = vector_field.frame_rate
    box_width, box_height = vector_field.box_width, vector_field.box_height

    radius = vector_field.particle_to_add_radius
    vector_field.particles = [Pathfinder(radius // 3, radius, vector_field) for _ in
range(30)]

    font = pygame.font.SysFont("comicsans", int(box_width // 2.6))
    clock = pygame.time.Clock()

    while True:
        for event in pygame.event.get():  # Event handler
            if event.type == pygame.QUIT:
                pygame.quit()
                return
            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_q:
                    pygame.quit()
                    return
                elif event.key == pygame.K_a:  # Switch between adding particles and
changing goal
                    vector_field.is_adding_particles = not vector_field.is_adding_particles
                elif event.key == pygame.K_c:  # Toggle collisions. Disable to reduce
latency
                    vector_field.enable_collision_between_particles = not
vector_field.enable_collision_between_particles
                elif event.key == pygame.K_EQUALS:  # Plus symbol --> increase radius
                    vector_field.particle_to_add_radius += 1
                elif event.key == pygame.K_MINUS:  # Minus symbol --> decrease radius
                    vector_field.particle_to_add_radius =
max(vector_field.particle_to_add_radius - 1, 3)
                elif event.key == pygame.K_h:  # Toggle distance field gradient
                    vector_field.draw_heatmap = not vector_field.draw_heatmap
                elif event.key == pygame.K_g:  # Toggle metadata
                    vector_field.draw_grid = not vector_field.draw_grid
                elif event.key == pygame.K_r:
                    vector_field.clear_obstacles()
```

```
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 3:
                    cell =
vector_field.index_to_coord(vector_field.hash_position(pygame.mouse.get_pos()))
                    vector_field.toggle_adding_cells(cell)  # Set toggle type accordingly

            click_event = pygame.mouse.get_pressed()
            if any(click_event):
                pos_cell_index =
vector_field.index_to_coord(vector_field.hash_position(pygame.mouse.get_pos()))
                if click_event[0]:  # Set new goal
                    vector_field.update_velocity_field(pos_cell_index)

                elif click_event[2]:
                    if not vector_field.is_adding_particles:
                        vector_field.toggle_blocked_cell(pos_cell_index)
                    else:
                        vector_field.add_particle(pygame.mouse.get_pos())

        screen.fill((186, 217, 210))
        if vector_field.draw_heatmap:
            vector_field.display_heatmap(screen)

        if vector_field.enable_collision_between_particles:  # Particle collisions
            for particle in vector_field.particles:
                particle.collision_event()

        vector_field.update()
        for particle in vector_field.particles:
            particle.update(screen)  # Standard particle update

        for particle in vector_field.particles:
            pygame.draw.circle(screen, (130, 46, 129), particle.get_position(),
particle.radius)

        for cell in vector_field.obstacles:  # Draw obstacles
            pygame.draw.rect(screen, (0, 0, 0, 0.5), (cell[0] * box_width, cell[1] *
box_height, box_width, box_height))

        if vector_field.draw_grid:  # Draw grid metadata if user selected it
            for x in vector_field.get_grid_coords(x=True):
                pygame.draw.line(screen, "#353252", (x, 0), (x, screen_height), 1)

            for y in vector_field.get_grid_coords(y=True):
                pygame.draw.line(screen, "#353252", (0, y), (screen_width, y), 1)

            for coord, cell, distance in zip(vector_field.get_grid_coords(),
vector_field.grid,
                                             vector_field.cell_distances):

                box_centre = np.array([coord[0] + box_width / 2, coord[1] + box_height / 2])
                line_radius = (box_width / 2.2) * cell.velocity
                if not any(np.isnan(cell.velocity)):
                    pygame.draw.line(screen, "#ff3542", box_centre, box_centre +
line_radius)
                if distance > 0:
                    number = font.render(f"{distance:.1f}", True, (255, 255, 255))
                    screen.blit(number, box_centre - (box_width // 4))
```

```
        pygame.display.update()
        clock.tick(frame_rate)


class Pathfinder(Particle):
    def __init__(self, mass, radius, container, position=None):
        super().__init__(mass, radius, container, position)
        self.damping = 0.7

    def check_for_collision_X(self, obstacle_x, obstacle_width):  # Alternative collision
detection = efficient
        radius = self.radius
        if self.next_position[0] + self.radius > obstacle_x + obstacle_width or
self.next_position[0] - radius < obstacle_x:
            return False
        return True

    def collision_event_obstacles(self, obstacle_pos, obstacle_width):  # Obstacle collision
handler
        # overriding the parent method with this new idea instead
        # runs much faster than the full collision resolution of the Particle Class
        # I used the spatial map to check if the particle is currently in an obstacle
        # Below I am checking if it has hit an obstacle from the sides, or from the top or
bottom
        x_collision = self.check_for_collision_X(obstacle_pos[0], obstacle_width)
        if x_collision:  # if collision in x direction
            self.velocity[0] *= -1
        else:  # If not x collision, then it is in y collision
            self.velocity[1] *= -1
        self.next_position = self.position + self.velocity * -1 * self.container.dt

    def collision_event(self, track_collisions=False):  # Particle collision handler
        try:
            cell_index = self.container.hash_position(self.position)
            particles_to_check = self.container.grid[cell_index].cell_list
            for particle in particles_to_check:
                if self.is_collision(particle, save_collisions=track_collisions):
                    self.resolve_static_collision(particle)
                    return

        except Exception as e:
            print("Debug: Particle collision", e)  # Debugging


class VelocityField(SpatialMap):
    def __init__(self, noOfRows, noOfCols, max_velocity):
        screen_size = (pygame.display.Info().current_w, pygame.display.Info().current_h)
        super().__init__(noOfRows, noOfCols, screen_size=screen_size)
        self.cell_distances = np.zeros_like(self.grid)  # Used in pathfinding algorithm
        self.damping = 0.80  # Energy factor in particle collision with walls
        self.obstacles = set()  # Stores obstacles.
        self.goal = np.array([0, 0])
        self.particle_max_velocity = max_velocity  # Desired velocity magnitude in steering
behaviours
        self.is_adding_particles = False
        self.is_adding_cells = False
        self.enable_collision_between_particles = False
        self.draw_heatmap = True
        self.particle_to_add_radius = 5
        self.max_distance = 0
```

```
    def display_heatmap(self, screen):  # drawing a colour gradient depending on cell
distance
        for i in range(self.cols):
            for j in range(self.rows):
                distance = self.cell_distances[self.coord_to_index((i, j))]
                if distance > 0:
                    if np.isinf(distance):  # give inf cell a harsh colour
                        pygame.draw.rect(screen, (255, 160, 255),
                                         (i * self.box_width, j * self.box_height,
self.box_width, self.box_height))
                    else:
                        norm_distance = distance / self.max_distance  # Normalising distance
between 0 and 1
                        colour = np.array([255 * (1 - norm_distance), 190, 255 *
norm_distance], dtype=int)
                        pygame.draw.rect(screen, colour,
                                         (i * self.box_width, j * self.box_height,
self.box_width, self.box_height))

    def find_max_distance(self):  # Time saving step. Only find max_distance once when new
goal is set
        self.max_distance = np.max(list(filter(lambda x: np.isfinite(x),
self.cell_distances)))

    def print_visited(self):  # Testing if algorithm is working
        for i in range(self.rows):
            for j in range(self.cols):
                print(self.cell_distances[self.coord_to_index((i, j))], end=" | ")
            print()

    def toggle_adding_cells(self, mouse_cell):  # ... and adding particles
        if mouse_cell in self.obstacles:
            self.is_adding_cells = False
        else:
            self.is_adding_cells = True

    def toggle_blocked_cell(self, coord):  # Easy to use obstacle toggling
        if self.is_adding_cells:
            if coord not in self.obstacles:
                self.obstacles.add(coord)
        else:
            if coord in self.obstacles:
                self.obstacles.remove(coord)

    def clear_obstacles(self):
        self.obstacles.clear()

    def add_particle(self, mouse_position):
        obj = Pathfinder(self.particle_to_add_radius // 3, self.particle_to_add_radius,
self,
                         np.array(mouse_position, dtype=float))

        self.particles.append(obj)

    def generate_heatmap(self, goal_coords):  # Calculate distance field
        self.cell_distances = np.empty_like(self.grid)
        # Initialise cell distances with infinity, obstacles with -1
        for cell_index, cell in enumerate(self.cell_distances):
            cell_coord = self.index_to_coord(cell_index)
```

```
            if cell_coord in self.obstacles:
                self.cell_distances[cell_index] = -1
            else:
                self.cell_distances[cell_index] = float('inf')

        # Set distance to the goal cell to 0
        goal_coords = np.clip(np.array(goal_coords), np.zeros_like(goal_coords),
np.array([self.cols, self.rows]) - 1)
        goal_index = self.coord_to_index(goal_coords)
        self.cell_distances[goal_index] = 0

        # Initialise queue for breadth-first search with coordinates of goal cell
        queue = [goal_coords]

        while queue:
            current_coord = queue.pop(0)  # Pull from queue
            current_index = self.coord_to_index(current_coord)
            current_distance = self.cell_distances[current_index]
            neighbouring_coords = self.get_neighbouring_coords(current_coord,
include_diagonal=True)
            for next_coord in neighbouring_coords:
                try:
                    next_index = self.coord_to_index(next_coord)

                    if next_coord not in self.obstacles and self.cell_distances[next_index]
== float('inf'):
                        # update distance if  cell is not blocked and not visited
                        change_x = abs(next_coord[0] - current_coord[0])
                        change_y = abs(next_coord[1] - current_coord[1])
                        if change_x == 1 and change_y == 1:
                            path_cost = 1.41421  # np.sqrt(2) to 5 d.p: diagonal movement
                        else:  # Orthogonal movement
                            path_cost = 1
                        self.cell_distances[next_index] = current_distance + path_cost
                        queue.append(next_coord)
                except IndexError:  # Invalid coordinate --> ignore
                    pass

    def calculate_vectors(self):  # Calculate velocity field from distance field
        for cell_coord in self.obstacles:
            self.cell_distances[self.coord_to_index(cell_coord)] = -1

        for counter, (values) in enumerate(zip(self.cell_distances, self.grid)):  # Gather
necessary values
            distance, cell = values
            if self.index_to_coord(counter) in self.obstacles:
                cell.velocity = np.array([0, 0])  # Set velocity of blocked cells to nothing
                continue
            coords = self.get_neighbouring_coords(self.index_to_coord(counter),
placeholder_for_boundary=True)

            distances = [0, 0, 0, 0]  # right, left, up , down etc. Temp distances for
validation purposes
            for index, eachcoord in enumerate(coords):
                if eachcoord and eachcoord not in self.obstacles:
                    distances[index] = self.cell_distances[self.coord_to_index(eachcoord)]
                else:
                    distances[index] = -1

            # Handling velocity of cells adjacent to blocked cells
```

```
            dist_copy = distances.copy()
            for index, distance in enumerate(distances):
                if distance == -1:  # Blocked cell
                    if index == 0 and distances[1] != -1:
                        dist_copy[0] = distances[1] + 1  # Change from + 1 to + 2 if
repulsive effect desired
                    elif index == 1 and distances[0] != -1:
                        dist_copy[1] = distances[0] + 1
                    if index == 2 and distances[3] != -1:
                        dist_copy[2] = distances[3] + 1
                    elif index == 3 and distances[2] != -1:
                        dist_copy[3] = distances[2] + 1

            # Find velocity components
            x_vector = dist_copy[1] - dist_copy[0]
            y_vector = dist_copy[2] - dist_copy[3]

            # Apply normalised velocity to cell
            cell.velocity = self.normalise_vector(np.array([x_vector, y_vector]))

    def update_velocity_field(self, coords_of_goal):  # New goal has been set, so rerun
algorithm
        if not any(np.isnan(coords_of_goal)):
            if not (self.goal[0] == coords_of_goal[0] and self.goal[1] == coords_of_goal[
                1]) and coords_of_goal not in self.obstacles:
                self.goal = coords_of_goal
                self.generate_heatmap(coords_of_goal)
                self.calculate_vectors()
                self.find_max_distance()  # Time saving for displaying heatmap

    def calculate_avoidance_force(self, position):  # Steering behaviour, now deprecated
        radius = 1 * self.box_width
        avoidance_strength = 150
        steering_force = np.zeros(2)

        for obstacle in self.obstacles:
            distance = np.linalg.norm(position - obstacle)
            if distance < radius:
                # Adjust steering force to help avoid the obstacle
                steering_force += avoidance_strength * (position - obstacle) / distance
        return steering_force

    def calculate_collision_avoidance(self, particle):  # Steering behaviour, now deprecated
        magnitude = 1000
        ahead = particle.position + self.normalise_vector(particle.velocity) *
self.box_width
        box_centre_add = self.box_width / 2
        for coord in self.obstacles:
            distance_vector = ahead - (self.undo_hash_position(coord) + box_centre_add)
            if self.get_magnitude(distance_vector) < self.box_width:
                return self.normalise_vector(distance_vector) * magnitude
        return 0

    def zero_vel_for_obstacles(self, particle):  # Lose energy on wall collision, now
deprecated
        pos = particle.position
        current_cell = self.index_to_coord(self.hash_position(pos))
        if current_cell in self.obstacles:
            particle.velocity *= 0
        else:
```

```
            return

    def update(self):  # update and apply field
        field_strength = 0.02
        for eachCell in self.grid:

            # Applying cell velocity onto particles
            desired_velocity = eachCell.velocity * self.particle_max_velocity
            if any(np.isnan(desired_velocity)) or any(np.isinf(desired_velocity)):  #
Debugging
                continue
            for eachParticle in eachCell.cell_list:  # Get particles within cell via spatial
map
                steering_force = desired_velocity - eachParticle.velocity  # Change

                eachParticle.velocity += (steering_force * field_strength)  # Steadily apply
desired velocity

                # Obstacle collision handling
                coord = self.index_to_coord(self.hash_position(eachParticle.next_position))
                if coord in self.obstacles:
                    eachParticle.collision_event_obstacles(coord, self.box_width)
```

idealGasLawSimulation.py

```
from Simulations.SimulationFiles.baseClasses import *
import pygame


def run():
    pygame.init()
    screen_width, screen_height = pygame.display.Info().current_w,
pygame.display.Info().current_h
    screen = pygame.display.set_mode((screen_width, screen_height))
    pygame.display.set_caption("Ideal Gas Law Simulation")

    container = Container(32, 18, (screen_width, screen_height))
    frame_rate = container.frame_rate
    clock = pygame.time.Clock()

    while True:
        screen.fill((78, 198, 140))

        # Draw layout
        container.draw_walls(screen)
        container.draw_widgets(screen, pygame.mouse.get_pos())

        # Update particles
        for index, particle in enumerate(container.particles):
            particle.update(screen, custom_dimensions=container.dimensions,
vector_field=False)

        for particle in container.particles:
```

```
            particle.collision_event()
            pygame.draw.circle(screen, particle.colour, particle.position, particle.radius)

        # Handle collisions
        completed = set()
        for ball_i, ball_j in container.colliding_balls_pairs:
            completed.add(ball_i)
            if ball_j not in completed:
                ball_i.resolve_dynamic_collision(ball_j)
                if container.spark_button.text == "Collision Sparks Enabled":
                    pygame.draw.line(screen, (0, 255, 0), ball_i.position, ball_j.position,
width=5)
        container.colliding_balls_pairs.clear()

        # Event handler
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                return

            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_q:
                    pygame.quit()
                    return

            # Selecting container wall
            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 3:  # RMB
                    if container.selected_wall_check(event.pos):
                        pygame.mouse.get_rel()
                        container.wall_selected = container.selected_wall_index(event.pos)

                elif event.button == 1:  # LMB
                    if container.reset_button.click_check(event.pos):  # Reset button
                        container.initialise_container()

                    if container.within_wall_check(event.pos):  # Add particles
                        container.add_particle(event.pos)
                        container.rms_velocity = container.calculate_rms_velocity()

                    for widget in container.widgets:  # General widget handler
                        if widget.click_check(event.pos):
                            widget.is_clicked = not widget.is_clicked
                            break

            elif event.type == pygame.MOUSEBUTTONUP:
                if event.button == 3 and container.wall_selected is not None:
                    container.wall_selected = None  # Release container wall

                if event.button == 1:  # Release temperature slider
                    container.temp_slider.is_clicked = False

            if container.wall_selected is not None:  # Adjust wall dimensions if wall is
selected
                container.change_wall_dimensions(pygame.mouse.get_rel())  # Move container
wall

        pygame.display.update()
        clock.tick(frame_rate)
```

```
class GasParticle(Particle):
    def __init__(self, mass, particle_radius, container, velocity=None, position=None,
colour=(123, 12, 255)):
        super().__init__(mass, particle_radius, container, velocity=velocity,
position=position)
        self.colour = colour  # Distinguishes heavy and light particles
        self.damping = 1

    def update(self, screen, custom_dimensions=None, vector_field=False):
        super().update(screen, custom_dimensions=custom_dimensions,
vector_field=vector_field)


class Widget:
    def __init__(self, position, size, colour, text=None, slider=False, parent=None,
hover=True, alt_text=None, dynamic=False):
        self.position = np.array(position, dtype=float)  # Top left corner
        self.size = np.array(size)  # (width, height)
        self.colour = np.array(colour)
        self.default_colour = self.colour.copy()
        self.slider = slider
        self.parent = parent  # i.e. the container the widget is bound to
        self.text = text
        self.alt_text = alt_text
        self.font = pygame.font.SysFont("Arial", 30)
        self.hover = hover  # Should the widget respond the cursor hovering over it?
        self.dynamic = dynamic  # Does the widget move?
        self.is_clicked = False
        if slider:  # Currently only used for the temperature
            self.knob = self.position + self.size // 2
            self.knob_rest_pos = self.knob.copy()
            self.knob_value = 273

    def draw(self, screen, mouse_pos):  # Drawing the widget on the screen
        pygame.draw.rect(screen, self.colour, tuple(self.position) + tuple(self.size))

        if self.slider:
            pygame.draw.circle(screen, self.colour, (self.position[0], self.position[1] +
self.size[1] // 2),
                               self.size[1] // 2)
            pygame.draw.circle(screen, self.colour,
                               (self.position[0] + self.size[0], self.position[1] +
self.size[1] // 2),
                               self.size[1] // 2)
            pygame.draw.circle(screen, (180, 180, 230), self.knob, self.size[1])

        else:
            if self.hover and self.position[0] < mouse_pos[0] < self.position[0] +
self.size[0] and self.position[1] < \
                    mouse_pos[1] < self.position[1] + self.size[1]:
                pygame.draw.rect(screen, self.colour * 0.9, (self.position[0],
self.position[1], self.size[0], self.size[1]))
            text = self.font.render(self.text, True, (0, 0, 0))
            pos = self.position + self.size // 2
            screen.blit(text, (pos[0] - text.get_width() // 2, pos[1] - text.get_height() //
2))

    def update(self):  # Check for changes
        if self.is_clicked:
```

```
            if self.slider:
                self.knob[0] = max(min(pygame.mouse.get_pos()[0], self.knob_rest_pos[0] +
0.5 * self.size[0]),
                                   self.position[0])
                self.knob_value += 0.01 * (self.knob_rest_pos[0] - self.knob[0])
                self.knob_value = max(1, self.knob_value)
                self.parent.temperature_change(self.knob_value)

        elif self.slider:
            difference = self.knob_rest_pos - self.knob
            self.knob += difference * 0.2

        if self.dynamic and self.parent:
            dim = self.parent.dimensions
            pressure_sizes = 250, 100
            self.position = np.array(
                [(dim[0] + dim[2] - pressure_sizes[0]) // 2, dim[1] - pressure_sizes[1] -
self.parent.wall_radius])

    def click_check(self, pos):  # Checks whether widget has been interacted with
        if self.slider:
            distance = np.sqrt((self.knob[0] - pos[0]) ** 2 + (self.knob[1] - pos[1]) ** 2)

            if distance < self.size[1]:
                return True

        else:
            if self.position[0] < pos[0] < self.position[0] + self.size[0] and
self.position[1] < pos[1] < \
                    self.position[1] + self.size[1]:

                if self.alt_text:
                    self.text, self.alt_text = self.alt_text, self.text

                self.colour = self.default_colour
                return True
        return False


class Container(SpatialMap):
    def __init__(self, rows, columns, screen_size):
        # Initialising variables
        super().__init__(rows, columns, screen_size=screen_size)
        self.rms_velocity = None
        self.R = 8.3145  # Boltzmann's constant
        self.px_to_metres = 0.1
        self.dimensions = np.array([200, 200, 1000, 850])  # left, top, right, bottom:
container wall positions
        self.font = pygame.font.SysFont("comicsans", int(self.box_width // 2.6))
        self.small_font = pygame.font.SysFont("comicsans", int(self.box_width // 4))
        self.colliding_balls_pairs = []  # Used for collision resolution
        self.wall_selected = None
        self.wall_radius = 20  # Wall width
        self.damping = 1  # No energy lost during collision - ideal gas law
        self.temperature = 293
        self.initial_temperature = 293  # Needed for resetting container

        # Creating widgets
        dim = self.dimensions
        pressure_sizes = 250, 100
```

```
        pressure_pos = ((dim[0] + dim[2] - pressure_sizes[0]) // 2, dim[1] -
pressure_sizes[1] - self.wall_radius)
        self.pressure_display = Widget(pressure_pos, pressure_sizes, (255, 255, 200),
parent=self, dynamic=True,
                                       hover=False)
        self.temp_slider = Widget((1480, 410), (400, 30), (81, 228, 81), slider=True,
parent=self)
        self.particle_button = Widget((1560, 630), (250, 100), (166, 179, 215), text="Heavy
Particles",
                                      alt_text="Light Particles")
        self.reset_button = Widget((1480, 900), (400, 100), (242, 125, 125), text="RESET")
        self.spark_button = Widget((1490, 770), (380, 75), (237, 234, 116), text="Collision
Sparks Enabled",
                                   alt_text="Collision Sparks Disabled")
        self.widgets = [self.pressure_display, self.temp_slider, self.particle_button,
self.reset_button,
                        self.spark_button]

        self.initialise_container()

    def initialise_container(self):  # Also used for resetting the container
        dim = self.dimensions
        base_v = 55  # Initial velocity of particles
        self.temp_slider.knob_value = self.initial_temperature
        self.temperature = self.initial_temperature

        self.particles.clear()  # Remove all particles to reset container
        self.particles.extend([GasParticle(0.1, 8, self, position=np.array(
            [randint(dim[0], dim[2]), randint(dim[1], dim[3])]), velocity=np.array(
            [randint(-base_v, base_v), randint(-base_v, base_v)], dtype=float)) for _ in
range(50)])
        self.rms_velocity = self.calculate_rms_velocity()
        self.pressure_display.text = f"{self.calculate_pressure() * 1000:<.2f} mPa"

    def calculate_pressure(self):  # P = (nRT)/V
        width, height = (self.dimensions[2] - self.dimensions[0]) / self.px_to_metres, (
                self.dimensions[3] - self.dimensions[1]) / self.px_to_metres
        pressure = (len(self.particles) * self.R * self.temperature) / (width * height)
        return pressure

    def calculate_rms_velocity(self):
        total_square_vel = 0.0
        for particle in self.particles:
            squared_velocity_magnitude = np.sum((particle.velocity / self.px_to_metres) **
2)
            total_square_vel += 0.5 * particle.mass * squared_velocity_magnitude

        rms = np.sqrt(total_square_vel / len(self.particles))
        return rms

    def add_particle(self, mouse_position):  # Add heavy or light particle at cursor
        if self.particle_button.is_clicked:
            obj = GasParticle(0.06, 5, self, position=np.array(mouse_position, dtype=float))
            obj.colour = np.array([255, 60, 60])
        else:
            obj = GasParticle(0.1, 8, self, position=np.array(mouse_position, dtype=float))

        self.particles.append(obj)
        self.pressure_display.text = f"{self.calculate_pressure() * 1000:<.2f} mPa"
```

```
    def temperature_change(self, new_temperature):  # i.e. when slider is changed
        change = new_temperature - self.temperature
        if abs(change) > 0.5:  # Only update if big enough interval
            old_temperature = self.temperature
            self.temperature = new_temperature
            temperature_ratio = new_temperature / old_temperature

            # Update simulation parameters
            self.rms_velocity = self.calculate_rms_velocity()
            self.pressure_display.text = f"{self.calculate_pressure() * 1000:<.2f} mPa"
            for index, particle in enumerate(self.particles):
                particle.velocity *= temperature_ratio

    def draw_widgets(self, screen, mouse_pos):  # Widget handler
        for widget in self.widgets:
            widget.draw(screen, mouse_pos)
            widget.update()

    def draw_walls(self, screen):  # draw container + text
        screen_width, screen_height = screen.get_width(), screen.get_height()
        dim = self.dimensions
        radius = self.wall_radius
        width = dim[2] - dim[0]
        height = dim[3] - dim[1]
        pygame.draw.rect(screen, (161, 221, 208), (0.75 * screen_width, 0, 0.25 *
screen_width, screen_height))
        pygame.draw.rect(screen, (200, 200, 200), (
            dim[0] - radius, dim[1] - radius, width + 2 * radius,
            height + 2 * radius))
        pygame.draw.rect(screen, (255, 255, 255), (dim[0], dim[1], width, height))

        # Output the widget text
        text = self.small_font.render(f"Heat Up", True, (10, 10, 10))
        screen.blit(text, (0.79 * screen_width - 0.5 * text.get_width(), 450))
        text = self.small_font.render(f"Cool Down", True, (10, 10, 10))
        screen.blit(text, (0.96 * screen_width - 0.5 * text.get_width(), 450))
        text = self.font.render(f"{round(self.temp_slider.knob_value, 1)} \u00B0K", True,
(10, 10, 10))
        screen.blit(text, (0.875 * screen_width - 0.5 * text.get_width(), 325))
        text = self.small_font.render("Root Mean Square Speed", True, (10, 10, 10))
        screen.blit(text, (0.875 * screen_width - 0.5 * text.get_width(), 100))
        text = self.small_font.render("AKA The Average Speed:", True, (10, 10, 10))
        screen.blit(text, (0.875 * screen_width - 0.5 * text.get_width(), 130))
        text = self.font.render(f"{round(self.rms_velocity, 1)} m/s", True, (10, 10, 10))
        screen.blit(text, (0.875 * screen_width - 0.5 * text.get_width(), 170))
        text = self.small_font.render("Spawn particles", True, (10, 10, 10))
        screen.blit(text, (0.875 * screen_width - 0.5 * text.get_width(), 550))
        text = self.small_font.render("with your left click!", True, (10, 10, 10))
        screen.blit(text, (0.875 * screen_width - 0.5 * text.get_width(), 580))

    def selected_wall_check(self, mouse_pos):  # checking if ANY wall has been selected
        dim = self.dimensions
        radius = self.wall_radius
        checks = np.array([
            dim[0] - radius < mouse_pos[0] < dim[2] + radius and not dim[0] < mouse_pos[0] <
dim[2],
            dim[1] - radius < mouse_pos[1] < dim[3] + radius and not dim[1] < mouse_pos[1] <
dim[3]
        ])
        if checks.any():
```

```
            return True
        return False

    def within_wall_check(self, mouse_pos):  # Checks if mouse is inside walls, not
including walls themselves
        dim = self.dimensions
        if dim[0] < mouse_pos[0] < dim[2] and dim[1] < mouse_pos[1] < dim[3]:
            return True
        return False

    def selected_wall_index(self, mouse_pos):  # Returns index of the selected wall
        dim = self.dimensions
        radius = self.wall_radius
        if dim[0] - radius < mouse_pos[0] < dim[0]:  # left wall
            return 0

        elif dim[1] - radius < mouse_pos[1] < dim[1]:  # top wall
            return 1

        elif dim[2] < mouse_pos[0] < dim[2] + radius:  # right wall
            return 2

        elif dim[3] < mouse_pos[1] < dim[3] + radius:  # bottom wall
            return 3
        else:
            return None

    def change_wall_dimensions(self, change):  # Update container dimensions
        index = self.wall_selected
        dim = self.dimensions.copy()
        if index % 2:  # If the top or bottom wall is selected
            dim[index] += change[1]
        else:  # If left or right wall is selected
            dim[index] += change[0]

        # Container dimensions validation
        if not dim[2] + self.wall_radius < self.screen_width * 0.75:
            dim[2] = self.screen_width * 0.75 - self.wall_radius
        if 20 < dim[2] - dim[0] and dim[3] - dim[1] > 20:
            self.dimensions = dim
        self.pressure_display.text = f"{self.calculate_pressure() * 1000:<.2f} mPa"
```

fluidFlowSimulation

```
# No longer supported nor included in the program
from Simulations.SimulationFiles.baseClasses import *
import pygame


def run():
    pygame.init()
    screen_width, screen_height = 1920, 1080
    screen = pygame.display.set_mode((screen_width, screen_height))  # Full screen
    pygame.display.set_caption("Ideal Gas Law Simulation (deprecated)")
    rows, columns = 18, 32
```

```
    spatial_map = FluidSpatialMap(rows, columns)
    frame_rate = spatial_map.frame_rate
    particles = [FluidParticle(1, 3, spatial_map) for _ in
range(spatial_map.no_of_particles)]
    spatial_map.calculate_rest_density(particles)
    clock = pygame.time.Clock()

    while True:
        for event in pygame.event.get():  # Event handler
            if event.type == pygame.KEYUP:
                if event.key == pygame.K_q:
                    pygame.quit()
                    return

        # Drawing grid
        screen.fill((90, 69, 50))
        if spatial_map.draw_grid:

            for x in spatial_map.get_grid_coords(x=True):
                pygame.draw.line(screen, "#353252", (x, 0), (x, screen_height), 1)

            for y in spatial_map.get_grid_coords(y=True):
                pygame.draw.line(screen, "#353252", (0, y), (screen_width, y), 1)

        # Simulation logic
        for particle in particles:
            particle.update_density()  # Must be done separately from pressure and viscosity
calculations

        for particle in particles:
            particle.update_pressure()
            particle.calculate_pressure_force()
            particle.update(screen)
            pygame.draw.circle(screen, (123, 12, 90), particle.position, particle.radius)

        pygame.display.update()
        clock.tick(frame_rate)


class FluidParticle(Particle):
    def __init__(self, mass, radius, spatial_map):
        super().__init__(mass, radius, spatial_map)
        self.radius = radius  # visual only
        self.mass = mass
        self.spatial_map = spatial_map
        self.force = np.zeros(2, dtype=float)
        self.stiffness_constant = 10  # used for pressure
        self.pressure_force = np.zeros(2, dtype=float)
        self.density = None
        self.pressure = None
        self.calculate_density()
        self.calculate_pressure()

    def calculate_density(self):
        density = 0
        neighbouring_particles = self.spatial_map.get_neighbouring_particles(self)
        for neighbour_particle in neighbouring_particles:
            if neighbour_particle != self:  # don't include self in density
                distance = self.spatial_map.get_magnitude(neighbour_particle.position -
```

```
self.position)
                influence = self.spatial_map.kernel.calculate_density_contribution(distance)
                density += influence * neighbour_particle.mass  # scale by particle's mass
        self.density = density

    def update_density(self):
        self.calculate_density()

    def get_density(self):
        return self.density

    def update_pressure(self):
        self.calculate_pressure()

    def get_pressure_force(self):  # interpolate pressure force
        return self.pressure

    def calculate_pressure(self):  # ideal gas law application
        self.pressure = self.stiffness_constant * (self.density -
self.spatial_map.rest_density)

    def calculate_pressure_force(self):
        self.pressure_force = np.zeros(2, dtype=float)
        neighbouring_particles = self.spatial_map.get_neighbouring_particles(self)
        for neighbour_particle in neighbouring_particles:  # Use pressure of neighbouring
particles
            if self != neighbour_particle:
                distance = self.spatial_map.get_magnitude(self.position -
neighbour_particle.position)
                self.pressure_force -= self.calculate_pressure_contribution(self,
neighbour_particle, distance)
        self.velocity += self.container.dt * self.pressure_force / self.mass

    def calculate_pressure_contribution(self, particle, neighbour_particle, dist):
        # similar function to self.calculate_pressure_force. Currently testing this
implementation
        direction_vector = self.spatial_map.normalise_vector(particle.position -
neighbour_particle.position)
        if particle.density == 0 or neighbour_particle.density == 0:
            return np.array([0, 0])
        pressure = particle.pressure / (particle.density ** 2)
        neighbour_pressure = neighbour_particle.pressure / (neighbour_particle.density ** 2)
        smoothing_kernel_gradient =
self.spatial_map.kernel.cubic_spline_kernel_gradient(dist)
        pressure_force = -neighbour_particle.mass * 0.5 * (
                pressure + neighbour_pressure) * smoothing_kernel_gradient *
direction_vector
        return pressure_force

    def calculate_property(self):  # generic property finder, like for pressure
        property = 0
        neighbouring_particles = self.spatial_map.get_neighbouring_particles(self)

        for neighbour_particle in neighbouring_particles:
            distance = self.spatial_map.get_magnitude(neighbour_particle.position -
self.position)
            influence = self.spatial_map.kernel.calculate_density_contribution(distance)
            property += influence * (neighbour_particle.mass / neighbour_particle.density)
        return property
```

```
    def apply_forces(self):  # apply the navier-stokes equation to get delta velocity
        self.force = np.zeros_like(self.velocity)
        self.force = self.force + self.calculate_pressure_force()
        self.velocity += self.container.dt * np.array((self.force / self.mass), dtype=float)

    def get_position(self):
        return int(self.next_position[0]), int(self.next_position[1])


class SmoothingKernel:
    def __init__(self, smoothing_length, poly_6=False, gaussian=False, cubic_spline=False,
spiky=False, test=True):
        # the radius within which a neighbouring particle will have an impact
        self.h = smoothing_length / 2 if cubic_spline else smoothing_length

        # The chosen smoothing_kernel
        self.cubic_spline = cubic_spline
        self.poly_6 = poly_6
        self.gaussian = gaussian
        self.spiky = spiky
        self.test = test

        if poly_6: # Assigning normalisation constant
            self.normalisation_constant = 315 / (64 * np.pi * self.h ** 2)  # ** 9
        elif gaussian:
            self.normalisation_constant = 1 / (np.sqrt(2 * np.pi) * self.h)
        elif cubic_spline:
            self.normalisation_constant = 10 / (7 * np.pi * self.h ** 2)
        elif spiky:
            self.normalisation_constant = 15 / (np.pi * (self.h ** 6))  # ** 6
        elif test:
            self.normalisation_constant = 1

    def calculate_density_contribution(self, particle_radius):
        if self.poly_6:
            return self.poly_6_kernel(particle_radius)

        elif self.cubic_spline:
            return self.cubic_spline_kernel(particle_radius)

        elif self.gaussian:
            return self.gaussian_kernel(particle_radius)

        elif self.spiky:
            return self.spikey_kernel(particle_radius)

        elif self.test:
            return self.test_kernel(particle_radius)
        raise ValueError("Unknown kernel type")

    def get_normalised_density(self, density):
        return self.normalisation_constant * density

    def test_kernel(self, particle_radius):  # Personal testing kernel
        return ((max(0, particle_radius ** 2 - self.h ** 2)) ** 3) / (np.pi * self.h ** 2 /
4)

    def cubic_spline_kernel(self, particle_radius):
        ratio = particle_radius / self.h
```

```
        if 0 <= ratio < 1:
            return 1 - (1.5 * ratio ** 2) + (0.75 * ratio ** 3)
        elif 1 <= ratio < 2:
            return 0.25 * ((2 - ratio) ** 3)
        else:
            return 0

    def cubic_spline_kernel_gradient(self, particle_radius):
        ratio = particle_radius / self.h
        if 0 <= ratio < 1:
            return -3 * ratio
        elif 1 <= ratio < 2:
            return -0.75 * (2 - ratio) ** 2
        else:
            return 0

    def poly_6_kernel(self, particle_radius):
        if particle_radius <= self.h:  # if particle is within smoothing radius
            return self.normalisation_constant * ((self.h ** 2 - particle_radius ** 2) ** 3)
        return 0

    def gaussian_kernel(self, particle_radius):
        return 0

    def spikey_kernel(self, particle_radius):
        if particle_radius <= self.h:
            return self.normalisation_constant * (self.h - particle_radius) ** 3
        return 0


class FluidSpatialMap(SpatialMap):
    def __init__(self, noOfRows, noOfCols):
        screen_size = (pygame.display.Info().current_w, pygame.display.Info().current_h)
        super().__init__(noOfRows, noOfCols, screen_size=screen_size)
        self.no_of_particles = 30
        self.rest_density = 100  # Tinker until reaching desired effect
        self.smoothing_radius = self.box_width  # To ensure that spatial grid can be used
        self.kernel = SmoothingKernel(self.smoothing_radius, cubic_spline=True)
        self.pressure_kernel = SmoothingKernel(self.smoothing_radius, spiky=True)

    def calculate_rest_density(self, particle_list):
        total_density = 0
        for particle in particle_list:
            total_density += particle.density
        self.set_rest_density(total_density / self.no_of_particles)  # rest density

    def set_rest_density(self, rest_density):
        self.rest_density = rest_density
```

database.py

```
import mysql.connector


class Database:
```

```
    def __init__(self, host_, user_, password_, database_):  # Connect to the server
        self.__db = mysql.connector.connect(
            host=host_,
            user=user_,
            password=password_,
            database=database_,
            autocommit=True
        )

        self.__conn = self.__db.cursor()

    def initialise_default_db(self):
        try:
            # Removing old tables
            self.__conn.execute("DROP TABLE IF EXISTS user_settings;")
            self.__conn.execute("DROP TABLE IF EXISTS students;")
            self.__conn.execute("DROP TABLE IF EXISTS teachers;")
            self.__conn.execute("DROP TABLE IF EXISTS users;")

            # Creating new tables
            self.__conn.execute("""
            CREATE TABLE users (
                user_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
                email VARCHAR(256) UNIQUE NOT NULL,
                password_hash CHAR(64) NOT NULL,
                full_name VARCHAR(64),
                date_of_birth DATE,
                is_teacher BOOLEAN DEFAULT FALSE
            );""")

            self.__conn.execute("""
            CREATE TABLE teachers (
                teacher_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
                user_id INT NOT NULL,
                FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
            );""")

            self.__conn.execute("""
            CREATE TABLE students (
                student_id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
                user_id INT NOT NULL,
                teacher_id INT NOT NULL,
                FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
                FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id) ON DELETE CASCADE
            );""")

            self.__conn.execute("""
            CREATE TABLE user_settings (
                id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
                user_id INT NOT NULL,
                pathfinder_rows INT DEFAULT 18,
                pathfinder_cols INT DEFAULT 32,
                pathfinding_velocity INT DEFAULT 40,
                projectile_score INT DEFAULT 0,
                projectile_max_level INT DEFAULT 1,
                FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
            );""")

            # Creating teachers and a couple student account for placeholders
            self.create_user("admin",
```

```
"5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8", "No class",
                                "2000-01-01", is_teacher=True)  # password
            self.create_user("teacher@email.com",
"1057a9604e04b274da5a4de0c8f4b4868d9b230989f8c8c6a28221143cc5a755",
                                "Mr Smith", "2000-01-01", is_teacher=True)  # teacher
            self.create_user("m.stevens@gmail.com",
"16a3a923a6143b7b1ebc73ea64ef23fc41ed7b26b883f51c8e90bb32a9cb3dc4",
                                "Mr Stevens", "2000-01-01", is_teacher=True)  # stevens
            self.create_user("student1", "placeholder", "Student One", "2000-01-01",
teacher_email="m.stevens@gmail.com")
            self.create_user("student2", "placeholder", "Student Two", "2000-01-01",
teacher_email="m.stevens@gmail.com")
            self.create_user("student3", "placeholder", "Student Three", "2000-01-01",
teacher_email="teacher@email.com")
            self.create_user("student4", "placeholder", "Student Four", "2000-01-01",
teacher_email="teacher@email.com")
            print("Database initialised successfully.")

        except Exception as e:
            print("Error initialising database:", e)

    def create_user(self, email, password_hash, full_name, date_of_birth, is_teacher=False,
teacher_email=None):
        # Create either a student or a teacher account
        try:
            if not is_teacher:  # Get teacher_id from email and then perform pseudocode
                teacher_id = self.get_teacher_id(teacher_email)

            self.__conn.execute("""
            INSERT INTO users (email, password_hash, full_name, date_of_birth, is_teacher)
VALUES (%s, %s, %s, %s, %s);
            """, (email, password_hash, full_name, date_of_birth, is_teacher))

            user_id = self.__conn.lastrowid
            self.__conn.execute("""
            INSERT INTO user_settings (user_id) VALUES (%s);
            """, (user_id,))

            if is_teacher:
                self.__conn.execute("INSERT INTO teachers (user_id) VALUES (%s);",
(user_id,))
            else:  # Is a student
                self.__conn.execute("INSERT INTO students (user_id, teacher_id) VALUES (%s,
%s);", (user_id, teacher_id))
            return True

        except:
            print("Could not create user")
            return False

    def get_teacher_id(self, email):  # Get teacher_id from their email address
        self.__conn.execute("""
        SELECT user_id FROM users WHERE email = %s;
        """, (email,))
        result = self.__conn.fetchone()
        if result is None:
            print("Teacher not in database")
            return None
        return result[0]
```

```
    def get_teacher_email_by_name(self, name):
        self.__conn.execute("""
        SELECT email FROM users WHERE full_name = %s;
        """, (name,))
        result = self.__conn.fetchone()
        if result is None:
            print("Teacher not in database")
            return None
        return result[0]

    def get_teachers_teacher_id(self, user_id):  # Get teacher_id of a teacher via user_id
        self.__conn.execute("""
        SELECT teachers.teacher_id
        FROM users, teachers
        WHERE users.user_id = %s AND users.user_id = teachers.user_id;
        """, (user_id,))
        result = self.__conn.fetchone()
        return result[0]

    def get_projectile_rankings(self, teacher_id):  # Used for projectile motion leaderboard
        self.__conn.execute("""
        SELECT full_name, projectile_score
        FROM users, user_settings, students
        WHERE users.user_id = students.user_id
        AND users.user_id = user_settings.user_id
        AND students.teacher_id = %s
        ORDER BY projectile_score DESC;
        """, (teacher_id,))
        result = self.__conn.fetchall()
        return result

    def get_teacher_names(self):  # Get all teachers for account creation stage
        self.__conn.execute("""
        SELECT full_name FROM users, teachers WHERE users.user_id = teachers.user_id;
        """)
        result = self.__conn.fetchall()
        return result

    def get_teacher_id_by_user_id(self, user_id):  # Get teacher_id of a student
        self.__conn.execute("""
        SELECT teacher_id FROM students WHERE user_id = %s;
        """, (user_id,))
        result = self.__conn.fetchone()
        return result[0]

    def verify_login(self, email, password_hash):
        try:
            self.__conn.execute("""
            SELECT * FROM users WHERE email = %s AND password_hash = %s;
            """, (email, password_hash))
            result = self.__conn.fetchone()
            assert result is not None
            return result
        except AssertionError:
            print("Incorrect login details")
            return None
        except:
            print("Unexpected error when verifying login")
            return None
```

```
    def get_user_settings(self, user_id):  # Fetch the user settings from the according
user_id
        self.__conn.execute("""
        SELECT * FROM user_settings WHERE user_id = %s;
        """, (user_id,))
        result = self.__conn.fetchone()
        return result

    def save_and_shut_down(self, user_settings):  # Save new settings to the database
        settings = user_settings.copy()
        settings.append(settings[0])
        self.__conn.execute("""
        UPDATE user_settings SET pathfinder_rows = %s, pathfinder_cols = %s,
pathfinding_velocity = %s, projectile_score = %s, projectile_max_level = %s WHERE user_id =
%s;
        """, settings[2:])


if __name__ == "__main__":
    db = Database("localhost", "root", "2121", "NEA")
    db.initialise_default_db()
```

main.py

```
import sys
import numpy as np
from PyQt6.QtWidgets import QMainWindow, QWidget, QStackedLayout, QToolBar, QGridLayout,
QLabel, QLineEdit, QPushButton, \
    QDateEdit, QComboBox, QTableWidget, QSizePolicy, QSpinBox, QSlider, QMessageBox,
QApplication, QTableWidgetItem
from PyQt6.QtGui import QFont, QColor, QAction, QCursor, QBrush
from PyQt6.QtCore import Qt, QDate
from Simulations import pathfinderSimulation, projectileMotionSimulation,
idealGasLawSimulation, fluidFlowSimulation
from database import Database
from hashlib import sha256
import re


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.user_info = None
        self.user_settings = None
        self.teacher_id = None
        self.database = Database("localhost", "root", "2121", "NEA")  # Readying database
        self.setWindowTitle("Physics Simulator")
        self.layout = QStackedLayout()
        self.index = QWidget()
        self.index.setLayout(self.layout)
        self.setCentralWidget(self.index)
        self.toolbar = QToolBar()  # How the user navigates the UI
        self.addToolBar(self.toolbar)
        self.toolbar.setMovable(False)
        self.toolbar.hide()  # User can not interact with program until successful login
```

```
        self.showFullScreen()  # The program is the focus
        self.setFont(QFont("Helvetica", 15))

        self.login_layout = QGridLayout()
        self.login = QWidget()
        self.login.setLayout(self.login_layout)
        self.login.setAutoFillBackground(True)  #
        palette = self.login.palette()  #
        palette.setColor(self.login.backgroundRole(), QColor(255, 249, 196))
        self.login.setPalette(palette)

        self.login_label = QLabel("Physics Simulator")
        self.login_label.setStyleSheet("""
        font-family: 'Times New Roman';
        font-size: 100px;""")
        self.login_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.login_layout.addWidget(self.login_label, 0, 2, 1, 2)

        self.email = QLineEdit()  # Email field
        self.email.setPlaceholderText("Enter email address")
        self.email.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.login_layout.addWidget(self.email, 7, 0, 1, 6)

        self.password = QLineEdit()  # Password field
        self.password.setPlaceholderText("Enter password")
        self.password.setEchoMode(QLineEdit.EchoMode.Password)
        self.password.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.login_layout.addWidget(self.password, 8, 0, 1, 6)

        self.login_button = QPushButton("Log in")  # Log in button
        self.login_button.released.connect(self.login_or_register)
        self.login_button.setMaximumWidth(350)
        self.login_layout.addWidget(self.login_button, 9, 3, 1, 1)

        self.toggle_login = QPushButton("Don't Have\nAn Account?")
        self.toggle_login.setMaximumWidth(350)
        self.toggle_login.released.connect(self.toggle_login_register)
        self.login_layout.addWidget(self.toggle_login, 9, 2, 1, 1)

        # Registration widgets
        self.full_name = QLineEdit()  # Name field
        self.full_name.setPlaceholderText("Enter your full name")
        self.full_name.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.login_layout.addWidget(self.full_name, 4, 1, 1, 4)
        self.full_name.hide()

        self.date_of_birth = QDateEdit()  # Birth date field
        self.date_of_birth.setDisplayFormat("dd/MM/yyyy")
        self.date_of_birth.setAlignment(Qt.AlignmentFlag.AlignRight)
        self.date_of_birth.setCalendarPopup(True)
        self.login_layout.addWidget(self.date_of_birth, 6, 1, 1, 2)
        self.date_of_birth.hide()

        self.date_of_birth_label = QLabel("Date of Birth")
        self.login_layout.addWidget(self.date_of_birth_label, 6, 1, 1, 1)
        self.date_of_birth_label.hide()

        teachers = []
        for record in self.database.get_teacher_names():  # Class selection
            teachers.append(f"{record[0]}")
```

```
        self.teacher_dropdown = QComboBox()
        self.teacher_dropdown.addItems(teachers)
        self.login_layout.addWidget(self.teacher_dropdown, 6, 3, 1, 2)
        self.teacher_dropdown.hide()

        self.layout.addWidget(self.login)
        self.login.setStyleSheet("""
            QLabel {
                font-family: 'Comic Sans MS';
                font-size: 30px;
                align: right;
                margin-left: 30px;
            }

            QLineEdit {
                margin: 20px 200px 0px 200px;;
                border: 10px solid '#ffab40';
                font-size: 30px;
                padding: 10px;
                width: 600px;
            }

            QPushButton, QDateEdit, QComboBox {
                font-size: 40px;
                padding: 10px 20px;
                color: green;
                border-radius: 10%;
                display: inline-block;
                position: fixed;
                margin: 30px 30px 30px 0px;
                transition-duration: 0.9s;
                border: 2px solid;
                align: right;
            }
            QPushButton {background-color: '#d7ffcc';font-family: 'Comic Sans MS';}
            QPushButton:hover {background-color: '#abff94';}
        """)

        # Home / Sign Out Page
        self.home_page_button = QAction("Home Page")
        self.toolbar.addAction(self.home_page_button)
        self.toolbar.addSeparator()
        self.home_page_button.triggered.connect(lambda: self.changeIndex(1))

        self.home_page_layout = QGridLayout()
        self.home_page = QWidget()
        self.home_page.setLayout(self.home_page_layout)

        self.home_page.setAutoFillBackground(True)  # Setting background colour
        palette = self.home_page.palette()
        palette.setColor(self.home_page.backgroundRole(), QColor(249, 247, 243))
        self.home_page.setPalette(palette)

        self.home_page.setStyleSheet("""
        QTableWidget {
            font-family: 'Comic Sans MS';
            font-size: 45px;
            padding: 20px;
            border: 0px solid;
        }
```

```
        QPushButton {
            font-size: 65px;
            padding: 20px;
            color: red;
            border-radius: 20%;
            display: inline-block;
            position: fixed;
            margin: 30px;
            border: 3px solid;
            background-color: '#d7ffcc';
            font-family: 'Comic Sans MS';
        }

        QPushButton:hover {background-color: '#abff94';}

        QLabel {
            font-family: 'Comic Sans MS';
            color: '#32a142';
            font-size: 40px;
            margin-bottom: 10px;
            padding-bottom: 10px;
            border: none;
        }
        """)
        self.logoff_button = QPushButton("Log Out")  # Exits the program
        self.logoff_button.setFixedSize(700, 300)
        self.logoff_button.released.connect(self.log_off)
        self.home_page_layout.addWidget(self.logoff_button, 1, 0, 1, 1)

        self.projectile_leaderboard_label = QLabel("Projectile Motion\nWeekly Leaderboard")
        self.home_page_layout.addWidget(self.projectile_leaderboard_label, 0, 1)
        self.weeklyScore = 0  # Score is from projectile motion weekly level
        self.projectile_leaderboard = QTableWidget()
        self.projectile_leaderboard.setColumnCount(3)
        self.projectile_leaderboard.setHorizontalHeaderLabels(["Ranking", "Full Name",
"Score"])
        self.projectile_leaderboard.verticalHeader().setVisible(False)
        self.projectile_leaderboard.setSizePolicy(QSizePolicy.Policy.Preferred,
QSizePolicy.Policy.Preferred)
        self.projectile_leaderboard.setAlternatingRowColors(True)
        self.projectile_leaderboard.setMinimumHeight(460)
        self.projectile_leaderboard.setColumnWidth(1, 120)
        self.projectile_leaderboard.setStyleSheet("background-color: transparent;")
        header = self.projectile_leaderboard.horizontalHeader()
        header.setStyleSheet(
            "QHeaderView::section { background-color: lightblue; color: white; font-weight:
bold; font-size: 45px}")
        self.layout.addWidget(self.home_page)

        # Projectile Motion Simulation
        self.projectile_button = QAction("Projectile Motion", self)
        self.toolbar.addAction(self.projectile_button)
        self.projectile_button.triggered.connect(lambda: self.changeIndex(2))
        self.projectile_widget_layout = QGridLayout()
        self.projectile_widget = QWidget()

        self.projectile_widget.setAutoFillBackground(True)  # Setting background colour
        palette = self.projectile_widget.palette()  #
        palette.setColor(self.projectile_widget.backgroundRole(), QColor(232, 179, 220))
```

```
        self.projectile_widget.setPalette(palette)

        self.projectile_widget.setStyleSheet("""
            QLineEdit {
                margin: 60;
                border: 3px solid '#0000ff';
            }


            QLabel {
                font-family: 'Comic Sans MS';
                font-size: 29px;
                margin-bottom: 10px;
                padding-bottom: 10px;
                border: none;
            }
            QPushButton, QSpinBox {
                font-family: 'Comic Sans MS';
                font-size: 30px;
                padding: 20px 32px;
                border-radius: 6%;
                display: inline-block;
                position: fixed;
                margin: 0 0 0 0;
                transition-duration: 0.4s;
                border: 3px solid;
            }
            QPushButton {background-color: '#F9B5AC';}

            QPushButton:hover {
                background-color: '#ffb468';
                color: white;
            }

        """)
        self.projectile_widget.setLayout(self.projectile_widget_layout)

        self.projectile_sim_buttons = [QPushButton("Level\n" + str(x + 1)) for x in
range(9)]
        for index, button in enumerate(self.projectile_sim_buttons):
            button.setCheckable(False)
            button.setCursor(QCursor(Qt.CursorShape.ForbiddenCursor))
            self.projectile_widget_layout.addWidget(button,
self.projectile_sim_buttons.index(button) // 3,
                                                    (2 *
((self.projectile_sim_buttons.index(button)) % 3) + 3), 1, 1)
            button.setMaximumWidth(300)
            button.setMinimumHeight(150)
        self.weeklyButton = QPushButton("Level of\nthe Week")
        self.weeklyButton.setMaximumWidth(400)
        self.enable_projectile_button(self.weeklyButton)
        self.weeklyButton.released.connect(lambda: self.run_projectile_motion_sim("Weekly"))

        self.air_resistance_button = QPushButton("Air Resistance\nENABLED")  # Air
resistance button
        self.toggle_air_resistance_button()
        self.air_resistance_button.setMaximumWidth(240)
        self.air_resistance_button.released.connect(self.toggle_air_resistance_button)
        self.air_resistance_button.setCheckable(True)
        self.projectile_widget_layout.addWidget(self.air_resistance_button, 2, 0, 1, 1)
```

```
        self.projectile_instruction = QLabel()  # Instructions text
        self.projectile_instruction.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.projectile_instruction.setWordWrap(True)
        self.projectile_instruction.setText("""Welcome to the
Projectile Motion Simulation!
The aim of the game is to fire the ball into the target.
The closer to the centre you are, the more score!
Use SUVAT equations to calculate the required velocity.
Get 100 score to unlock the next level.

Controls:
LMB to move the ball
RMB to project the ball
t: show parameters table
v: toggle velocity view
q: quit""")
        self.projectile_widget_layout.addWidget(self.projectile_instruction, 0, 0, 2, 3)

        self.layout.addWidget(self.projectile_widget)

        # Vector Field Pathfinding Simulation
        self.pathfinding_button = QAction("Vector Field Pathfinding", self)
        self.toolbar.addAction(self.pathfinding_button)
        self.pathfinding_button.triggered.connect(lambda: self.changeIndex(3))
        self.pathfinding_layout = QGridLayout()
        self.pathfinding = QWidget()

        self.pathfinding.setAutoFillBackground(True)  # Setting background colour
        palette = self.pathfinding.palette()
        palette.setColor(self.pathfinding.backgroundRole(), QColor(209, 245, 255))
        self.pathfinding.setPalette(palette)

        self.pathfinding.setStyleSheet("""
            QLabel{
                font-family: 'Comic Sans MS';
                font-size: 30px;
                color: '#000000';
                align: left;
            }

            QPushButton, QSpinBox, QSlider{
                background-color: '#aff8f0';
                font-family: 'Comic Sans MS';
                font-size: 30px;
                padding: 20px;
                color: '#540d6e';
                border-radius: 6%;
                display: inline-block;
                position: fixed;
                margin: 0 0 0 0;
                transition: 0.4s;
                border: 3px solid;
            }

            QPushButton:hover {
                background-color: '#83f28f';
                color: white;
            }
        """)
```

```
        self.pathfinding_rows = QSpinBox()  # no of rows for simulation
        self.pathfinding_rows.setRange(4, 100)
        self.pathfinding_rows.setMaximumWidth(300)
        self.pathfinding_layout.addWidget(self.pathfinding_rows, 0, 1)
        pathfinder_rows_label = QLabel("No. of Rows")
        pathfinder_rows_label.setMaximumWidth(400)
        self.pathfinding_layout.addWidget(pathfinder_rows_label, 0, 0)

        self.pathfinding_cols = QSpinBox()  # no of cols for simulation
        self.pathfinding_cols.setRange(4, 100)
        self.pathfinding_cols.setMaximumWidth(300)
        self.pathfinding_layout.addWidget(self.pathfinding_cols, 1, 1)
        pathfinder_columns_label = QLabel("No. of Columns")
        pathfinder_columns_label.setMaximumWidth(400)
        self.pathfinding_layout.addWidget(pathfinder_columns_label, 1, 0)

        self.pathfinding_speed = QSlider()  # desired velocity magnitude for steering
behaviour in simulation
        self.pathfinding_speed.setOrientation(Qt.Orientation.Horizontal)
        self.pathfinding_speed.setMaximumWidth(1300)
        self.pathfinding_speed.setRange(1, 100)
        self.pathfinding_speed.setTickPosition(QSlider.TickPosition.TicksBothSides)
        self.pathfinding_layout.addWidget(self.pathfinding_speed, 2, 1, 1, 1)

        self.pathfinding_speed_label = QLabel(f"Particle speed =
{self.pathfinding_speed.value()}")
        self.pathfinding_speed_label.setFixedWidth(350)
        self.pathfinding_speed.valueChanged.connect(
            lambda: self.pathfinding_speed_label.setText(f"Particle speed =
{self.pathfinding_speed.value()}"))
        self.pathfinding_layout.addWidget(self.pathfinding_speed_label, 2, 0, 1, 1)

        self.pathfinding_run = QPushButton("Run")  # Run the pathfinder
        self.pathfinding_run.setMinimumHeight(100)
        self.pathfinding_layout.addWidget(self.pathfinding_run, 3, 1, 1, 3)
        self.pathfinding_run.released.connect(self.run_pathfinder)

        self.pathfinder_instruction = QLabel()  # Instructions text
        self.pathfinder_instruction.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.pathfinder_instruction.setWordWrap(True)
        self.pathfinder_instruction.setText("""Welcome to the
        Vector Field Pathfinder!
        Build mazes, obstacles, and paths...
        The particles will find their way to the goal!

        Controls:
        LMB: Set the goal
        a: Change between the obstacle toggle and adding particles
        RMB: Change cells into obstacles, or vice versa
        RMB|alternative: Add particles! Use the + and - key to change their size!
        c: Enable collisions! Turn this off if the program is slow
        g: Show grid information
        h: Show distance heatmap
        r: Clear the field off all obstacle
        q: Quit""")

        self.pathfinding_layout.addWidget(self.pathfinder_instruction, 0, 2, 2, 3)
        self.pathfinding.setLayout(self.pathfinding_layout)
        self.layout.addWidget(self.pathfinding)
```

```
        # Ideal Gas Law Simulation
        self.ideal_gas_button = QAction("Ideal Gas Simulation", self)
        self.toolbar.addAction(self.ideal_gas_button)
        self.ideal_gas_button.triggered.connect(lambda: self.changeIndex(4))
        self.ideal_gas_layout = QGridLayout()
        self.ideal_gas = QWidget()
        self.ideal_gas.setLayout(self.ideal_gas_layout)
        self.layout.addWidget(self.ideal_gas)

        self.ideal_gas.setAutoFillBackground(True)  # Setting background colour
        palette = self.ideal_gas.palette()
        palette.setColor(self.ideal_gas.backgroundRole(), QColor(200, 249, 202))
        self.ideal_gas.setPalette(palette)

        self.ideal_gas.setStyleSheet("""
                    QLineEdit {
                        margin: 60;
                        border: 3px solid '#00ff00';
                    }

                    QPushButton {
                        background-color: '#7AF0B7';
                        font-family: 'Comic Sans MS';
                        font-size: 30px;
                        padding: 20px;
                        margin: 30px;
                        color: '#6B5E62';
                        border-radius: 6%;
                        display: inline-block;
                        position: fixed;
                        transition: 0.4s;
                        border: 3px solid;
                    }

                    QPushButton:hover {
                        background-color: '#83f28f';
                        color: white;
                    }""")

        self.ideal_gas_run = QPushButton("Run ideal gas simulation")  # Runs the simulation
        self.ideal_gas_run.released.connect(self.run_ideal_gas_sim)
        self.ideal_gas_run.setMinimumHeight(150)
        self.ideal_gas_layout.addWidget(self.ideal_gas_run, 3, 1, 1, 1)

        self.ideal_gas_instruction = QLabel()
        self.ideal_gas_instruction.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.ideal_gas_instruction.setWordWrap(True)
        self.ideal_gas_instruction.setText("""Welcome to the
Ideal Gas Simulation!
PV = nRT
Change the conditions and watch how the pressure
and the particles react!

Controls:
Drag the walls to change the volume of the container!
Click inside the container to add a particle!
Use the buttons!
q: Quit""")
        self.ideal_gas_layout.addWidget(self.ideal_gas_instruction, 0, 0, 2, 3)
```

```
        # General stylesheet
        self.setStyleSheet("""
            QLabel{
                font-family: 'Comic Sans MS';
                font-size: 30px;
            }""")

    def changeIndex(self, newIndex):  # Switch between layers
        self.layout.setCurrentIndex(newIndex)

    def attempt_login(self):  # Send data to database and fetch user information
        email, password = self.email.text().strip(),
sha256(self.password.text().encode()).hexdigest()
        user_info = self.database.verify_login(email, password)
        if user_info:
            self.user_info = user_info  # Get user information like email and name
            self.user_settings = self.database.get_user_settings(self.user_info[0])  # Get
user settings from user id

            if not user_info[-1]:  # If is_teacher field is False:
                self.teacher_id = self.database.get_teacher_id_by_user_id(self.user_info[0])
            else:  # User is a teacher
                self.teacher_id = None

            self.initialise_program()  # Prepare simulation with the user settings
            self.show_toolbar()  # User now has access to simulations
            self.changeIndex(1)  # Switch to the home / sign out page
            return True

        else:
            print(email, password)
            QMessageBox.critical(self, "Verification Failed", "Invalid email or password")
            return False

    def login_or_register(self):
        if self.toggle_login.text()[0] == "A":
            self.create_new_db_user()
        else:
            self.attempt_login()

    def create_new_db_user(self):
        valid = True
        detailed_text = ""
        try:
            if len(self.password.text()) < 6:
                detailed_text += "Password must be at least 6 characters.\n"
                valid = False

            if self.email.text() == "":
                detailed_text += "No email given.\n"
                valid = False

            elif not re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$",
self.email.text()):
                detailed_text += "Email is not valid.\n"
                valid = False

            if len(self.email.text()) > 256:
                detailed_text += "Email is too long.\n"
```

```
                valid = False

            if self.full_name.text() == "":
                detailed_text += "Full name is not given.\n"
                valid = False

            if self.date_of_birth.date().addYears(16) > QDate.currentDate():
                detailed_text += "You must be at least 16 years old.\n"
                valid = False

            assert valid  # Ensure that fields have passed validation
            teacher =
self.database.get_teacher_email_by_name(self.teacher_dropdown.currentText())  # Get
teacher_id
            user_created = self.database.create_user(self.email.text(),

sha256(self.password.text().encode()).hexdigest(),
                                                     self.full_name.text(),

self.date_of_birth.date().toString("yyyy-MM-dd"),
                                                     teacher_email=teacher)
            if not user_created:  # If database-side issue, then call an error
                raise AssertionError

            self.toggle_login_register()  # prompt user to login
            return True

        except AssertionError:
            box = QMessageBox()
            box.setText("Could not create a new account\nPlease check you have entered your
information correctly.")
            box.setDetailedText(detailed_text)
            box.setIcon(QMessageBox.Icon.Warning)
            box.setWindowTitle("Registration Failed")
            box.exec()

            return False

    def toggle_login_register(self):
        if self.toggle_login.text()[0] == "A":  # If box says: Already Registered?
            self.toggle_login.setText("Don't have\n an account?")
            self.login_button.setText("Log in")
            self.full_name.hide()
            self.date_of_birth.hide()
            self.date_of_birth_label.hide()
            self.date_of_birth_label.hide()
            self.teacher_dropdown.hide()

        else:  # Switch to Register format
            self.toggle_login.setText("Already have\n an account?")
            self.login_button.setText("Register")
            self.full_name.show()
            self.date_of_birth.show()
            self.date_of_birth_label.show()
            self.teacher_dropdown.show()

    def show_toolbar(self):  # i.e. Enable access to the program and simulations
        self.toolbar.setVisible(True)

    def initialise_program(self):
```

```
        self.penetration_factor_button = QSpinBox()  # Penetration factor for creating
projectile motion levels
        self.penetration_factor_button.setAlignment(Qt.AlignmentFlag.AlignRight)
        self.penetration_factor_button.setMaximumWidth(180)
        self.penetration_factor_button.setSuffix("%")
        self.penetration_factor_button.setRange(1, 100)
        self.penetration_factor_button.setValue(15)

        if self.teacher_id is None:  # No teacher_id means user is a teacher
            self.projectile_instruction.setText(
                self.projectile_instruction.text() + "\nTeachers can use the spinbox to
control the penetration factor!")
            self.weeklyButton.setMaximumWidth(350)
            self.projectile_widget_layout.addWidget(self.weeklyButton, 2, 2, 1, 1)
            max_level = len(self.projectile_sim_buttons)
            self.projectile_widget_layout.addWidget(self.penetration_factor_button, 2, 1, 1,
1)
        else:  # Do not add the projectile_factor_button to the layout
            self.weeklyButton.setFixedWidth(750)
            self.projectile_widget_layout.addWidget(self.weeklyButton, 2, 1, 1, 2)
            max_level = self.user_settings[-1]

        self.pathfinding_rows.setValue(self.user_settings[2])
        self.pathfinding_cols.setValue(self.user_settings[3])
        self.pathfinding_speed.setValue(self.user_settings[4])
        self.weeklyScore = self.user_settings[5]
        if self.weeklyScore > 0:
            self.weeklyButton.setText(f"Level of the Week\nCurrent Score:
{self.weeklyScore}")
        self.fill_projectile_leaderboard()  # Projectile motion weekly leaderboard
        for index, button in enumerate(self.projectile_sim_buttons[:max_level]):
            self.enable_projectile_button(button, index)

    def fill_projectile_leaderboard(self):
        row_count = 10
        self.projectile_leaderboard.clearContents()
        teacher_id = self.teacher_id
        if self.teacher_id is None:
            teacher_id = self.database.get_teachers_teacher_id(self.user_info[0])
            print("teacher_id: ", teacher_id)
        fields = self.database.get_projectile_rankings(teacher_id)
        self.projectile_leaderboard.setRowCount(row_count + 1)

        for index, (name, score) in enumerate(fields[:row_count]):  # Fill the leaderboard
with fetched data
            print(index, name, score)
            record = (QTableWidgetItem(str(index + 1)), QTableWidgetItem(name),
QTableWidgetItem(str(score)))
            for column, item in enumerate(record):
                item.setTextAlignment(Qt.AlignmentFlag.AlignCenter)
                item.setFont(QFont("Comic Sans MS", 25))
                self.projectile_leaderboard.setItem(index, column, item)

        user_record = (
            QTableWidgetItem("-"), QTableWidgetItem(self.user_info[3]),
QTableWidgetItem(str(self.weeklyScore)))
        for column, item in enumerate(user_record):  # Have the last row as the user's score
            item.setTextAlignment(Qt.AlignmentFlag.AlignCenter)
            item.setFont(QFont("Comic Sans MS", 25))
            item.setBackground(QBrush(QColor(80, 30, 50, 125)))
```

```
            self.projectile_leaderboard.setItem(row_count, column, item)

        self.home_page_layout.addWidget(self.projectile_leaderboard, 1, 1, 2, 1)

    def enable_projectile_button(self, button, index=-1):  # Buttons are disabled by default
        button.setCheckable(True)
        if index >= 0:
            button.released.connect(lambda index=index: self.run_projectile_motion_sim(index
+ 1))
        button.setCursor(QCursor(Qt.CursorShape.PointingHandCursor))
        button.setObjectName("free_button")
        button.setStyleSheet("""
            QPushButton:hover{
                background-color: #83f28f;
            } """)

    def toggle_air_resistance_button(self):
        text = self.air_resistance_button.text()
        if "DISABLED" in text:  # Then enable air resistance
            self.air_resistance_button.setText("Air Resistance\nENABLED")
            self.air_resistance_button.setStyleSheet("""
            QPushButton {
            font-family: 'Times New Roman';
            background-color: rgba(0, 255, 0, 0.3);
            color: black;
            } """)
        else:  # Then disable air resistance
            self.air_resistance_button.setText("Air Resistance\nDISABLED")
            self.air_resistance_button.setStyleSheet("""
            QPushButton {
            font-family: 'Times New Roman';
            background-color: rgba(255, 0, 0, 0.3);
            color: black;
            }""")

    def run_projectile_motion_sim(self, level_no):
        if self.teacher_id is None and str(level_no) == "Weekly":
            projectileMotionSimulation.draw_mode(level_no,
self.penetration_factor_button.value() / 100)

        score = projectileMotionSimulation.run(level_no,
self.air_resistance_button.isChecked())
        if score:
            if score > 100 and level_no != "Weekly":
                print("Winner!")
                print(level_no)
                button = self.projectile_sim_buttons[level_no]
                self.enable_projectile_button(button, level_no)
            else:
                if score > self.weeklyScore:
                    self.weeklyScore = score
                    self.weeklyButton.setText(f"Level of the Week\nCurrent Score:
{self.weeklyScore}")
                    self.save_to_database()
                    self.fill_projectile_leaderboard()

    def run_pathfinder(self):
        row = self.pathfinding_rows.value()
        col = self.pathfinding_cols.value()
        if row / col != self.height() / self.width():  # If cells are rectangular
```

```
            warning_box = QMessageBox()
            warning_box.setWindowTitle("Rectangular cells")
            warning_box.setIcon(QMessageBox.Icon.Question)
            warning_box.addButton(QPushButton("Apply square grid"),
QMessageBox.ButtonRole.ApplyRole)
            warning_box.setStandardButtons(QMessageBox.StandardButton.No |
QMessageBox.StandardButton.Yes)
            warning_box.setText(
                "With the current grid configuration, the cells in the grid will NOT be
square.\nWould you like to continue anyway?")
            user_input = warning_box.exec()  # Either Yes for continue, No, or No and
implement square grid
            if user_input == QMessageBox.StandardButton.No:  # Don't run sim
                return
            if user_input != QMessageBox.StandardButton.Yes:  # Apply square grid and don't
run sim
                self.correct_grid_ratio()
                return
        speed = self.pathfinding_speed.value() * 20
        pathfinderSimulation.run(row, col, speed)

    def correct_grid_ratio(self):
        width, height = self.width(), self.height()
        gcd = np.gcd(width, height)  # greatest common denominator
        cols, rows = width // gcd, height // gcd
        while rows < 15 or cols < 15:  # i.e. if the grid is too small as it is
            rows *= 2
            cols *= 2
        if rows > 100 or cols > 100:  # If the screen size is such that the smallest grid
size is impractical
            rows, cols = 18, 32  # Simply apply a default grid size of 18x32
        self.pathfinding_rows.setValue(rows)
        self.pathfinding_cols.setValue(cols)

    def run_ideal_gas_sim(self):
        try:
            idealGasLawSimulation.run()
        except Exception as e:
            print(e)

    def run_fluid_flow_sim(self):
        try:
            fluidFlowSimulation.run()
        except Exception as e:
            print(e)

    def log_off(self):
        self.save_to_database()
        self.close()

    def save_to_database(self):
        print(self.user_settings)
        self.user_settings = list(self.user_settings)

        max_level = 1
        for index, button in enumerate(self.projectile_sim_buttons):
            print(button.objectName())
            if button.objectName() == "free_button":
                max_level = index + 1
        self.user_settings[-1] = max_level
```

```
        self.user_settings[2] = int(self.pathfinding_rows.value())
        self.user_settings[3] = int(self.pathfinding_cols.value())
        self.user_settings[4] = int(self.pathfinding_speed.value())
        self.user_settings[5] = int(self.weeklyScore)
        print(self.user_settings, "new ")

        self.database.save_and_shut_down(self.user_settings)


app = QApplication(sys.argv)
window = MainWindow()
window.show()
sys.exit(app.exec())
```