­­

**OCR A LEVEL**

**Computer Science Project**

**H446-03­­­**

**Name:**  Katy Thackray

**Candidate Number:** 5120

**Project Title:** Gas pressure model

[Analysis 5](#_Toc98146811)

[Analysing the problem 5](#_Toc98146812)

[Goals of the project 5](#_Toc98146813)

[Stakeholders 5](#_Toc98146814)

[Why this project is suitable for a computational solution 6](#_Toc98146815)

[Interview 7](#_Toc98146816)

[Interview Questions 7](#_Toc98146817)

[Questions aimed at teachers. 7](#_Toc98146818)

[Questions aimed at students. 8](#_Toc98146819)

[Interview Responses 9](#_Toc98146820)

[Changes to my solution based on responses 11](#_Toc98146821)

[Research 12](#_Toc98146822)

[Existing similar solutions 12](#_Toc98146823)

[Example 1: Gas Law simulator hosted by The University of Texas 12](#_Toc98146824)

[Example 2: Gas Molecules Simulation hosted by Falstad.com 14](#_Toc98146825)

[Example 3: Gas laws simulation hosted by TeachChemistry 16](#_Toc98146826)

[Further research 17](#_Toc98146827)

[Proposed solution 18](#_Toc98146828)

[Limitations 20](#_Toc98146829)

[Requirements 21](#_Toc98146830)

[Software and hardware requirements 21](#_Toc98146831)

[Success criteria 21](#_Toc98146832)

[Extra features criteria 23](#_Toc98146833)

[Design 24](#_Toc98146834)

[Breakdown of system 25](#_Toc98146835)

[System decomposition 25](#_Toc98146836)

[Class structure 31](#_Toc98146837)

[Key variables 33](#_Toc98146838)

[Inputs and outputs 35](#_Toc98146839)

[User interface design 37](#_Toc98146840)

[Algorithms 41](#_Toc98146841)

[Main loop 41](#_Toc98146842)

[Collisions 42](#_Toc98146843)

[Changing mode 46](#_Toc98146844)

[Moving particles 46](#_Toc98146845)

[Checking a collision with the container 47](#_Toc98146846)

[Changing the temperature of the simulation 47](#_Toc98146847)

[Calculating particle colour 48](#_Toc98146848)

[Testing 51](#_Toc98146849)

[Testing method during and after development 51](#_Toc98146850)

[Post development testing 51](#_Toc98146851)

[Iterative testing 51](#_Toc98146852)

[Testing sub routines during development 51](#_Toc98146853)

[Testing checklist 54](#_Toc98146854)

[Development and testing 55](#_Toc98146855)

[Project organization 55](#_Toc98146856)

[Agile development 55](#_Toc98146857)

[Stage 1 – Setting up the base of the project 55](#_Toc98146858)

[Goals of this stage 55](#_Toc98146859)

[Stage 1 – Development 55](#_Toc98146860)

[Stage 1 – Testing 65](#_Toc98146861)

[Stage 1 – Review 69](#_Toc98146862)

[Stage 2 – Detect and respond to collisions between particles 70](#_Toc98146863)

[Goals of this stage 70](#_Toc98146864)

[Stage 2 – Development 70](#_Toc98146865)

[Stage 2 – Testing 75](#_Toc98146866)

[Stage 2 – Review 80](#_Toc98146867)

[Stage 3 – Simulation and interface development 80](#_Toc98146868)

[Goals of this stage 80](#_Toc98146869)

[Stage 3 – Development 80](#_Toc98146870)

[Stage 3 – Testing 94](#_Toc98146871)

[Stage 3 – Review 104](#_Toc98146872)

[Stage 4 – Modes 104](#_Toc98146873)

[Goals of this stage 104](#_Toc98146874)

[Stage 4 – Development 104](#_Toc98146875)

[Stage 4 – Testing 115](#_Toc98146876)

[Stage 4 – Review 117](#_Toc98146877)

[Stage 5 – More Advanced interface features 119](#_Toc98146878)

[Goals of this stage 119](#_Toc98146879)

[Stage 5 – Development 119](#_Toc98146880)

[Stage 5 – Testing 131](#_Toc98146881)

[Stage 5 – Review 134](#_Toc98146882)

[Testing against checklist from the design stage 135](#_Toc98146883)

[Project Evaluation 137](#_Toc98146884)

[Testing usability of the application 137](#_Toc98146885)

[Testing against success criteria 137](#_Toc98146886)

# Analysis

## Analysing the problem

### Goals of the project

My goal is to make an interactive and simple to use program that can help people simulate and visualise how gas pressure works and how different factors affect it. I want to have a simple and usable user interface where you can change different variables of the simulation such as temperature and watch as the change in environmental factors affects the gas particles.

Essentially the model will present a container filled with particles. Variables will be set to a default which can be changed by the user. The particles will collide with one another as well as the container and the frequency of this will depend on factors that can be tweaked by the user. There are also other possible features I could add, such as different types of gases with different masses.

### Stakeholders

#### Stakeholders of this application

#### The demographic of this program, for the most part, would be physics teachers and physics students as the goal of the project are to demonstrate the behaviour of gas in a fixed container. This program will hopefully help people learning physics to visualize gas pressure and equations relating to it. The interactive nature of the software means the variables can be changed to demonstrate how gas pressure is affected which teachers or students can tweak. Physics students are a key stakeholder for this project because someone who is learning physics may want to test out concepts they have learnt in a simulator. The program should benefit them in learning and understanding physics and should not require them to have much prior knowledge to be able to use it.

#### Stakeholder representatives and how they will contribute

#### The representatives of my stakeholders will be my Physics teacher from my school and an A level Physics student. A physics teacher who has experience teaching physics could use this software to help students understand how the variables involved on different equations all relate to each other and how they apply in a graphical model. My physics teacher would be a helpful example of a stakeholder since he will understand the details of gas pressure and how to best represent it in a model. Having the opinion that understands gas pressure well will be helpful to understand if my model accurately simulates gas pressure.

A physics student is another stakeholder for the application and will contribute to this project as I can get feedback from non-experts on how to lay out the interface to make it simple to use and features that will make this simulator more effective for learning. The person I will be questioning in this project is a year 12 physics student so they will have a good understanding of physics however I would like this piece of software to be usable for students from a large range of years that may not necessarily have a lot of knowledge in physics. Therefore, I plan to make the user interface simple, and it will not use too technical words to describe different buttons or elements of the interface so if someone who is not familiar with physics wanted to use the software, they would not have a problem navigating the user interface.

### Why this project is suitable for a computational solution

Visualizing and experimenting with gas pressure would be best done computationally, with other alternatives such as using real life equipment takes more time, money and possibly has safety risks. It is also the easiest to use without having to set up a lot of equipment. A computer can store the data of each virtual particle and carry out calculations very quickly to present the current conditions of the system simultaneously as the simulation itself is being presented. I believe that all the maths involved in this project can be implemented in this model in a structured way using certain computational methods.

#### Abstraction

Abstraction will be necessary both for implementing the physics in this project as well as presenting the results to the user. This project will not be able to accurately represent all aspects of physics so instead the physics will be abstracted down to the core functionality that is relevant to the model, so the project does not need to be overcomplicated with procedures that are not relevant. This project is focused on gas pressure so it will not delve into other topics such as the state change of gas or any intermolecular forces between the particles that may be relevant in the real-life behaviour of gas.

**Representational abstraction**

To simplify rendering particles will be modelled as circles, which also makes testing for collisions much simpler since circles have an equal radius from all sides, calculating whether one collides with something does not take much computationally. Particles can be modelled as hard spheres with fully elastic collisions to reduce complexity more. The gas molecules will also be modelled as monatomic meaning they will only consist of one singular particle rather than be bound to other atoms, which most gas molecules are in real life, as representing the behaviour of gas doesn’t require bounded particles. This will both simplify the procedure and representation of the simulation.

The container holding the particles will be modelled as a rectangle since that is a simple geometric shape where calculations will not be too complex to apply to. The physics used in general will also be simplified and will treat the simulation to be closed where energy is not lost from the system and irrelevant factors will be abstracted away.

**Information hiding**

There will be classes for components in the simulation such as particles or the container. These will have their own methods and internal functionality that can be hidden as private methods.This will keep the code organized and make the development phase simpler. A class could be responsible for making all the calculations relevant to the simulation while another will use that data and present it on the interface. This way I can keep my project modular so I can develop each part of the project independently.

**Procedural abstraction**

Some procedures can be generalized to be used for multiple different uses. One example where I can apply procedural abstraction would be for drawing different elements of the simulation as they will all have a colour, position and size so a general sub routine could be used for each of these elements. Or procedural abstraction can be applied for procedures involving the particles to work for particles regardless of their size or mass instead of needing to write separate sub procedures.

**Problem reduction and ideal gases**

The problem of representing gas pressure can be reduced by modelling the gas as an ideal gas. An ideal gas is a concept of a gas that perfectly obeys Boyles law and therefore there are a list of assumptions made about the gas. In reality, ideal gases don’t exist, but they can be used as a concept to present the behaviour of gases mathematically. Below are a few of the assumptions made for ideal gases.

* Volume of individual particles is negligible compared to volume of the gas
* Collisions are elastic (no loss in kinetic energy)
* No forces between the particles (unless they are colliding)
* Temperature is directly proportional to average kinetic energy of the particles
* Particles are monoatomic
* Particles move randomly
* The gas does not change state

For calculating statistics, I can use these assumptions to reduce calculations I would need to make. I will not need to include the volume of the gas particles, loss in kinetic energy or intermolecular forces for my solution which will make this problem much more manageable. Other assumptions such as the gas not changing state or particles being monoatomic can be used to reduce the complexity of modelling the gases in my simulation as I won’t need to worry about implementing this in my model.

#### Top-Down Approach

The best solution for me to solve this problem would be to use a top-down approach to separate each aspect of it into manageable sections. A top-down approach can be applied to this software by splitting the solution into three main components, the management of the simulation, rendering the simulation and the user interface for the simulation. Managing the simulation can be broken down further by updating the positions of the particles as well as updating any variables that need to be updated as the simulation changes. Rendering can be broken down into all the elements that will need to be rendered and how they will be rendered to the screen. The user interface will manage buttons for interacting and statistics to inform the user what is happening in the simulation. These sub tasks can be broken down further to smaller and more approachable problems. As each part of this problem can be broken down into many different sub tasks, a computational approach to solving it will be appropriate as a system with many modular components and sub procedures can be used to solve this problem as a whole.

#### Thinking Ahead

My solution will include many steps as well as components which will need to interact with each other to work and performance is also a factor that I will consider, so I will need to carefully plan out each component and how it will work so that my code is organised as well as optimised. Having a clear plan for how the software will be designed before programming it will make the development process as well as final working product of my model much simpler.

All the inputs will be through the graphical user interface. The inputs will consist of buttons and sliders to change values rather than directly entering a value for a certain variable to be since that will make the program less simple to interact with and validation will also be simpler to maintain in the inputs with sliders since they can have a predefined minimum and maximum which the user cannot go out of bounds. The main output will be the graphics output from the simulation as well as the values that certain variables, such as temperature or average velocity, are at.

#### Object Oriented Programming

Object oriented programming can be integrated into this project to organize and compartmentalize each part of the program and allowing for a more modular design to the software. Particles can be implemented as a class with all their data and methods encapsulated by that class rather than just having a massive set of data all in one table to store information about the particles which would be much less robust and would complicate the system. A class could represent the simulation as a whole and store a list of these particle classes while having methods that operate on them. Polymorphism could be used to abstract the simulation by generalization such as with a particle, where there is a base class including general methods, a particle would need and multiple sub classes that extend from it with their own different behaviour and functionality. This problem will include a lot of data and operations that will need to be carried out on that data, which is an ideal situation to use object-oriented programming, therefore this programming paradigm suits this problem.

## Interview

### Interview Questions

The goal of the interview is to get an understanding on what features of this project will be most helpful, how teachers or students would like the user interface to be laid out. It is also to get a general opinion of this software and if or how they would like to use it.

### Questions aimed at teachers.

**General questions:**

* Do you think simulation/visualizer programs are a useful tool in teaching? Why?
* Have you used any software before to help teach students a certain concept in physics?
* If yes, what kind of simulations did you use and do you think they were helpful? Why?
* How in depth would you want an educational physics simulation to go into the subject if you were using it in a class?

**Specific questions about my idea for a gas pressure model:**

* For using a gas pressure model would you prefer the user interface to be simplistic or have many features, buttons and presented information.
* Are there any features or you would want included?
* Do you think mathematical accuracy, or the visual representation of gas pressure is more important in the model? Why?

My questions here are quite open to allow for further questions to be asked so I can get a more detailed response. The general questions are to establish the opinion and history of teachers using software to help teach physics and get their opinion on how effective they were for teaching.

Questions 1, 2 and 3 are to get an idea of what my physics teacher’s experience with visualizers/simulations is and allows for further input about their experience.

Question 4 aims to get an opinion of how in depth they would want a simulation to be. I hope to get an understanding on what features of a physics simulation are good for educating from a teacher's point of view.

The more specific questions are there to get an idea of how in depth the project should be mathematically as well as how I should balance the complexity of the user interface with the inputs and interactive features I may want to include.

Question 5 investigates how they would like the user interface to look. Since they are a representative of the target audience for this project it is important to have an understanding on how they would prefer the programs interface to be.

Question 6 is quite an open question that will help me get an understanding of what a teacher would want to see in a program like this. This could include extra functionalities, elements of the interface or anything they would like to see included in the software.

Question 7 focuses on what my priorities should be regarding this program between mathematical accuracy and the visual representation of gas pressure it provides. Since this is something, I will need to balance it would be helpful to get a stakeholder's opinion.

### Questions aimed at students.

**General questions:**

* Do you find gas pressure in physics as a concept hard to visualize? Why?
* Have you used or been presented with any software that helps you visualize certain physics concepts?
* If yes, what features about them did you like or dislike?

**Specific questions about my idea for a gas pressure model:**

* Would you find a piece of software that can model gas pressure helpful?
* How would you want the interface for this software to look?
* Would you prefer complex details to be hidden or displayed on the interface by default? Why?

The general questions are to understand if students have used or seen software that model physics concepts and how effective they were from a learner's point of view.

Question 1 directly asks the student If they find visualising physics challenging since the goal of this project is to help people visualize and understand gas pressure. It would be helpful for me to know if they may benefit from this project since that would affect their responses to the later questions.

Question 2 and 3 focuses on the students experience with simulation/visualizer software as well as what features they liked about them which I can consider for my project to benefit my target demographic.

The more specific questions will hopefully help me get an outline of how students would like the user interface to be like since usability of the software is most important with learners as I want the software to be able to be used by people who are not necessarily familiar with physics.

Question 4 is to have a clear answer on if they think this kind of program would be useful to them.

Question 5 and 6 is to investigate how a student would want the interface layout to be as well as their opinion on if more in depth details are shown in the interface.

I do not ask about certain functionalities or features that they may want included since it requires some knowledge about gas pressure that not everyone may be familiar with.

### Interview Responses

Here is a summary of the responses I received from my interview with stakeholders.

**Responses from my physics teacher**

1. **Do you think simulation/visualizer programs are a useful tool in teaching? Why?**

“*Yes, I use them for many areas of teaching to cement understanding, some people are more visual learners and a simulation can be helpful to them*”

1. **Have you used any software before to help teach students a certain concept in physics?**

*“Yes, I have used them in the past for F=mxa and for building circuit, both for visual learners and a way to show the class something that would take a long time with real equipment. "*

1. **How in depth would you want an educational physics simulation to go into the subject if you were using it in a class?**

**“***Depends upon the year group being taught, for example at GCSE we discuss P x V = constant but at A-Level we use PxV = NkT, So I prefer it if it can combine complex features with a simple presentation and more simple features***”**

1. **For using a gas pressure model would you prefer the user interface to be simplistic or have many features, buttons and presented information?**

***I showed an image of an example of a gas pressure model here since I thought this question could be improved by having a teacher’s opinion on the interface of an already existing solution. The example I used was from the*** [***first solution I researched***](#_Existing_similar_solutions)***. Since it is quite simple to be given to analyse for a question.***

“*The one you have demonstrated above looks good, but year 9's doing particle model would not understand Rms so you may want RMS as well as Average part hidden*”

1. **Are there any features or you would want included?**

“*You may want to call gas A and gas B Hydrogen and Oxygen so there is a comparison they can link to in terms of periodic table and size*"

1. **Do you think mathematical accuracy, or the visual representation of gas pressure is more important in a model? Why?**

“*The P x V = constant is very important, but I think the visual representation is more so because to understand the laws relating to gas you must first understand the concept of gas pressure*”

**Response Analysis**

The stakeholder has experience using simulations for teaching. They mention in question 3 that different amounts of complexity relating to the subject is necessary for different levels of teaching. Since I want my simulation to be usable for a wide range of ages and levels in physics, I believe I should include these more advanced functionalities.

My physics teacher points out it should have a simple presentation. The solution could be very lightweight and have some more complex features available in the settings that the user can choose to enable to balance having these functionalities along with a clear interface.

There is also mention of the existing solution I presented having too many details that require a certain level of understanding so I should try to make sure my solution does overcomplicate the interface with stats that may not be understandable to everyone. He also believes having two types of gas would be beneficial since their behaviour can be compared. The response to question 6 is quite useful since it highlights the importance that P x V = constant so I will have to make sure this is true in my solution.

**Responses from students**

1. **Do you find gas pressure in physics as a concept hard to visualize? Why?**

“*Yes, questions about this topic I often struggle with. I just don’t understand how the equations work and I find the concept of pressure quite confusing.*”

1. **Have you used or been presented with any software that helps you visualize certain physics concepts?**

“*Yes, in class we used a visualizer for relative speeds. You could add objects that go at some percentage of relative speeds compare their length and time dilation*.”

1. **If yes, what features about them did you like or dislike?**

“*I liked that it was a simple and easy to use and you could add many different objects going at different speeds to compare. However, there wasn’t an easy way to reset the simulation and you just had to delete all the objects you added. Also, someone who doesn’t know about relative speeds would not understand what the simulation is showing at all*”

1. **Would you find a piece of software that can be model a gas pressure helpful?**

“*I think so if it is easy to use and not overly complicated. Having something where I can simulate gas pressure may help me understand it more.*”

1. **How would you want the interface for this software to look?**

“*I would like it to be simple and straightforward where it’s clear what each button does*”

1. **Would you prefer complex details to be hidden or displayed on the interface by default? Why?**

“*I think by default I would like them hidden however there could be an option to show them if someone was interested in seeing them*”

**Response Analysis**

The response to question 1 shows that the student would likely benefit from a piece of software like this as they have a hard time understanding gas pressure meaning that their responses should be quite beneficial. The student mentions using a visualizer for relative speeds with the main complaints about not having a reset button and the simulation not including much of an explanation by what is showed by the simulation. The solution would need to include a reset button in the interface which will change all the variables back to their default values. I could also add a help option that may give certain features of the simulation some explanation.

### Changes to my solution based on responses

After interviewing my stakeholder representatives, I have decided to add these features/improvements to my solution to improve the experience of people using my application.

* Buttons such as a help button and a reset button to make the application more usable
* It’s important the application represents equations such as and other laws properly.
* The interface should have simple presentation with certain statistics being optional for the user
* Having an option to switch the unit of temperature between temperature and kelvin so students below an A level understanding of physics can still understand what is happening in the model

## Research

### Existing similar solutions

### Diagram Description automatically generatedExample 1: Gas Law simulator hosted by The University of Texas[[1]](#footnote-2)

**Analysis**

An example of an existing similar solution is one of the simulations hosted by the university of Texas’s Chemistry department to simulate gas laws. It has a very simple interface with a few features you can interact with. Since this simulation hosted by the university’s the stakeholders for this example are university students while the stakeholders for my project will be GCSE to A level physics students and teachers. Therefore, some elements of this example may not work well for my solution however this solution has the similar goal of presenting the behaviour of gas and there are certainly some features I can use to adapt my project. Below is my analysis of some elements of the application which have been labelled on the diagram

1. This example only uses the kelvin unit to represent temperature, while this may be appropriate for university students as kelvin is the standard unit of temperature. However, with the stakeholders of my application, there will be wider ranges of understanding of thermal physics so they may not be familiar with kelvin and therefore this an alternative unit should be provided so this statistic can still be informative.
2. Two types of gas molecules for gas are used in this example which is a helpful way to show how mass can affect the energy and velocities of the particles. It should show the user how particles of larger mass will retain the same average kinetic energy but lower velocities.
3. These statistics can be quite helpful for someone who understands their meaning however people who don’t understand the terms rms (root mean square speed) or KE (kinetic energy) will not find these statistics useful so having a slightly more detailed label for these stats will be best for my program.
4. The presentation of the simulation is quite practical with inputs being presented as actual equipment such as clicking on a gas pump to increase volume. Having the application present the equipment that would be used for a real-life experiment involving gas (such as the gas pump) pressure is a good way to show how this would work in an actual experiment as well and I think it is a good feature.
5. In this application the particles moving in the cylinder are more of an animation rather than an actual view of what is going on. For example, shown in label 5 there are two particles passing through each other, while they should bounce of each other and transfer energy in the process. I think having the model of the showing what is happening and including the collision of particles would be a more immersive experience for the user as they can see exactly what is going on in the simulation as well as have the statistics be directly linked to that view of the simulation so changes in energies after a collision can be seen in real time.

#### Parts I can apply to my solution

The parts I will apply to my solution from analysing this example are listed below. The practical presentation element I’ve added as an extra feature as despite being a good way for the user to visualize it would require me to draw the apparatus for my project which could be very time consuming and take away focus from the actual functionality of the project.

* Simple user interface (However with some more features)
* The practical presentation of this simulation (As a possible extra feature)
* Having two different types of gases that can be used

### A picture containing graphical user interface Description automatically generatedExample 2: Gas Molecules Simulation hosted by Falstad.com[[2]](#footnote-3)

**Analysis**

This gas simulator is a java applet hosted on a website that has many physics related simulations freely available to use. Its purpose is to demonstrate the kinetic theory of gases, therefore there is a focus on how energy is transferred between particles. Unlike the first solution it has a much larger focus on functionality between the model shown on the right and the statistics which is how I will also be implementing my application so analysing this application was very beneficial to further developing my proposed solution. Below is my analysis on some of the applications features I would like to focus on.

1. The statistics and equations shown could be informative, especially if a user wanted to test them against the formulae they know however they do lack units and there is not much clarity on what each value means. For example, PV/NkT should theoretically be 1 for an ideal gas so this statistic can show how close the model is to representing an ideal gas but there is no explanation for this. Incorporating these statistics would be helpful however a clearer presentation would be necessary for these statistics to be informative to some of my stakeholders.
2. Having particles be coloured depending on their kinetic energy is a great way to see how the energy between particles are transferred as you can see the particles change colour as they collide showing the effect of the collision. Since these particles have the same mass, this can also represent their change in velocities which may also be a helpful thing to visualize as the momentum of particles does is proportional to pressure.
3. This application has multiple modes that can present how an ideal gas would behave in different scenarios such as the *1 Gas, One moving molecule*option which will present many stationary particles and one fast moving particle and you can watch as the one particle collides and causes the other particles to move and collide, distributing energy across the gas. These different scenarios for the gas allow for more exploration of the simulation and gives more uses to the simulation.
4. The graph underneath the simulation shows the distribution of the particles’ kinetic energy and will move as the temperature changes showing how the mean kinetic energy will increase as temperature does. This is a relevant feature for a simulator presenting kinetic theory however, my solution does have more of a focus on the gas laws, therefore this feature may not be utilized as much.
5. The stop button is also a useful feature if the user wanted to calculate the difference between two variables at two different points in time and allows the user to see the exact state of the simulation at that point in time.

#### Parts I can apply to my solution

This solution had many interesting features that, despite having more of a focus on kinetic theory, that would also be helpful in presenting/simulating the gas laws to my stakeholders. Including different scenarios would be an extra feature as my focus is the gas laws however it could still be useful for some of my stakeholders as it’s related to gas pressure it could be implemented if there is still time after developing the core elements of the application.

* Different scenarios for the simulation to be in (Possible extra feature)
* Particles being coloured based on their kinetic energy
* The inclusion of valuable statistics (however with some more explanation and units)
* A pause button

### A picture containing diagram Description automatically generatedExample 3: Gas laws simulation hosted by TeachChemistry**[[3]](#footnote-4)**

**Analysis:**

This web application is hosted by teachchemistry.org and has more of a focus on the experiment regarding gas pressure rather than the theory as the process of doing an experiment is simulated in this application. Despite my project being more about getting an accurate model for the gas laws, this software has some unique qualities that I could apply to my solution. Below is an analysis of the relevant parts of the program.

1. Allowing the user to plot a graph of data they have gotten by changing variables is an interesting idea as it gives this more practical experience than other simulations and allows the user to graphically see how variables change proportionally to one another. Since the main stakeholders for this program are chemistry students and chemistry being generally a more practical/experimental subject than physics this feature may not be as relevant to my solution and my problem is more focused on visualising/exploring gas pressure rather than the experiment.
2. Having gas laws be separated into different modes would be a very helpful solution to showing how each of them work. For each mode a variable is left constant so the user can see how two other variables will affect each other when altered. This use of different modes to help visualize the gas laws separately would be helpful especially for my stakeholders taking the physics A level because these laws are a part of the syllabus.
3. The statistics are shown clearly with units which makes them much more readable and generally more informative. The only statistics shown are the ones which are relevant to the law being shown which does mean everything on the interface is relevant to the purpose of the simulation.
4. The model being used to help visualize is much less robust than the previous so there is much less to explore in the program as the experimental aspect of the program is straightforward. Since I want my program to be used in a way that people can explore the gas laws and freely experiment with them by using a simulation, I would require a much more robust model and one that directly effects the statistics shown so all the mathematics going on can be presented visually on screen.

#### Parts I can apply to my solution

Having modes dedicated to gas law will help users understand each law individually and I think would be beneficial to my stakeholders. The interface of this simulation is also quite straightforward and doesn’t require too much understanding of physics to use which I would like to carry forward to my solution. Below are the aspects of this solution I will carry forward to my solution.

* Data shown clearly with units
* Different modes focused on a specific gas law
* An option to display graphs on how variables have changed (as an extra feature)

### Further research

**Handling collisions between particles**

In my simulation particles are going to collide with each other in the container and when they do their velocities and energies will likely need to change. The conservation of momentum must be conserved to simulate this and the collision between particles can be modelled as elastic (no energy is lost) to simplify the calculations. One challenge would be calculating the change in velocity for each particle in the collision, so I researched this aspect and found this useful equation from a web page about simulating ideal gas particles[[4]](#footnote-5)

A picture containing text, clock

Description automatically generated

This formula shows how the velocity of particle i () would change as it collides with particle j (). With the each particle’s position being represented by and velocity by This formula shows I will need to get the vector going fromto (), perform a dot product with it and the difference in velocity between and , multiply by - and then divide by . Using this formula, I can simulate velocities changing as particles collide.

**Rendering engine**

I will need some way to render the simulation to the screen. For this project I will just need something very simple since all it would really need to do is draw the particles to the screen at their positions, manage inputs from the mouse and update. I also would like it to be cross platform, so the software is accessible for multiple operating systems. I chose to use the olcPixelGameEngine[[5]](#footnote-6) because it fits this description and is very simple to implement since it is just a header file I will need to include in my project.

## Proposed solution

#### Main features of proposed solution considering research

After researching other solutions and interviewing stakeholders, I have a summary of the features of the proposed solution.

|  |  |  |
| --- | --- | --- |
| **Features** | **Justification/Limitation** | **References** |
| 2D graphics for the simulation | Using 2D graphics to represent gas pressure will greatly reduce the complexity and the computation needed for the simulation to run. Presenting gas pressure in 2D will also make it clearer for the user to see what is happening. In real life gas particles are 3 dimensional so there is a slight lack in realism however this is minimal. | All the similar solutions I researched used 2D graphics |
| Modelling gas as an ideal gas | Modelling the gas as an ideal gas will still represent gas in most conditions accurately however greatly reducing calculations I will need to make and simplifying the problem by making assumptions about the gas I’m modelling. While this may not model real gas perfectly it will be appropriate for my simulation which mainly focuses on the gas laws. | The abstraction section of this stage highlights why this is a suitable approach for problem reduction |
| Different modes | The use of different modes will allow the user to experiment with gas with different preconditions or environments. This will make the project more helpful to my A level stakeholders especially as they can experiment with each of the main gas laws individually. | A feature I decided to add after analysing the solution 2nd Example of the similar solutions I researched |
| Each gas law being shown in each mode | Each mode can focus on one of the gas laws by keeping one of the variables constant so the user can see the proportionality between variables in the simulation. The laws that will be shown in my simulation will be Boyles law, Charles law and the pressure law. | I added this to my features after looking at the 3rd example I researched |
| Statistics shown on the user interface | It’s important that the simulation shows statistics about what is happening as it will give some actual values to be shown along with the visuals of the simulation. Statistics will include the value of each of the variables and other relevant information such as the average kinetic energy or the mean velocity of the particles. These values are used in many formulas related to gas pressure so including them will allow the user to explore the mathematics of the simulation if they want to. It’s also important all the statistics have units so what they represent is clear to the user. | All the simulations I researched had statistics shown on the interface, however in different ways. |
| A reset and pause button included in the interface | These two buttons will allow the user to control the simulation more by being able to revert it back to its default state or pause it’s running. The pause button could be useful if the user wants to see the exact value the statistics are at that point. | I added a reset button as a feature after my interview responses and a pause button after researching Example 2 |
| Two types of gas particles that can be used | The simulation will have two types of particles with one heavier and larger than the other. This will help users see the difference in between behaviour of particles depending on their mass as different gas particles do behave differently from each other. | From researching the 1st similar solution |
| The colour of particles in the model based of their energy values | This would be an option which the use can toggle for particles to be a colour that represents their energy value as a colour relative to a scale of colours that corresponds to an energy. This will allow the user to see in real time how collisions have affected the particles or how particles kinetic energies have changed as temperature has increased. | From analysing example 2’s approach to representing energy in particles |
| Unit of temperature can be switched between kelvin and degrees | This will make my application be more accessible to users of different levels in physics so the unit of temperature can be presented how the user prefers. If a unit is not understood well by the user it is not very useful In showing information to them. | I concluded I would add this feature after my interview response analysis |
| A help button | A help button is a necessary usability feature as it will give guidance on how to use the simulation which may not be apparent for someone using it for the first time especially and the application should be made accessible as possible | From my interview responses |

#### Extra features

|  |  |  |
| --- | --- | --- |
| **Feature** | **Justification/Limitation** | **References** |
| Giving an option for the user to view graphs produced from the data of the simulation | This feature may be helpful for users as they can see graphically how variables have changed. This feature however would require a very large amount of extra processing for a feature that many of the users will not use. Therefore, this will be one of the extra features which I could include If all my main features have been implemented. | Example 3 plotted and presented graphs which clearly presents how the laws affected each variable |
| Sprites of experimental equipment in the user interface | Representing inputs/outputs as practical equipment in the user interface may be helpful understanding the practical application of these experiments involving gas pressure However, this is a feature which will likely not change the experience of using the application for most users and may take a lot of time designing so this will be an extra feature I will consider. | The solutions 1 and 3 included this and made the user interface look more interesting as well as practical |
| Providing different scenarios | Different scenarios for the simulation to be in such as the ones described reviewing the solution hosted by Falstad.com could give the user more insight on how gases behave under certain circumstances. Different scenarios could be implemented as modes like how the gas laws will be implemented. This feature, however, does move away from my focus on presenting the gas laws so it is not a main feature. | The solution I researched hosted by Falstad.com had many different interesting examples to experiment with adding to the functionality of the application |

#### Features I’ve decided to not include

|  |  |  |
| --- | --- | --- |
| Feature | Why it won’t be included | References |
| Gravity affecting particles | On smaller scales gravity does not have a large affect on the trajectory of gas particles and therefore implementing gravity in the simulation adds needless complexity to it. | The 2nd example I looked at included this as a feature however there was not a reason to use it because it did not show much about the behaviour of gas |
| An energy distribution graph to represent distribution of energy across the particles | This feature could be informative if my solution was focused on the energy of particles however the focus is on the gas laws and producing an energy distribution graph would require a large amount of processing make the user interface more cluttered | The 2nd example includes an energy distribution graph however as established analysing that solution it does have a larger emphasis on energy than the gas laws |
| Allowing the user to plot the graphs themselves using the interface | While this was an interesting feature, I came across researching other solutions, it did not really add much to the application and such a feature would require a lot of code to implement therefore it is not viable for my application. | Example 3 included this as a feature. The user would change a variable and then click the ‘Add Data’ button to add it to the graph. While this was quite interactive it would be more effective to simply present the graphs. |

#### Stakeholder feedback on proposed features

To make sure my physic teacher who represents my stakeholders thinks all these features will be useful additions to the simulation I explained all the features I plan to include to get his opinion on them.

“*The features you plan to include are all relevant for a simulation about the gas laws. I think the feature where each law will be in a different mode sounds like it would be very helpful in a teaching environment. The feature where the particles will have colours according to their kinetic energy isn’t directly related to the gas laws but the distribution of energy across all the particles would clarify that particles in gas have a range of energies which change as they collide with each other so it could be a useful feature. Your proposed solution for the simulation covers all the features I would need for a simulation to teach the gas laws. The extra features don’t seem like they would be necessary for most people using the application so I don’t think there should be too much focus on them.”*

As the proposed solution and features is sufficient for my stakeholders then I will make sure to have all these features are included in the application. As my physics teacher doesn’t think that the extra features will add a lot to the application I will only consider adding them after all the main features have been developed if it takes less time than expected.

### Limitations

I believe that this project is very achievable however there are some limitations to my solution. There will be limitations to the project due to constraints such as time or resources. The program will not be able to emulate all real-world physics phenomena's that may come into play when it comes to gas pressure as this would take a lot of processing power and would needlessly increase complexity. Instead, I will instead focus on the core environmental factors that would affect the collisions of particles in a container such as temperature, number of particles, volume of container as well as the gas laws. The solution will also be limited by my own understanding of the concept, for example I will not be able to present gas pressure at a university level since I would need a strong understanding of it to be able to simulate it in a program, so I am limited to an A level understanding of gas pressure in this project. The abstraction section in this project details how I will model gas pressure without the complexity of real-world physics.

## Requirements

### Software and hardware requirements

#### Hardware

**A computer that can run software with standard IO peripherals –** This software will require a mouse/trackpad to interact with the user interface and click buttons. A screen will be needed to show the simulation as well.

Since there will likely be a lot of processing that needs to be done with many particles in the simulation this software will need a computer with at least 2Gb of RAM to run well. Almost all computers nowadays will have more than 2Gb of RAM, so I don’t believe this amount of RAM is too high of a spec even for school computers.

The program should not take more than 500KB of storage as there is no database that will be needed and all that should need to be stored is the program itself. Any storage device with more than 1MB of available storage space will be sufficient to run this application.

#### Software

**Window, Mac or Linux operating system –** These operating systems are supported by this piece of software, the code is cross platform and would just need to be compiled for each of the operating systems.

### Success criteria

|  |  |  |  |
| --- | --- | --- | --- |
| No. | **Criteria** | **Explanation/Justification** | **How to evidence** |
| 1 | Current state of simulation shown in the window | The state of the simulation needs to be updated and rendered in real time for the user, so it is clear what is going on. | Screenshot of window showing the state of the simulation |
| 2 | A clear user interface to interact with the simulation | An interface is needed for the user to interact with the simulation and one that does not require any knowledge for someone to use is ideal for this piece of software to be accessible to anyone. | Screenshot of window with a clear interface, large text and large buttons with distinct colours to make them easy to read |
| 3 | Different variables that control the simulation | For this to be a useful simulation it’s state will need to change based on a handful of variables. This is necessary for the simulation to be interactive to the user. | Buttons being used to change variables shown in a screen along with screenshots of variables being changed in the simulation. |
| 4 | Reset button | Change’s simulation state to default state. If the user changes many variables, this will allow them to easily revert the simulation back to its default state. | Testing evidence to show the reset button works. |
| 5 | A button to switch between modes | To allow the user to change what mode the simulation is in as my simulation will include modes there will need to be a way to switch between them. | Testing evidence showing the button will switch to each mode |
| 6 | Help button | This button will give the user some guidance if they don’t know how to use the program | A screenshot of the help text that comes up after clicking the button |
| 7 | Pause button | This will allow the user to stop the simulation if they want to see the exact state of the simulation at that point in time | Testing evidence of the button working with video evidence of it The pause buttons stopping and starting the simulation |
| 8 | Multiple modes/examples | To provide multiple different ways the user can experiment with the software and demonstrate different gas laws. | Showing a screenshot of the program in each mode as the and evidence from development |
| 9 | Default mode | The default mode will not be related to a specific law and therefore should allow the user to change any of the variables in the simulation with none needing to be kept constant. | Screenshot of application on start-up where any variable can be changed |
| 10 | Charles law mode | The user can experiment with the volume and temperature to see how gases expand when heated | Screenshot of the program in this mode |
| 11 | Boyles law mode | User can experiment with the volume and pressure at a constant temperature to understand their proportionality. | Screenshot of the program in this mode |
| 12 | Pressure law mode | To allow the user to experiment with relationship between pressure and temperature while volume is kept constant. | Screenshot of the program in this mode |
| 13 | Brownian motion mode | This is mode will include one very large particle amongst many smaller particles. It will help the user understand the random motion of particles by how they change the path of the large particle. | Screenshot of the program in this mode |
| 14 | A different interface for each mode | This is necessary as each mode will have different parts of the simulation that can be changed. | Screenshot of each mode’s interface |
| 15 | Upper and lower limits for some variables in the simulation | Variables can be changed by the user so it is important that the program has limitations for each variable so they cannot be given absurd values. | A screenshot of the code that makes sure variables are kept within limits |
| 16 | Gas laws being represented | This software should present gas laws accurately for this to be a valid simulation of gas pressure. | Providing the code for each law and testing evidence in the development stage |
| 17 | Options to represent data | These can be useful if the user wants to have more insight on what is going on in the simulation. This data may include number of collisions, average speed, average energy etc | Screenshot of the options on the interface along with the code of each option shown. |
| 18 | Ability to calculate pressure, rms and average kinetic energy | This is so these statistics can be presented to the user | Code for calculating these values and screenshot of interface with these values displayed |
| 19 | Option to have particles be coloured in proportion to their energy | This feature will be used to show how energy is transferred when particles collide and show the scale of energy of all the particles in the simulation. | A screenshot of with this option enabled showing the colour scale of particles |
| 20 | Units being shown alongside statistics | Units are very important for understanding the meaning behind a value of something, so they are necessary in my solution | Units being used in the interface of my application |
| 21 | Top-down approach | Since there are multiple components that will need to work together to make the solution having a top-down approach to designing and developing the program a top-down approach would be appropriate. | Design stage resembling a top-down approach |
| 22 | Modular structure of the application | Having the structure of the application be modular will make development and testing stage much more organized and simpler | The code of my programming being separated into different modules for different purposes |

### Extra features criteria

Below are the criteria to evidence the optional features that I may implement along with a summary of why these features are not a part of the main set of criteria. If I decide to implement any of these features I will have some criteria to review it against

|  |  |  |
| --- | --- | --- |
| **Criteria** | **How to evidence** | **Why it’s not part of the main criteria** |
| Giving an option for the user to view graphs produced from the data of the simulation | The code shown along with the graphs produced by the simulation | This may require a lot of processing power and is not the main point of the application |
| Sprites of experimental equipment in the user interface | An image of the user interface with practical equipment presented as inputs/outputs in the user interface | I will need to draw the assets for this feature which may take time away from developing the more important elements of the simulator. |
| Different scenarios for the user to experiment with gas laws | The code shown for different scenarios | The focus of the simulator is to present gas laws so it would be an extra feature as it’s also not a core part of my solution |

# Design

* Breakdown of the system
  + System Decomposition
  + Class Structure
  + User interface design
  + Key variables
  + Inputs and outputs
* Algorithms
  + Main loop of application
  + Updating the Simulation
  + Checking for collisions
  + Handling user input
  + Changing mode
  + Calculating particle colour
* Testing design
  + Testing method
  + Testing data
  + Testing checklist

## Breakdown of system

### System decomposition

The figure above shows an overview of how the simulation will operate and a breakdown of its components. The three main operations of the software are rendering the simulation, managing the user interface, and controlling the simulation. These three operations are split into other operations that make up the task they are a part of.

Operating the simulation will involve updating and managing its own state and processing its data which can be used for further calculations regarding the simulation or for statistics. The user interface will need to change itself if the mode is switched, detect and respond to buttons being clicked, and update any statistics displayed to the screen so everything is kept up to date. Rendering will be split into the different parts of the window that need to be drawn and some further calculations will also need to be done so they are rendered in the correct position and colour.

**Success criteria links:**

• Top-down approach

• Different variables that control the simulation

#### Rendering

There are three main elements that the simulation will need to render, the particles, interface elements and any sprites and images that will be used. While having sprites used in the interface of the program is not one of my main features, having the functionality to draw sprites should be included if I were to incorporate any sprites into my application, such as one of my extra features where the interface will have sprites resembling apparatus that would be used in an experiment. Below is the summary of the sub tasks in this diagram.

**Summary of sub tasks**

Particles

* Determine colour – If the option for scaling the colour of the particles based on their kinetic energy is enabled, there will be a sub routine that will calculate this colour before rendering the particle. If the option is disabled the colour will be a default colour held as a field in the particle class
* Find on screen position - The particles position is in the context of the simulations space so this will need to be altered to be in the right position on the window
* Use correct radius – There are two types of particles so the radius will vary meaning the radius of the particle will have to be queried before rendering

Buttons

* Centre text in the centre of the button – To have a clean interface the buttons’ text should be centred. This can be done by function that uses the length of the button and the length of the string to calculate the offset that the text is drawn inside the button
* Colour the button differently when clicked – Usually when clicking a button on an interface it changes colour for a brief period of time which lets the user know that the application registered that the button has been clicked. This can be simply implemented by making the buttons colour slightly darker if it was clicked in the last second

Text displays

* Retrieve data being displayed – As stats are shown in by the text displays, they will need to be updated every time the simulation updates, so before rendering the text display will retrieve the value it needs from the simulation
* Concatenate data with title and unit – the text display will have a title for the data e.g. ‘Temperature’, data e.g. ‘30’ and a unit e.g. ‘**°**C’. These will all need to concatenated to form the string that will be rendered and this may include type casting.

Sprites/images

* Load sprite – The file for the sprite will need to be loaded in along with any parameters related to the sprite that will be used.

#### Simulation Decomposition

Diagram

Description automatically generatedThe tasks that need to be done to operate the simulation can be separated into two groups, Managing the simulation, and processing the data regarding the simulation. Some of the more computationally heavy tasks would be checking for collisions and calculating averages as these will require each particle to be processed on for each loop so it is important the algorithms for them are efficient.

Most of these tasks will be managed by one class that will hold all the information about the simulation so that all the data necessary for calculations can be easily accessed within that class. This part of the software will need to take inputs from the user interface regarding user input and it will need to calculate statistics to display on the interface. As shown by the diagram the larger tasks are made up of smaller sub tasks which are described in more detail below.

**Managing the simulation – Summary of sub tasks**

* Compare each particles position to other particles – This task will consist of calculating the distance between each particle and checks if they collide. This is a necessary function for collisions to be detected in my program
* Calculate change in velocities – Calculate the resulting velocities of both particles after they have collided. Energy transfers between particles when they collide so their velocity must change
* Calculate change in energy transfers – This will be done after calculating a change in velocities and will just use a simple formula using the resulting velocity previously calculated.
* Measure the time between each frame – Time for each frame needs to be measured to keep the speed of the simulation consistent regardless of performance. This will be done by recording the time before and after processing a frame and finding the difference.
* Run each particle move method – A simple loop which runs the ‘move’ method for each particle. It will pass in the time for each frame as this is relevant to the distance the particle moves relating to its velocity.  This method will be needed to update the particle positions.
* Make sure areas are kept in defined bounds – Each variable will need to have limits and therefore there will be some validation to check that the variables stay within the defined limits.
* Control the increase and decrease in temperature – The increase in temperature will also increase the kinetic energy of the particles within the container and this will be done over a period of time, so the kinetic energy of particles doesn’t increase too rapidly.
* Update values when variables included their calculation have changed – For every update of the simulation variables such as rms or average kinetic energy may change so they need to be recalculated each frame which can be done in a sub routine.

**Processing data – Summary of sub tasks**

* Decide what data needs to be processed – There are some options on the user interface to enable/disable certain statistics to be shown. To improve performance, stats that are not enabled shouldn’t be calculated unless necessary for another calculation. The system can ignore any calculations that aren’t enabled in the interface.
* Calculate rms – In thermal physics the root square speed is used to show the average speed of particles. This will be a sub routine which finds the average of the square of each particle’s speed and the square roots it.
* Calculate pressure – This will be done using the equation   so the rms should’ve already been calculated to avoid calculating it twice.
* Calculate average energy – The average energy of the particles will be calculated so the user can have an idea of how average energy will change as temperature or other variables change.

**Success criteria links:**

* Upper and lower limits for some variables in the simulation
* Multiple modes/examples
* Gas laws being represented
* Ability to calculate pressure, rms and average kinetic energy

#### Diagram Description automatically generatedUser interface decomposition

This diagram represents all the inputs and outputs that the user interface will consist of. Text displays will be what display information about the simulation while the buttons will be inputs for the user interface. On the left the buttons used to alter the simulation are shown however these will vary depending on what the current mode of the simulation is as some modes will need to keep some variables constant so they cannot be altered by the user. Below shows what variables can be changed by which modes.

|  |  |
| --- | --- |
| **Variable that can be changed** | **Modes** |
| Temperature | Default, Charles law, Pressure law, Brownian motion |
| Volume | Default, Boyles law, Charles law , Brownian motion |
| Number of particles | Default, Boyles law, Pressure law |

As each interface will have a different set of buttons and different functionality, I will have a parent class that contains all the general buttons, text displays and layout with each mode being a sub class of the parent class that adds its own set of buttons, methods, and restrictions (such as the need for keeping some variables constant).

As mentioned, while breaking down the simulation into smaller tasks, the user interface will need to communicate with the simulation handler so the parent class for the user interface will also contain a reference to the class operating the simulation and it is important that it’s a reference and not a copy so that the user interface is accessing the instance that is being used and changes can be made to it.

**Text displays – Summary of each display**

* RMS – The root mean square speed, will represent the average speed of each particle. Will be useful for any calculations the user may want to make or just to generally see if it is decreasing or increasing.
* Constant variables – This will make it clear to the user when a variable is being kept constant so it’s obvious when other variables are being altered by the simulation to keep that variable constant.
* Average energy – The average kinetic energy is similar to presenting the velocity however it will still be useful for calculations an demonstrating how energy differs between different types of gases.
* Temperature – Temperature should be displayed so the user understands the environment in the container and as it is a value that they can alter it will show the change they have made numerically
* Number of particles – The number of particles will be helpful as the user can keep track on the number of particles when the simulation has many of them all flying around.

**Buttons – Summary of each button**

Options

* Change mode – This button will allow the user to cycle through each mode. Below on the diagram each mode that can be used is listed.
* Reset simulation – From my research I found that resetting the simulation is a usability feature I would apply to my solution as it makes the simulation much easier to interact with.
* Help button – This button will present instructions about using the program and is a necessary usability feature as it’s important that the user understands how to use the application.
* Pause button – The pause button will stop the particle controller from updating so the user can view the exact state of the simulation at any point in time.
* Count collisions – This will display the number of collisions of the particles with the container. The number of particles with the container is relevant as it would be what causes the pressure of a container with gases inside.
* Particles coloured based on their energy – This is an option which can be toggled to colour the particles based on their energy so the user can have a visual representation of the energy distributed across all the particles.
* Switch temperature unit between kelvin and Celsius – Allowing the user to choose their preferred unit for temperature will make my program accessible to users with different levels of understanding in physics as they will still be able to understand what the value of temperature means.

Altering the simulation

* Change temperature – Increase or decrease the temperature of the model. Increasing the temperature allows the user to essentially increase/decrease the energy in the simulation and see the results of that.
* Change volume – Increase or decrease the volume of the model. This will also be seen visually as the container will decrease in size. This will show how a larger container will affect particle collisions and how the expansion of gas decreases pressure.
* Change number of particles – Increase or decrease the number of particles of type A (the lighter particles) or of type B. Increasing the number of particles is essential to showing how amounts of particles affect pressure of a container.

**Success criteria links:**

* Help, reset, pause button, change mode button
* A different interface for each mode
* Options to represent data

### Timeline Description automatically generated Class structure

The diagram on the previous page presents my design for the classes/structs and how they will work together to form the structure of this program. Red boxes represent structs, green represents classes, yellow represents Enums which is just a set of defined constants and purple represents external classes that are not a part of my own code.

#### Polymorphism

The class **olc::PixelGameEngine** is the class that **Simulation** will inherit from to use the engine that will allow me to render to the screen each frame. Listed under **olc::PixelGameEngine** are the main methods inherited from that class that I will be using leaving out less relevant methods that are private or I will not need to use.

Each mode is also a sub class of the Gui class as they will inherit all the methods for managing the interface of the program and will override the Gui class’s constructor and adjust() method as some modes will need to adjust variables to keep another constant. The modes will also have different variables to keep constant and different default states so their properties will differ along with some of their overriden methods.

#### Use of structs

Some of the components in the diagram are structs and not classes as structs are generally simpler and by default public unlike classes so they are a suitable way to store a defined set of related data that can be easily accessed by the class containing the instance of that struct.

#### Use of Enums

Enums are not related to object-oriented programming however I included one in this diagram for clarity on what the type PARTICLE\_TYPE is. As particles of different masses/sizes will be an instance of the same class (sub classes are not necessary as no extra functionality is needed) I will still need a way to categorize them to show different statistics for each particle type (such as comparing their average velocities) so using an Enum will be the most simple and clear way to do this as it does not require comparing strings or using integers that don’t clearly show what type the instance is.

#### Description of each object

Each class will help encapsulate data and manage part of the program. The Particle class will likely have many instances while the program is running while the **Particle\_Controller** or the **Simulation** class will be static as they will help manage the state of the application. It is important that references are used when passing these classes as parameters otherwise other instance will be made and any changes within that function will not have applied to the relevant instance being used. Below are two tables detailing each class and its purpose.

#### Classes

|  |  |
| --- | --- |
| **Class** | **Purpose and brief description** |
| Simulation | This class will be a static class that will contain the main loop that will be iterated through during the runtime of the application. It will also render all the GUI elements and the simulation to the window each frame. |
| Container | A static class containing the dimensions of the container that the particles will be moving around in. It will have methods that can be used to change these dimensions. |
| Particle\_Controller | A static class that operates the simulation. It will contain a vector with all the particles, variables such as temperature, and many methods that will operate on this data. |
| Particle | A class containing data for each particle including mass, energy, position and velocity. It will also contain methods that can be used by the particle controller such as a move method or check for a collision with the container. |
| Gui | The Gui class will be a parent class that the interface for each mode will be based on and will contain all the elements and methods that won’t change between modes. |
| Button | Buttons within the interface will be instances of this class. This class will contain properties of the button as well as a lambda function which will be ran when the button is clicked. |
| Boyles\_Mode | This is a child class of the Gui class and will be the interface for the Boyles law mode.Also It will contain methods that will adjust pressure when volume is adjusted (and vice versa) to obey Boyles law. |
| Charles\_Mode | This is a child class of the Gui class and will manage the interface for the Charles law example. Also, It will keep the pressure constant using methods managing temperature/volume. |
| Pressure\_Mode | This is a child class of the Gui class and will manage the interface for the pressure law mode. It will adjust pressure based on temperature. |
| Brownian\_Mode | This is a child class of the Gui class and will manage the interface for the Brownian-motion mode interface. It will add the large particle representing the smoke molecule to the simulation. |
| Default\_Mode | This is a child class of the Gui class and will be the default interface. It will not need to keep any variables constant as any factor in the simulation can be changed by the user. |

**Structs**

|  |  |
| --- | --- |
| **Struct name** | **Description and purpose** |
| State | The state struct will hold all relevant information regarding the state of the simulation. This struct will be necessary when changing mode or resetting the simulation as variables such as temperature, number of particles and volume will need to be put back to a default value that will be defined in the state struct. |
| Collision | The collision struct will contain references to two particles that have collided, one or more of these structs will be stored in a vector after a function has checked all particles for collisions and passed as a parameter to the method that processes and handles particle collisions. Essentially this struct will be used to store information about collisions each frame. |
| Text\_Display | This will display data to the user. It will have visual properties such as position or colour and properties regarding its content such as the unit, title and the data itself |

**Success criteria links:**

* Multiple modes/examples
* Modular structure of the application S

### Key variables

This section is to give a description for the key variables that will be needed in the program. Not all variables are described in this section as some are less relevant or are just references to another class.

**Frequently used external classes**

As my program will need to use positions for particles and other elements, as well as vectors for the particles velocity I will need a class to represent a vector. The types **olc::vf2d** and **olc::vi2d** defined with the game engine I am using and can be used as type for 2D vectors as they include many useful methods relating to vectors such as performing dot products or normalization. The classes **olc::vi2d** refers to a 2D vector of integers while **olc::vf2d** refers to a 2D vector of floats and these types will be used frequently in the application.

The class from the standard library, **std::vector**, will also be used a lot in the program as it manages a dynamically allocated array that can change in size which is useful to store any data where the amount of elements is not clear.

**Particle**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Identifier** | **Purpose and justification** |
| Mass | Float | mass | The particles mass property which will be helpful for calculations regarding collisions and energy. |
| Kinetic energy | Float | kinetic\_energy | The particles kinetic energy which will affect its velocity and other calculations. It will also be used to colour them based on their energy |
| Radius | Int | radius | The particles radius in pixels for drawing it to the screen and used when detecting collisions between the particles. |
| Velocity | olc::vf2d | velocity | The velocity (speed and direction) of the particle. Will determine the next position of the particle. |
| Simulation Position | olc::vf2d | position | Position of the particle within the simulation space (not screen space). The position will be relative to the simulations space to reduce the need to convert it every time a calculation is done. |

**Particle Controller**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Identifier** | **Purpose and justification** |
| Number of particles | Int | particle\_count | The number of particles that are in the simulation. Keeping track of the number of particles in the simulation will be necessary for any calculations where the amount of particles is relevant (such as the average kinetic energy) |
| Temperature | Float | temperature | The temperature of the container that the particles are in. As this variable changes the particles energy will also change so it is necessary that this variable Is recorded and updated. |
| Change in energy | Float | delta\_energy | This refers to the change in energy for the particles when the temperature increases. This will be used to increase/decrease the particles energy as the temperature increase/decreases |
| Time between frames | Float | time\_between\_frames | This is the time the last frame took to process and is in seconds. This is so that the movement of particles in the simulation is not dependant on frame rate. |
| Particles | std::vector<Particle> | particles | This will store all the instances of the particles in the simulation. A vector is used as it is dynamic so the size of the data structure can change. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Identifier** | **Purpose and justification** |
| Width of container | int | width | The horizontal length of the container variable will be a constant. This is needed for drawing it to the screen. |
| Height of container | int | height | The vertical length of the container. This can be increased or decrease to change area of the container. This value will need to be stored for drawing the container |
| Area of container | int | area | The area of the container. This value will be used as the volume of the container. While this is the area of the container it can act as the volume as this simulation is modelled as 2D |
| Position of container | olc::vi2d | position | Position of the container in the window. Will be used for drawing it to the screen. |

**Container**

**Button**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Identifier** | **Purpose and justification** |
| Screen Position | olc::vi2d | position | The position of the button on the screen for drawing it in the correct place |
| Size | olc::vi2d | size | The horizontal and vertical size of the button for Drawing it’s dimensions to the screen |
| Colour | Olc::Pixel | colour | The colour of the button. This may change based on the type of button or mode so this variable will be used to store it’s colour. |
| Text | std::string | text | text on the button. This is to summarize the purpose of the button in one or two words. |
| Text colour | olc::Pixel | text\_colour | Colour of text on the button. As the colour of the button may change, the colour of the text should as well so that the text can be changed to a colour that can be easily read despite the colour of the button |
| Buttons function | std::function<void>(void) | func | Function that will be called when the button is pressed. This is stored as a variable so each instance can have a different function rather than needing sub classes for each button. |

### Inputs and outputs

|  |  |  |
| --- | --- | --- |
| **Input** | **Process** | **Output** |
| Help button | The mode is checked, and the appropriate text is taken to be rendered | A box will show up with text that gives guidance on using the current mode of the simulation |
| Reset button | The default state is found, and the particle controller sets its variables to the state’s variables changing it to that state | The simulation is reverted to the defined default state that it was previously in when the application started |
| Change mode button | The mode is switched to the next in the list of modes by switching the current mode class to another and resetting the state of the simulation | The mode has been switched and therefore the simulation will behave differently. The user interface will also change in buttons and layout. |
| Simulation buttons | These buttons will alter the model in some way by incrementing or decrementing different variables | The simulation will now have these changes applied and other variables such as pressure may be altered by this change. will be a visual change in the simulation as well |
| Toggle data display buttons | The display is removed from the list of displays that need to be updated and rendered when turned off. It is added to this list if toggled on | Data will stop being updated and displayed If toggled off. If toggled on the data related to that button will be presented. |

#### Validation

All the variables that can be changed by the user will need to have validation to reduce unexpected behaviour and the possibility of the application crashing. Limits will need to be defined for each of these variables. The value of these limits will be set during development as I will have a better idea of the capability of the simulation at that stage.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Need for validation** | **Validation** |
| Any constant variables | Most mode of the simulation will have variables that will need to be kept constant and therefore buttons used to change these constant variables should not respond to the user clicking on them if the current mode has that variable constant. | The variable is constant in the current mode so the button should not call any method when clicked by the user. |
| The variable is not constant in the current mode so it should call its corresponding method when clicked by the user. |
| Temperature | Extremely high temperatures will drastically increase particle velocity and collision detection may fail because of the large distances they go each frame. The particles also may go so fast their paths can’t be tracked by the user, which isn’t ideal | upper limit which will be determined in a later stage, at which the temperature cannot be increased from |
| lower limit of zero (as there is no temperature colder than that) |
| Number of particles | Having an extreme number of particles could drastically decrease performance of the application so it is important to have a reasonable upper limit. | Upper limit will be determined in a later stage after testing with different amounts of particles |
| Lower limit of zero as the number of particles cannot be a negative number |
| Volume | If the user increases the volume to much the container may stretch out of the bounds of the window so this needs to be clamped. The container also shouldn’t be so small that it disappears, so a lower limit is necessary. | Upper limit around the point where the top of the container is getting close to the top of the window |
| Lower limit around 4 times the radius of the large particle to avoid the container being able to fit particles and allow them to move around inside |

### User interface design

Graphical user interface

Description automatically generated

The image above is my plan for the applications presentation. On the left there is the view of the simulation while on the right there is the interface with all the buttons and statistics the interface will consist of. The actual window of the application will differ in aspect ratio however this presents the general layout of the application.

Each of the key features of the interface are split into different figures which form the application and are described below.

#### Explanation/justification of user interface design

**Figure 1** – Figure 1 shows the container and particles within it. This is where the simulation will be rendered (however there will likely be more space horizontally but for the purpose of this layout the buttons of the interface have been made larger than I plan them to be).

Above the container there is blank space on the window. This is intentional as the container will be able to change in size -from the volume button- so there needs to be some space on the window to allow the container to increase in size on the window.

**Figure 2** – Figure 2 contains the general buttons that will not change for different modes. The help button will present some instructions for using the simulator if the user is confused. This button present different information for each mode so there is specific guidance about that mode. Next to the help button is the button to change to a different mode which will switch to a different mode when clicked. The reset button allows the user to reset the simulation to it’s original state so it is a good usability feature if the user wants to revert the simulation back to it’s default values. There is also a title that states the current mode so it is clear to the user what mode the simulation is currently in.

**Figure 3** – Figure 3 contains all the interactive buttons that can change the variables of the simulation. This includes the number of particles (Light particles and heavy particles), the temperature and the volume. There is also a pause button to pause the simulation whenever the user wants to which I found was a useful feature for the user in my research.

Different modes

As described in the user interface section in the ‘Break-down of the system’ part of this stage, each mode will have one or more variable kept constant so some of these buttons should not be used in certain modes. To indicate this the button will be greyed out and will not make any changes if clicked,

A screenshot of a computer

Description automatically generated with medium confidence

Above shows how this part of the interface will look for different modes. Each button that cannot be changed by the user is clearly greyed out to show this.

Button change colour when clicked

Buttons will also change colour for a small amount of time when they are clicked to show that the user interface has registered that the user has clicked the button and shows that the interface is responsive to the users input which makes the program more usable because of this. This will not work for buttons that have been greyed out as they should not respond to the user clicking on them.

Below shows the difference in colour a button will have when clicked. The left button is the original button and, on the right, it is made slightly darker for a small amount of time after being clicked.

Text

Description automatically generated

**Figure 4** – Figure 4 is where the main stats related to the simulation are presented. There are also buttons next to some of them with check marks to show they are enabled. The purpose of these buttons is to toggle the statistics on and off. If the user is not interested in the statistics, they can toggle it off and this could also make the application perform better as less calculations will need to be done each frame.

Enabling statistics and units

A tick is used to represent the option to be toggled on as a tick is a common symbol known by most users. Next to the number related to each statistic a unit is also given to so the user can understand that value rather than having arbitrary numbers with no meaning which would not be helpful to the user.

**Figure 5** – Figure 5 has other options related to stats. The ‘use kelvin’ button allows the user to switch between Celsius and kelvin. This is to allow users of different skill levels in physics to use and understand the application. Most GCSE students will only be familiar with Celsius while A level students will also understand Kelvin as a unit.

The ‘Colour Particles’ button will enable colouring the particles based on their energy relative to the average energy of the particles. This is a feature that I thought would be helpful in my research so this button will enable this feature.

The ‘More Information’ button and ‘Other statistics’ buttons I decided to add to make this application more usable for both people with advanced knowledge of thermal physics and people with limited knowledge in the subject. The ‘More Information’ button will give some insight into what the statistics may mean regarding the program. It may describe what each statistic is showing about the simulation such as the pressure which is showing the force of the particles on the container.

The ‘Other Statistics’ button is aimed more at advanced users who may want to see some of the inner workings of the application by seeing other statistics not shown on the interface. This may include the standard deviation of energy or other results of equations that are used in the simulation. This will give more functionality to the application for advanced users if they are interested in these stats.

Both buttons will work like the help button, showing a box on screen with the relevant information.

#### Stakeholder feedback on user interface design

The purpose of designing the user interface is to make sure that it is ideal for my stakeholders, so I showed the representatives of my stakeholders this design and they were happy with the layout of the design as it seems clear and easy to follow for a range of different end users.

My physics teacher mentioned that it may be better to present the average kinetic energy for each type of particle rather than presenting an average kinetic energy for all the particles as this may help show that the energy between two different types of particles in the same environment, despite having different masses, should be the same (on average) which is an idea many students forget in exams.

After getting this feedback I decided to add this feature to my user interface design. It would also require taking an average of both types of particles separately however this is not much more computationally demanding as it would be to take one average of all the particles.

Graphical user interface

Description automatically generated

The image above shows the change I made to my design based on stakeholder feedback with the average energy now separated. It’s important the simulation is accurate enough to produce results that show the average energy is about the same for the two types of particles, so the user does not have an incorrect understanding on how energy differs based on mass of the particle.

**Success criteria links:**

• Simple interface

• A button to switch between modes, reset button, help button

• Current state of simulation shown in the window

## Algorithms

Following on from the system design, these algorithms will present the program flow of each task that needs to be done. The algorithms for each procedure should be appropriately broken up into smaller subroutines make code more reusable and reduce complexity of the subroutines.

### Main loop

Diagram

Description automatically generatedAfter the program has started and the simulation has been initialized it will enter a loop where the same set of tasks will be carried out, and this will be executed for every frame of the window. These tasks will update the simulation, interface, and check for user input. Below is a high-level flowchart presenting the tasks the main loop will carry out.

The time between frames will need to be calculated for the particle controller to use to make sure speed of the simulation is not affected by frame rate. My method for doing this will be to record the start and end time of every loop and then minus the end time from the start time. Different classes will be handling these different procedures however all the necessary methods will be called from the same function within the Simulation class so the simulation class will need to contain an instance of the classes that will execute tasks within the loop. Here is the pseudocode for the main loop.

**Update simulation pseudocode**

This procedure will be a method in the particle controller class as it has access to all the particles. This will also be the costliest method in the application as it will include all the collision testing as well moving all the particles. Below is the pseudocode for this method. The simulation should also not update if it is paused so an if statement is required to check if the simulation is paused and skip this procedure if it is.

****

**Update user interface**

The user interface will need to update the data being displayed as well as adjust the simulation to keep constant variables constant as other variables are changed.

****

### Collisions

The simulation will need to check for and handle collisions between particles so I will need to have a procedure to solve each of these problems. Since there could be multiple collisions on one particle at once these procedures should be separate rather than handling collisions at the same time as checking for them because both collisions need to be processed to alter that particle correctly.

These methods will be check\_collisions() and handle\_collisions() and will be part of the ParticleController class as it controls all of the particles in the simulation. To transfer all the collision data processed from the check\_collisions() method to the handle\_collisions() method I will use a simple struct ,Collision, that holds a pointer to the two colliding particles which has been described in more detail in the class section.

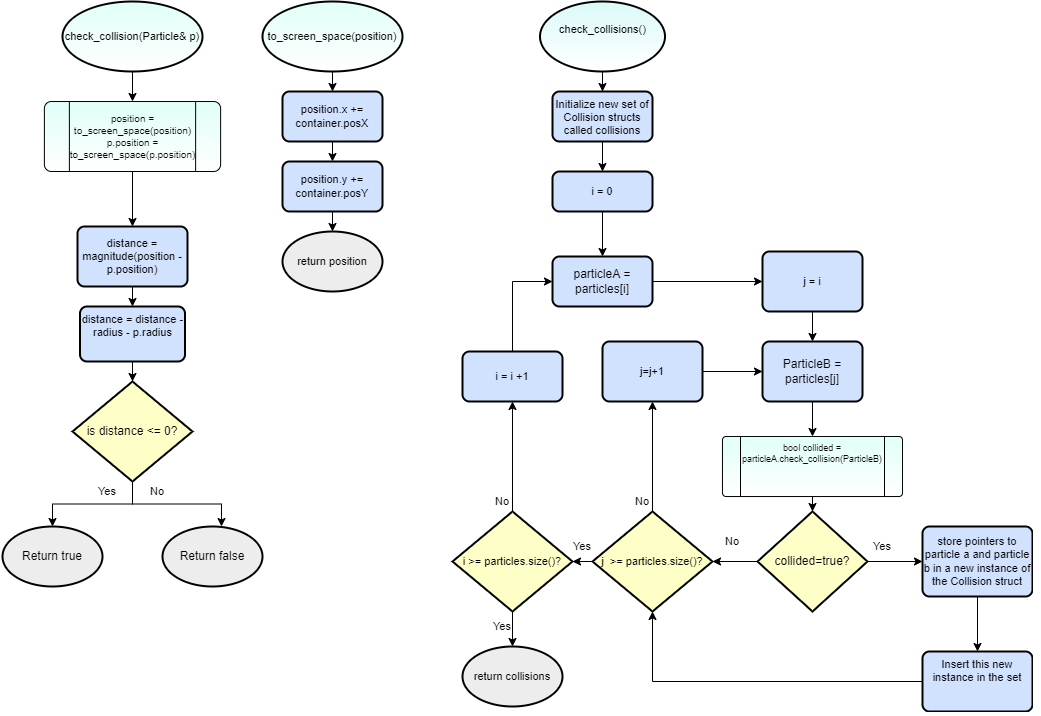
#### Checking for collisions

**Procedure of checking collisions between many particles**

Testing for a collision between two particles is a simple procedure of taking the distance between the two centres of the particles and subtracting the radii of the particles. If the distance is zero or negative the two particles have collided. There may also be a small value subtracted as well acting as the resolution to reduce errors of particles overlapping. To check for a collision between every particle in the simulation I will need to do this with every pair of particles. While this method will have a complexity of O(n²) it will only have a large effect on performance with thousands of particles which is much more than is needed in the simulation.

**Storing collision data**

Each collision will be stored as a collision struct with a reference to the two particles as fields. All the collisions will be stored in a set which is a container in the C++ standard template library that stores unique values in order. This is helpful as a copy of the same collision could occur if a pair of particles are checked twice leaving them to be operated on as if they collided twice. The steps for checking each pair of particles will be as follows

* Negate the position of one point from another and find the magnitude
* Minus the two radii and a resolution value from the distance
* If distance is now below zero, define a new collision struct holding the references to the two particles and add it to the set, otherwise move on to the next pair of particles

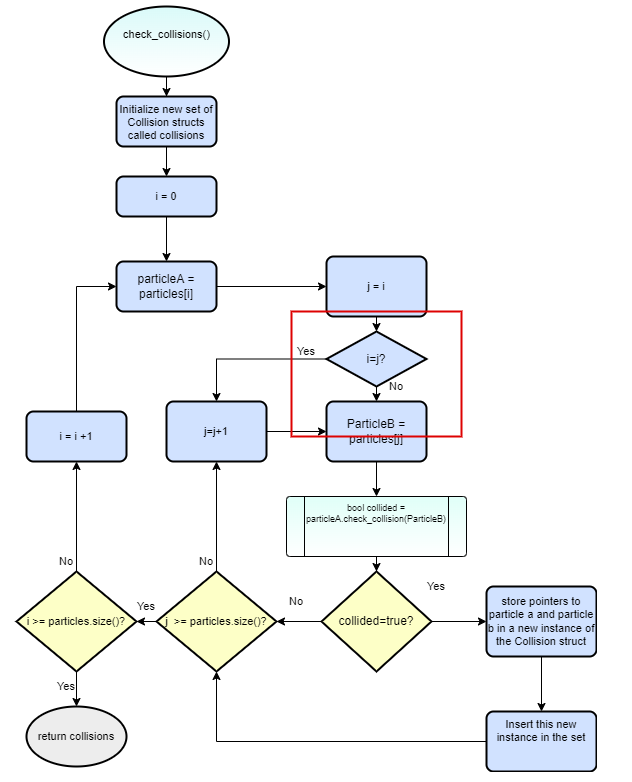
|  |
| --- |
| **Particles** |
| Particle\_1 |
| Particle\_2 |
| Particle\_3 |

As the check\_collisions function involves a lot of sequence and selection I decided to use a trace table to track the variables involved. 3 particles were in the particles vector. Particle\_1 and Particle\_3 in this situation have collided for the sake of this trace table while Particle\_2 has not collided with anything.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **i** | **j** | **particleA** | **particleB** | **collided** | **Collisions.size()** |
| 0 | 0 | Particle\_1 | Particle\_1 | true | 1 |

After the first row of the trace table, I noticed that Particle\_1 would be checked with itself as j is set to the value of I and would detect a collision as they are at the same position because it is the same particle. To stop this, I can set j equal to i+1 instead but if i is equal to the last item in the vector then then an error would be thrown since it j would be larger than the number of elements in the vector.

To solve these issues, I can either set j=i+1, check if j is larger than the length of the vector and if it is skip that iteration however it would be simpler to check if the two particles are the same and subsequently skip that iteration.

This change is shown in the diagram on the right.

The vector of particles is zero based so the size will return a number one larger than the index of the largest element however for this flowchart we can imagine size just returns the index of the last element.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **i** | **j** | **i=j?** | **particleA** | **particleB** | **collided** | **Collisions.size()** |
| 0 | 0 | true | Particle\_1 | Particle\_1 | false | 0 |
| 0 | 1 | false | Particle\_1 | Particle\_2 | false | 0 |
| 0 | 2 | false | Particle\_1 | Particle\_3 | true | 1 |
| 1 | 1 | true | Particle\_2 | Particle\_2 | false | 0 |
| 1 | 2 | false | Particle\_2 | Particle\_3 | false | 0 |

Above is the trace table for the new version of the flowchart and shows that it is working as it should because the collision between Particle\_1 and Particle\_3 is detected, and no other collision is recorded (as there are no other colliding particles).

Below is the pseudocode for the check\_collisions method. SET refers to the type std::set. The second for loop starts at i rather than 0 because when those two particles would have already been tested for a collision.

****

#### Handling collisions

**Limitations of current formula**

One of the assumptions made for modelling gases is that all collisions are be purely elastic meaning no energy is lost when the two particles collide so the total energy of the particles will be the same before and after a collision. In the research section of this project, I found a formula that can be used to find the resulting velocity of the particle after it collides with another assuming that the collision is elastic which is appropriate for a simulation. However, the formula is also making other assumptions regarding the scenario of the collision which do not work for my program.

This formula also assumes that the two particles colliding are of the same mass, which in my program may not be the case as larger particles will also be included. When more massive particles collide with smaller ones with the same velocity they should not have been as largely impacted as two small particles colliding with each-other since the momentum of the smaller particle would be smaller compared to the momentum of the larger despite moving at the same velocity.

**Changing the formula to work in my application**

To account for this, I will need to consider mass in this equation. Energy and momentum still need to be conserved. The change in velocities will need to be scaled in terms of the difference between their mass and the sum of their masses along with velocities. I will need to do this in a way that energy is still conserved.

My method of doing this will be to multiply the change in velocity that has already been calculated by the particles mass over the sum of their masses. Note that the numerator is because the larger the mass the smaller the change in velocity will be so it is the other particles mass that needs to be divided. The result is also multiplied by two so that sum of energy will be the same as before rather than half the original total energy.

In the development phase of this project, I can test this formula by running the function and checking if the energy before is equal to the energy afterwards. As well as their velocities, energy will also change as energy transfers between particles when they collide. The new energy value can be calculated from the formula . Since this function is just a process and does not include any if statements or other types of selection, the function written in pseudocode should be enough to represent this subroutine.

This process can be broken down into a few steps:

* Negate the position of particle B from the position of particle A, store in a variable
* Negate the velocity of particle B from the velocity of particle A, store in a variable
* Plug these values into the formula using the appropriate operations on them
* Add the resulting value to the current velocity
* Calculate the new energy value from the new velocity and particles mass.



The function for calculating the velocity will be a method in the particle class the original velocity of the first particle needs to be stored in a temporary variable as it is needed to calculate the velocity of the second particle but gets overwritten after its new velocity is calculated.

### Changing mode

For the simulation to change to different modes it will need to change its state. My method of doing this is to have a struct, which each mode will have an instance of, that holds variables to change the particle controller. There will be simple function that can load the state in the particle controller



The reset button could also utilize states to reset the simulation back to a default state

### Moving particles

The particle also has a move method which will be called after testing for collisions with the container since velocity will change if a collision has occurred. The seconds between frames will keep the speed of the simulation constant and the line commented out would be used if gravity were implemented however it is unlikely gravity will be implemented in this simulation to reduce unnecessary complexity.



### Diagram Description automatically generatedChecking a collision with the container

Checking for a collision between the container will need to be done for all particles every frame. Energy will not be transferred between the container and the particle so if there was a collision with the container, the particle will keep the same speed (with a change in direction). This would not be the case but to reduce overcomplicating the simulation modelling the container this way would be appropriate.

The particle will require checking all four sides of the container with the particle’s position plus or minus its radius depending which side of the container it is. Since this function only really requires the particles dimensions and container dimensions, it can be a method within the particle class and be checked in the move method before it moves removing the need for the particle controller to check this by looping through every particle. Rather than return a Boolean variable, this method will just make the necessary changes which is to inverse the particle velocity in the direction of the side it hit.



### Changing the temperature of the simulation

Diagram

Description automatically generatedA change in the temperature of the simulation will usually be due to the user pressing a button to change it. The algorithm here shows how this change would be made. As temperature is modelled as directly proportional to the particle’s temperature the kinetic energies of particles will need to be updated to account for this increase in temperature. The change in temperature is also checked with the limits before it is applied so that the values of the variables stay within the valid range.

The function to change the temperature will be a method in the particle controller as it manages the simulation meaning the button will need a reference to the particle controller for this algorithm.

This algorithm will also be applicable for a change in any other variable leaving out updating the particles energies as that should not be necessary for changing other variables such as volume.



### Calculating particle colour

#### Colours used

One of the features of this application is the option to have the particles be different colours based on their energies. The colours used is the first step to this problem. The colour scale used to represent the energies of the particles will be a scale from red to orange, yellow and then white. This colour scale will be useful as objects that go brighter can be conceived as getting hotter so the particles can be perceived as gaining energy as they get a brighter colour.

Rather than using a continuous scale for the colour of the particles, a discrete scale will be used which will reduce the amount of computation and code needed to achieve this as less calculations will need to take place. Each discrete colour will have a corresponding value representing the distance away from the mean kinetic energy and particles will have the colour corresponding to the value it is closest to.

To find these discrete colours I would implement I used an online tool hosted on Github[[6]](#footnote-7) which allowed me to generate a colour scale and tweak it to be appropriate for this application. Below is the colour scale I used. It resembles the scale I described before and has clear variations between the colours which will make the differences in particles energies more obvious.

A picture containing graphical user interface

Description automatically generated

#### Colour scale compared with standard deviations

The scale includes 13 colours so the 7th colour will be used for particles very close to the average kinetic energy. The other colours will be based on the number of standard deviations away from the mean the particles kinetic energy is. For each one colour away from the middle colour could be 0.3 standard deviations away from the average kinetic energy however this may be too large, or too small of an interval between each colour. Different values will need to be experimented with in the development and testing stage in the project to find the value that optimally makes the particles differentiable from each other.

**Calculating standard deviation**

The standard deviation is the average distance away from the mean and will be used against the particles energy to see how get a numerical value of how far it is away from the mean. The standard deviation calculated with the formula below where would be the number of particles, represents the average kinetic energy and would be the sum of every particle kinetic energy squared. This calculation will not be ran for each particle as it is a general value that will be used for to compare the energies of the particles to the mean and therefore will only need to run every update of the simulation.

**Link between colour and Z value**

The Z value is the number of standard deviations away from the mean. The diagram below shows how colours will vary with Z value. The numbers below the scale on this diagram represents the Z value that colour represents however the particles Z value will not be discrete, so each colour will have a range. X is any value in the range and Z is the mid value of the colour in the inequality below. 0.15 is used as it is half of the interval 0.3 used in this model.

.

Rectangle

Description automatically generated with low confidenceThe far end of the colour scale has no maximum limit so every particle will have a colour and the same applies for the nearer end of the scale. The value of 0.3 is used for each interval but as mentioned previously this will likely be tweaked along with the domain of each colour at a later stage.

**Implementation**

A subroutine will be used to find the correct colour for each particle. An array will be used which stores each colour in order according to the scale. The sub routine will calculate the index of the array that the particle will use. The first step will be to find the Z value.

* The particles continuous Z value will need to be calculated first and can be found by finding the difference between the mean and particles kinetic energy, then dividing by the standard deviation.
* Once the Z value is found the max number of standard deviations away on the scale is added to it and then that number is divided by the interval. In this case the max number of standard deviations away will be 1.8 (the particle may have a larger Z value than this, but this value is where the scale stops changing) and the interval is 0.3. This will find where on the scale the particle belongs however this value is unbounded and not an integer yet.
* The value is then clamped between 0 and 12 as there are no colours before/after these indexes in the array and otherwise there would be an index-out-of-range exception thrown.
* The value is rounded to the nearest integer and then used to retrieve the colour from the array that the particle will have.

This process will be a method in the particle controller class as it has access to all the particles to be able to calculate the mean. The pseudocode for this set of tasks is shown below.

****

#### The link between these sub-routines and classes

The class diagram in the Class structure section of this stage shows what class these different methods will be a part of. Most of the methods shown here will be within the particle controller and particle class as these classes will be doing the most computation. The classes in this project are at some level of abstraction, such as the particle class which manages what the particles do but does not control exactly how a particle handles or detects a collision which is controlled by the particle class. This can be seen in these sub routines as other methods are called from a class being operated on. This helps keep the code more organized and separate tasks further. These levels of abstraction will make sure my program is modular and easy to alter/maintain.

## Testing

### Testing method during and after development

Each function should be well tested to make sure it is working properly so the program can be tested and improved upon later in the development process. It is important to test these subroutines with different parameters or data to make sure they can work appropriately in different situations.

#### Development testing

During the development, testing should include different sets of data for the methods to process (such as having a very large number of particles in the simulation) and see if the result is what was expected. This may be visually as the simulation is draw to the screen. For debug purposes some variables can be printed to the console, and I can also make use of breakpoints and other features of debug mode in visual studio.

#### Testing performance of procedures

As the simulation is running in real time, the application’s performance may not be to the standard of stakeholder requirements if it is not coded efficiently. A particularly costly procedure would be handling particle collisions as the complexity of that algorithm is O() so the amount of processing will increase by quadratic order for each particle. Therefore, it is important to test performance at key stages in development by examining the frame rate with different amounts of particles.

As my main stakeholders are students and teachers, this application will need to run well on a school computer meaning I will need to test how the application runs on a school computer as well as my own.

### Post development testing

After development is complete, the application can be destructively tested and tested against the success criteria. This will ensure all the required features are there and that the program can make sure limits are not broken due to user input.

I will also present the solution to stakeholders to make sure their requirements are met and if there were some changes, they would like to be made to the current prototype. I will also ensure with my physics teacher that my simulation accurately represents gas pressure.

### Iterative testing

The development of this project will be done in iterations with a working prototype of the application being produced from each iteration. Each prototype will be tested and reviewed to plan the next iteration, and this will repeat until the program is complete.

### Testing sub routines during development

The data below will be used to check If each function is working correctly for each scenario given. If the function does not output, the expected results then there is likely a bug or issue with it. These methods will need to be tested to make sure the simulation is functioning as expected.

|  |  |
| --- | --- |
| **Sub routine** | **Need for testing** |
| Move particles | Moving the particles needs to work perfectly as this will be one of the most used routines in the simulation. Bugs in this routine could cause the whole simulation to not function correctly |
| Change number of particles | This sub routine will run when a button has been pressed so there is a need for validation. This should always be tested to make sure that the validation is working on all cases |
| Change volume | Validation will also be used for this sub-routine and it will need to make sure the size of the container does not become larger than the window. Each case will need to be tested. |
| Change temperature | This method will need validation to make sure energy levels of particles don’t get so high that it causes the simulation to not function correctly. More about this is described in the inputs/outputs section. Each case will need to be tested. |
| Handling collisions between the container | If a collision is not handled between a particle and the container then the particle will continue to move outside of the model leaving the window so this routine must be tested a lot to make sure this will not happen. |
| Check for a collision between particles | If a collision is not recorded when it should be allowing for two particles to pass straight through each other onscreen, the simulation would not be reliable to show gas pressure as relevant calculations are not being done correctly. Also, if collisions are recorded when particles have collided this would make the particles behave very strangely and would make my model of gas very inaccurate. |
| Handling collisions between particles | There are many different scenarios of two particles colliding and if there is a bug in this sub routine it could create add more or remove energy from the simulation or cause particles to go in the wrong direction. Therefore, it should be thoroughly tested. |
| Loading a state of the simulation | Loading a state will change many of the values of the current variables within the simulation so it must be checked that this works properly so all the values are changed correctly |
| Loading a mode of the simulation | Loading a mode will change the state of the simulation along with the rules applied to it and will need to be tested to make sure modes are being loaded in and implemented without error |
| Checking for a button press | The simulation must respond to every button press otherwise it would make the program much less responsive and therefore less usable so it must be checked that this method works |

**Testing data**

The output of many subroutines in this application will be dependent on the situation the simulation is in rather than a specific value of a variable so as a scenario is used for many of the methods to make sure their results are valid for all possible scenarios.

|  |  |  |
| --- | --- | --- |
| **Function being tested** | **Input/Scenario (if applicable)** | **Expected results** |
| Move particles(time\_between\_frames) | - | Particles will move according to their velocities and the time in seconds for each iteration of the simulation |
| Change number of particles(amount) | The change in particle number will still be within the defined limits of the application | Run the function and add or delete that number of particles |
| The change in particles will be under the defined limits (a negative number of particles) | Stop deleting particles when the number of particles reaches zero to avoid errors |
| The change in particles will be over the defined limits | Only add the number of particles that will reach the upper limit and no higher. |
| Change volume(amount) | (Similar scenarios to the above function) | The volume changes to the new amount or one of the limits if the change would surpass that |
| Change temperature | (Similar scenarios to the above function) | The temperature changes to the new amount or one of the limits if the change would surpass that. Particles should increase in speed |
| Handle a collision between a particle and container | Particle doesn’t collide with container | The Particles velocity stays the same |
| Particle collides with right or left side of container | The x component of the particle’s velocity is multiplied by -1 |
| Particle collides with top or bottom side of container | The y component of the particle’s velocity is multiplied by -1 |
| Check for a collision between two particles | Two particles with a distance between them less than the sum of their radius. | Collision struct is created to record the collision to be handled later |
| Two particles with a distance between them smaller than the sum of their radius. | Nothing changes |
| Handle a collision that has happened between two particles | Similar velocities and are going at close to opposite directions | Direction of velocity will inverse but not change much in magnitude |
| Similar velocities and directions are not near opposite | One particle will likely be much more deflected than the other |
| Different velocities and going in directions near opposite | The faster particle will decrease in speed and the slower particle will increase in speed |
| Different velocities and different directions | There are many different outcomes depending on the differences between the particles |
| Load state(state) | - | The state of the simulation will change to the given struct |
| Load mode(mode) | - | The mode of the application will change to the given mode as well as the state changing to the modes default state |
| Check for a button press | A button has been pressed | The button’s function will be called |
| A button has not been pressed | Nothing changes |

**Variable/Constants data**

Default variables of the simulation will need to be tested to see if they are suitable for the simulation. There are many constants and variables that will be used for the calculations in this application however these constants are based on real life physics and my application is only simulating real-world physics so they may not be suitable.

This is because particles are very small in real life however in my simulation they will be modelled as circles that the user can see. The size of the container I will use in my application would only be a few hundred atoms long. Gas particles at room temperature, on average, go about 500m/s. from this information you can deduce that using real life constants to simulate the behaviour of gas particles will cause them to go way to fast for the user to even see and thus these numbers must be altered and tested.

While it’s important that the laws related to the variables are followed to have an accurate representation of gas pressure, the actual values of these variables matter much less so the data used will be purely on how well it works with the simulation.

The value of these constants will be determined during development when I can test them on the current prototype of the simulation however below shows what will need to be tested.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Why this needs to be tested** | **Initial value** |
| Volume | As the simulation is presented in 2D the area will need to be used as if it was the volume. While the visual area (in pixels) of the volume will be dependent on the layout of the window, the volume to use for calculating relevant statistics (such as pressure) will need to be based of a realistic value (which will change proportionally to how it changes on screen) to produce statistics that are at reasonable values. | would be a reasonable base area to use. This number can then be multiplied by the height in pixels to get the volume and this number can be shown in the volume statistic. |
| Boltzmann constant | This number is a constant which relates the temperature and average kinetic energy of gas. The real constant will likely not be suitable as the scale of particles are much larger in my model meaning the Boltzmann constant would give them an energy much lower or higher than they need to work in the simulation. | 1.3806503 × 10-23  Is the real life Boltzmann constant so I will start by using this value. |
| Z value for average kinetic energy | This number was mentioned in the algorithms section and will determine will depend on how the energy is distributed across particles. This value will determine how far away from the mean a particle needs too be from the average kinetic energy to be a different colour. Ideally this value will not be so high that almost all particles are the same colour but not so low that many of the particles are on the outer bounds of the scale. | 0.3 as it was the constant I used as I was designing the algorithm |

**Testing buttons**

|  |  |  |
| --- | --- | --- |
| **Button being tested** | **Scenario (if applicable)** | **Expected result** |
| Help button | Different modes | A box will show up with some text detailing how to use the program and will vary with different modes |
| Exit button | - | Application closes |
| Change mode | - | The mode is switched to the next one in the loop |
| Reset button | Simulation is already in its default state | Nothing changes |
| Simulation is not in its default state | The simulation is reset to a default state |
| The pause button | Simulation is paused | Resume simulation |
| Simulation is running | Pause the simulation |

### Testing checklist

The table below will be used to make sure the application has been well tested in the development stage and is a list of all things that will need to be tested to make sure the application is functioning correctly.

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 1. | Reset button resets simulation |  |
| 2. | Temperature, particles and volume will not go above or below their defined limits |  |
| 3. | Help buttons renders help text to the screen |  |
| 4. | Particles will not leave the container |  |
| 5. | It will be correctly detected when a button has been clicked |  |
| 6. | Changing mode will switch the mode correctly and will loop around the modes |  |
| 7. | Loading modes will correctly load its state and conditions such as constant variables |  |
| 8. | Particle collisions are correctly detected |  |
| 9. | Particle collisions are correctly handled |  |
| 10. | Energy is conserved when particles of the same mass collide |  |
| 11. | Energy is conserved when particles of different masses collide |  |
| 12. | Performance is maintained when checking for collisions between all particles |  |
| 13. | Changes to the simulation from the interface are applied |  |
| 14. | Constant variables cannot be changed by the user |  |
| 15. | Changing temperature will appropriately change the pressure |  |
| 16. | Rms and average kinetic energy don’t change unless temperature changes |  |
| 17. | Both large and small particles can be in the simulation |  |
| 18. | Text displays update to show the current data |  |
| 19. | Speed of particles movement is not affected by frame rate |  |
| 20. | The pause button will halt the simulation if the simulation is running |  |
| 21. | The pause button will allow the simulation to run if it is paused |  |
| 22. | The help text is different for each mode |  |
| 23. | Button to change temperatures unit will toggle the unit between ‘K’ to ‘°C’ |  |
| 24. | The colour scale of particles based on their kinetic energy can be enabled and disabled |  |
| 25. | If the particle colour scale is toggled on particles are coloured based on their energies |  |
| 26. | When enabled, the colour scale of particles is not affected by temperature |  |
| 27. | The values used for the colour scales means particles have a wide range of colours |  |
| 28. | Each mode works and represents the proportionality between variables properly |  |
| 29. | The interface is clear and simple to use for stakeholders |  |

# Development and testing

## Project organization

To keep my code organized and easy to maintain I will split the larger classes into header and C++ files. Classes that don’t include many methods will be defined in one header file as they will not require as much code. Separating the declarations and definitions of the classes like this will allow for me to easily keep track of all the methods/fields that class currently has and allows me to edit methods in the C++ files without having to recompile the whole project which I would need to do if I edited a header file.

I will have two folders containing all the code. One folder will contain the header files I will need for my project, and another will contain all the source files. Separating header files and source files will also keep the project organized and easier to navigate.

## Agile development

I will split this this part of the project into different stages each with a section of development, testing and evaluation. This way I can have different prototypes for my project and focus on certain features for each stage. This will make development more modular so I can test and evaluate each stage after developing it to make sure it meets my stakeholders’ requirements and is following my initial plan from the analysis stage. The earlier stages will be setting up the basic features of the application while later stages will implement more advanced features on top of the prototype made so far.

## Stage 1 – Setting up the base of the project

### Goals of this stage

To be able to code the required features of the application, I will first need a basic prototype of the system with the basic functionality which can be built upon in later stages.

This first prototype will have particles that can move every frame at some velocity, a button that can add more particles and a container which holds the particles and prevents them from moving leaving it. Coding these fundamental parts of the application will provide a platform for the more complex functionality to be added and tested.

### Stage 1 – Development

#### Simulation class

The simulation class controls the whole application so it will be the first class I develop. As mentioned in the ‘Class structure’ section of the design stage it will inherit from the PixelGameEngine class which is the rendering engine I will be using to render the simulation.

Simulation.h

The code above is the header file for the Simulation class. It will include many of the other header files as it will need to use these classes, that I will define in them later in this stage, since it controls the operation of the whole application. The class inherits the PixelGameEngine class publicly, so all the public members of that class stay public in the Simulation class as some of them will need to be accessed by the main function which creates an instance of this class.

1- Simulation class design

Text

Description automatically generatedIn the code I haven’t yet added the method DrawTextDisplays() method as I will add text displays in the next stage when the simulation is developed enough for there to be statistics that the text displays can show on the interface.

Diagram

Description automatically generatedI also decided to change the identifier from the ‘currentGui’ to ‘currentMode’ as really what this field will hold is the current mode in use however the variable type is the Gui class as all the modes will inherit from the Gui class as decided in the design section.

- Designed relationship between Modes and Gui class

**Simulation.cpp**

While defining each of the methods of the simulation class I would also define methods in the other class whenever they were needed.

The code below shows the constructor for the simulation class which simply consists of setting the title of the window, recording the time, and storing it in a variable and setting the current mode to the default mode.

**Constructor**

In the constructor of **Simulation** a mode called **testMode** is constructed and used for the **currentMode** field. This is a class I made as a placeholder mode for me to test each **Button** and text display before I develop each individual mode which will need to be done once I already have a functioning simulation as they will need to utilize these features of the simulation that have not been developed yet.

**Creating the simulation**

This method is inherited from the **PixelGameEngine** class and must return true. It will run once when the Simulation has been constructed. Right now, it just uses the **Particle\_Controller** to add a single particle to the simulation however in the future I might adjust this to be more particles.

**Updating the application each frame**

****

3 Program loop design

Diagram

Description automatically generatedThis is an overridden method also inherited from the **PixelGameEngine** and will be the program loop that will control the application as a until it is closed. This method differs from the original design with some of the procedures. As the rendering engine already calculates the time between frames and stores it as the variable **fElapsedTime** meaning I will not need record start/end times and calculate the change in time.

The user interface right now is very minimal and therefore doesn’t need updating. Once I have added text displays this method will be necessary, however. Checking If the window is closed is also handled by the **PixleGameEngine** so this will not need to be done in this method.

Checking if buttons are pressed

**** This method simply consists of checking if the left mouse button has been pressed and then ,if it has, looping through each button to check if the position of the cursor was in it’s region on the screen. Checking if the left mouse button has been clicked is done first because if the mouse button was not clicked then it would be pointless to compute which button the cursor is over, if any.

**Drawing methods**

****After writing the code for each of the classes I wrote the drawing methods for them. The methods mostly consist of retrieving the objects dimensions/colour and drawing them appropriately, (such as drawing a circle for the particles or drawing a rectangle for the container). In the design stage I had two separate methods for drawing the particles -one was intended to loop through each particle and the other would be called each iteration to draw it- but this being separated into two methods is not necessary as it does not really reduce complexity and the single method still only has a few lines of code. I also removed the method ‘DrawParticle’ from **Simulation.h** leaving just ‘DrawParticles’ as to not cause any errors about the method not being defined.

#### User interface

**Gui.h**

As I would need to have a test mode to test this first prototype of the application, I coded the first version of the **Gui** class since modes will inherit from the **Gui** class. I added the basic methods and fields to this class for the application to work such as visual aspects (size, position and colour) and the necessary functionality of storing and adding buttons to the interface.

 Regarding position/size of the window macros are used to simplify the code and make changing these values easier. The macros represent constant values and I decided to define these macros in one file called ‘**Utils.h‘**. I decided to define these macros later on when most of the stage has been finished as finding appropriate values would require testing which would require having a working prototype first.

**testMode (testMode.h)**

Below is the code for the **testMode** class. My implementation of this mode system means that there is not a lot of code needed for each mode as many of the properties are inherited already from the **Gui** class.

The class is static as there should not be multiple instances of it since only one instance is needed to manage the interface. In the constructor for this class a **Button** is added to the buttons vector with the inherited **addButton** method. In the constructor the properties for the button are provided, the buttons method is then assigned using a lamda function. The function in this code will add a new (light) particle to the simulation.

Table

Description automatically generated with low confidence



PARTICLE\_TYPE which is used to specify the type of particle added to the simulation is an Enum which I used to simplify categorizing the particle types as I planned in the design stage. This Enum will be defined in the file **Utils.h** like the macros that I used in the **Gui** class because many different classes may need to use this Enum.

4 PARTICLE\_TYPE enum from design stage

**Button**



The code above for the button class will likely not need to change as it is a very simple class. The field **func** is the function that will run when the button has been clicked. This way I can have buttons that do different things without needing sub classes for each button and instead just have different instances of the button class. The field is of type **std::function<void(void)>** as none of the buttons will return or take in a value and will simply run a procedure.

#### Particles and particle controller

**Particle controller class (ParticleController.h)**

The code above is my current implementation of the particle controller class. For now, it is all in one header file however as I write more methods for the class (which will happen as the simulation increases in complexity) it will be separated into a C++ file and a header file to make the class more maintainable.

**Add particles to simulation**

The **add\_particle** method will add a particle to the simulation with a set velocity. Once temperature is implemented the velocity will be based on the current temperature of the container.

**Update Simulation**

The simulation currently will just have particles that move around so that is the only procedure called in the **update** method for each particle. I have also defined a macro **MAX\_PARTICLES** that would be the maximum number of particles the simulation can hold however this will likely change after I test how the application performs with many particles (which will need to be done once the simulation carries out more complex calculations on the particles.

**Particle class (Particle.h)**

**** This is the current implementation of the particle class. The only methods currently are the constructor, **move**, **check\_collisions\_with\_container** and the getters/setters for the class. The fields such as **kinetic\_energy** and **mass** have been set up however there are no methods that currently do anything with them as the purpose is to simply have a moving particle on screen in this first basic prototype. These methods will allow an instance to be made of a particle, the particle to move around the screen and hopefully bounce of the sides of the container.

#### Entry point

**Main.cpp**

Main.cpp is the entry-point of the application and contains the main function. The main function will construct an instance of the Simulation class and start it. Once the class has finished running (which is detected when the window is closed) the main function ends along with the program.

#### Configuring IDE

As I am using Visual Studio 2019 for this project I will need to configure it to work with this project. Since I am not importing any libraries all I really need to configure is the include directory and platform configuration.

5 Solution explorer

A screenshot of a computer

Description automatically generated with medium confidenceGraphical user interface, text, application

Description automatically generated6 Properties widow for visual studio

$(SolutionDir) represents the directory for the project and include is the folder in which the header files are stored. I chose all platforms as there isn’t a specific need for the project to be x64 or x86 however I will run the application as an x64 program as that is my computers architecture.S

The image on the right shows all the source and header files in my project.

### Stage 1 – Testing

Below is an image of the program after compiling and running it. While the current layout does differ from my design, as the simulation becomes more functional, I can add more to the interface and change it’s appearance to fit my origional design.Graphical user interface

Description automatically generated

#### Testing particle movement

Particles in the container move at a steady velocity however the velocity of 20 ,pixels per second, I origionally set meant the particle moves very slow on the screen.

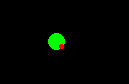
I increased it to 400 which was a more ideal speed for the particle however these values are fairly arbitary as the initial velocity will be calculated from the current temperature rather than a constant value but for testing this stage of the application I will need to test a range of speeds for the particle.

#### Testing particle collisions with container

**Direction marker to aid in debugging**

Taking screenshots of the simulation shows the positions of all the particles but it does not show the direction they are going as that can only be indicated by watching the application run. Because of this I added a line of code that will make debugging the program simpler.

 I added the last line in the method that draws the particle which will draw a small red circle at the edge of the circle in the direction that it is going.

This image on the left shows an example of this. The red circle shows that the particle is moving downwards and to the right. This can be used to make sure that particles are going in the right direction after colliding with the side of the container.

Using the current speed, I’ve set for the particles (400 pixels per second) I tested this method and it showed to be successful for all the sides of the container

An example of the particle successfully bouncing of the right side of the wall is shown by these two images as the particle has not passed outside of the container and its velocity has been flipped (as the particle is going quite fast it goes quite far from the side before I can screenshot it but the change in direction can still be observed.

Shape, rectangle

Description automatically generated A picture containing text, night sky

Description automatically generated

After watching the particle moving around the container and bouncing of the sides I can confidently say the function works for the particle at that speed however in the simulation the speed of particles will vary so I decided to alter the speed of the particle and see the results.

|  |  |  |
| --- | --- | --- |
| **Speeds (pixels per second)** | **Pass or fails to react to a collision** | **Any additional details** |
| 10 | Pass | It took about a minute before the particle actually got to the side |
| 100 | Pass |  |
| 500 | Pass |  |
| 1000 | Pass |  |
| 2000 | Pass |  |
| 10000 | Pass | The particle is going so fast that it seems to jump across the other side of the container each second however it still bounces of the sides |

After testing with a range of speeds of particles I can confidently say that this method should work for all speeds of particles in the simulation. The speed of 10,000 pixels per second would likely not be a speed that particles can go up to once I add limits to how high the temperature can increase to, however having the test still pass with that value means this function works well enough.

#### Testing button responsiveness

Currently the only button on the screen is one that should add particles to the simulation however testing that the buttons responds to being clicked will mean all buttons should respond in the same way as they are instances of the same class.

Rather than testing if particles are added to the simulation, it would be simpler to just print text to the screen when the button is clicked to confirm the button is responding to being clicked.



There are 3 main tests relating to clicking the button that should be passed for my application to work properly and that is that the buttons respond to clicks when the mouse is over it, the button doesn't respond to clicks that are not over it and when the mouse button is held down on the button, the button only responds once rather than many times. This is necessary because otherwise the user could accidentally increase a variable rapidly.

* Button responds to being clicked when mouse hovers over it
* Button does not respond when mouse is clicked and is not over the button
* When the button is held it only registers one click rather than many

After clicking within the area of the button 3 times the console output was:

Text

Description automatically generated

After holding down the mouse button on the button the console output was:



And after clicking multiple times in different areas of the window (other than other the button) there was no output in the console.

|  |  |  |
| --- | --- | --- |
| **Test** | **Clicks registered** | **Pass or fail** |
| Clicking on the button 3 times | 3 | Pass |
| Holding the mouse button on the button | 1 | Pass |
| Clicking in areas on the window other than the button | 0 | Pass |

As all the tests for the button was successful, there is evidence that that this procedure is functioning correctly.

#### Testing adding particles

In this section I will need to test:

* Particles are successfully added every time
* Particles always appear within the bounds of the simulation
* Adding a reasonably large number of particles doesn’t significantly slow the simulation

**Test that particles are successfully added and appear in bounds of the simulation**

**A picture containing graphical user interface

Description automatically generated**First to tests that particle would be added to the simulation successfully, I pressed the ADD button 3 times.

7 After adding particles

8 Before adding particles

**A picture containing text

Description automatically generated**The result here shows that 3 particles were added which means this function is generally working however after trying a few more times a particle was spawned out of the bounds of the container meaning there must be some error in the function that adds a new particle regarding dimensions.

Graphical user interface

Description automatically generated with medium confidence

**Test simulation performance is consistent**

To test that the simulations performance does not drop too significantly after adding a reasonably large number of particles I added about one hundred to the simulation along with recording the initial fps before and after.

Background pattern

Description automatically generated

Shape, rectangle

Description automatically generated

The image above shows the simulation before and after this change. There are particles out of bounds of the simulation, but they should have the same effect on performance.

The fps will of course vary depending on device, operating system and the applications already running on the device however only the change is relevant in this test as I am testing the effect on performance when adding particles.

The results above show a small change in frames per second after adding one hundred particles. The fps does vary slightly so it is challenging to record the exact change when it comes to performance, but this change does seem to be negligible.

As collisions have not yet been implemented between particles this test will need to be done again because collisions will likely have a larger impact on performance when there are many particles.

|  |  |
| --- | --- |
| **Test** | **Pass or fail** |
| Particles are successfully added every time | Pass |
| Particles always appear within the bounds of the simulation | Fail |
| Adding a reasonably large number of particles doesn’t significantly slow the simulation | Pass |

As the second test failed, I will need to find the error in the function which adds a new particle, specifically regarding the position it initially appears when added. Below is the code that generates the random position which the particle starts at.



It finds a random integer and takes the modulus with 100 minus the radius so that the particle does not have a position where any of it will be outside the container.

However, I have missed out the fact that I would also need to do this at the start as right now the particle could have x or y coordinate of 0 which would mean it’s centre is at the top/left side meaning it is outside the container.

I changed the code accordingly to account for this.

 instead of ‘100 - rescale\_length(RADIUS)’ I’ve changed it to be ‘100 - 2 \* rescale\_length((RADIUS)’ as the range of the particles position needs to be the length of the container minus twice the radius to account for both sides.

The radius is also added at the end so, if the radius was 5 for example, instead of the range that each component being 0-90 it would be 5-95 to prevent the particle’s being too close to the edge that it would cross the boundary.

To test if this change in the code worked, I added a very large number of particles again to see if I would get any ones that would appear outside the container however they all were set inside of it.

Graphical user interface

Description automatically generated

Editing this section of the code seemed to have fixed the error so now particles will only initially be inside of the container.

### Stage 1 – Review

The basic implementation of the system for the application in this stage has been developed and tested successfully. As this stage is more about setting up the application rather than developing its core features, there is not much of a need for feedback to be received from stakeholders, so I did not show my stakeholders this current prototype of the application.

**Testing checklist**

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 4. | Particles will not leave the container | ✓ |
| 5. | It will be correctly detected when a button has been clicked | ✓ |
| 19. | Speed of particles movement is not affected by frame rate | ✓ |

**Next steps**

For the next stage of the simulation, I will implement collisions between particles. As this is a large problem that requires a large a lot of computation and testing, I decided to tackle this problem as its own stage. Implementing collision detection and responds between particles into the system will not require the implementation of any other parts of the simulation I have not already developed so I should be able to develop this part of the solution as my next stage.

## Stage 2 – Detect and respond to collisions between particles

### Goals of this stage

Having collisions be handled between particles in the simulation is one of the core features of this application described in the analysis section of this project. In the design stage I broke this problem down into two sections: detecting the collisions and handling the collisions detected.

In this stage I will be coding the algorithm I designed to check for and handle collisions, test these two sub routines and test how this feature affects the performance of the application depending on the amount of particles present.

### Stage 2 – Development

#### Collision struct

When designing this algorithm, I decided to use a struct that would store a pointer to two particles. This would be used to essentially store collisions to later be processed once every particle has been tested with every other particle for collisions. This was to prevent errors in cases that multiple particles collide with one causing erroneous results of handling the collisions.

 In the file ‘**ParticleController.h**’ (the class that **the ParticleController** is defined) I defined the **Collisions** struct using the above code.

I defined it in this file as the **ParticleController** is the only class which would need access to this struct as it will handle all the collisions between particles. I also decided to use the **set** data structure to store these collisions as sets only hold unique values meaning I would not have any duplicate **Collision** structs in the set which would cause errors as the subroutine to handle the collision would be called twice on a particle when it should only be called once.

#### Organizing the ParticleController class

Before writing the subroutine to check for collisions, I will split up the particle controller to a header (.h) file and a C++ (.cpp) to split up the declarations and definitions. This will help keep it organized as I add more methods and fields.

Below is the code for **ParticleController.h** and is very similar to the original file however the definitions have been moved to ParticleController.cpp and I have added some macros at the top to define dimensions of the particles in the simulation.

****

#### Coding the subroutines

The two subroutines I will need to write will be **check\_collisions** and **handle\_collisions**. The diagram below shows how I planned to integrate these methods as members of the **ParticleController** class, and I will be rusing this diagram to keep my development consistent with my design.

Text

Description automatically generatedThe **check\_collisions** does not take any parameters however the **handle\_collisions** method will take in a **set** of **Collision**s which is produced by the **check\_collisions** method.

****

Now that I have added declarations of these methods to the header file I will define them in the file ParticleController.cpp

Check and Handle collision methods

**Checking for collisions**

Diagram

Description automatically generated

10 Flowchart to check for collisions

The above flowchart shows the structure of the subroutines I will need to write. I have already written the to\_screen\_space function so what is left is **check\_collsion(Particle p)** and **check\_collisions**

**check\_collision(Particle p)**

As **check\_collisions** uses the **check\_collision(Particle p)** method I will write **check\_collision(Particle p)** first. This method will be in the Particle class as planned and will check if there is a collision between the particle the method is being run from and the particle passed as a parameter.

 This method mostly follows the same process as the flowchart. The positions are converted to screen coordinates, the distance is calculated between them and checked. I added a collision resolution instead of 0. This is because an error could be caused by inaccuracy’s due to the jump between frames where a particle can move far before it can be checked if it will collide. A small resolution will reduce this error.

**check\_collisions**

**** The function above shows my definition for the **check\_collisions()** method. It follows the same procedure of the flowchart from the design stage and I made sure to take into the account the problem I found when using a tracetable against my original designed flowchart with the line “if (i != j) {“ which makes sure I am not checking the particles against themselves.

Diagram

Description automatically generated

11 Fix I made to my original flowchart

To check that there were no syntax errors in the subroutines I just wrote I attempted to compile the program. Even though I haven’t called these methods yet, compiling the program will still tell me if I have any errors with these sub routines that the IDE can detect.

**Fixing error with Collision struct**

When attempting to compile I got an error message stating:

 And an error from the build output read: After doing some research on this error [[7]](#footnote-8), I found out I need a < operator in my collision struct for it to be contained in a set. This is because sets are ordered data structure so an operator to compare the structs is needed to order them. As the order of the set does not actually matter to me, I will just write a simple operator function that returns true for any case.

 After recompiling the application, it built successfully meaning the error has been fixed.

**Handle collisions subroutine**

This sub routine will handle the collision of two particles based on the direction and magnitude of their velocities, mass, and energy. I have already worked out the equations I will be using in the analysis and design stage so now I will need to develop and test them.

A picture containing text, clock

Description automatically generated  
13 Edit I made in the design stage to take mass into account

12 Equation from Analysis further research to find velocity of colliding

This method will be in the particle class as it will be called for each particle in a collision. The particle it has been collided with is passed into the subroutine along with its velocity as these variables are needed by the equation. The 2 equations above will be used in the method to calculate resulting velocity and kinetic energy



**Handling detected collisions**

Right now, I have a method that checks each particle for collisions and a method in the particle class which will handle a given collision and alter the kinetic energy and velocity of the particle resulting from the collision. However, I need a method which will loop through all the currently detected collisions and handle them one by one. This will be in the Particle Controller class and will be called right after all the collisions have been checked for.

The above code loops through each collision in the set and runs the handle\_collision() method on each of the particles which I developed earlier. I need a temporary variable to store particle a’s velocity as it is needed for particle B’s calculation but gets modified before that however the original value is needed. I have already set this up to be called in the check\_collisions method after all collisions have been detected.

**Implementing these new methods in the main loop**

Collisions will need to be handled every iteration of the main loop as a collision can happen at any point in the running of the simulation. Only check\_collisions needs to be called as it calls the other necessary subroutines which handle the rest of the process.

In the above code I simply called check\_collisions() in the update method which is what is meant to update the simulation every frame.

### Stage 2 – Testing

#### Checking for collisions testing

The first part of testing this stage will be testing the detection of collisions between particles in the simulation because this is the first step **of handling collisions** between particles. To do this I will print to the console every time a collision has been detected. I will not run the handle\_collision() function as I should test that they are being properly detected before testing that collisions are being dealt with as intended.

I will do this by watching the particles move on screen and when I observer two colliding I will check the console for this detection. Having a way to identify each particle to check that the correct particles that have collided are identified will make this process easier to test so I added a few lines of code for the sake of identifying particles individually.

I gave each particle a unique id which is set when it is constructed and is equal to its number in the list of particles:

//For debugging

particles.back().id = particle\_count;

I added a line of code to draw on the number corresponding to the particle’s id next to it on the screen:

DrawString(pos, std::to\_string(p.id), olc::WHITE,3);

In the handle\_collisions() method I commented out the body of the function and added a line that would output the collision:

std::cout << "Collision between particle " << collision.a->id << " and particle " << collision.b->id << " Have collided\n";

The image on the below is a screenshot of the simulation with these modifications. Each particle is numbered and these numbers I can compare with the ones identified in the console.

Graphical user interface, application

Description automatically generated

Now I will run the program and see if collisions that I see are identified by the console. For this test to be successful:

* The console detects collisions
* The console detects collisions regardless of speed or direction of particles

**Testing that collision detection works**

A picture containing text

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidenceThese two images show the console and simulation screen at the same time and you can see a collision between particle 1 and 2 happening and the program recognizes shown by the text in the console. The lines are repeated many times as they pass through each other which means it will continue to detect their collision until they have passed each other so this is expected.

**Testing collision detection works with different speeds and directions**

Testing that collision detection will work for any direction can just be done by doing the previous test many times as each time there is a collision the direction of particles varies. The procedure used to test for a collision also does not compute the direction the particles so there is not a reason for it to only work when particles collide in certain directions.

To test that particle detection works at different speeds I will vary the particle speed, run the application, and check to see if collisions are still being detected. There may be bugs I don’t know about in the procedure when particles are going at lower or more likely higher speeds.

For higher speeds I could not keep track of particles on the screen in real time, so I had to record them along with the console output and slow down the recording to be able to tell if a collision was detected

|  |  |  |
| --- | --- | --- |
| **Speed (pixels per second)** | **Result** | **Additional notes** |
| 10 | Pass |  |
| 50 (default speed I’ve used so far) | Pass |  |
| 500 | Pass |  |
| 1000 | Pass | Not many repeated lines in the console as collisions only last a few frames which is expected |
| 5000 | Fail | Particles move so fast that they jump past each other in most cases even if they are on course for a collision |

These results show that my algorithm is suitable for the program as there is not much I can do for particles going at extremely high speeds other than attempt to limit the speed particles can go. It is unlikely particles will reach a speed of 5000 so this it is not a problem that the test is unsuccessful for this speed.

#### Handling collisions testing

Handling the collisions between particles would mean that particles will ping of each-other after colliding. The kinetic energy would need to be conserved since I am modelling the collision as elastic. First, I will simply test that particles are bouncing off each other as intended after colliding as that problem would have to be tackled before making sure that the total energy of the particles is the same.

I will test this by recording the simulation in action and taking a screenshot before and after two particles have collided in the recording. I will remove the lines that print that a collision the detection and uncomment the line that calls **handle\_collisions().**

Testing results:

Graphical user interface, application

Description automatically generatedA screenshot of a computer

Description automatically generated with low confidence

14 After collision

Before collision

Here particle 2 and particle 8 are about to collide shown by their position and direction. Then they rebound after the collision. The speed of the particles also changed, with particle 8 gaining speed after the collision and particle 2 losing speed. Particle 2 was originally going faster than particle 8 so this is an expected result.

Graphical user interface, application

Description automatically generatedHowever after doing more testing the procedure was clearly not working for all particles. While the collision was always noticed by the console sometimes two particles would seemingly combine into one rather than rebound as they are supposed to.

In this image particle 4 and 2 have collided but instead of bouncing of each other they have stuck together and being detected as a collision.

##### Analysing the error

After reviewing my code for the **handle\_collision()** method I found that the particles are likely colliding this way because after the **handle\_collision()** method is called on them the first time, they are still close enough to count as a collision in the frame after and so it is called again.

This may practically reverse the effect it had, causing them to change directions again and go towards each other despite colliding and this would keep happening each frame, so the particles are practically stuck together.

##### Fixing the error

To fix this I could find a way to check if particles collided in the last frame however that would be quite inefficient and add a lot of complexity to the program. However, It may be easier to check the directions of the two colliding particles after the **handle\_collision()** method has been called. If the two particles are going towards each other, then that means that this error has occurred as they should not be going towards each other after colliding, they should always bounce off each other.

If it appears they are still going towards each other, then their velocity directions are both reversed to avoid this error. This may mean particle movement is slightly less accurate to real-life, but it will not make a large impact on the simulation.

**Checking if particles are coming towards each other**

To check this, I will need to get the angle between the two vectors first. This can be done using the dot product on the normalized velocity of one particle with the vector between their positions. Then if this angle is less than 90° then they are going towards each other.

if (acos(-vec.norm().dot(velocity.norm())) < PI / 2).

The if statement above does all these steps. PI / 2 refers to 90° in radians as the **acos** method returns an angle in radians. **vec** refers to the vector between the two particles and **velocity** refers to the velocity of one of the particles.

**Reversing velocities**

Reversing the velocities will just mean multiplying each component by -1.

set\_velocity({ velocity.x \* -1, velocity.y \* -1 });

This leaves my correction to the error as:

if (acos(-vec.norm().dot(velocity.norm())) < PI / 2) {

set\_velocity({ velocity.x \* -1, velocity.y \* -1 });

}

##### Testing again

I tested this method again using the same method as before and this error did not come up again. All collisions seem to be working as expected. All particles are bouncing of each other one colliding and I cannot get the original error, so it has likely been fixed.

However, I will also need to test if kinetic energy is being conserved to know that this method is functioning as it is supposed to. I can do this by printing the kinetic energy of both particles before and after the **handle\_collisions()** method and checking their sum is the same.

##### Testing that energy is conserved

In the handle\_collisions() method I added lines that will print the energies of the particles and the sum of the particles’ energies before and after the collision is handled

ParticleController.cpp



**Results**

**Text

Description automatically generated** Console output

The image above is a screenshot of the console after two collisions. While the kinetic energy contained by the particles differs after collisions the sum does stay the same showing that the collisions are elastic as intended. However currently I will also need to test this with particles of different mass. In the design stage I made an edit to the equation I was using regarding collisions to work with particles of different masses colliding.

For this I will give particles random masses when adding them for the sake of testing this.

Text

Description automatically generated

18 Console output

The kinetic energy is also kept the same for particles of different masses shown by the console output. Generally, more energy is transferred to the less massive particle after a collision which is accurate to how collisions of particles work.

### Stage 2 – Review

Collisions have been set up correctly and will work with particles of different masses, speeds and directions. Once temperature is introduced in the simulation there will be more variation in the energies of particles. Making sure collisions are elastic will be very helpful later when I need to make sure that temperature, pressure and the mean kinetic energy are consistent, and that energy is not being added to the simulation other than when temperature is changed.

**Testing checklist**

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 8. | Particle collisions are correctly detected | ✓ |
| 9. | Particle collisions are correctly handled | ✓ |
| 10. | Energy is conserved when particles of the same mass collide | ✓ |
| 11. | Energy is conserved when particles of different masses collide | ✓ |

**Next steps**

Now that the fundamental parts of the simulation have been developed, I will introduce temperature, change in volume, the two types of particles that can be used and the interface to interact with the simulation. I will also add the **Text\_Display** class which will be used to display statistics on the interface.

## Stage 3 – Simulation and interface development

### Goals of this stage

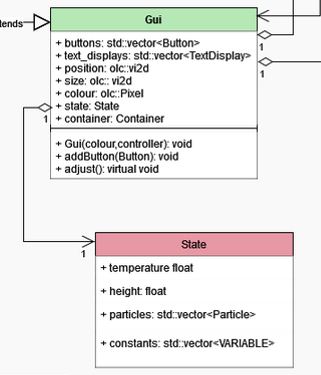
The goals of this stage will be to add the necessary features to the simulation outlined in my requirements. This would include temperature, volume, two different types of particles and ways to interact with these features on the interface. The interface will also need to present the statistics regarding these variables so I will develop the **Text\_Display** struct outlined in the design stage which will be used to present the values of variables such as temperature on the interface.

I will also add the general buttons of the interface including the reset, pause and help button. This will require implementing the **State** struct, specifically for the reset button.

### Stage 3 – Development

First, I will develop the **State** struct as that will be needed for the development of modes and for the reset button to work. The purpose of states will be to store the attributes of different **modes** to be loaded in when they are selected, or in the case of the reset button, to store the default state to set the simulation back to its original state.

#### Coding states into the simulation

My original design shows that a **State** would store the height of the container, temperature, constants in the simulation and a vector of particles. As constant variables are not yet a feature of the simulation it will simply hold the temperature, height, and particles. Instead of a vector of particles It may be more efficient to simply hold the number of particles in the state which can then be added once it is loaded in. This will reduce the space in memory a state will hold especially if there are many particles for the given state.

To start I will define the state struct in the Particle controller class as every other class that uses this struct will include the **Particle\_Controller.h** header file.



19 ParticleController.h – State struct

I will also need to include methods to load the state in which will simply rewrite the variables and add the needed particles. This method will be called from the **Container** class so I will write the method **load\_state**(state) for that class.

Pseudocode for load state method

 The pseudocode above was my original design for this method but this will need to be separated as the container handles the height and the particle controller handles the temperature and particles.

21 Container.h

Once the container class has altered the dimensions of the container according to the attributes of the state parameter passed in, it calls the **load\_state** method on the particle controller which Will take care of the other attributes to be loaded.

ParticleController.h

I will also add a button and default state into the current prototype interface to be able to test this. In the constructor of the **testGui** class which inherits from the **Gui** class I will add this button.

testGui.h - Constructor

I have also defined the state to be used to reset. Once modes are implemented, they will each have their own state defined as member of their class. Later in this stage I will test the button to see if it is implemented correctly.

Instead of using states, I could have a method that simply resets all the variables however this method is much more applicable for the future when modes are implemented because they can each have their own state which gets loaded. Coding new modes into the simulation will be made much easier.

#### Temperature

While I have added the temperature field to the simulation, I will need to implement it fully into the system. When temperature increases, the average kinetic energy of the particles should to. The actual value of the temperature does not matter very much if what matters is the relationships between the variables.

When temperature is changed, I will change the kinetic energy of the particles proportionally with it. I will do this over time as heating often does not happen instantly. Having a loop that adds energy to each particle until it is enough that it matches the current temperature of the simulation will need to be done when the temperature has changed.

**Subroutines**

For organization I separated the Particle class into a header file and a .cpp file so reduce the file size as the number of subroutines in the class get larger.

Particle.cpp

The above sub routine was added to recalculate the velocity after it’s energy has been changed. This will need to be used when energy changes due to a temperature change.

ParticleController.cpp

 This method will be used to update the energies of particles after a temperature change. It will be called multiple times until **delta\_energy**=0 which means energy has changed enough. Delta energy will be a member of the class as multiple methods will need to keep track of it.

ParticleController.cpp - update()

 This if statement has been added to the **update**() method to check if energies need to be updated.

ParticleController.cpp

 This subroutine will be used to change temperature. This will be the subroutine that buttons will use to change the temperature of the simulation. The Boltzmann constant used in the equation has been changed (from 1.38\*10^-23 to 1.38\*10^-1) to fit the needs of the simulation as in real life particles are much smaller however this will need to be larger as the speed the particles are going are pixels per second rather than m/s and it is more important that the change in temperature looks like it has a consistent relationship with the particles energy on-screen than the actual values involved.

Now I can add a button which should alter the temperature when pressed.

testGui.h - Constructor



#### Changing volume of container

As this is a 2D application really area is changing however in the context of gas pressure I will refer to it as volume as that is how I will use it in calculations. To vary the volume the height of the container can be increased of decreased using a button.

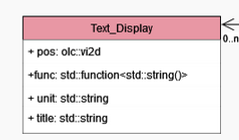
As I have already taken into account that the height of the container may vary while developing the application, all I will need to do now is to add the button to change the volume.

Figure testGui - Constructor

#### Adding stats to the interface

**Text display struct**

To add statistics to the interface I will need to code one of the structs I planned in the design stage. This will be the **TextDisplay** struct and will be updated every single frame to keep information up to date.

On the left shows what the struct will contain. This includes an on-screen position, title, unit and a function. The unit will be the unit of the variable displayed, e.g. Kelvin for temperature and the function will be ran every frame and will be what updates the display and it returns a string which will be the actual data that needs to be displayed along with the title and unit.

Gui.h

Inside the Gui class I also need to add a vector of these structs similar to the vector of buttons that each sub class will contain.

Figure Gui.h - Class declaration



**Adding text displays**

Similar to buttons, text displays will be added to a sub class of the Gui class in the constructor. For this I will need a simple method to add a text display to the end of the vector in the parent class.

Figure Gui.h - Protected method



**Drawing text displays on the interface**

The Simulation class will need to draw each text display onto the screen. Before doing that it will need to run the text displays function to retrieve the data to display.

**Example**

As an example for testing I will make an instance of a text display in the testGui class.

Figure testGui.h - Constructor

The function I am passing as a parameter is rather long because I want to have the temperature set to 2 decimal places to keep the interface looking consistent which requires messing with data types and some string manipulation.

#### ­­Further improvement to the user interface

The user interface right now is very simplistic and lacks a lot of the functionality it should for the application. In this section I will add more buttons, stats and improve on the presentation of the interface.

First, I will add some of the usability features highlighted in the requirement such as a pause and a help button as these will make the program more accessible to users.

Graphical user interface

Description automatically generated

Figure Appearance of the current working prototype

**Pause button**

Adding the functionality to pause should be very simple. All I will need is a Boolean value to indicate whether the simulation is paused which toggles when the pause button is clicked. If the simulation is paused, then it will not update.



34 Pseudocode from design stage where pausing is implemented

In the Particle Controller class I defined the paused variable as a public member and set as false initially because the application will not start out paused.



The **OnUserUpdate()** method in the **Simulation** class will also need to check the simulation is not paused before updating it. The simulation however will still draw everything on screen, it will just not update the simulation.

Simulation.cpp - OnUserUpdate()

Then I can just add a button which will toggle the value of paused so the user can pause the simulation.

testGui.h - Constructor



**Help button**

The help button will present text on screen to give the user an idea of what the program is and how to use it. Different modes will have different text presented by the help button so guidance can be more specific. To do this I will need to add a field to the **Gui** class which each mode will be based on.

Gui.h – Gui class

 The string assigned to the variable is rather generic and gives information about the application as a whole so this will not change for each mode. In the constructor of each mode I can add to this string.

testGui.h - Constructor

I will need a way of presenting this text on screen and a button which will toggle this. A small text box can be made in the corner of the screen to present this information when the button is pressed. A Boolean value which is true when the text should be presented and false when it should not be on screen.

ParticleController.h – ParticleController class

The simulation can check this value and draw the help dialog on screen when it is true. It is set to false as the application will not start out with a help dialog on screen.

Simulation.cpp

The method above will be used every frame and will draw the help dialog if the Boolean value **help\_dialog** is true. As well as the text some rectangles are drawn around it to make it more presentable. Now all I would need is a button which changes the value of **help\_dialog** when clicked.

testGui.h - Constructor



When this button is clicked a help dialog is presented as shown in the screenshot below.

Graphical user interface, application

Description automatically generated

By clicking the button again, it will disappear. Further improvements could be made on the presentation of it however it does server it’s purpose. More guidance may be added so I gave some extra room for longer strings.

#### Adding to the interface

Right now, I have buttons to add particles, increase temperature and increase volume along with the more general buttons such so I should add the option to remove particles, decrease temperature and decrease volume. I also should implement heavy particles into the program as well.

Adding heavy particles will be quite simple as I have already coded my methods so that they would work with particles of different masses and sizes.

Removing particles will require a method in the particle controller class to do this. All I need to do is add the button to the testGui class.

testGui.h



**Removing particles**

Removing particles will require it’s own method in the particle controller to be ran by the button.

ParticleController.cpp

In this method, light particles and heavy particles are distinguished so if there is a certain type to be removed then it will just be a particle of that type which will be removed. The vectors **light\_particles** and **heavy\_particles** hold the indexes of light particles and heavy particles respectively. I added this to easily remove particles of either type without having to search through the particles to find one of the given type.

**Validating removing particles**

There is a chance this method could run where there are no particles of the given type. With the current code this would cause the program to crash so I will make sure that this is not the case.

I added two if statements that will make sure that there is at least one particle of that type before running the code. If there is not then then nothing will happen.

Adding particles is already validated since I have an if statement that checks if the number of particles is below the maximum number which is currently set to 200.

**Decreasing temperature and decreasing volume.**

testGui.h - Constructor

****The buttons above run the same methods to increase temperature/volume but enter a minus value which I have made sure should work in these methods.

I will also need to add some validation to these methods to make sure that they don’t go to high or too low.

**Validating temperature and volume**

ParticleController.cpp - increment\_temperature()

Using std::max and std::min I make sure the temperature is within the boundaries of 0 and 700. 0 is the minimum temperature as 0K is absolute zero. 700 is the largest as the particles go very fast at this temperature so the limit should be around this point.

I use an if statement as well as I only want to make an energy change if the temperature has changed, if it hasn’t changed there should be energy change in the particles.

Container.h

The validation here is to ensure the container does not get so big that it moves out of the window and so small that the particles cannot even fit. These boundaries will ensure that the volume stays within these limits.

**Changing appearance of interface**

For the button class I have added a different constructor where a text colour can be inputted if I want to have buttons with text that is not black however I will keep the original constructor as well so I don’t have to edit the initialization of the current buttons.

Button.h

I have also added a text at the top of the window which will say what mode the simulation is currently in. To do this I have added a field called **name** to the Gui class

std::string name;

Which I draw every frame:

std::string mode\_string = "Current Mode: " + currentMode.name;

DrawStringDecal({ int(((0.57 \* WINDOW\_WIDTH) - 200) / 2 + 0.4 \* WINDOW\_WIDTH), 10 }, mode\_string, olc::BLACK);

Text

Description automatically generatedIn the user interface section of the design stage I also mentioned that I would like to have buttons change colour when clicked.

48 Example I used from design stage

To do this I can add a Boolean field to the button class called clicked. When this is true the colour of the button will be darkened slightly and when the button is clicked this can be set to true. When the mouse button is released, it can be turned off.

Button.h



The number passed as a parameter to **CheckButtonPress()** refers to whether the left mouse button has been pressed or released. If it has been released the method will set the button’s **clicked** field that the mouse is hovering over to false. If the left mouse button has been pressed then the function will check which button has been pressed call its function and set **clicked** to true.

Simulation.cpp - OnUserUpdate()



Simulation.cpp

**Improving CheckButtonPress() method efficiency**

The method currently loops through each button in a for loop to check if the mouse is over it using a for loop. This is not optimal as even if the button has been found it will carry on. Changing it to a while loop would make it more efficient.



Button when not clicked

53 Example of colour change when button is pressed

The left image has lower quality as the amount of time the button stays grey is not long enough to screenshot. I recorded the application to take this image however this resulted in lower quality.

**Current presentation of the interface**

As the interface of the current prototype was very simplistic and not necessary very clear or visually appealing, I have made some edits to the presentation. This mostly consisted of moving buttons around, changing the dimensions of the interface and assigning different colours to the application as well.

**Graphical user interface

Description automatically generated**

The button “Change Mode” on the top right corner currently does not do anything however once I implement different modes the button will have a use. I also hope to add more stats rather than just temperature in a later stage of development.

**Comparism to design of user interface**

Graphical user interface

Description automatically generated

54 Interface design

My initial interface looks fairly like my current design however there are some key differences. Some buttons are shuffled around however this is not very significant. Of course, there are much more options in the lower part of the interface, but this has yet to be developed.

My initial design used buttons with a + or – to indicate whether to add or decrease to the given variable however I think my current prototype is just as clear with the descriptions on the buttons. The statistic for each variable is also shown in the top right however now looking at my original design and my current prototype, I plan to move them to the middle (in between the buttons to add or decrease each variable) as there is space for that on my current prototype’s interface. Including the stat I currently have for the temperature shown at the bottom of the interface.

### Stage 3 – Testing

#### Testing states

To test my implementation of states I can test the Reset button which uses this feature I have added. I will test this using multiple different states as the ‘reset’ state and see if the button behaves as expected.

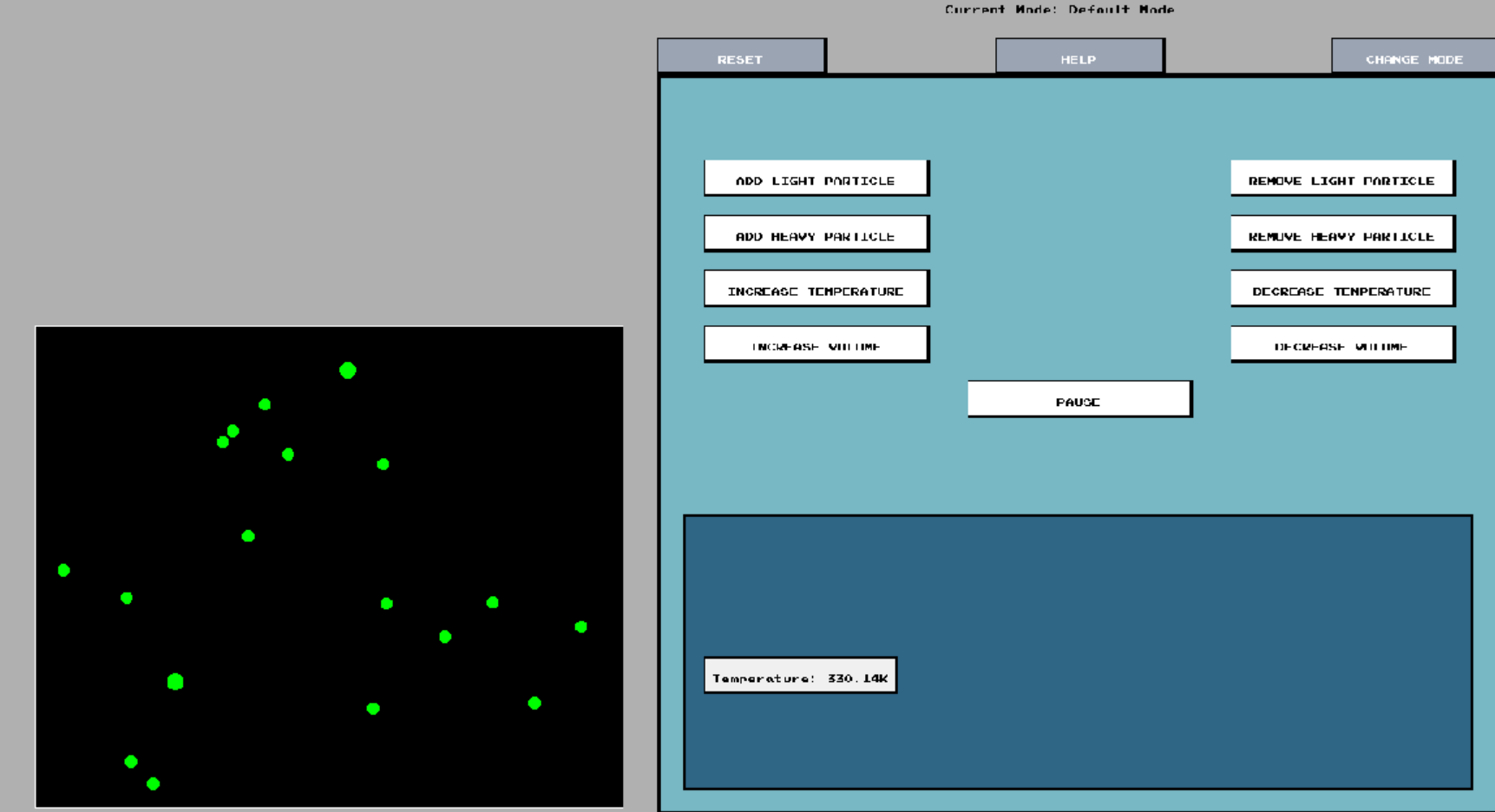
The current reset state is:



The first number refers to the temperature in Kelvin, the second is the height of the container and the third is the number of particles in the simulation. Pressing the reset button should revert the simulation to this state

So the expected outcome when clicking the reset button is the temperature to change to 303.15, the height of the container to become its original height and all the particles in the simulation to be replaced by a single particle.

Before clicking reset button



After clicking reset button

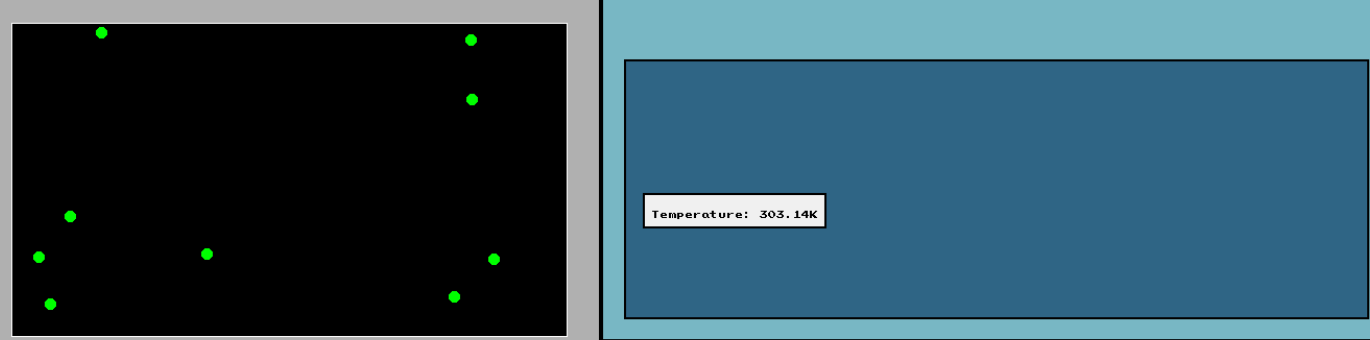


These results are as expected so the reset button likely works however I should also test other reset states to make sure there are not any bugs.

State Reset\_state = { 200.10,0.6 \* WINDOW\_HEIGHT,20 };

Here is a different state to reset to with a smaller temperature, larger height of container and more particles

Before adding resetting



After pressing the reset button

Graphical user interface

Description automatically generated

The temperature and volume had changed to the expected values but the number of particles after resetting is still one rather than 20. There may be an issue with adding multiple particles in a single frame possibly since everything else seems to be working.

**Identifying problem**

Currently the add\_particle method will add a particle in a random position in the simulator.

ParticleController.cpp

The randomness must be a problem with this function as there would be no other reason for it to not work when running it multiple times in the same frame. What must be happening is that the particles are spawned in the same position at once when added in the same frame.

When I wrote this procedure I put the line srand(time(NULL)) at the top to set the rand() function’s seed. However, in the same frame this will produce the same result and therefore all particles will be spawned at the exact same position with the same speed and only one particle is visible.

**Fixing error**

To fix this error all I will need to do is remove the line rand(time(NULL)) and put it in the constructor so the seed is just set once as that is all that is necessary.

ParticleController.cpp



**Testing the error has been fixed**

Before clicking the reset button

A screenshot of a computer

Description automatically generated with medium confidence

After clicking the reset button

Graphical user interface

Description automatically generated

Here the reset button is shown to be working as 20 particles have been added after clicking it. This fix also means in any other situation where I would like to add multiple particles at once it should work as well. Most of the modes I would like to have multiple particles as their default state, so it is good I have found this bug before coding modes into the application.

#### Testing changing the volume

**Testing increasing the volume**

Testing that changing the volume works and there are no unwanted side effects is necessary as being able to change the volume of the container is a key feature of the application in my requirements.

I test both increasing and decreasing the volume by clicking the increase and decrease volume button

Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated

After increasing volume

Before increasing volume

The button worked as expected and the particle still bounces of the walls of the container correctly however when increasing the volume, the particle moved up a few pixels slightly with the container which is not what I expected to happen. This can be seen when observing the change but not in an image as it is instantaneous.

I have coded the particles position to be 0 – 100 in both the x and y axis relative to their position in the container to make calculations simpler. This is likely the cause of this error as the position will be different on screen when the volume increases/decreases. I can add a method to correct particle positions to correct this when the volume is changed.

ParticleController.cpp

This function will run every time volume changes and will correct the particle positions so they will not move with the container when its size is being adjusted. Their position on the screen will stay the same and just their position relative to the container will change. This should not be too in efficient to run as it will only be called when the volume is altered.

**A group of green lights

Description automatically generated with low confidenceA picture containing text, green, electronics, light

Description automatically generatedTesting decreasing the volume**

**66** Before decreasing volume

**67** After decreasing volume

As you can see the test has failed because as the volume is decreased some particles move out of the container. This must be because particles at the top are past the position that the top of the container will be when its height has decreased. This will not work for this application as it is a requirement that all particles are kept in the container

**Fixing the error**

Originally the particles were moving with the container when the volume changes which I edited so their positions would be adjusted so they would stay in the same place on screen. However, in the case of decreasing the volume the particles that would otherwise be pushed out of the container should not have their positions corrected and instead should move downwards with the container.

Now, all particles at a certain distance away from the top of the container will be corrected but the ones by the top that will likely leave the container will not have their positions corrected. Now I can test decreasing the volume again with this change.

A picture containing green, light, colorful

Description automatically generated

A picture containing background pattern

Description automatically generated

69 Before decreasing volume

68 After decreasing volume

The button worked as expected without any errors this time. I tried a lot of times, but no particles left the container, so this change has fixed the error I had. Now volume can be increased and decreased without causing any unwanted effects on the simulation.

#### Testing validation

The table below shows the variables that need to be validated and their limits. These limits I have decided on in development when and are based on my needs for validation for each variable highlighted in the design stage.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Upper limit** | **Lower limit** |
| Particles | 200 | 0 |
| Temperature | 700K | 0K |
| Height of container | 600 pixels | 200 pixels |

This section is to test that all the validation is working, and these variables are not going over the limits I have set. The table below shows the results for each test I did. To do this testing I would click each button on the interface which would either add/decrease each variable to reach the limit.

**Particles**

The results below show that the validation for the particles is working. On the left when trying to remove more particles the application will not do anything and when trying to add more than 200 it will not add anymore.

Shape, rectangle

Description automatically generatedA picture containing text, display

Description automatically generated

70 Maximum particles the button will add

Maximum particles the button will remove

**Temperature**

72 Minimum temperature

 Maximum temperature



After getting to the limits of 0 and 700, clicking the decrease temperature button at 0K or increase button at 700K did not do anything which is as expected.

**Graphical user interface, application

Description automatically generatedVolume**

**74** Maximum volume

**Graphical user interface

Description automatically generated**The volume also works as expected and will not exceed the given values.

**75** Minimum volume

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Can go over upper limit** | **Can go under lower limit?** | **Buttons used** |
| Particles | No | No | Add/Remove particles |
| Temperature | No | No | Increase/decrease temperature |
| Height of container (volume) | No | No | Increase/decrease volume |

#### Testing the pause button

The pause button is quite straight forward, it just stops the simulation updating after being clicked and allows it to resume when clicked again. To test it have just clicked the button a few times to make sure it was doing this. The button was working as it was supposed to. I also reset the simulation and then paused to see if anything changed there but it still worked as expected.

#### Testing performance

As the program has become more developed and complex, more computational power is being used each frame to process and render the simulation. This section is to make sure that the application still performs well as it is important that it runs well even when there are many particles on screen.

When initially running the application, I get a frame rate of about 150 with one particle on screen which is not very high considering this is on a high spec computer and with one particle on the simulation. To see if there are other factors that may be causing this lower than expected frame rate I will test how rendering the interface impacts performance.

**Impact on performance from user interface**

To test If drawing the user interface had a large impact on the performance I ran the program with and without drawing the ui elements.



76 Simulation.cpp - OnUserUpdate()

Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generatedA picture containing graphical user interface

Description automatically generated A picture containing graphical user interface

Description automatically generated

The images above show the program right after being launched and the average fps it seems to stay at. Clearly drawing the interface has a very large impact on the frame rate which is certainly not ideal. The interface is still there, buttons can still be clicked, but it is simply not being drawn.

After drawing different elements of the user interface to see what is causing this massive drop-in frame rate, it seems drawing large boxes on the screen causes this.

**Using a different draw method to optimize rendering**

Since the drawing of each element was handled by the rendering engine I was using, I did some research to find out why this had such a large effect on frame rate. I found that there are two different methods for drawing elements on screen. For example there is **FillRectangle()** and **FillRectangleDecal()**.

The difference is that FillRectangle() is drawn by software while FillRectangleDecal() will send the given data to the GPU and the GPU will store it and draw it each time when the sub routine is called which is much faster than using the CPU. Currently I have been using FillRectangle() so this seemed like it could fix this problem.

I changed most of my drawing methods to this one to test if performance would improve.

Example of using FillRectangleDecal()



**Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated**

**Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated**

This change in my code has made the simulation run much more efficiently as much less processing is needed to be done by the CPU each frame. The frames per second on the right is very similar to before when I didn’t draw the interface at all so clearly the interface doesn’t heavily effect performance anymore.

**Testing performance against number of particles**

As the number of particles increases the more particles to be processed and updated.

The collision algorithm especially will affect frame rate as the number of particles as it has a complexity of O(). While there are other methods of detecting collisions with complexities of o(n) they often require a lot of processing in regards to splitting the container into cells, allocating each particle into cells and comparing them which will likely be less efficient and add more complexity when using < 300 particles and more efficient when dealing with a much larger number of particles as the amount of processing increases with quadratic order and will need far too much computational power to run at a solid frame rate. However, I do not think this application will need over 200 particles to simulate gas particles.

Below is a table of the frame rate at intervals of 40 particles. Here I am using light particles however using heavy particles or both will not affect performance as the same amount of processing is done for each type of particle.

|  |  |
| --- | --- |
| **Number of particles** | **Observed frame rate (2s.f)** |
| 0 | 860 |
| 40 | 800 |
| 80 | 630 |
| 120 | 480 |
| 160 | 360 |
| 200 | 270 |

The table here shows that frame rate does drop relatively fast when the number of particles increases. While 270 frames per second is still a high frame rate to run at it shows a limitation to the number of particles that can be added. On lower spec hardware this may also be lower.

While I am satisfied that the frame rate can still run well with a lot of particles, I may need to check how this program will run on school laptops to see how viable this application is for use in schools.

**Testing performance on a school computer**

When running the application on a school laptop the frame rate fluctuates between 280 and 310 frames per second with just one particle on the screen.

Initial frame rate



I will record the average frame rate against the number of particles while the number of particles increase as I did before to test how performance changes based on the number of particles. If the frame rate remains relatively high above 30 fps, then the application is running to a high enough performance to run on well a school computer. This is important as teachers and students are my main stakeholders for this application.

|  |  |
| --- | --- |
| **Number of particles** | **Observed frame rate (2s.f)** |
| 0 | 300 |
| 40 | 290 |
| 80 | 270 |
| 120 | 260 |
| 160 | 230 |
| 200 | 180 |

The lowest value the frame rate seems to reach is about 150 on a school laptop which is much higher than 30 frames per second, so the application will certainly be able to run on most school computers perfectly fine. Therefore, it will be accessible in that regard and processing power should not stop stakeholders from being able to use the application even with the maximum number of particles.

### Stage 3 – Review

In this stage the simulation has developed much more as a whole with a much more intuitive interface and added features. The interface has been developed more to be more user friendly and closer to my original design. Most of the core parts of the application have been added now so what is left are the more advanced features of my application.

**Testing checklist**

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 1. | Reset button resets simulation | ✓ |
| 2. | Temperature, particles and volume will not go above or below their defined limits | ✓ |
| 3. | Help buttons renders help text to the screen | ✓ |
| 12. | Performance is maintained when checking for collisions between all particles | ✓ |
| 13. | Changes to the simulation from the interface are applied | ✓ |
| 17. | Both large and small particles can be in the simulation | ✓ |
| 18. | Text displays update to show the current data | ✓ |
| 20. | The pause button will halt the simulation if the simulation is running | ✓ |
| 21. | The pause button will allow the simulation to run if it is paused | ✓ |

**Next Steps**

My next step will be to implement Modes into the application. Modes will be sub classes of the Gui class which has already been coded. The **testGui** class has been the mode I’ve been using to run and test the simulation so far. This I will change to be the default mode and the other modes will be Charles law, Boyles law, the pressure law and Brownian motion.

## Stage 4 – Modes

### Goals of this stage

In this stage of development, I will add modes to the application. These modes will have certain variables that need to be kept constant to obey the law that is related to that mode. For example, in the Boyles law mode pressure temperature will be kept constant to show the relationship between pressure and volume.

### Stage 4 – Development

#### Changing modes

A button will be used to change the mode of the program. When clicked it will switch to the next mode in an array of pointers to different modes. Since there will only be 5 modes this shouldn’t be annoying for the user to switch between modes.

The modes are stored and controlled by the simulation class so the method to change modes will be there however the buttons do not have access to the simulation class so for the simulation class to know to change mode a Boolean variable in the container class will be checked by the simulation class and toggled by the ‘change mode’ button. When this variable is true the simulation will change mode and reset this to false.

Simulation.h - Simulation class



I have copied the **testMode** class and changed the name to **BoylesMode** to test changing modes so the two modes stored in the **modes** array will be these two modes for now.

Simulation.cpp - Constructor



In the constructor for the simulation to initialize each mode and store it as a pointer in the array.

Simulation.cpp - OnUserUpdate()



Each frame the Boolean value of **change\_mode** is checked to see if the simulation should change mode.

This procedure will set the **currentMode** to the new mode and load its default state. Default state is a field in the gui class I added of type **State**  and is also what the reset button will reset to when clicked. The use of the modulus is so that the mode index will loop back to zero rather going out of range of the array.All the ‘Change Mode’ button does is change the change\_mode variable in the container to true when clicked which I set up the simulation class to respond to.

The **testMode** class’s default struct contains 1 particle and the default state for the **Boyles-law** mode is contains 10 particles and a larger volume to test if clicking ‘Change Mode’ will work properly.

The results on the next page show that the simulation is changing mode successfully. I tested this now as I will need this to work will developing the different modes.

Graphical user interface

Description automatically generated

Application when launching

Graphical user interface

Description automatically generated

Application after clicking 'Change Mode

The text at the top ,above the reset, “help” and “change mode” buttons, states the current mode and the state of the simulation with a larger volume and more particles shows that the mode has been successfully changed and loaded in as the defined states of each mode are being loaded in.

#### Adding constant variables

Most of the modes will require keeping a variable constant. In my original design of the interface I had the idea of buttons which would usually be clicked to change a variable, be greyed out when it is being kept constant so the user knows that clicking it will not do anything. I will also implement this in this stage of development as well.

A screenshot of a computer

Description automatically generated with medium confidence

Buttons for designed modes

**Constant enum**

An enum for classifying different variables will be useful to identify what variables to make constant and as a variable in the class for each mode. Each button will also have an enum to identify if it should run its function when clicked or not. For example, if a button to increase temperature was clicked in a mode where temperature should be kept constant the enum can be used to identify that the buttons will not run its function when clicked.

Utils.h



Buttons that do not edit particles, temperature or volume will be given the value OTHER and modes that do not have any constant variables will store NONE as the variable they will keep constant.

Simulation.cpp - CheckButtonPress()

In the constructor each button is assigned it’s corresponding enum for example the ‘Add Light Particle’ button is assigned the value CONSTANT::PARTICLE as it changes the number of particles.

**Greying out buttons when keeping variables constant**

To achieve this all need to do is edit the method which draws each button.

Simulation.cpp - DrawButton()

Now the button will also be grey if it **CONSTANT** variable is the same as the modes **CONSTANT** variable.

In the Boyles-law mode the temperature will be kept constant so I set const\_variable in the BoyleMode class to CONSTANT::TEMPERATURE to test this out.

The buttons will also not change the temperature when clicked in this mode.

Graphical user interface

Description automatically generated

#### Adding more statistics

The point of the modes is to show the proportionality between different variables so having statistics on the interface will be needed to show this.

**Average kinetic energy**

As stated in the design stage, there will be separate statistics for the average energy of each particle type. Calculating the average kinetic energy will require getting a total of the kinetic energy and dividing it by the number of particles of that type.

 This method will do this procedure on a given type of particle. It may seem like a lot of computation to do each frame but with a maximum amount of particles being 200, looping through a simple task 200 times will not make a large effect on performance.

Gui.h

 The code above is adding these text displays to the interface.

**Collisions per second**

To be able to calculate the amount of collisions in the last second I will need to keep track of time and keep track of collisions. By collisions I am referring to collisions with the container as these collisions are what affect pressure.

The check\_collision\_with\_container() method will return a Boolean value depending on if there was a collision and this will get passed back to the particle controller where it can increment the collision counter if the result was true.

This method will need to run every frame. The collisions per second will update after every second with the value of the **collision\_counter** which also gets reset every second. This way the collisions per second can be calculated regardless of the speed the simulation is running at.

**Pressure**

Originally, I was planning on displaying a value for pressure on the interface calculated from an equation. The problem with this is that the simulation is not going to produce realistic values at all because it simply does not have realistic conditions for a realistic pressure. (Most containers have billions of particles while my simulation will have <1000 for example).

What really matters is how it changes in proportion to other variables so instead I will show a bar that may increase or decrease in size relative to the collisions per second. This is because the pressure is dependant on collisions per second. The maximum collisions per second is around 500 when I have 200 particles and container at the smallest volume so it will be on a scale from 0 to 500.

Simulation.cpp - DrawGui()

 From the image below you can see the bar which is supposed to represent pressure. I think this will be an easy way to see how pressure changes with volume rather than a number which would represent the pressure.

Graphical user interface

Description automatically generated

The image above shows all the changes I have made to the interface. I have also moved the statistics for the particles, temperature and volume to the top and changed the colour of the text displays slightly to convey that they are not buttons.

#### Boyles law

Boyles law states that the pressure a gas exerts is inversely proportional to the volume occupied by it. The pressure bar at the bottom will be useful at conveying this. Temperature should remain constant as It would affect the pressure. I have already created this class to test the ‘Change Mode’ button so I will just need to edit the class.

As the only variable that can be changed in this mode is volume, no adjustments will need to be made to other variables since temperature and particles will be kept constant.

BoylesMode.h - Constructor

The last parameter in the constructor is the default state which starts with 150 particles and a large volume. A large volume is started with so the user can see how the pressure increases with decreasing volume. 150 particles is enough to see a difference in pressure as volume changes so that is the default setting for Boyles mode.

The colour for the mode is also different to differentiate it from other modes. The help text has changed so the user can have some idea of what this mode is about and how it should be used.

Graphical user interface

Description automatically generated

The image above is how the application looks in with this mode. As the volume decreases pressure can be seen to increase and vice versa. The temperature buttons are also greyed out because temperature is kept constant. The help text is also shown below. The text at the top stays the same for each mode where more detail is given about the specific mode below.

Graphical user interface

Description automatically generated

#### Charles’s law

Charles’s law states that as temperature increases gas expands to keep pressure constant. This I will not be able to truly implement as I cannot keep pressure truly constant because it is calculated from the collisions that take place which I cannot guarantee to be constant.

This is because Charles law only really applies to idea gases and in application (especially with a small sample of particles) there are many limitations to being able to simulate idea gases.

Instead I will show Charles law by increasing the volume as temperature increases to show the core principle of this law and the help text for this mode will explain why pressure is kept constant and could help the user understand the concept of ideal gases better.

**Adjusting volume and temperature**

If the user increased the temperature, then the volume should also increase according to Charles’s law. Also if the user increased the volume then the temperature should increase proportionally. To do this I will need a couple methods in the **ParticleController** to adjust the volume/temperature.

ParticleController.cpp

These methods will calculate the value to adjust temperature/volume and return it. The numbers used in these methods are related to the maximum and minimum values for temperature and volume. In **adjust\_volume** temperature is divided by 700 because it ranges from 0 to 700 and the method will use this to calculate what the height of the container should be.

CharlesMode.h

 The adjustments() method in the CharlesMode class utilizes these functions to adjust either volume in response to a button being pressed. Each time a button is clicked the adjustments() method is run in the case that there needs to be any adjustments.

**Charles law mode**

CharlesMode.h

 In the constructor, other elements of the mode are set up. It will also start with a 150 particles however a smaller volume to show how temperature increases as volume does. The help text describes why this is not a perfect representation of Charles’s to inform the user and describes ideal gases in the process.

Graphical user interface

Description automatically generated  
The number of particles will be kept constant as it is not relevant to Charles law and will affect the pressure even more which is not ideal.

The images above show the application in this mode and the one below shows the help text that comes up when clicking ‘Help’.

Text, letter

Description automatically generated

#### The Pressure law

The pressure law states that temperature is proportional to temperature when volume is kept constant. For this law I will need to keep pressure constant and allow temperature to be changed. As temperature is increased pressure should also increase due to the larger number of collisions and higher energy collisions.

PressureMode.h

The above code is the constructor for the **PressureMode** class. Since the variable being kept constant is the volume, there is no need for an **adjustments()** method. In this mode I have also changed the colour of the interface to make each mode look different in some way however I have made sure that the colour of the interface contrasts with the simulation and buttons so it is clear to understand for all users.

The image below shows the application in the Pressure-Law mode and the help button has been clicked to show the help text on the left.

Graphical user interface

Description automatically generated

#### Brownian-Motion mode

Brownian motion states that particles in fluids move randomly because they are bombarded with other particles in the fluid. This can be demonstrated by having one large particle in a fluid of smaller particles. The smaller particles will cause the large particle to move randomly around in the container.

The number of particles will remain constant as the purpose of this mode is to show the random movement of particles.

I will also need to add another particle type which will be much larger and heavier than the other two types. States will also need to be edited to allow these large particles to be added.

Utils.h



The ‘VERY\_HEAVY’ particle type will be this very large particle which will only be present in the Brownian motion mode. I will also need to define it’s size and mass which I will do using macro to making editing them simpler. This particle will not be as easily moved around as the lighter particles so a larger mass is needed so that velocity is not too high.

Utils.h



With these values the particle will have a radius 6 times larger than the light particle meaning it’s area will be 25 times larger and a mass 20 times larger than the light particles so it will be one very large particle in a sea of smaller particles. The state struct will need to be edited to allow modes to add this new type of particle to the simulation.

ParticleController.h

In the **add\_particle()** method and **load\_state()** method these particles will now be added in the same way the light and heavy particles can be added.

BrownianMode.h

The last argument in the constructor is the state. As you can see it has a high temperature and lower volume so particles will have more collisions with the large particle. The 1 at the end indicates there will be one very large particle. The help text briefly explains Brownian motion and the number of particles will be kept constant.

Graphical user interface

Description automatically generated

Here is the application in this mode. I have made the large particle a different colour to the rest to make it more easily seen by users. Volume and temperature can be edited so the user can see how the motion of the large particle changes as particles are faster or more compact however the main point is that it will move randomly in the container.

### Stage 4 – Testing

The modes that have been developed in this stage have been built on the current procedures and classes that have already been developed and tested and the ‘Change Mode’ button has already been tested so there is not much testing that needs to be done in this stage. I will test that the modes show each law correctly as this is the required criteria of each mode.

#### Boyles-law mode

This mode should show that pressure will increase as volume decreases. To test this, I will simply switch to that mode, decrease the volume and see if the pressure is increasing as intended.

Graphical user interface

Description automatically generated Initial volume

100 Initial pressure

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

101 Pressure after decreasing volume

102 Minimum volume

These screenshots show that this mode is wokrking as intended. Pressure is increasing fairly proporionally with the decrease in volume which is a correct representation with Boyles law.

#### Charles’s-law mode

Charle’s law states that when pressure is constant, temperature and volume are directly proportional. In my application I cannot keep pressure constant as it is calculated by the number of collisions per second which I can’t control as described in the development stage.

When increasing temperature, even with an increase in volume, pressure still increases and there is no way for me to keep it constant if I am calculating it based on the amountt of collisions per second which constantly fluctuates even when the state is not being changed.

However as temperature increases the volume does increase proportionally in this mode so that principle of Charles’s law is being followed.

#### Pressure-law mode

For this mode to work as intended, as temperature increases, pressure should increase as well at a fairly constant rate. Below are the results from testing this.

For this I decided to show a value for the pressure on screen to make a table of values. The actual value of the pressure does not matter and is unlikely to be realistic but the change in pressure after each increase in temperature is what is important.

|  |  |
| --- | --- |
| **Temperature** | **Average pressure** |
| 309 | 81 |
| 390 | 111 |
| 471 | 143 |
| 552 | 166 |
| 633 | 190 |

The temperature is increasing by 81K each time since there is a noticable enough different with those intervals and the pressure seems to increase by about 20-30 each time so it appears to be proportional as it is increasing by approximately the same amount each time.

#### Brownian-motion mode

The purpose of this mode is to show the random motion of particles in a fluid using one large particle in a container of many smaller particles. Therefore the large particle in this mode should move around in a fairly random manner. This can only be tested by observing the how it moves in the simulation.

**Observations**

Background pattern

Description automatically generatedBackground pattern

Description automatically generatedThe larger mass this particle has means it does not move around as easily as the smaller particles and rather gets pushed around in random directions because of all the particles colliding with it. Sometimes it will be pushed to the side of the container but eventually get pushed out towards the centre again. This behavior is as expected and shows the mode is functioning as expected as it represents Brownian motion well.

### Stage 4 – Review

This stage in development has been successful as different modes have been added into the application. Most of the modes work well and meet the criteria I have set however there is still the issue of pressure not being able to be kept constant due to the inability to keep pressure constant.

**Testing checklist**

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 6. | Changing mode will switch the mode correctly and will loop around the modes | ✓ |
| 7. | Loading modes will correctly load its state and conditions such as constant variables | ✓ |
| 14. | Constant variables cannot be changed by the user | ✓ |
| 15. | Changing temperature will appropriately change the pressure | ✓ |
| 16. | Rms and average kinetic energy don’t change unless temperature changes | ✓ |
| 22. | The help text is different for each mode | ✓ |
| 28. | Each mode works and represents the proportionality between variables properly | ✓ |
| 29. | The interface is clear and simple to use for stakeholders | ✓ |

**Stakeholder review**

#### Questions

For stakeholder testing I will demonstrate the software to them on a school computer and ask them the following questions to receive feedback on the project.

1. Do you think this application represents the gas laws accurately?
2. Do you think this would be a useful tool for students to understand the gas laws?
3. Is the interface clear and simple to use? If not, why?
4. Are there any changes you would make to the presentation of the interface?
5. Are there any statistics or buttons you would add to the application?

Question 1,2 and 3 are to get a general idea how well the application meets the initial requirements for the stakeholders and question 4 and 5 are to see if there would be anything they think the simulator is missing.

#### Response

1. **Do you think this application represents the gas laws accurately?**

“*Yes, the gas laws are being presented accurately enough for a simulation as the gas laws seem to be followed accurately by the simulation.*”

1. **Do you think this would be a useful tool for students to understand the gas laws?**

“*Yes, the modes are particularly useful for showing how different laws work.* *A mode that shows the amount law would also be a helpful addition to show students how the number of particles affects pressure so perhaps that could be added.*”

1. **Is the interface clear and simple to use? If not, why?**

“*The interface is sufficiently simple to use, the buttons are clear, and I think the help button is useful for explaining to students what is happening in each mode.*”

1. **Are there any changes you would make to the presentation of the interface?**

“*The interface is fairly simple, but most users will not know the application has different modes just by looking at it and I think it should be made clear that the application can be switched to different modes, so there should be some text encouraging the user to use different modes.*”

1. **Are there any statistics or buttons you would add to the application?**

"*I think a button where I can add many particles at once would be useful and would save a lot of time rather than clicking the ‘add light particle’ button many times. Also, I think a scale, like the scale showing the level of pressure, would be an effective way to show temperature as it is easier than reading the values of temperature.*”

The response from my physics teacher shows the application’s current features are working as intended for my stakeholders but the interface could have some improvements. The suggestions of usability features that could be added to the application along with another mode I could certainly add as additions to the application.

**Next steps**

The application is nearly finished now however there are some more features from my initial requirements to be added. This includes particles having different colours based on their energies. I have left this feature to do by the end of development as the project would already have to have most features in place to program this feature. Most of the next stage will consist of adding the final features to the program. After the last stage I will do post development testing to make sure this application meets the initial requirements.

I will also add the features my stakeholder suggested in the stake holder testing section to make sure the application will fit the requirements of my stakeholders. This will include an additional mode that represents the amount law, buttons that will add/remove many particles at once and a scale to show the temperature on the interface.

## Stage 5 – More Advanced interface features

### Goals of this stage

In this stage I will develop the more advanced features I have planned to add to the interface. This will include colouring particles based on their energy, allowing the user to switch between Celsius and Kelvin as the unit for temperature and other statistics for the interface.

### Stage 5 – Development

#### Colouring particles based on their energy

This feature will be an option that the user can enable. It will give each particle a colour from a scale of colours based on how far away they are from the mean kinetic energy. Particles with energy much lower than the mean will have a lower colour on the scale and particles with energy much higher than the mean will have a higher colour on the scale.

The colours have already been decided on in the design stage along with the algorithm that will be used to calculate each particles colour.

The algorithm below was my planned algorithm from the design stage which I will implement in this development stage.

****

**Calculating standard deviation**

These procedures will be defined in the particle controller class as they will need access to all the particles. First, I will need to calculate the standard deviation of particle kinetic energies so I will need to write a sub routine for that. I will use the equation below to calculate standard deviation.

denotes the sum of each particles kinetic energy squared and represents the mean kinetic energy.

ParticleController.h

****This algorithm uses the equation above to calculate the standard deviation. It uses the variable **average\_ke\_all\_particlesa** as the mean. This variable will be a field in the **ParticleController** class and will be updated each frame as there are multiple times this variable needs to be used and it would be more efficient to calculate once and store as a variable in the class. It is calculated using the same method from before to find the average kinetic energy of particles.

**Finding Z values**

Now this procedure has been made the find\_z\_value function can be developed. The z value refers to the number of standard deviations the particle’s kinetic energy is away from the mean. Each particle will have a z\_value stored as a variable so instead of returning a z value this sub routine could update the z values of each particle.

****This sub-routine follows my designed pseudocode quite closely however it will loop through each particle to set it’s **z\_value**. This will need to happen each frame so that the z values will be up to date.

The code above shows the update method and the **update\_particle\_z\_value** method will be run along with other methods to keep variables up to date such as **get\_average\_kinetic\_energy().**

**Calculating colour**

Now the z values can be calculated, all that needs to be done is to code the method that will find the correct colour for each particle. This will work by having an array of colours in order of the colour scale and the method will calculate the correct index using the particle’s **z\_value**

****This method differs in some ways to the pseudocode. The max value will refer to the magnitude of the largest z value on the scale. Anything with a larger z value will just have the highest index on the scale. This is added on the first line of the function to change the scale of z values from -MAX\_Z\_VALUE/2 < z < Z\_MAX\_VALUE/2 to 0 < z < MAX\_Z\_VALUE. This is because the indexes will be in range 0 <= index <= 12.

Once the index is calculated a colour is returned. The array of colours will therefore be stored in the particle controller class. Now I will need to define this array of colours

In my original design of this algorithm, I set the MAX\_Z\_VALUE to be 1.8 so that will be used for now however this will need to be tested to find a suitable value that will best show the scale of different energies the particles have.

#define MAX\_Z\_VALUE 1.8f

**Defining colours array**

In the design stage when choosing the colours to use for this feature I copied them to a text file as hexadecimal colour codes.

A picture containing table

Description automatically generated

In my application colours are defined using integer values ranging from 0 to 255 so I will convert these colours to denary and into an array of colours in same order they are in now as they were copied in order of the scale.

In the **ParticleController** I will first need to declare this array. There are 13 colours in total so the array will need to be that size.

olc::Pixel colours[13];

Then in the constructor I will set each element of the array to the corresponding colour.



**Adding this feature into the simulation**

Now all the subroutines have been developed, the simulation will just need to call the correct method to find the particles colour and a button needs to be added to toggle this feature.

Gui.h - constructor

I have added a Boolean variable called **colour\_particles to** the **ParticleController** class to indicate if particles should be coloured based on their energy or just kept as a constant colour. This button will toggle this value when clicked.

Simulation.cpp

This method now will check if the particles should be coloured based on their energy and if the variable **colour\_particles** is true, the **get\_particle\_colour()** method will be called. Particles of type **VERY\_HEAVY** (the ones used in the Brownian-motion mode) will stay the same colour to stand out more from the other particles as their energy is not relevant.

To improve performance I will also make sure that calculations regarding finding each particle’s z value and calculating the standard deviation is only done when **colour\_particles** is enabled as they only need to be computed when this condition is met.

ParticleController.cpp - update()



#### Calculating rms

The root mean square speed (rms) is a statistic that will be shown on the interface as it shows the average speed of particles. To calculate the root mean square speed I will have to loop through each particle, square the magnitude of the velocity and add it to a total which will be divided by the total number of particles and then square rooted.

ParticleController.cpp



If the number of particles is 0 then infinity would be returned so there is an if statement in place to make sure the returned value would be 0 in that case. Now the text display for rms can be added to the interface.

Gui.h - add\_text\_displays

****The function set in the text display will make sure that the rms will be shown to 2 decimal places. The rms will not be accurate to real life like most of the statistics in the application however the way it changes depending on other variables will show users the relationship between the rms and other variables such as temperature.

#### Switching between Kelvin and Centigrade for the temperature unit

This feature is to make sure the application is accessible to users with different levels of understanding in this topic as many people are not familiar with the Kelvin unit. To convert a temperature in Kelvin to Celsius I will just need to subtract 273.15 from the temperature.

I will need the temperature display’s update method to be edited so it will check if the temperature should be converted to Celsius or not.

Gui,h - add\_text\_displays

The unit will also have to change however this will need to be done when drawing the text display as it will be simpler to do than change the unit in the lambda function of this text display.

Simulation.cpp - DrawTextDisplays()

Here the unit is being stored as a local variable and changed if the current text display has title “Temperature” and the Boolean variable **use\_kelvin** is true. The button to change the unit will just need to toggle this Boolean value the same way the pause or change mode buttons work.

Gui.h



#### Additional features suggested by stakeholders

After showing a prototype of the application to my stakeholder representative in stage 4, there were some features which he suggested which would improve usability of the application which I will add in this stage. These features include:

* Another mode to represent the amount law
* Buttons to add/remove multiple particles at once
* A scale to represent temperature like how pressure is represented

**Amount law mode**

Why it would be a good addition

The amount law is about the relationship between pressure and the number of particles present in the container. This would be a good addition to the application because understanding that pressure is dependent on the number of particles is important in thermal physics and should not require much code to implement as modes have already been integrated into the application.

Development

For this mode temperature and volume will need to be kept constant while particles can be added or removed because volume and temperature are not relevant to the amount law. My current implementation of modes only allows modes to have one constant variable so I will need to change how constant variables are implemented to account for the amount law.

Instead of having a field in the class which holds the variable to be kept constant I will have a method overridden from the base class **Gui** which takes in a type of **CONSTANT** and returns a Boolean value that represents if the variable should be kept constant or not.

Gui.h



By default, it will return false in the **Gui** class so by default no variable will be kept constant if this method is not overridden. The field **const\_variable** has also been removed as it is not needed anymore.

BoylesMode.h



The code above is the overridden method in the Boyles-Law mode as temperature is kept constant in Boyles law. For each mode I have done the same thing and return true if the passed in variable is equal to the variable that needs to be kept constant.

The code in **CheckButtonPress()** will also need to change as it used the **const\_variable** field to check if a variable is being kept constant.

Simulation.cpp

If the **check\_constant** method returns false, then the buttons method can be run as it doesn’t need to be kept constant.

Now that modes can have multiple constants, I can add the Amount-Law mode.

AmountMode.h

The initial state of the mode has 50 particles which the user can increase or decrease and see how this affects the pressure in the container. The help text explains how pressure will change according to the number of particles and links it to the equation as that is a relevant equation in A level thermal physics and linking an equation to the amount law gives more context to the equation. Therefore, making them easier to understand and remember.

The application in the "Amount-Law" mode

Graphical user interface

Description automatically generated

The image above shows the application in this mode at its default state. The mode has a light blue colour for the interface to contrast with the other modes and other colours on screen and the temperature is kept constant along with volume.

**Adding/removing multiple particles in the simulation**

Why it would be a good addition

This feature was mentioned by my stakeholder as adding particles individually would take up a lot of time especially when trying to get to the maximum amount of particles. To solve this, I will add buttons that will add/remove 10 particles from the simulation.

These buttons will only add/remove light particles as they are the main type of particles that will be used in the simulation, the heavy particles are mostly to show how energy and speed differ with particles mass so having buttons that add/remove a large amount of them isn’t necessary.

Development

These buttons will have very simple functions as they will just need to run **add\_particles()** or **remove\_particle()** in a for loop.

There should be no issues with validation here because validation is handled within both the **add\_particle()** and **remove\_particle()** methods**.** For example if the number of particles was at 195 and the user clicked the ‘+ 10 particles” button then only 5 would be added as the **add\_particle()** method checks each time before adding a particle that the maximum number of particles ,200, will not be exceeded.

Gui.h - add\_variable\_buttons()

As I may want to change the number of particles these buttons will add/remove in the future, I use the value n to represent the number of particles to add or remove so only that value would need to change if I wanted to change the amount to 5 for example.

Interface after adding the buttons

Graphical user interface

Description automatically generated

**Adding a scale to represent temperature**

Why it would be a good addition

A scale would be a very simple and visual way to present temperature as it does not require interpreting values to get an understanding of how high the temperature is and instead can just be visually shown. Volume is already visually shown by the size of the container and the amount of particles is visually shown by how many particles you can see in the container so a scale to represent temperature in a visual way would makes sense.

Development

As the temperature will be between 0 and 700 this will be the limits of the scale as well. A temperature of 0 would be at the start of the scale and a temperature of 700 would be at the end.

Simulation.cpp - DrawGui()

The **temperature\_length** will be where on the scale the temperature is and it is multiplied by 250 as the scale will be 250 pixels long on the screen. It will also be red as red is a colour associated with temperature. Because of this I changed the colour of the pressure to blue so there is some contrast between the two scales.

Interface statistics section

Diagram

Description automatically generated

Application in default mode

Graphical user interface

Description automatically generatedWith these added features to the application, the simulation now more closely resembles by original design with the extra features I have added as a response to stakeholder feedback.

### Stage 5 – Testing

#### Testing particle colouring based on energy

I will need to test if this algorithm works and test for an appropriate value for MAX\_Z\_VALUE so that particles are represented as a range of colours.

**Initial testing**

By initially running the application and enabling the feature by pressing ‘Toggle particle colour scale’ I would expect the first particle to appear yellow/orange as it would have a z value of 0 giving it the middle colour of the scale. Below shows a screenshot of the result.

**A yellow moon in a black sky

Description automatically generated with medium confidence**The colour of the particle is the same as the colour at the centre of the scale so this part of the algorithm is working

122 Central colour on scale

Now I should test how the simulation would look like with many particles. Most particles should be bright red to pale yellow with a few being dark red and white which are on the further ends of the scale.

**Testing Z values for the colour scale**

Background pattern

Description automatically generated****This screenshot shows many particles that are white and dark red but not very many particles in the centre of the scale. This means that most particles are on either side of the scale show the **MAX\_Z\_VALUE** needs to be made larger.

**123** Colour scale used

I will set the value to 3 rather than 1.8 and see how this effects the particles.

#define MAX\_Z\_VALUE 3.0f

108 Many particles, MAX\_Z\_VALUE = 1.8

**A picture containing light, close

Description automatically generated**The particles in this screenshot have colours much more in between the ends of the scale which makes the variation in energy difference much easier to see.

3 seems to be a good value to set for the maximum z value so I will keep the macro **MAX\_Z\_VALUE** set as 3.

124 Many particles, MAX\_Z\_VALUE = 3

**Testing consistency of scale**

As standard deviation is used in the algorithm to calculate each particle’s z value, the colour scale of particles should remain consistent meaning particles will not all become one colour at very high or low temperatures as the z value is relative to mean kinetic energy.

To test this, I will need to observe the feature at different temperatures as average kinetic energy varies.

 Background pattern

Description automatically generated

Background pattern

Description automatically generated

Background pattern

Description automatically generated

The screenshots above show the simulation at 1K, 309K and the maximum temperature 700K. While the average kinetic energy varies the colour scale is shown to be consistent across temperatures, so this algorithm is working as intended.

#### Testing changing the temperature unit

To make sure this button works for different temperatures I will test using this button at 30K, 309K and 628K. the table below shows the expected corresponding values in Celsius. In this test I will also convert it back to Kelvin to make sure both converting to Celsius and converting to Kelvin works.

|  |  |  |  |
| --- | --- | --- | --- |
| **Expected temperature (K)** | **Expected temperature (°C)** | **Successful K to °C?** | **Successful °C to K?** |
| 30 | -243.15 |  |  |
| 309 | 35.85 |  |  |
| 678 | 404.85 |  |  |

Results using 309K

125 Value converted back to kelvin



126 Initial value

127 Value converted to Celsius

Results using 30K

128 Value converted back to Kelvin

129 Value converted to Celsius

130 Initial value



Results using 678K

131 Value converted back to Kelvin

132Value converted to Celsius

133 Initial value

These results are all the same as the expected values so there is no errors in the temperature conversion

|  |  |  |  |
| --- | --- | --- | --- |
| **Expected temperature (K)** | **Expected temperature (°C)** | **Successful K to °C?** | **Successful °C to K?** |
| 30 | -243.15 | ✓ | ✓ |
| 309 | 35.85 | ✓ | ✓ |
| 678 | 404.85 | ✓ | ✓ |

#### Testing the Amount-Law mode

The purpose of the amount law is to show the proportionality between the pressure and number of particles, To do this I will make a table and increment the number of particles by 20 each time to have enough readings to make a conclusion. The value for pressure is just where on the scale of 0 to 500 it is and doesn’t correspond to a real life value.

The increase in pressure should be a similar amount each time if the two variables are proportional.

|  |  |  |
| --- | --- | --- |
| **Number of particles** | **Pressure** | **Increase from last reading** |
| 0 | 0 | - |
| 20 | 15 | 15 |
| 40 | 34 | 19 |
| 60 | 56 | 22 |
| 80 | 73 | 23 |
| 100 | 96 | 23 |
| 120 | 120 | 24 |
| 140 | 147 | 27 |
| 160 | 167 | 20 |
| 180 | 191 | 24 |
| 200 | 215 | 24 |

The pressure did fluctuate by about 10 – 20 and the recorded values are in the middle of the values it would fluctuate between. The increase seems to be about 20 each time and is staying constant for the most part so this simulation is presenting the amount law accurately.

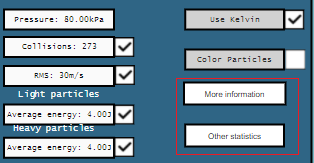
### Stage 5 – Review

In this stage I added the feature for colouring the particles based on their energy and other features such as the ability to toggle the temperature unit and the rms stat on the interface. Other usability features have also been added which were not part of the original requirements to make sure that the requirements of stakeholders are met. They have all been implemented successfully however some buttons I had planned to add from the design stage I have not developed.

**Testing checklist**

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 23. | Button to change temperatures unit will toggle the unit between ‘K’ to ‘°C’ | ✓ |
| 24. | The colour scale of particles based on their kinetic energy can be enabled and disabled | ✓ |
| 25. | If the particle colour scale is toggled on particles are coloured based on their energies | ✓ |
| 26. | When enabled, the colour scale of particles is not affected by temperature | ✓ |
| 27. | The values used for the colour scales means particles have a wide range of colours | ✓ |

User interface design- statistics section



This would include the ‘More information’ and the ‘Other statistics’ button. These buttons I did not decide to include because they did not seem like a useful addition to the application. The ‘More information’ button’s purpose was to describe what each statistic means/represents however most of the statistics are self-explanatory and the help button already goes into detail about how the variables change for each mode.

The ’Other statistics’ button would show other statistics relating to the simulation such as the standard deviation of energy for example. However, these statistics will not be relevant to most users because they don’t particularly relate to gas pressure, or the gas laws and all the relevant statistics are already shown on the interface.

### Testing against checklist from the design stage

While developing this application I ticked of each action on the list to test to make sure that I was thoroughly testing the software while developing it. Some of these actions such as No. 13 are not explicitly tested as it is shown to be successful from testing other actions. Since each button is working and changing the simulation as they are supposed to, the tests of those buttons being successful means No. 13 is also successful.

|  |  |  |
| --- | --- | --- |
| **No.** | **Action to test** | **Successful?** |
| 1. | Reset button resets simulation | ✓ |
| 2. | Temperature, particles, and volume will not go above or below their defined limits | ✓ |
| 3. | Help buttons renders help text to the screen | ✓ |
| 4. | Particles will not leave the container | ✓ |
| 5. | It will be correctly detected when a button has been clicked | ✓ |
| 6. | Changing mode will switch the mode correctly and will loop around the modes | ✓ |
| 7. | Loading modes will correctly load its state and conditions such as constant variables | ✓ |
| 8. | Particle collisions are correctly detected | ✓ |
| 9. | Particle collisions are correctly handled | ✓ |
| 10. | Energy is conserved when particles of the same mass collide | ✓ |
| 11. | Energy is conserved when particles of different masses collide | ✓ |
| 12. | Performance is maintained when checking for collisions between all particles | ✓ |
| 13. | Changes to the simulation from the interface are applied | ✓ |
| 14. | Constant variables cannot be changed by the user | ✓ |
| 15. | Changing temperature will proportionally change the pressure | ✓ |
| 16. | RMS and average kinetic energy don’t change unless temperature changes | ✓ |
| 17. | Both large and small particles can be added in the simulation | ✓ |
| 18. | Text displays update to show the current data | ✓ |
| 19. | Speed of particles movement is not affected by frame rate | ✓ |
| 20. | The pause button will halt the simulation if the simulation is running | ✓ |
| 21. | The pause button will allow the simulation to run if it is paused | ✓ |
| 22. | The help text is different for each mode | ✓ |
| 23. | Button to change temperatures unit will toggle the unit between ‘K’ to ‘°C’ | ✓ |
| 24. | The colour scale of particles based on their kinetic energy can be enabled and disabled | ✓ |
| 25. | If the particle colour scale is toggled on particles are coloured based on their energies | ✓ |
| 26. | When enabled, the colour scale of particles is not affected by temperature | ✓ |
| 27. | The values used for the colour scales means particles have a wide range of colours | ✓ |
| 28. | Each mode works and represents the proportionality between variables properly | ✓ |
| 29. | The interface is clear and simple to use for stakeholders | ✓ |

# Project Evaluation

## Testing usability of the application

Post development testing

## Testing against success criteria

#### Success criteria number 1 – 3

|  |  |  |  |
| --- | --- | --- | --- |
| No. | **Criteria** | **Explanation/Justification** | **How to evidence** |
| 1 | Current state of simulation shown in the window | The state of the simulation needs to be updated and rendered in real time for the user, so it is clear what is going on. | Screenshot of window showing the state of the simulation |
| 2 | A clear interface to interact with the simulation | An interface is needed for the user to interact with the simulation and one that does not require any knowledge for someone to use is ideal for this piece of software to be accessible to anyone. | Screenshot of window with a clear interface, large text and large buttons with distinct colours to make them easy to read |
| 3 | Different variables that control the simulation | For this to be a useful simulation it’s state will need to change based on a handful of variables. This is necessary for the simulation to be interactive to the user. | Buttons being used to change variables shown in a screen along with screenshots of variables being changed in the simulation. |

**Graphical user interface

Description automatically generated**

1. **Current state of the simulation is shown in the window**

The first piece of criteria is to make sure that the simulation is being rendered on-screen for the user to see using the data currently stored about the particles and is updated each frame. The diagram, along with other screenshots of the application, shows this requirement has been met.

1. **A clear user interface to interact with the simulation**

An interface is needed so that the user can interact with the simulation, and it needs to be clear and easy to read to make it as accessible as possible. All the buttons are large and easy to read (the application is scaled down in the image above) and I have used colours that can easily be distinguished from each other to make sure that it is clear when something is a button or statistic.

Each mode also has a different colour to make it easier to distinguish between each mode however this was a decision made during development.

1. **Different variables that control the simulation**

The variables in the simulation are the number of particles (light and heavy particles), temperature and volume. All these variables can be increased or decreased within there defined limits unless the current mode has that variable set to be constant. Below is the evidence that each variable can be changed.

The main way to tell the temperature has changed is to see that the particles are going faster but the increase in temperature can only be shown by the temperature scale and statistics in an image.

**Graphical user interface

Description automatically generated**

**Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generated**

135 Changing the volume

136 Changing the number of light particles

137 Default state of simulation

**Graphical user interface

Description automatically generatedGraphical user interface

Description automatically generated**

138 Changing the temperature

139 Changing the number of heavy particles

#### Success criteria number 4 – 7

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Criteria** | **Explanation/Justification** | **How to evidence** |
| 4 | Reset button | Change’s simulation state to default state. If the user changes many variables, this will allow them to easily revert the simulation back to its default state. | Testing evidence to show the reset button works. |
| 5 | A button to switch between modes | To allow the user to change what mode the simulation is in as my simulation will include modes there will need to be a way to switch between them. | Testing evidence showing the button will switch to each mode |
| 6 | Help button | This button will give the user some guidance if they don’t know how to use the program | A screenshot of the help text that comes up after clicking the button |
| 7 | Pause button | This will allow the user to stop the simulation if they want to see the exact state of the simulation at that point in time | Testing evidence of the button working video evidence of it The pause buttons stopping and starting the simulation |

1. **Reset button**

Graphical user interface, diagram

Description automatically generatedThe reset button Is an essential feature for the program and as explained in the diagram, will reset the simulation to a default state that will vary based on the current mode. The diagrams below show the reset button functioning.

Graphical user interface, application

Description automatically generatedDiagram

Description automatically generatedHere is the evidence for the reset button. I used Boyles-Law mode as an example mode to use and its default state has 150 particles and a volume of 5.8. The video evidence shows the reset button for each mode.

1. **A button to switch modes**

The button that switches the mode the simulation is currently in will increment an index for an array of modes. To prevent the simulation from crashing when clicking the button at the last mode, it will loop round, using the modulus operator, to the first mode.

1. **A help button**

As well as give the user guidance on how to use each mode, the help button will give some details on what is happening each mode and why it is happening. This is to give some context to the simulation and relate it back to the gas laws.

In the initial success criteria, I did not plan to have different text from the help button for each mode but when developing I had this idea as more specific help can be given.

Text

Description automatically generatedHere is the evidence of the help buttons implementation in each mode. The text at the top is always the same for each mode as it gives more general advice on using the application with specific advice coming after

1. **Text

   Description automatically generatedPause button**

The pause button will stop the simulation from running when clicked and the simulation is running, if the simulation is paused then the simulation will resume.

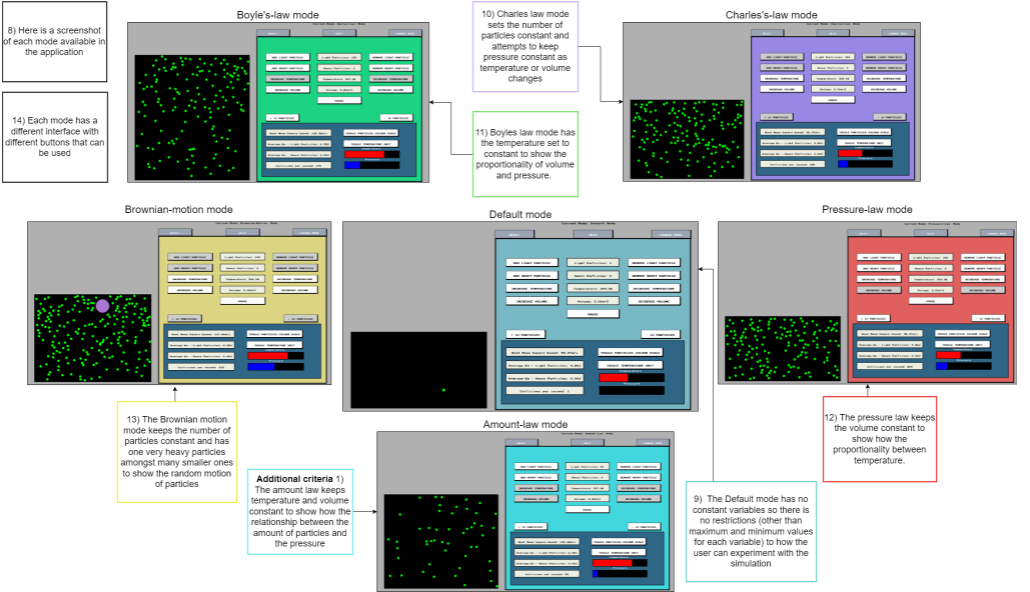
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Expected outcome** | **Actual outcome** | **Video evidence** |
| The pause button can pause the simulation from | Clicking the pause button while simulation is running | The simulation will stop updating, all values in the statistics will also stay the same | Same as expected |  |

Video evidence here

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Expected outcome** | **Actual outcome** | **Video evidence** |
| The pause button can resume the simulation when it is paused | Clicking the pause button while simulation is paused | The simulation will continue running and updating as normal | Same as expected |  |

#### Success criteria number 8 – 14

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Criteria** | **Explanation/Justification** | **Evidence** |
| 8 | Multiple modes/examples | To provide multiple different ways the user can experiment with the software and demonstrate different gas laws. | Showing a screenshot of the program in each mode as the and evidence from development |
| 9 | Default mode | The default mode will not be related to a specific law and therefore should allow the user to change any of the variables in the simulation with none needing to be kept constant. | Screenshot of application on start-up where any variable can be changed |
| 10 | Charles law mode | The user can experiment with the volume and temperature to see how gases expand when heated | Screenshot of the program in this mode |
| 11 | Boyles law mode | User can experiment with the volume and pressure at a constant temperature to understand their proportionality. | Screenshot of the program in this mode |
| 12 | Pressure law mode | To allow the user to experiment with relationship between pressure and temperature while volume is kept constant. | Screenshot of the program in this mode |
| 13 | Brownian motion mode | This is mode will include one very large particle amongst many smaller particles. It will help the user understand the random motion of particles by how they change the path of the large particle. | Screenshot of the program in this mode |
| 14 | A different interface for each mode | This is necessary as each mode will have different parts of the simulation that can be changed. | Screenshot of each mode’s interface |

****

**Extra success criteria (for amount law and other additions)**

**Development testing**

* **Get stakeholder feedback**
* **Add the date of change????**
* **Refer back to design – discuss flowcharts, testing and user interface design** !
* **Add success criteria to evaluation**

**Evaluation:**

**Post development testing**

**Remember It is 20 marks**

**Get user feedback on the simulation**

1. <https://ch301.cm.utexas.edu/simulations/js/idealgaslaw/> [↑](#footnote-ref-2)
2. Webpage: [**https://www.falstad.com/gas/**](https://www.falstad.com/gas/) [↑](#footnote-ref-3)
3. Webpage: [**https://teachchemistry.org/classroom-resources/the-gas-laws-simulation**](https://teachchemistry.org/classroom-resources/the-gas-laws-simulation) [↑](#footnote-ref-4)
4. Web page about simulating ideal gas particles: <https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2008/sdh78_cc459/sdh78_cc459/ideal_gas_simulator.html> [↑](#footnote-ref-5)
5. Github page for the rendering engine I will be using: <https://github.com/OneLoneCoder/olcPixelGameEngine> [↑](#footnote-ref-6)
6. Webpage of the application I used <https://hihayk.github.io/scale/> [↑](#footnote-ref-7)
7. Page that helped me fix the error: <https://stackoverflow.com/questions/61455321/c2676-binary-const-ty-does-not-define-this-operator-or-a-conversion-to> [↑](#footnote-ref-8)