1 Preamble

1.1 Table of contents

1.2 Abstract

The aim of this project was to create a software teaching tool that allows the user to investigate the properties of an ideal gas through the use of a particle simulation. The initial simulation, along with the mechanisms for the user to control the simulation, was created in Java, and allowed the user to investigate the properties of ideal gases. The functionality of this simulation was then extended to explain the more advanced concepts of heat engines and activation energy. This was achieved by including tools and graphics to relate the behaviour of the particles to the to the properties of the gas as a whole.

The software produced shows great promise, but to make it a worthy teaching tool it requires more work with teachers and students to refine its potential for learning and overall usability.

1.3 SVN repository

All software for this project can be found at:
https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2017/ljh590.

1.4 Acknowledgements

I would like to thank my project supervisor Steve Vickers for his continued support throughout the development of the project. His ideas and support for the physics of the simulation helped to shape the result of the project.

I would also like to thank Ela Claridge for the feedback and resources she provided to aid in the improvement and evaluation of the usability of the software.

2 Introduction

The aim of the work described in this report was to create a software teaching tool to allow users to investigate the behaviour of ideal gases in addition to some more advanced processes. The development of the project was split into two parts. The first part involved the creation of the initial simulation, along with the mechanisms for the user to control the simulation. This part was inspired by the PhET “Gas Properties” simulation (PhET Interactive Simulations, 2018), which is a brilliantly simple and user friendly simulation intended to teach students about the behaviour of ideal gases, and how this behaviour is affected by changes to properties such as the temperature of the gas. This idea provided the foundation for the project.

The second part of the development was more experimental and involved extending the functionality of the simulation to explain more advanced processes. To fully utilise the simulation, these processes had to be conceptually difficult to understand and benefit from the visual demonstrations that could be made possible by the simulation. For these reasons, the processes chosen to demonstrate were heat engines and activation energy. A heat engine uses heat energy in the system to perform mechanical work on its surroundings. The activation energy of a particle is the energy required to trigger some process, for instance breaking its bonds and causing a change of state.

The aim of the second part of development was to demonstrate these two processes as accurately and consistently as possible given the available time. To effectively explain the processes, the microscopic properties of the particles simulation (such as the speed and energy of individual particles) were related to the macroscopic properties of the gas as a whole (such as the total pressure exerted on the container and the temperature of the gas).

Since the end goal of the project was to create a capable teaching tool, it was crucial to present the data of the simulation in a concise and interesting manner. This was most easily accomplished by displaying graphs, which enabled the user to more easily relate the results

of the simulation to the theory of the process being presented. It was also important to carry over the excellent usability of the PhET simulation to ensure the software was as easy to learn and use as possible. A high level of usability allows the user to focus their attention on the process presented, rather than diverting their attention to navigate and understand how to use the software.

The model-view-controller design pattern was used to create the software and present the simulation to the user. This pattern is effective at allowing more flexible development which responds better to changes and experimentation in the software due to the logical separation of the major components. The user interface acts as the controller and passes its inputs to the simulation, which is considered as the model of the software. The views of the software then regularly update based on the simulation to display the particles and graphics.

3 Further background material

This section provides the necessary information to understand both the inspiration for the project and the physical processes demonstrated by the program.

3.1 PhET “Gas Properties” simulation

Physics Education Technology (PhET) is a project by the University of Colorado which provides many educational simulations. These simulations are designed to be user friendly and easy to learn so that the user is able to quickly explore and investigate it. One such simulation is the Gas Properties simulation, intended to teach students about the behaviour of ideal gases and how this behaviour changes when properties such as the temperature of the gas and the volume of the container vary.

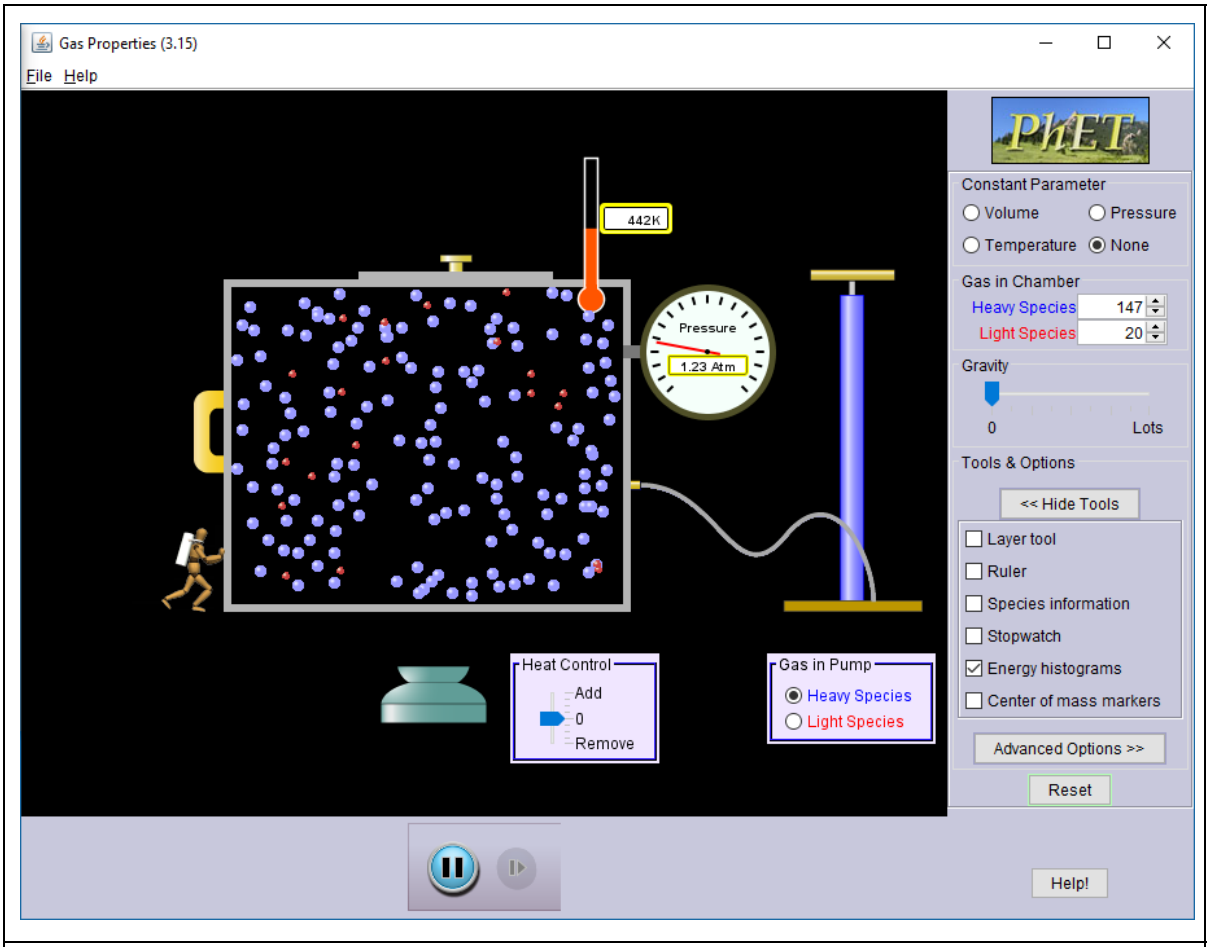


Figure 3.1.1: The PhET Gas Properties simulation (PhET Interactive Simulations, 2018).

As figure 3.1.1 shows, the simulation has a friendly interface which tries to keep its apparatus familiar and similar to a lab experiment. The average temperature of the particles is displayed using a thermometer and the pressure exerted by the particles on the container is displayed using a pressure gauge. The pump to the right of the container makes it clear to the user that the container is being filled by a gas. These are items that a typical user would instantly recognise and be confident in using in the context of the simulation.

A stove is placed below the container to allow the user to vary the temperature of the gas particles. When heat is added the stove emits a flame, and ice cubes appear above the stove when heat is removed. This makes it obvious to the user what the stove is doing to the gas. The user is also greeted by instant changes to the values displayed in the thermometer and pressure gauge, immediately notifying the user of the impact of adding or removing heat.

The handle on the left wall can be dragged to move the wall in or out. Moving the wall inwards decreases the volume of the container, which in turn increases the pressure exerted by the particles on the container, since the particles collide more frequently with the walls. Particles that collide with the wall moving inwards gain energy, increasing the average temperature of the particles and further increasing the pressure. Moving the wall outwards exhibits the opposite behaviour where both temperature and pressure decrease.

The door on top of the container can be slid open to allow the particles to escape the container, with the size of the resulting gap controlling the rate at which the particles escape. When the pressure is too high, the door is ejected and allows the particles to escape, conveying realism to the user by illustrating the container's pressure limit.

The design of the program makes it clear that usability and learnability are primary aims of the simulation. The user interacts directly with the apparatus and whenever they do so the user interface provides instantaneous feedback on their actions. This aids in making the simulation fun, interactive and easy to use.

In addition to the basic simulation, the program also provides a number of tools in the right sidebar to encourage further experimentation.

The "Constant Parameter" box allows the user to set either volume, pressure or temperature to be kept at a constant value. Setting the volume to be constant causes the left wall to no longer be moveable, thus enabling the user to explore the pressure law (with constant volume, the pressure exerted by a gas is proportional to its temperature). When the pressure is set be constant, the left wall is automatically moved to maintain a constant value, enabling the exploration of Charles' law (at a constant pressure, the volume of a gas is proportional to its temperature). To maintain a constant temperature the stove is automatically controlled. In this way, the user explores Boyle's law (at a constant temperature, the pressure and volume of a gas are inversely proportional).

Energy and speed histograms of the particles can be displayed which are regularly updated to reflect the current energy and speed of the particles. As the average temperature of the particles changes, the shapes of the distributions change to reflect this.

Several additional tools are available in the simulation to further investigate the behaviour of ideal gases. For example, two sizes of particles can be pumped into the container to demonstrate how changing the mass of a particle with the same energy will affect its speed. These tools are much less of a focus in the simulation but still provide valuable learning opportunities.

In summary, the PhET simulation provides an excellent introduction to ideal gases along with many tools for students to experiment with the gas' behaviour. These tools enable the user to effectively investigate how the properties of the gas vary in relation to each other.

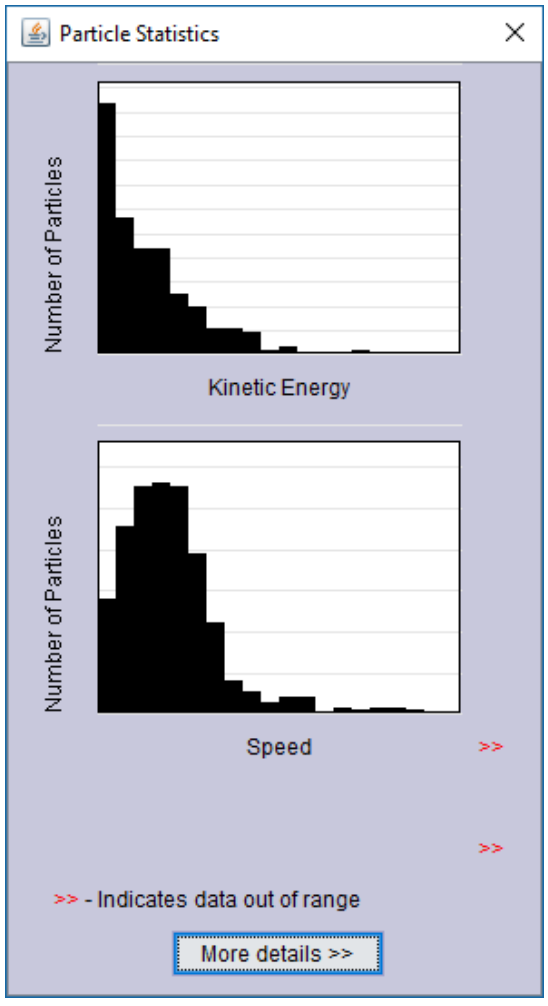


Figure 3.1.2: Speed and energy distributions of the particles as seen in the PhET simulation.

3.2 The Carnot cycle and heat engines

The Carnot cycle is a reversible process which provides an upper limit on the efficiency of any engine. The cycle is reversible because the engine can convert heat into work (as in a typical car engine), or conversely work can be applied to the system in order to remove heat from it (as in a refrigerator). The former case is called a heat engine, where heat energy in the system is used to perform work on its surroundings, for example by moving a piston. The Carnot cycle assumes that no energy is lost to wasteful processes such as friction.

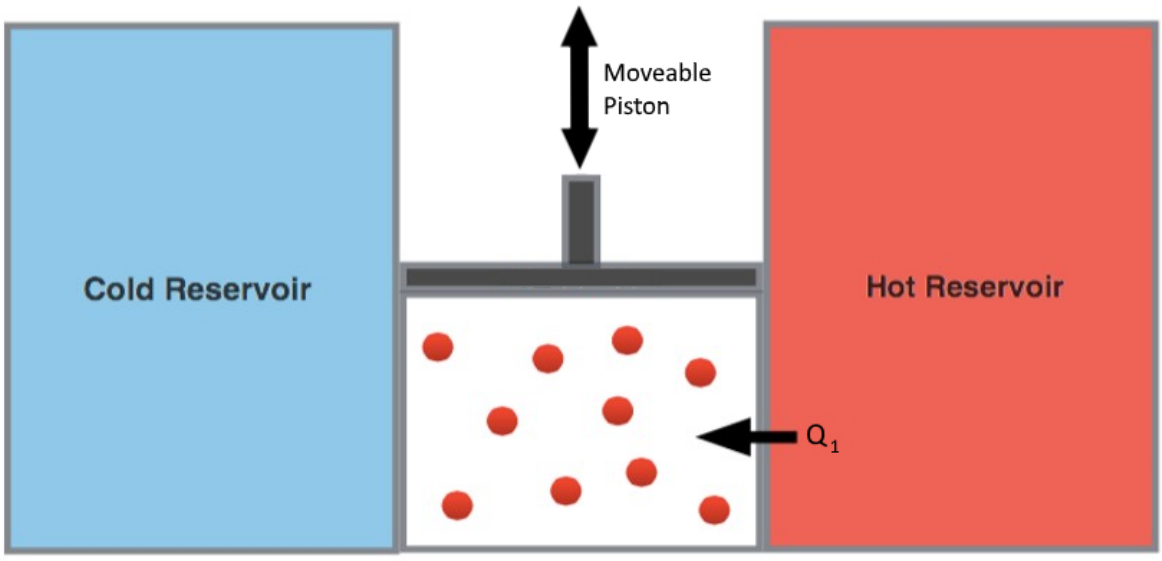


Figure 3.2.1: An overview of the model used in the Carnot cycle (during step 1 of the Carnot cycle when acting as a heat engine) (amended from BlyumJ, 2017).

In the model used in the Carnot cycle there are two "heat reservoirs" at temperatures T_H and T_C (hot and cold respectively). Their thermal capacity is so large that we assume their temperatures are unaffected by a single cycle. Figure 3.2.1 shows an overview of the model being used.

During a single cycle, an arbitrary amount of entropy ΔS is extracted from the hot reservoir, and deposited in the cold reservoir. The total amount of entropy remains unchanged.

The change in entropy is defined as $\Delta S = \frac{\Delta Q}{T}$, where ΔQ is the amount of heat energy transferred into the system, and T is the temperature of that system. When heat is transferred out of the system the sign of ΔQ is reversed, resulting in a decrease of entropy in the system.

An amount of energy $T_H \Delta S$ is also extracted from the hot reservoir and a smaller amount of energy $T_C \Delta S$ is deposited in the cold reservoir. The work done by the engine is equal to the difference in the two energies: $(T_H - T_C)\Delta S$.

The Carnot cycle when acting as a heat engine is a four-stage cycle as follows:

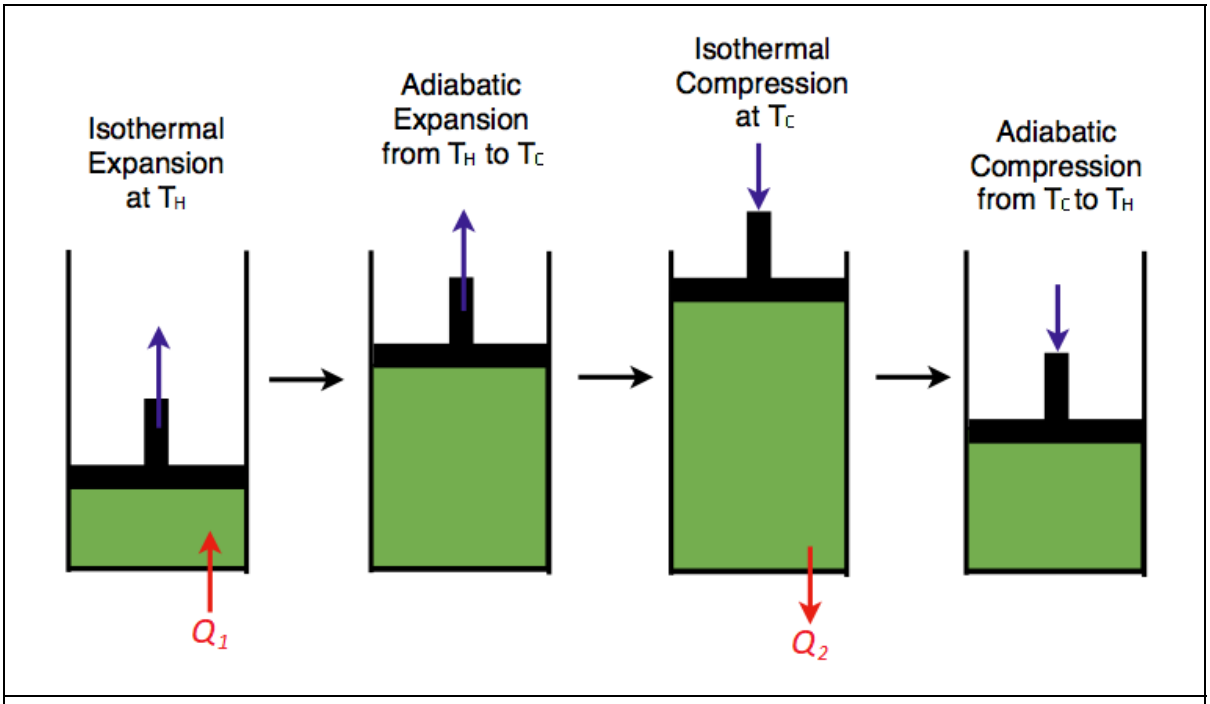


Figure 3.2.2: A gas-piston model of a Carnot heat engine (amended from Hou, 2017).

Step 1 (1 to 2 on figure 3.2.3, A to B in figure 3.2.4, container uninsulated): The gas is allowed to expand and it does work on the surroundings.The temperature of the gas does not change during the process, and thus the expansion is isothermal. The gas expansion is propelled by absorption of heat energy Q_1 from the high temperature reservoir and results in an increase of entropy of the gas in the amount $\Delta S_1 = \frac{Q_1}{T_H}$.

Step 2 (2 to 3 on figure 3.2.3, B to C in figure 3.2.4, container insulated): The mechanisms of the engine are assumed to be thermally insulated, thus they neither gain nor lose heat (an adiabatic process). The gas continues to expand, doing work on the surroundings, and losing an amount of internal energy equal to the work done by the system. The gas expansion causes it to cool to the "cold" temperature, T_C. The entropy remains unchanged.

Step 3 (3 to 4 on figure 3.2.3, C to D on figure 3.2.4, container uninsulated): Now the surroundings do work on the gas, causing an amount of heat energy Q_2 to leave the system to the low temperature reservoir, and the entropy of the system decreases in the amount $\Delta S_2 = \frac{Q_2}{T_C}$. (This is the same amount of entropy absorbed in step 1.)

Step 4 (4 to 1 on figure 3.2.3, D to A on figure 3.2.4, container insulated): During this step, the surroundings do work on the gas, increasing its internal energy and compressing it, causing the temperature to rise to T_H due to the work added to the system, but the entropy remains unchanged. At this point the gas is in the same state as at the start of step 1.

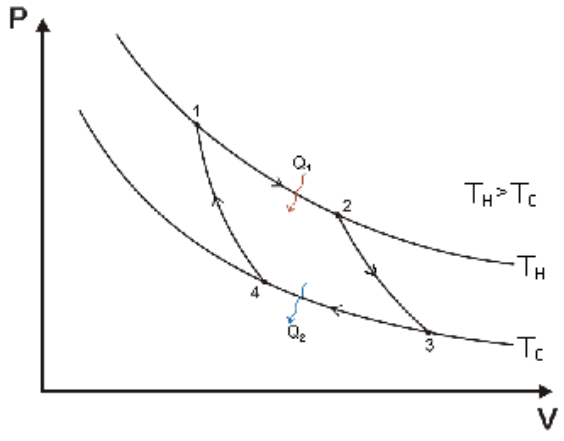


Figure 3.2.3: A pressure against volume diagram of the Carnot cycle (amended from Keta, 2006).

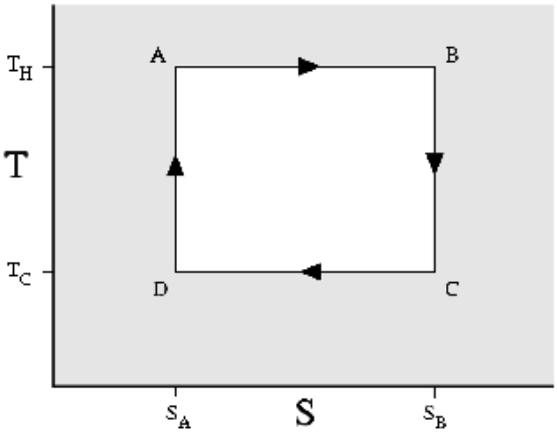


Figure 3.2.4: A temperature against entropy diagram of the Carnot cycle (amended from Wikimedia Commons contributors, 2017).

3.3 Activation energy

The activation energy, ε, is the amount of energy required to trigger a process. Examples of processes include:

a) A change of state: The particles need enough energy to break the bonds that hold them together.
b) Thermionic emission: Electrons released by a heated conductor need enough energy to escape from the attraction of the positive nuclei in the conductor.
c) Conduction in a semiconductor: Semiconductors will only start to conduct once there are electrons in a high-energy state. The electrons therefore need enough energy to reach this high-energy state.

In each of these examples, the activation energy comes from the random thermal energy of the particles. The ratio $\frac{\varepsilon}{kT}$ is very important (k is Boltzmann's constant, 1.38 x 10^{-23}, T is the temperature in kelvin). When kT is big enough compared with ε, some particles have enough energy to begin the process, but if the ratio $\frac{\varepsilon}{kT}$ is too high, no processes are triggered.

As $\frac{\varepsilon}{kT}$ gets down to around 15-30, the process starts to occur at a moderate rate. Some particles must therefore have energies of 15-30 times the average energy in order to trigger the process. Every time particles collide, there's a chance that one of them will gain extra energy above and beyond the average kT. If that happens several times in a row, a particle can gain energies much greater than the average. Only a tiny proportion of particles will reach an energy of 15 kT to 30 kT. However, this small fraction still represents a large number of particles due to the typically huge numbers of particles colliding billions of times each second.

Particles with different energies are in different energy states. For example, a particle with an energy of $kT + \varepsilon$; is in a higher energy state than a particle with an energy of kT. The Boltzmann factor, $e^{-\frac{\varepsilon}{kT}}$, is used to find the ratio of the numbers of particles in energy states ε joules apart.

For $\frac{\varepsilon}{kT}$ = 15 the Boltzmann factor is ~10^{-7}, and for $\frac{\varepsilon}{kT}$ = 30 it's only ~10^{-13}. That means that only about one in 10^{13} to one in 10^{7} particles have enough energy to overcome the activation energy. Even though this is a tiny proportion, it is important to remember how fast these particles are moving and how often they are colliding with each other. Even with so few particles having enough energy, the reaction can happen in a matter of seconds.

For any particular reaction, the values of ε and k are fixed. This means that the only variable that will have an affect on the Boltzmann factor is the temperature. A graph of the Boltzmann factor against temperature produces a curve as seen in figure 3.3.1. At low temperatures the Boltzmann factor is also very low, so very few particles will have energies at or above the activation energy and the reaction will occur very slowly. At high temperatures the Boltzmann factor approaches 1, so a much greater proportion of the particles will have sufficient energy to react and the reaction will occur very quickly. In between, the Boltzmann factor increases rapidly with temperature. So a small increase in temperature will have a large effect on the rate of reaction. The rate of a reaction with activation energy ε is therefore proportional to the Boltzmann factor, $e^{-\frac{\varepsilon}{kT}}$.

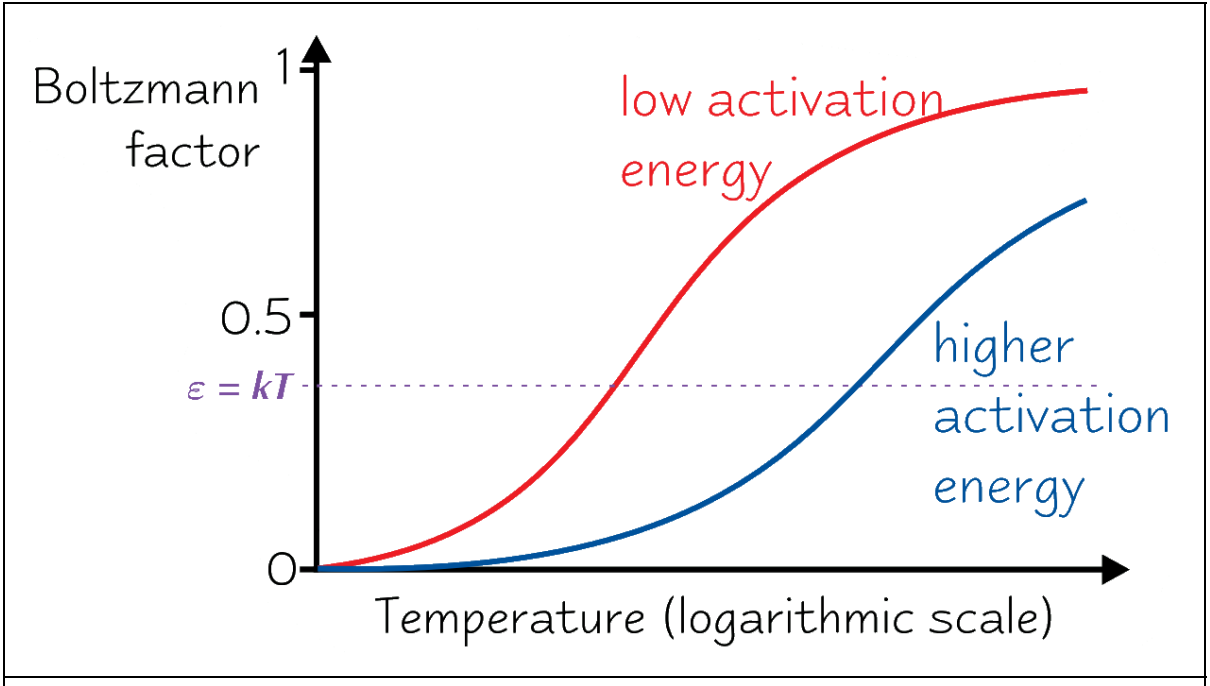


Figure 3.3.1: Boltzmann factor against absolute temperature (CGP Books, 2009).

4 Analysis and specification

The development of this project was split into two phases. The goal of the first phase was to create an initial simulation with the essential features of the PhET simulation. Since the goal was clear, the requirements here were well understood.

The second phase was a more experimental one where the requirements were not as well defined. The goals here were to demonstrate the processes of heat engines and activation energy. After reading around the subjects, the requirements became more clear. For example, to demonstrate heat engines it would be necessary to recreate the four steps of the Carnot cycle in the simulation, similarly to what is shown in figure 3.2.2. By further breaking down the problem, the essential requirements became obvious. Additional requirements were then added to aid the usability and learnability of the program; for example ensuring that the simulation ran smoothly, and providing background information to the user so that they are able to get the most out of the simulation.

Each requirement listed in this section has an importance attached to them. This importance corresponds to a category from the MoSCoW method:

Must	Features that absolutely have to be implemented. If any of these requirements are not met, the project will be considered a failure.
Should	Features that are important but not absolute musts.
Could	Features that are nice to have but are not essential.
Won't	Features that are not going to be implemented at this stage, although this may change at a later stage.

4.1 Requirements for the simulation phase

4.1.1 Functional requirements

Label	Title	Requirement	Importance
4.1.1.1	Simulate a large number of particles	The simulation should support up to 500 particles.	Must
4.1.1.2	Adjust the wall temperature	The temperature of the walls of the container should be adjustable, with a minimum temperature of 200 kelvin and a maximum of 4000 kelvin.	Must
4.1.1.3	Adjust the number of	The number of particles in the simulation should be adjustable in the range of 0 to 500.	Must

	particles	A change in the number of particles should be reflected instantly in the simulation.	
4.1.1.4	Adjust simulation speed	The speed of the simulation should be adjustable in the range of 0.25x to 4x speed.	Could
4.1.1.5	Display current temperature of particles	The current average temperature of the particles should be updated and displayed on the screen in real time.	Must
4.1.1.6	Display current pressure exerted by particles	The current average pressure exerted by the particles on the container should be updated and displayed on the screen in real time.	Must
4.1.1.7	Particles collide convincingly	Collisions between two particles and between a particle and a wall should be accurate representations of real physics and appear convincing to the user.	Must
4.1.1.8	Adjust the size of the container	The size of the container must be adjustable. This should be done by allowing the user to move exactly one of the walls inwards and outwards.	Must
4.1.1.9	Pause the simulation	The simulation should be able to be paused, stopping the movement of the particles and stopping any graphs from updating.	Should
4.1.1.10	Restart the simulation	The simulation should be able to be restarted, respawning the particles and setting the average particle temperature to be equal to the wall temperature.	Should
4.1.1.11	Reset the simulation	The simulation should be able to be reset, setting the value of the wall temperature, number of particles, simulation speed and any other values adjustable by the user back to their default values.	Could
4.1.1.12	Particles should be large enough to follow	The size of the particles should be large enough to allow the user to follow the movement of individual particles.	Must
4.1.1.13	Speed distribution graph	The program should display a speed distribution graph (percentage of particles against speed of the particles) which updates several times each second.	Must
4.1.1.14	Energy distribution graph	The program should display an energy distribution graph (percentage of particles against energy of the particles) which updates several times each second.	Must

4.1.2 Non-functional requirements

Label	Title	Requirement	Importance
4.1.2.1	Open within a reasonable amount of time	The program should open in at most 3 seconds.	Must
4.1.2.2	Run smoothly even with a large number of particles	The simulation should update at least 30 times per second even when simulating the maximum number of particles.	Must
4.1.2.3	Program should have a small file size	The size of the program (the executable JAR file) should be no larger than 5 megabytes.	Must
4.1.2.4	User inputs should be processed in real time	The program should respond to user inputs in at most 0.2 seconds.	Must
4.1.2.5	Simulation should not crash	The simulation must be resilient to run time bugs and should handle most fatal exceptions, resulting in no more than one per hour.	Must
4.1.2.6	Graphs update in real time	The distributions of the graphs should be recalculated and updated on the screen within 0.1 seconds to ensure that the distributions are representative of the current speeds and energies of the particles.	Must

4.2 Requirements for the experimental phase

The previous requirements from section 4.1 still apply in this phase.

4.2.1 Functional requirements

Label	Title	Requirement	Importance
4.2.1.1	Pressure vs volume graph	The program should display a pressure against volume graph which updates several times each second.	Must
4.2.1.2	Temperature vs entropy graph	The program should display a temperature against entropy graph which updates several times each second.	Must
4.2.1.3	Background	Background information of heat engines	Must

	information of heat engines	should be available inside the program. This should explain enough information for the user to understand and experiment with the simulation.	
4.2.1.4	Automatically move wall in	A button should cause the moveable wall to automatically move inwards at a slow and steady rate. This is necessary to automatically create a Carnot cycle.	Must
4.2.1.5	Automatically move wall out	A button should cause the moveable wall to automatically move outwards at a slow and steady rate.	Should
4.2.1.6	Allow particles to push the moveable wall	The user should be able to toggle whether or not the particles can push the moveable wall when they collide with it. This is necessary to automatically create a Carnot cycle.	Must
4.2.1.7	Toggle insulation of the walls of the container	The user should be able to toggle whether or not the walls of the container are insulated. Collisions with an insulated container are elastic, whereas collisions with an uninsulated container will cause the particles to move slightly towards the temperature of the walls. This is necessary to automatically create a Carnot cycle.	Must
4.2.1.8	Demonstrate Carnot cycle (heat engines, heat into work)	The program should be able to automatically demonstrate a Carnot heat engine through the following steps: 1. Container smallest it can be; particles push the moveable wall outwards; hot reservoir; container uninsulated 2. Container ~⅔ maximum size; particles still pushing; hot reservoir; container insulated 3. Container largest it can be; moveable wall moves in automatically; cold reservoir; container uninsulated 4. Container ~⅓ maximum size; moveable wall still moving in; cold reservoir; container insulated	Must
4.2.1.9	Demonstrate Carnot cycle (refrigeration, work to remove heat)	The program should be able to automatically demonstrate a Carnot refrigeration cycle through the following steps: 1. Container largest it can be; moveable wall moves in automatically; hot reservoir; container insulated 2. Container ~⅔ maximum size; moveable wall still moving in; hot reservoir; container	Could

		uninsulated 3. Container smallest it can be; particles push the moveable wall outwards; cold reservoir; container insulated 4. Container ~⅓ maximum size; particles still pushing; cold reservoir; container uninsulated	
4.2.1.10	Boltzmann factor vs reaction rate graph	The program should display a Boltzmann factor against reaction rate graph which updates several times each second.	Should
4.2.1.11	Background information of activation energy	Background information of activation energy should be available inside the program. This should explain enough information to the user for them to understand and experiment with the simulation.	Must
4.2.1.12	Adjust activation energy	The activation energy of the particles should be adjustable. The activation energy should range from 0 joules to a value where only a very small number of high energy particles reach the activation energy.	Must
4.2.1.13	Colour particles at activation energy	Particles that have energies at or above the activation energy should be coloured differently to highlight them.	Must
4.2.1.14	Make particles disappear at activation energy	Particles that reach the activation energy should disappear (be removed from the simulation).	Should
4.2.1.15	Animation when particles disappear at activation energy	Particles that reach the activation energy should disappear, and an animation should play at the location of the particle to make the user aware that the particle disappeared.	Could
4.2.1.16	Particles release energy when they disappear at activation energy	After particles reach the activation energy and disappear, particles nearby should gain energy, akin to an exothermic reaction.	Could
4.2.1.17	Informational help screen	An informational help screen should be available in the program to provide the user with instructions on how to use and get the most out of the program.	Should

4.2.2 Non-functional requirements

Label	Title	Requirement	Importance
4.2.2.1	Graphs updated in real time	New points on the graphs should be calculated and updated on the screen within 0.1 seconds to ensure that the graphs are representative of the current state of the particles.	Must

5 Design

5.1 Overview of the software structure

The software for this project was designed under the model-view-controller (MVC) design pattern. Using this pattern, development becomes more flexible to changes and experimentation due to the logical separation of the major components of the software.

The model receives user input via the controller and is responsible for managing the data of the software. In the software of the project a ‘model’ class is used to control the simulation and communicate to the ‘view’ components. The simulation itself regularly updates a list of particles as well as detecting and resolving any collisions involving the particles. Statistics such as temperature, pressure and entropy are also extracted from the particles.

The view in an MVC design pattern presents the data of the model in a meaningful way. The project included two view components: The *GraphView* class takes statistics such as temperature and pressure from the simulation and produces graphs that are frequently updated on the screen. *SimComponent* takes the list of particles from the simulation and draws them onto the screen, along with the container enclosing them.

The controller is responsible for handling user input and, when necessary, passing this input onto the model. The *ControlPanel* class is the project’s implementation of a controller, which creates and arranges the user-interactable components on the display. Input from these components is validated and then sent to the model to be handled.

5.2 Algorithm choices

5.2.1 Collision detection

5.2.1.1 Collision detection between two particles

Each particle is compared to every other particle to detect any collisions. This has time complexity $O(n^2)$, although provided the collision detection check is efficient and the number of particles is relatively small, this method is still reasonably efficient. An alternative approach is to split the container into segments and only compare each particle to other particles in its current segment. While this would improve the efficiency of the simulation, the previously stated solution was found to efficient enough without the added complexity of implementation.

To detect a collision, two particles are tested to find whether they overlap. If the distance between the centres of the two particles is less than the sum of their radiuses then a collision has occurred. This is very efficient and simple to implement, and detects collisions accurately enough for the purposes of the simulation. One improvement on this is to check whether the particles overlap using bounding boxes first, then if that check passes test the distance between them. Since the majority of particles will not be colliding in each iteration of the simulation, performing a slightly more efficient check first would yield slightly improved efficiency. Although this idea was not adopted, it is a simple idea that could be implemented in the future if the efficiency of the simulation needs to be improved.

A predictive method could be used to improve the accuracy of the collision detection. An implementation using parametric equations would be able to determine the exact point of collision so that collisions could be handled much more accurately. However, this method is much more expensive and complex than checking for overlaps, and the level of accuracy is not required for a simulation at this level. This predictive method was therefore not adopted.

5.2.1.2 Collision detection between a particle and a wall

Collision detection between a particle and a wall is very simply implemented using overlap checks. If the distance between the wall and the centre of the particle is less than the particle's radius then a collision has occurred. Similarly to collisions between particles, a predictive method could also be applied here to improve the accuracy of the collision detection. However, just as between particles this method was not implemented due to its added unnecessary complexity.

5.2.2 Resolving collisions

5.2.2.1 Resolving collisions between two particles

A simple seven step method was adopted to resolve collisions between particles using vectors (Berchek, 2009). This suited the implementation of particles because of their vector

representation, and the method itself could be easily translated into working code. The resulting collisions are accurate and the process is fairly efficient.

The method is as follows:

1. Find the unit normal and unit tangent vectors. The tangent vector is tangent to the particles' surfaces at the point of collision.

$$\vec{n} = \langle x_2 - x_1, y_2 - y_1 \rangle$$

$$\vec{un} = \frac{\vec{n}}{|\vec{n}|} = \frac{\vec{n}}{\sqrt{{n_x}^2 + {n_y}^2}}$$

$$\vec{ut} = \langle -\vec{un}_y, \vec{un}_x \rangle$$

2. Create the initial (before the collision) velocity vectors, $\vec{v_1}$ and $\vec{v_2}$.

3. Resolve the velocity vectors into normal and tangential components. This is done by taking the dot products between the velocity vectors and the unit normal and unit tangent vectors.

$$v_{1n} = \vec{un} \cdot \vec{v_1}, v_{1t} = \vec{ut} \cdot \vec{v_1}, v_{2n} = \vec{un} \cdot \vec{v_2}, v_{2t} = \vec{ut} \cdot \vec{v_2}$$

4. Find the new tangential velocities (after the collision).

$$v_{1t}^{'} = v_{1t}, v_{2t}^{'} = v_{2t}$$

5. Find the new normal velocities using the one-dimensional collision formulas (since we only need to consider the velocities of the particles in the normal direction).

$$v_{1n}^{'} = \frac{v_{1n}(m_1 - m_2) + 2m_2 v_{2n}}{m_1 + m_2}, v_{2n}^{'} = \frac{v_{2n}(m_2 - m_1) + 2m_1 v_{1n}}{m_1 + m_2}$$

6. Convert the scalar normal and tangential velocities into vectors.

$$\vec{v_{1n}^{'}} = v_{1n}^{'} \cdot \vec{un}, \vec{v_{1t}^{'}} = v_{1t}^{'} \cdot \vec{ut}, \vec{v_{2n}^{'}} = v_{2n}^{'} \cdot \vec{un}, \vec{v_{2t}^{'}} = v_{2t}^{'} \cdot \vec{ut}$$

7. Find the final velocity vectors by summing the normal and tangential components for each object.

$$\vec{v_1^{'}} = \vec{v_{1n}^{'}} + \vec{v_{1t}^{'}}, \vec{v_2^{'}} = \vec{v_{2n}^{'}} + \vec{v_{2t}^{'}}$$

5.2.2.2 Resolving collisions between a particle and a wall

Resolving these types of elastic collisions against stationary walls is simple. When a particle collides with either the left or right wall, reverse its x component of velocity. When a particle collides with either the top of bottom wall, reverse its y component of velocity. In both cases, the particle should be moved back into the container so that it is touching the wall.

Resolving these types of collisions for moving walls proved to be much more of a challenge. The basic idea was to use the one-dimensional collision formula, with subscript 1 denoting the particle and subscript 2 denoting the wall.

$$v_1 = \frac{u_1(m_1 - m_2) + 2m_2u_2}{m_1 + m_2}$$

To convincingly demonstrate the behaviour of ideal gases, the mass of the wall had to be found experimentally. See section 6.4 for details.

5.3 Usage of threads

Threads are used and managed in a way to avoid concurrent access problems across the software. In the main *Simulation* class only one thread is used, which regularly updates the particles in the simulation. Not only does this improve thread safety, it also made the design and implementation of the simulation much easier.

Several other threads are used to manipulate and extract information from the simulation. For example, *SimComponent* regularly retrieves a copy of the particles from the simulation to draw them onto the screen. Whenever a change is requested in the simulation that would affect an iteration in progress such as altering the number of particles, this change is stored and then updated in the next iteration. This prevents errors and odd behaviour in the simulation.

5.4 Other design decisions

Perhaps the most obvious design decision was to simulate the particles and container in two dimensions rather than three. This was primarily done to keep the simulation as simple to understand and follow for the user as possible. Additionally the simulation would be much less computationally expensive to render on the screen, as well as simpler to implement.

For the implementation of the simulation, a decision was made to store completed iterations of the simulation in a buffer to render that buffer on the screen. In this case, a buffer is a snapshot of the particles and the width of the container at the end of an iteration, after all collisions have been resolved. In this way, the movement of the particles appears more natural because the user does not see the particles overlapping with each other or with the walls of the container.

Storing multiple snapshots in the buffer helps to prevent any microstutters. This is because a backlog of snapshots is available even when the performance of the system in which the

simulation is running on varies. Storing multiple snapshots also provides the mechanisms for running a simple benchmark. The performance of the simulation can be tested by setting the maximum size of the buffer to a large number, for example 10,000, and timing how long it takes to fill the buffer. This method of benchmarking was used in the early phases of the development of the simulation to test the effects that small optimisations had on the overall performance of the program.

The number of particles in the simulation was limited to a maximum of 500. This was to keep the number of particles relatively low so that the user could keep track of what was going on; ideally the user should be able to follow the path of each particle (requirement 4.1.1.12). With a reasonable size for each particle, 500 particles barely fit into the container at its smallest size. Finally, although efficiency of the simulation is not so much of an issue, simulating a large number of particles may be a challenge for some older computers, which are common in an educational environment.

Other limits set in the simulation include the wall temperature and activation energy. The values for these were chosen to display a range of behaviours of the particles. The range of temperatures for the walls (200 K to 4000 K) are arbitrary values which showcase the particles moving from very slowly to very quickly. The range of activation energies (0 J to ~100 x 10^{-21} J) are also arbitrary values. These values were chosen since an activation energy of 0 J means that all of the particles are at or above the activation energy, and an activation energy of ~100 x 10^{-21} J leads to none or hardly any of the particles reaching the activation energy.

6 Implementation

The following sections describe and discuss the implementation of a few key functions that control the physics of the simulation. Lots of thought and experimentation was put into these functions to ensure that they demonstrate the physics as accurately and convincingly as possible.

6.1 Updating the simulation

The particles in the simulation are updated by the *actionPerformed* method. A *Timer* is set up which calls this function every 2 milliseconds to regularly update the simulation. This *Timer* can be paused and resumed, making it very easy to pause or resume the simulation.

The *Timer* controls a single thread which executes this method. Using a single thread makes thread safety less of a worry and implementation in general is much simple (see section 5.3).

The *actionPerformed* method works as follows: First, any requested changes to the buffer or number of particles are made. Then, if the buffer is not full, the positions and velocities of all particles are updated, and any detected collisions are resolved. Afterwards, the updated list of particles is appended to the buffer, and statistics such as the average temperature of the particles are recalculated.

Description	
Simulation: actionPerformed() - Function called every 2 ms which performs one iteration (a single step or frame) of the simulation. This updates the particles in the simulation as well as macroscopic properties such as temperature, pressure and entropy.	
Data structures	
buffer	A list of *SimBuffers*. A *SimBuffer* holds the relevant information to draw the particles and the container on the screen. It contains the list of particles from the simulation as well as the width of the container at that time. The list used to store the buffer is a *CopyOnWriteArrayList*. This is a list which allows concurrent reading and writing, which is useful for thread safety.
particles	A list which stores the particle currently in the simulation. A *CopyOnWriteArrayList* is used for its thread safety.
Particle	A *Particle* object holds the information required to represent a particle in the simulation. It stores both the position and velocity vectors of the particle, as well as its radius and mass.

Pseudocode

```
1  /* If a buffered list of particles is requested, the oldest
2  buffer stored must be removed */
3  if buffer requested:
4        remove first item from buffer
5
6  /* If the buffer must be emptied (when low response rates are
7  required, e.g. when moving the wall), update the list of
8  particles using the oldest buffer, then empty the entire buffer
9  */
10 if rollback requested:
11       update particles from oldest buffer
12       clear buffer
13
14 /* If the user has requested a change in the number of
15 particles, update the number of particles accordingly */
16 if number of particles changed:
17       n := newNumParticles - oldNumParticles
18       if (n > 0)
19             /* Spawn n particles in random locations with
20             velocity approximately equal to the average velocity
21             */
22             addParticles(n)
23       else
```

```
24              /* Remove (the absolute value of) n random particles
25              from the simulation */
26              removeParticles(abs(n))
27
28  /* If the buffer is not full, update the list of particles and
29  add the updated list to the buffer */
30  if buffer not full:
31        for each Particle p in particles:
32              /* Update the particle's position based on its
33              velocity */
34              p.move()
35
36              /* Check if the particle has collided with any walls
37              and if so, resolve the collision (see section 6.3)
38              */
39              collideWall(p)
40
41              /* Compare each particle to every other particle */
42              for each Particle q in particles:
43                    if p != q:
44                          /* Check for a collision between p and q.
45                          If a collision is detected, resolve it
46                          (see section 6.2) */
47                          collideParticle(p, q)
48
49        /* After all particles have been updated, check how many
50        of them have energies (i.e. speeds) above the activation
51        energy */
52        for each Particle p in particles:
53              if p.getSpeed() > activationEnergy:
54                    numReactions := numReactions + 1
55                    /* If "Particles disappear at activation
56                    energy" is enabled, remove the particle from
57                    the simulation */
58                    if particles disappear:
59                          particles.remove(p)
60        store numReactions
61        numReactions := 0
62
63        update buffer
64
65        /* Calculate new average temperature and pressure values
66        and store them */
67        calculateCurrT()
68        store temperature
69        calculateCurrP()
70        store pressure
71
72        /* If the container is not insulated, update the value of
73        entropy (heat is not transferred into/out of the
```

```
74          container when it is insulated) */
75          if container isn't insulated:
76                entropy := entropy + (heat transfer / current T)
```

6.2 Detecting and resolving collisions between two particles

The *collideParticle* method detects collisions between two given particles by checking for overlaps (see section 5.2.1.1), and if the two particles are overlapping then the collision is resolved using the method described in section 5.2.2.1.

The *collideParticle* method is called by the *actionPerformed* method (see section 6.1) when updating the list of particles. Since each particle is compared to all other particles, the collision detection must be reasonably efficient to avoid performance problems at high numbers of particles. When calculating the distance between the two particles, the square of the distance is taken. This avoids using the costly square root function which would severely impact performance.

In this method there is a check for a special case where particles are perfectly overlapping. This is typically caused by many particle moving very quickly out of the container so that they collide with two walls, and all of the particles are then placed back into the container at the same coordinates. This causes all of the particles overlapping to perform a collision with each other, but since the distance between each particle is 0, the velocity of the particles is also set to 0. This results in the particles becoming stuck in the corner of the container. The implemented solution is to slightly move the particles in the direction of their velocity so that the distance between them is not 0.

Description	
Simulation: collideParticle(Particle p, Particle q) - Checks if there has been a collision between two given particles, p and q. If there has been a collision, update their positions and velocities accordingly.	
Data structures	
Vector	A two-dimensional vector representation with corresponding x and y values.

Pseudocode

```
1  /* Calculate the squared distance between the particles. This
2  is much less costly for collision detection than calculating
3  the actual distance */
4  sqDist := squared distance between particles p and q
5
6  /* If the squared distance is less than double the radius
7  squared (i.e. if the particles are overlapping), then a
8  collision has occurred */
```

```
 9 if sqDist < (2 * Particle.radius) * (2 * Particle.radius):
10
11      v1 := velocity of p
12      v2 := velocity of q
13
14      /* If the particles are exactly on top of each other,
15      move them slightly. This is a special case which prevents
16      the particles' velocities from being set to 0 since the
17      distance between them is 0 */
18      if p and q are exactly on top of each other:
19           move p by 0.1 * v1
20           move q by 0.1 * v2
21           /* Recalculate distance between the particles */
22           sqDist := squared distance between particles p and q
23
24      /* Move the particles away from each other in the normal
25      direction so that they are no longer overlapping, but
26      rather their edges are now touching */
27      unitNormal := ((2 * Particle.radius) - sqrt(sqDist)) / 2
28      dist := vector distance between p and q
29      distUnit := unit vector of dist
30      move p by distUnit * unitNormal
31      move q by distUnit * -unitNormal
32
33      /* Recalculate distance between the particles */
34      dist := vector distance between p and q
35      distUnit := unit vector of dist
36
37      /* Find the normal and tangential components of the
38      particles' velocities (v1 and v2) */
39      distUnitTangent := tangent vector of distUnit
40
41      v1Normal := dot product of distUnit and v1
42      v1Tangent := dot product of distUnitTangent and v1
43
44      v2Normal := dot product of distUnit and v2
45      v2Tangent := dot product of distUnitTangent and v2
46
47      /* Calculate the new normal and tangential velocities.
48      The normal velocities are calculated from a
49      one-dimensional collision between the particles in the
50      normal direction */
51      v1NormalNew := distUnit * v2Normal
52      v1TangentNew := distUnitTangent * v1Tangent
53
54      v2NormalNew := distUnit * v1Normal
55      v2TangentNew := distUnitTangent * v2Tangent
56
57      /* The final velocity vectors are calculated by summing
58      the normal and tangential components for each particle */
```

```
59        v1New := v1NormalNew + v1TangentNew
60        v2New := v2NormalNew + v2TangentNew
61
62        /* Finally, the velocity of the particles is updated */
63        set p's velocity to v1New
64        set q's velocity to v2New
```

6.3 Detecting collisions between a particle and a wall

The *collideWall* method detects if a particle has collided with a wall using a similar overlap check used for detecting collisions between particles. In this case, the checks are one-dimensional, so they are very simple and efficient. For example, when checking if a particle has collided with the left wall, only the x position of the particle must be considered.

Description	
Simulation: collideWall(Particle p) - Checks if a given particle p has collided with a wall. If it has, call collideWallSpeed to update its velocity based on which wall it has collided with.	
Data structures	
container	A class to represent the container in which the particles are held. It stores the width and height of the container, as well as the number of pixels that the right wall has been moved since the last iteration of the simulation.

Pseudocode

```
 1  wallX := width of container
 2  wallY := height of container
 3  r := Particle.radius
 4
 5  /* If the particle's x position (which represents the centre of
 6  the particle) is less than its radius, then the particle has
 7  collided with the left/west wall */
 8  if p.getX() < r:
 9        p.setX(r)
10
11        /* Update the particle's velocity, given that it collided
12        with the west wall */
13        collideWallSpeed(p, Wall.W)
14
15  /* Has the particle collided with the right/east wall? */
16  else if p.getX() > (wallX - r):
17        p.setX(wallX - r)
18        collideWallSpeed(p, Wall.E)
19
20  /* Has the particle collided with the top/north wall? */
21  if p.getY() < r:
```

```
22          p.setY(r)
23          collideWallSpeed(p, Wall.N)
24
25  /* Has the particle collided with the bottom/south wall? */
26  else if p.getY() > (wallY - r):
27          p.setY(wallY - r)
28          collideWallSpeed(p, Wall.S)
```

6.4 Resolving collisions between a particle and a wall

The *collideWallSpeed* method updates the speed of a particle after it has collided with a wall. This method handles a number of cases based on the movement of the right wall and the insulation of the container.

The basic idea of the method is as follows:

1. If the particles are allowed to push the right wall, update the particle's velocity using the one-dimensional collision formula and move the right wall accordingly.
2. If the container is not insulated, move the particle's speed closer to its expected speed for the current wall temperature.
3. If the particle has collided with the right wall while the wall was moving, update the particle's velocity using the one-dimensional collision formula.
4. If the particle has collided with either the top or bottom stationary walls, reverse its y component of velocity.
5. If the particle has collided with either the left or right stationary walls, reverse its x component of velocity.

The *collideWallSpeed* method utilises a number of scaling values to ensure that the particle behaves as we would expect. Since the particles behave differently based on whether the container is insulated or uninsulated, two sets of scaling values were experimentally found to ensure the behaviour of the particles is as accurate as possible.

When a particle collide with a wall there are four cases to consider, with each case needing to deal with whether or not the container is insulated.

The first case is when a particle collides with a stationary wall. In this case, particles in an insulated container collide elastically with the walls. Particles colliding in an uninsulated container gain/lose a fraction of the energy needed to bring them to their expected energy at the wall temperature, thus keeping the average temperature of the particles approximately equal to the wall temperature.

To achieve this, particles slower than the expected speed must gain energy and particles faster than the expected speed must lose energy. Since the slower particles collide less

frequently with the walls than the faster particles, these particles must therefore gain more energy than the faster particles lose to ensure that the temperature of the gas remains constant.

The second case is when a particle collides with the right wall while it is moving inwards. Here, particles gain energy when they collide with the moving wall. In an uninsulated container, the high energy particles lose more energy when colliding with the walls of the container, while the opposite is true for low energy particles. This dissipates the energy introduced by the moving wall, maintaining a constant average particle temperature.

In an insulated container, the energy gained by the particles was tuned to suit the demonstration of the Carnot heat engine cycle. During step 4 of the cycle in the simulation, the average temperature of the particles must increase from 1000 K to 3000 K in a short amount of time, so the particles gain a larger proportion of energy when colliding with a wall.

The third case is when a particle collides with the right wall while it is moving outwards. In contrast to the previous case, particles lose energy during a collision of this nature. Therefore the tuning of the scaling values in this case follow an opposite pattern to before. When the container is uninsulated, high energy particles lose less energy when colliding with the walls of the container so that the average temperature does not fall. Since this case is not part of the heat engine cycle, the scaling values in an insulated container were simply chosen so that the interaction appeared convincing.

Finally, the fourth case is when a particle collides with the right wall while the particles are allowed to push it. Similarly to the second case, the energy lost by the particles in an insulated container was tuned to suit the demonstration of the Carnot heat engine cycle. During step 2 of the cycle in the simulation, the average temperature of the particles must decrease from 3000 K to 1000 K in a short amount of time, so the particles lose a larger proportion of energy when colliding with a wall. The scaling values for the particles in an uninsulated container also chosen according to the heat engine cycle. In step 1 of the cycle, the particles must remain at a constant temperature while pushing the wall outwards. The particles gain a larger proportion of energy to offset the energy they lose by pushing the right wall.

Ideally most of the scaling values would not be necessary. However, they were needed in this case to ensure the particles behave as we would expect, likely due to the speed at which the right wall automatically moves in comparison to the speed of the particles. In the simulation, the movement of the wall causes huge changes to the energies of the particles, so scaling values are put in place to try to control how much energy is gained or lost. In reality the right wall would be moving several orders of magnitude slower than the particles themselves, so the amount of energy gained/lost by the particles would be much more insignificant. Therefore the movement of the wall would not have such a significant impact on the speed of the particles, so the scaling values would likely not be required.

Description

Simulation: collideWallSpeed(Particle p, Wall w) - Given a particle p and a wall w, calculate the velocity the particle should have after colliding with the wall.

Pseudocode

```
/* Calculate the energy of the particle before changing its
speed */
prevEnergy := energy of the particle
wallSpeed := distance wall has moved since last iteration

/* If the wall is moving in (wallSpeed < 0) then limit the
size of wallSpeed so that particles do not gain too much
energy when colliding with it. This affects particles in an
insulated container more, so the wall speed is limited more
in this case */
if walls are insulated && wallSpeed < -5:
      wallSpeed := -5
else if walls are not insulated && wallSpeed < -10:
      wallSpeed := -10

/* If particles are allowed to push the right wall */
if particles can push the wall && w = Wall.E
  && the container is not at max size:
      /* Scaling values for the one-dimensional collision
      formula below. These values were selected experimentally
      to allow particles in an uninsulated container to remain
      at roughly constant temperature while pushing the right
      wall, while particles in an insulated container should
      lose a sufficient amount of energy to demonstrate heat
      engines (in particular step 2 of the Carnot heat engine
      cycle) */
      if walls are insulated:
            wallM := 10
            particleM := 1
            factor := 5
      else:
            wallM := 12
            particleM := 25
            factor := 6

      vx := p's x component of velocity

      /* One-dimensional momentum calculation to calculate the
      particle's new velocity (Based on the equation:
      v1 = ((u1 * (m1 - m2)) + (2 * m2 * u2)) / (m1 + m2) ) */
      if wallSpeed < 0:
            newVX := ((vx * (partM - (wallM / fact))) + (2 *
            wallM * wallSpeed)) / (partM + (wallM / fact))
      else:
```

```
45              newVX := (vx * (partM - (wallM / fact))) / (partM +
46              (wallM / fact))
47
48        set p's x component of velocity to newVX
49
50        /* The particle's y component of velocity is halved when
51        the container is insulated so that the particles pushing
52        the right wall can lose enough energy to properly
53        demonstrate step of the Carnot heat engine cycle */
54        if walls are insulated:
55              halve p's y component of velocity
56
57        /* Move the right wall outwards by a number of pixels
58        equal to wallVX */
59        wallVX := 2 * ((2 * vx) / (1 + wallM))
60        push the right wall outwards by wallVX
61
62  /* When the walls are uninsulated and a particle collides
63  with any stationary wall, move its speed closer to the
64  expected speed for the current wall temperature */
65  if walls are not insulated &&
66    (wallSpeed = 0 || w != Wall.E):
67        expected := calculate expected speed of p
68        actual := calculate current speed of p
69        difference := expected - actual
70
71        /* Scale the particle's speed to try to keep the average
72        temperature of the particles approximately equal to the
73        wall temperature. A value of 1 means that the velocity
74        of a particle will be set to its expected velocity */
75        if wallSpeed > 0: /* Wall moving out */
76              scaleSpeedUp := 1
77              scaleSlowDown := 15
78        else if wallSpeed < 0: /* Wall moving in */
79              scaleSpeedUp := 1.75
80              scaleSlowDown := 1.75
81        else: /* Wall stationary */
82              scaleSpeedUp := 1
83              scaleSlowDown := 4.1
84
85        /* Move the particle's speed closer to its expected
86        speed */
87        if difference > 0:
88              ratio := (actual + (difference / scaleSpeedUp)) /
89              actual
90        else:
91              ratio := (actual + (difference / scaleSlowDown)) /
92              actual
93
94        scale p's velocity by ratio
```

```
 95
 96 /* Update the particle's velocity based on which wall it
 97 collided with */
 98 switch w:
 99       case N:
100             reverse p's y component of velocity
101       case E:
102             vx := p's x component of velocity
103
104             /* (Special case) If the wall is moving right and
105             the particle is moving slower than the wall but
106             still collides with it, halve and reverse its x
107             component of velocity */
108             if particles cannot push the wall && wallSpeed > 0
109               && vx < wallSpeed:
110                   halve and reverse p's x component of velocity
111
112             /* If a particle collides with a moving wall */
113             else if particles cannot push the wall
114               && wallSpeed != 0:
115                   /* Similarly to lines 27-34, these scaling
116                   values were chosen to keep the average
117                   temperature of particles in an uninsulated
118                   container approximately constant, while
119                   particles in an insulated container need to
120                   gain enough energy to properly demonstrate
121                   step 4 of the Carnot heat engine cycle */
122                   if walls are insulated:
123                         wallM := 2.75
124                         particleM := 1
125                   else:
126                         wallM := 1
127                         particleM := 5
128
129                   /* The particle should behave differently
130                   when the wall is moving in different
131                   directions (Based on the equation:
132                   v1 = ((u1 * (m1 - m2)) + (2 * m2 * u2)) / (m1
133                   + m2) ) */
134                   if wallSpeed > 0:
135                         newVX := ((vx * (1 - 5)) + (2 * 5 *
136                         wallSpeed)) / (1 + 5)
137                   else:
138                         newVX := ((vx * (particleM - wallM)) +
139                         (3 * wallM * wallSpeed)) / (particleM +
140                         wallM)
141
142                   set p's x component of velocity to newVX
143
144             /* If the wall is stationary */
```

```
145             else:
146                   reverse p's x component of velocity
147       case S:
148             reverse p's y component of velocity
149       case W:
150             reverse p's x component of velocity
151
152 /* Calculate and store the heat transfer to/from the particle
153 p. The right/east wall is always considered insulated, so
154 heat transfer is only calculated for collisions with other
155 walls */
156 if w != E:
157       afterEnergy := energy of the particle
158       heatTransfer := afterEnergy - prevEnergy
159       store heatTransfer
```

7 Testing

7.1 Testing strategy

The functionality of the software was tested against the described functionality of the formal requirements (see section 4) to ensure the simulation implemented the required features correctly.

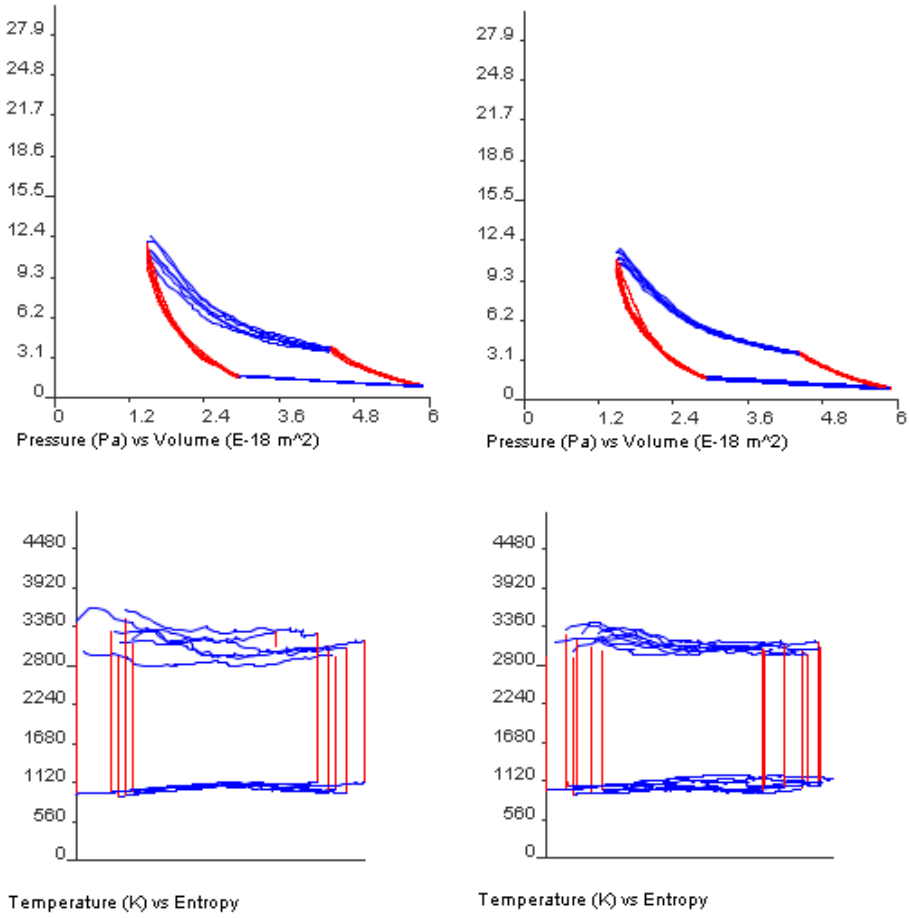
Since a large and experimental part of the software was demonstrating the physics accurately and convincingly, the majority of the testing went into ensuring that this was the case. The expected results of the simulation were well understood due to the background research conducted. The physics of the simulation were therefore tested by comparing the simulation's actual output (for example, behaviour of the particles, shapes of the graphs) with the expected results, and ensuring that the correct results were produced consistently.

By verifying that the macroscopic properties of the gas or shapes of the graphs were correct, then not only have the requirements to demonstrate the processes been met, but the microscopic properties of the gas of the underlying physics of the simulation must also be correct.

7.2 Examples of testing

The component which required the most testing and experimentation was the demonstration of the Carnot heat engine cycle (requirement 4.2.1.8). The goal was to produce pressure-volume (requirement 4.2.1.1) and temperature-entropy (requirement 4.2.1.2) graphs that were as close as possible to those expected. This testing was key in improving the accuracy of the simulated heat engine cycle. Several other requirements had to be met for this demonstration to be implemented. For example, the temperature slider (requirement

4.1.1.2) and the insulation of the walls (requirement 4.2.1.7) are key components of the demonstration.



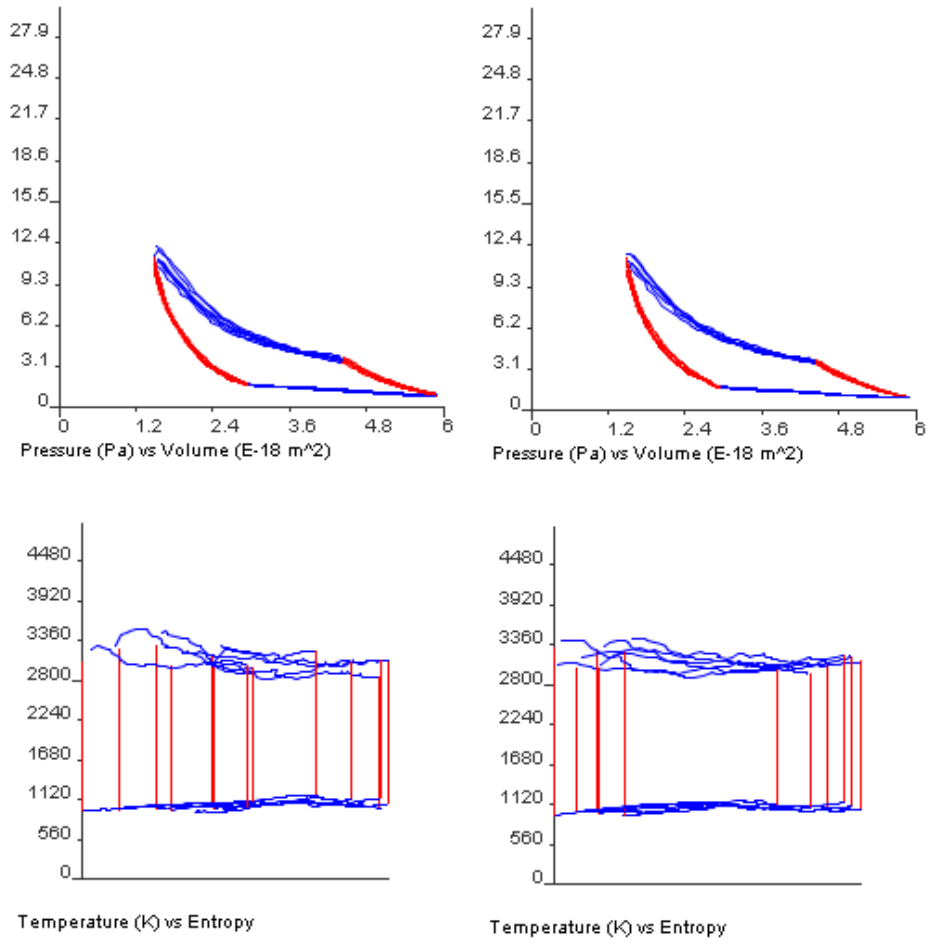


Figure 7.2.1: Four examples of the pressure-volume and temperature-entropy graphs produced by the Carnot heat engine cycle demonstration in the simulation.

Figure 7.2.1 shows four independent tests of simulation's Carnot heat engine cycle demonstration. The pressure-volume graphs are shaped roughly how we would expect in comparison to figure 3.2.3, and the appearance of these graphs is very consistent. The temperature-entropy graphs clearly display the distinct rectangular shape that is characteristic of the Carnot cycle, albeit much less consistently than the pressure-volume graphs.

Demonstrating activation energy required much less experimentation, but the correctness, similarly to heat engines, relies heavily on the underlying physics of the simulation. The graphs displayed in the activation energy mode are the speed and energy distributions, and the Boltzmann factor-reactions per iteration graph.

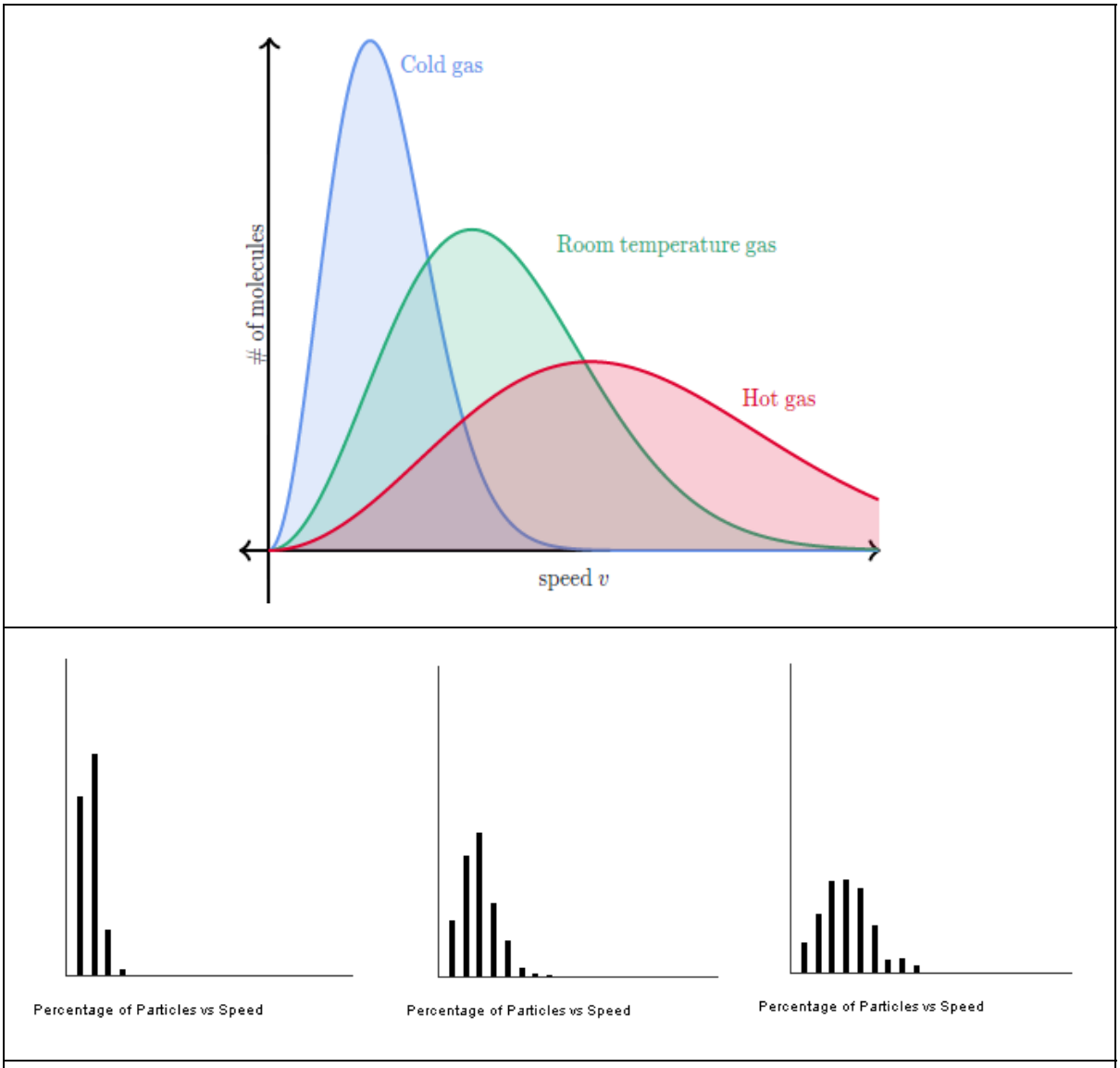


Figure 7.2.2: A comparison of the expected speed (Maxwell-Boltzmann) distribution (top, Khan Academy, 2018) to the distributions produced by the simulation (bottom) at 500 K, 2000 K, and 4000 K respectively.

The speed distributions (requirement 4.1.1.13) produced by the simulation display the expected change in shape as the temperature is increased: As the temperature of the gas increases, the peak of the distribution shifts to the right due to a larger proportion of particles having more energy. In order to maintain the same area under the curve (which represents the total number of particles in the gas), the height of the distribution decreases.

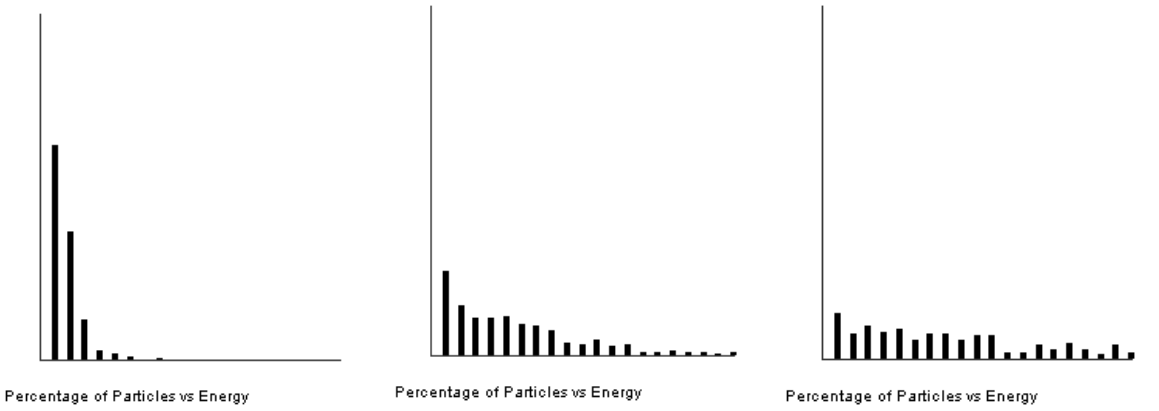


Figure 7.2.3: Energy distributions produced by the simulation at 500 K, 2000K, and 4000 K respectively.

As expected, the energy distributions (requirement 4.1.1.14) produced by the simulation also show that as the temperature of the gas increases, the distribution becomes shorter and wider. The energy distributions are more spread out than the speed distributions as the energy of a particle is proportional to its velocity squared.

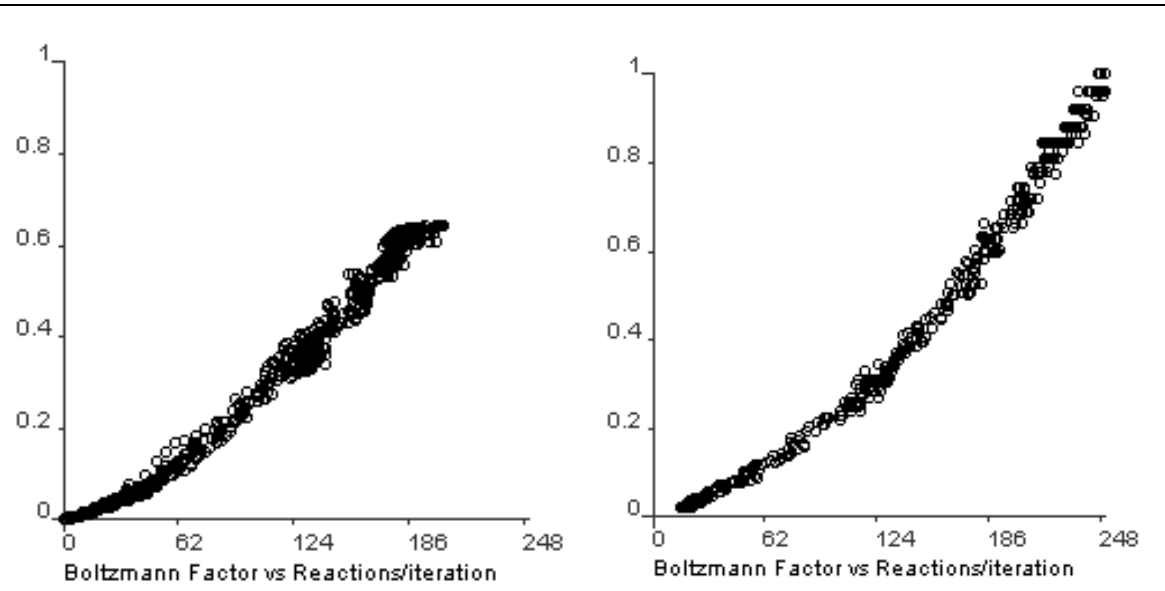


Figure 7.2.4: Two graphs of Boltzmann factor against reactions per iteration. The left graph was produced by varying the temperature of the gas; the right graph was produced by varying the activation energy of the particles.

The Boltzmann factor against reactions per iteration graph (requirement 4.2.1.10) should display a proportional relationship since the rate of a reaction is proportional to the Boltzmann factor. The shape of the graph is consistent whether the temperature of the gas increases or the activation energy decreases. This is because the Boltzmann factor, $e^{-\frac{\varepsilon}{kT}}$,

increases as the temperature of the gas increases, and increases as the activation energy decreases.

The accuracy and consistency of these graphs is a testament to the underlying physics of the simulation. Because the macroscopic properties of the graphs are correct, the physics of the simulation, i.e. the microscopic properties of the gas, must also be correct.

8 User interface

A good user interface is vital for a teaching tool. A study by the University of Colorado (Adams *et al*, 2008), the creators of the PhET Gas Properties simulation, found through interviews that “the simulation must function intuitively or the student's attention is focused on how to use the simulation rather than on the topic presented.” This is why the PhET simulations feature intuitive interfaces which allow the user to instantly begin experimenting and learning.

To reflect the intuitive design of the Gas Properties simulation, Nielsen’s ten usability heuristics (Nielsen, 1995) were used to guide the interface design of the software. These heuristics are all about creating a simple and easy to learn interface, much like that of the PhET simulations.

The first principle, “visibility of system status”, is all about providing feedback to the user so that they remain informed about what is going on. This is a key design principle of the PhET simulations, as it should be for all teaching tools of an experimental nature. Often the first thing that users will do upon seeing a simulation for the first time is to experiment with each of the components of the user interface to gain an understanding of their effects. It is therefore vital that the major components of the simulation give sufficient feedback to aid the user’s understanding of the software. This is evident in the sliders and buttons in the simulation, which provide immediate feedback about the effect of the changes that the user made.

To improve the usability of the simulation, an attempt was made to keep the user interface simple and consistent. Components are labeled using simple language that the user understands, and related components are kept together so that the user can quickly grasp how to use the program.

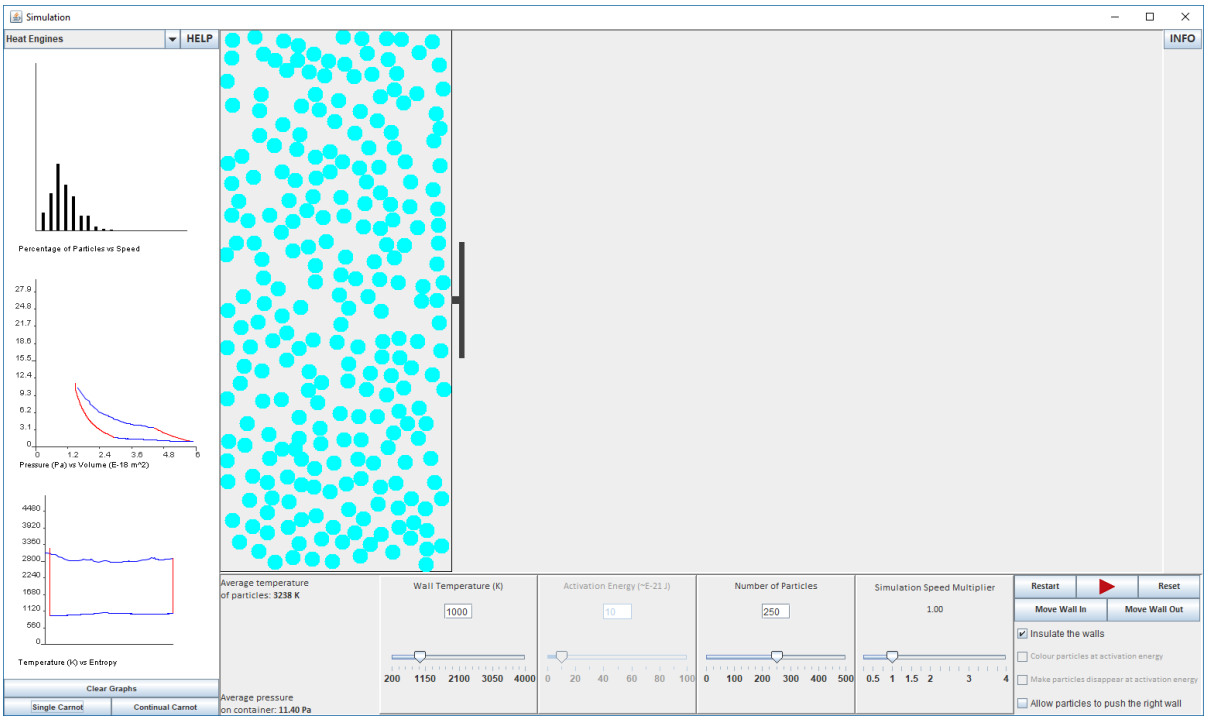


Figure 8.1: A screenshot of the final version of the software. Macroscopic properties of the gas are displayed on the left, and the simulation is manipulated using the control bar at the bottom.

According to Nielsen's tenth principle, it is better if the system can be used without documentation, although sometimes it is necessary to supply such documentation. The simulation's user interface can be quite daunting at first due to its numerous components, each of which require a solid understanding in order to effectively experiment in the program. This is why the "INFO" button in the top right of the software is included. This provides basic documentation on all of the components in the simulation and describes how to use them, enabling the user to become confident in their use and experimentation of the simulation.

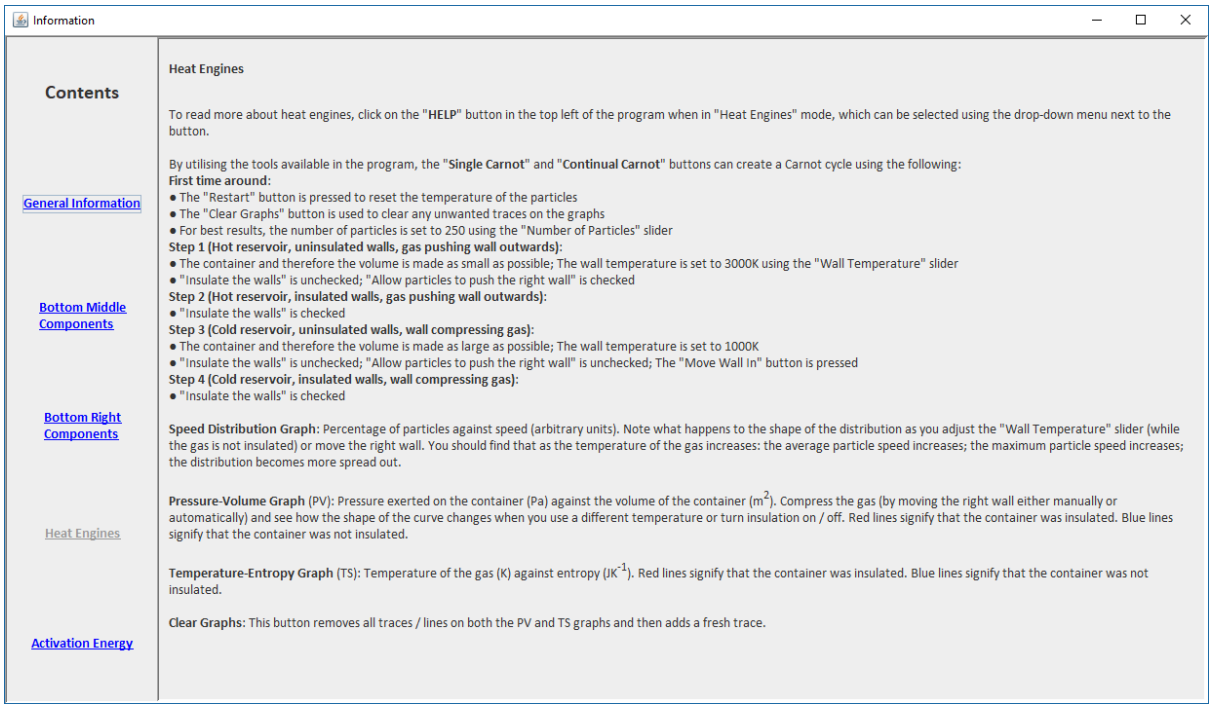


Figure 8.2: The information screen to provide the user with instructions on how to use and get the most out of the program.

The sixth principle, “recognition rather than recall”, is handled particularly well in the PhET simulation. Many of the important functions of the user interface are incorporated into the apparatus of the simulation. For example, the user can adjust the temperature of the particles by adding or removing heat using the stove. The image of the stove is something that the user would instantly recognise as a way to control the temperature. In comparison, the temperature in the project’s simulation is controlled using a slider, which may not provide sufficient feedback upon use. The slider may also blend in to the sliders that surround it, making the user work harder to effectively navigate the tools in the simulation.

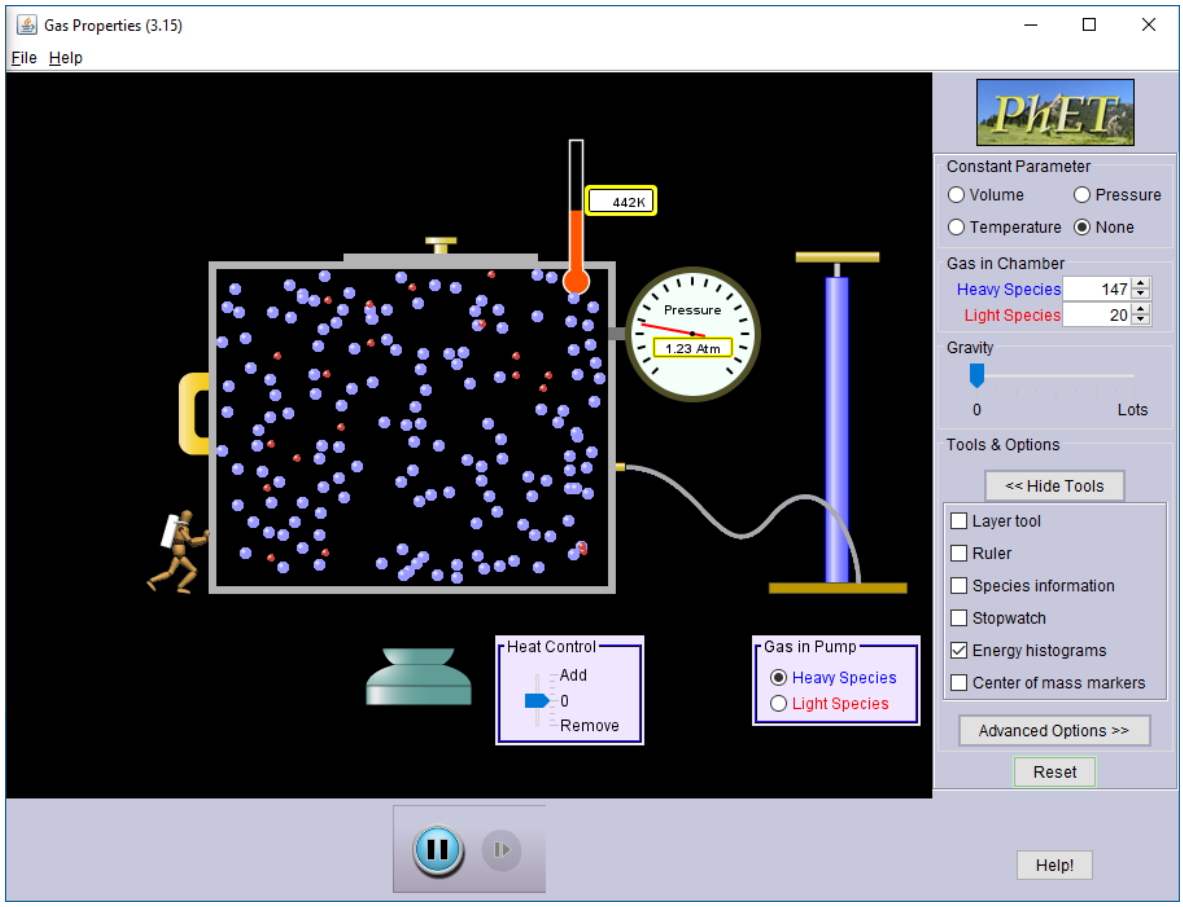


Figure 8.3: The PhET Gas Properties simulation (PhET Interactive Simulations, 2018).

The user interface is a component that can always be refined. The current interface of the simulation is far from perfect, so it would be necessary to work extensively with teachers and students to uncover what they need for the interface to match their expectations.

9 Project management

The development of the software for this project was split into two phases: creating the particle simulation; and exploring the physics behind heat engines and activation energy. During the first phase, the initial simulation was created along with the mechanisms to allow the user to control the simulation, for example the wall temperature and number of particles. The second phase of development implemented the functionality used to demonstrate heat engines and activation energy. Both the simulation and user interface were also refined during this phase.

9.1 Creating the particle simulation

The waterfall model was adopted to create the particle simulation. In this model, development is separated into five distinct phases: requirements; design; implementation;

testing; and maintenance. These phases are executed sequentially to produce a finished piece of software upon completion of the model.

The waterfall model is ideal when requirements are well understood and changes during development will be limited. This was the case for the particle simulation where only the basic functionality required to display the broad microscopic and macroscopic properties of the simulation were needed to begin the second experimental phase.

In hindsight, the model used for this phase of development was a very good fit. Since the aims and requirements of the simulation were largely known beforehand, the phases of the waterfall model naturally fell into place when development began. Requirements could be implemented one at a time until the simulation had its full intended functionality.

9.2 Exploring heat engines and activation energy

Since this phase of development required a more experimental approach, iterative development was used. Here, development is broken down into increments, each involving all stages of development but only delivering part of the required functionality. This allows developers to take advantage of what was learned during development of earlier increments, as well as enabling them to experiment with ideas to see what works best.

Experimentation was a large part of this phase of development. Requirements were not well understood unlike the previous phase of development and many ideas had to be tested. This testing helped to further understand the requirements and refine the functionality of the program over time.

The user interface is an example of a component which relied heavily on iterative development. The usability and learnability of the program are aspects which require lots of user testing and there is always room for improvement. The initial interface created from the first phase of development required much testing and experimentation to evolve into what it became at the end of the project, and with more testing and experimentation, more improvements would follow.

The physics of the simulation also required much experimentation. It took many iterations to find the ideal parameters to most accurately demonstrate the behaviour of ideal gases and in particular heat engines. This was supported by the flexibility of iterative development, which enabled these parameters in the simulation to be revisited after any impactful changes were made to other components of the software.

10 Results and evaluation

10.1 Overview of the completed software

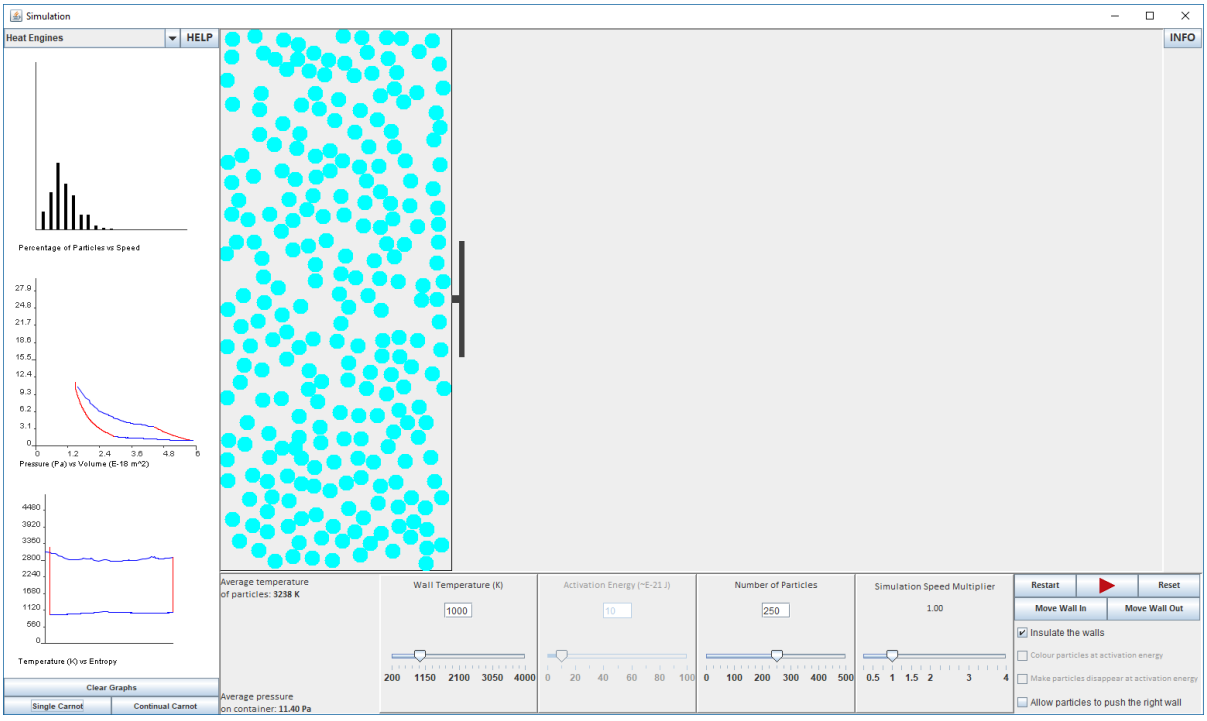


Figure 10.1.1: A screenshot of the final version of the software.

The above screenshot shows the final version of the software in the heat engines mode. Particles collide with the walls and each other in the container located in the centre of the software. The handle attached to the right wall allows the user to move this wall. The parameters of the simulation can be altered using the control bar at the bottom of the software. Finally, the graphs on the left of the simulation display some of the macroscopic properties of the gas being simulated.

In terms of requirements, all requirements from section 4 that are labeled “Must” and “Should” have been implemented, as well as a few “Could” requirements. In this respect the project should be considered a success.

Regarding the overall performance of the simulation, the performance under normal circumstances (default parameters, e.g. 250 particles, standard simulation speed) is reasonable and meets the specified performance of requirement 4.1.2.2. However, the simulation becomes taxing when the number of particles is large and/or the simulation is sped up, although this is to be expected. On some older and slower machines, this can make the simulation almost unusable. It is clear that optimisations can be made to improve

the performance of the simulation, despite its current performance meeting the minimum expectations.

The reliability of the software also meets the specified reliability of requirement 4.1.2.5. Efficient and correct usage exception handling has proved successful in removing fatal run-time bugs. Extensive testing revealed no occurrences of bugs or errors leading to program crashes.

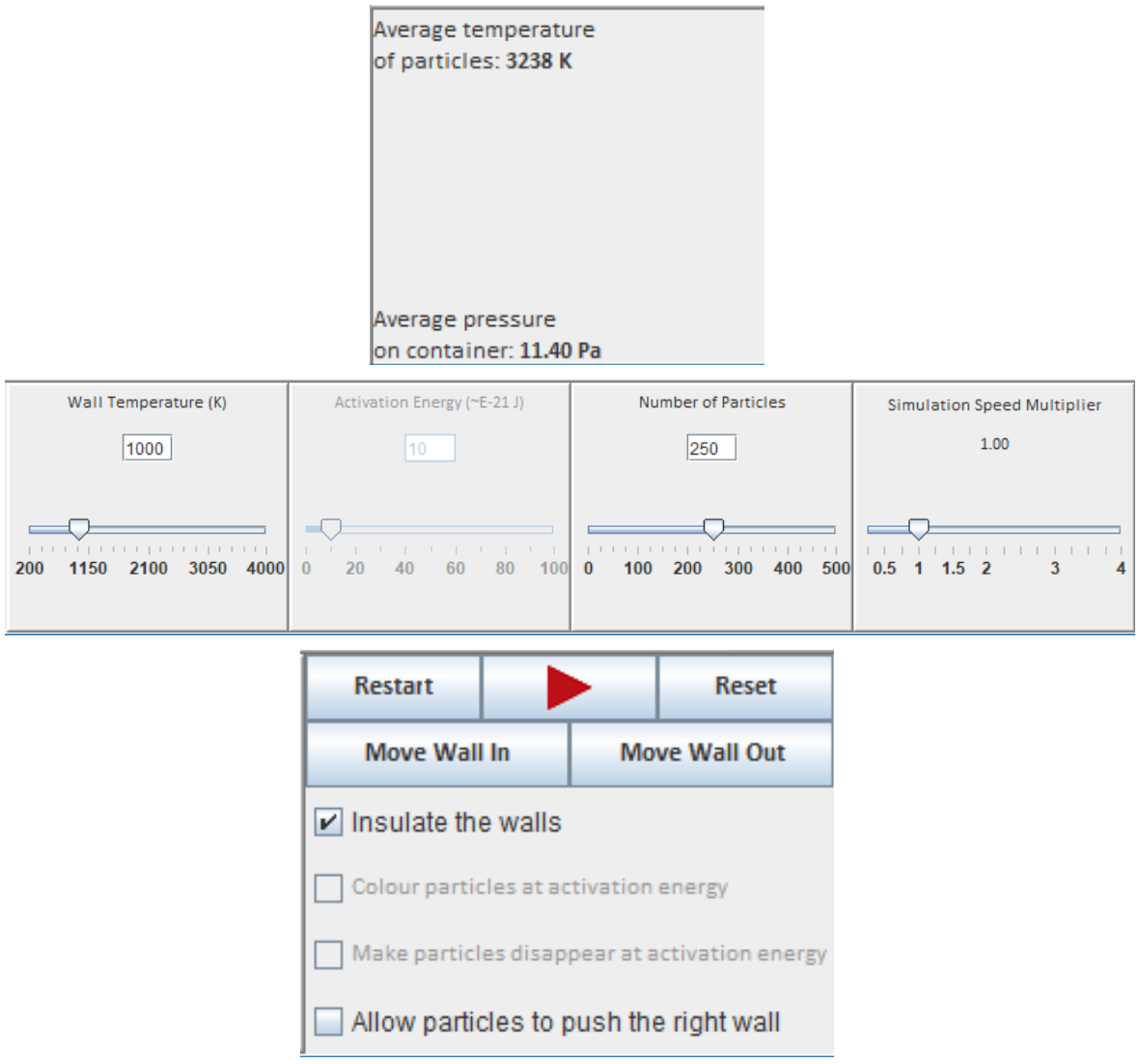


Figure 10.1.2: Enlarged screenshots of the bottom control bar.

The bottom control bar provides the tools and options to enable effective exploration in the simulation. While the primary function of these controls is to aid the demonstration of heat engines and activation energy, they can also be used to investigate the general behaviour of ideal gases.

The control bar is split into three sets of components: the temperature and pressure labels; the sliders; and the controls and options. The temperature and pressure labels display the

temperature of the particles and the pressure exerted by the particles on the container, averaged over the last ten iterations (one sixth of a second at the default simulation speed). These are the most basic comparators of the microscopic and macroscopic properties of the gas.

The sliders allow the user to adjust the variable parameters of the simulation. These include the wall temperature, activation energy, number of particles, and simulation speed. Changing the value of each of these sliders produces immediate results so that the user receives instantaneous feedback about the effect of the changes that were made. The values of the wall temperature, activation energy, and number of particles sliders can also be adjusted by inputting a positive integer into the text fields above the sliders, then pressing the ‘enter’ key.

The additional controls and options on the right of the control bar provide tools for controlling and manipulating the simulation. The “Move Wall In” and “Move Wall Out” buttons move the right wall automatically at a slow and steady speed. This is helpful when trying to produce smooth graphs as it is very difficult to manually move the wall at a constant rate. The checkboxes control how the particles interact with the container in addition to how they act when reaching the specified activation energy.

While in heat engines mode the components relating to activation energy are disabled. This is to separate out the two modes so that the user is not overloaded with information.

Almost all input is received through buttons and sliders, so validation is usually not required since the user is unable to provide erroneous inputs. The values of the Wall Temperature, Activation Energy, and Number of Particles sliders can be manually entered. This input is well sanitised and the user is unable to enter erroneous data, as the text fields only allow positive integers in the correct range to be entered. The software is therefore extremely robust.

In terms of design, despite being feature-rich the control bar is densely packed with components which do not individually stand out. This makes it more difficult for the user to find a particular component at a glance.

The drop-down menu above the graphs allows the user to switch between heat engines and activation energy modes. The help button next to it opens a window offering background information to aid the user's understanding of heat engines (see figure 10.1.4).

In this mode there are three graphs: a histogram showing the distribution of the speeds of the particles; a pressure (y-axis) against volume (x-axis) graph; and a temperature (y-axis) against entropy (x-axis) graph.

The pressure-volume and temperature-entropy graphs are primarily used to demonstrate the Carnot heat engine cycle. This can be done by using the "Single Carnot" or "Continual Carnot" buttons, which create either one cycle or as many as the user would like.

The simulation is typically very reliable and the results produced are consistent. However, when demonstrating the heat engine cycle, the physics of the simulation are tuned to produce consistent results at the standard simulation speed and 250 particles. Although the simulation adjusts to these numbers at the start of the cycle, the accuracy of the results is drastically reduced if the user changes any parameters during the cycle, including the simulation speed.

See section 7.2 for more information on the accuracy and consistency of each of the graphs.

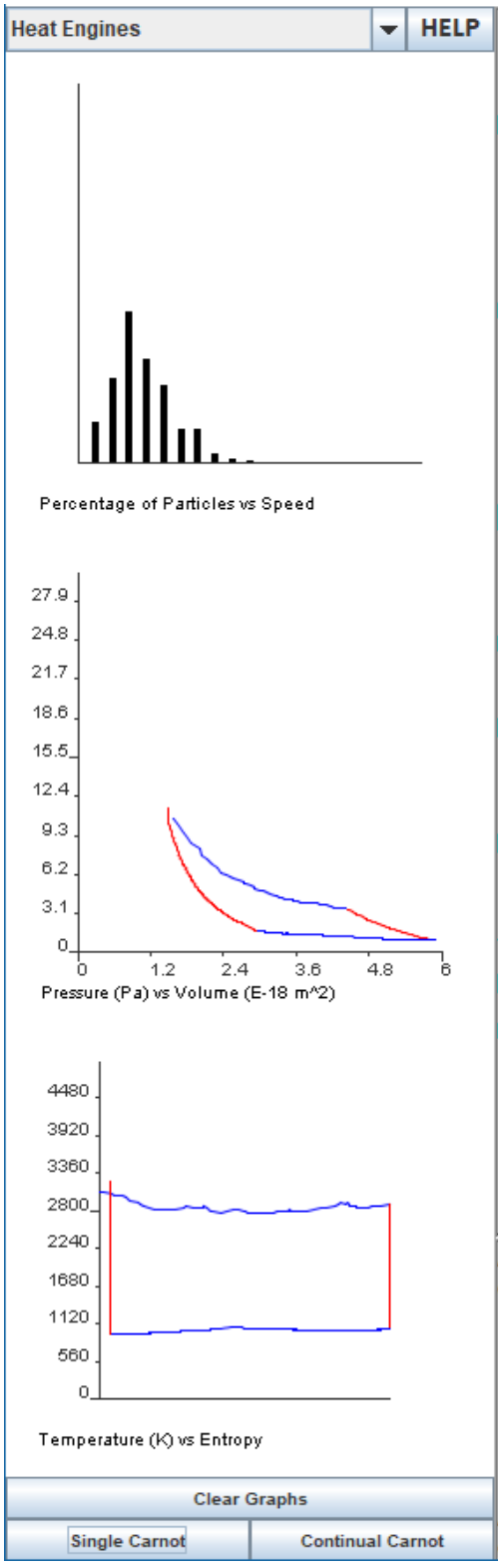


Figure 10.1.3: The left bar for the heat engines mode of the software.

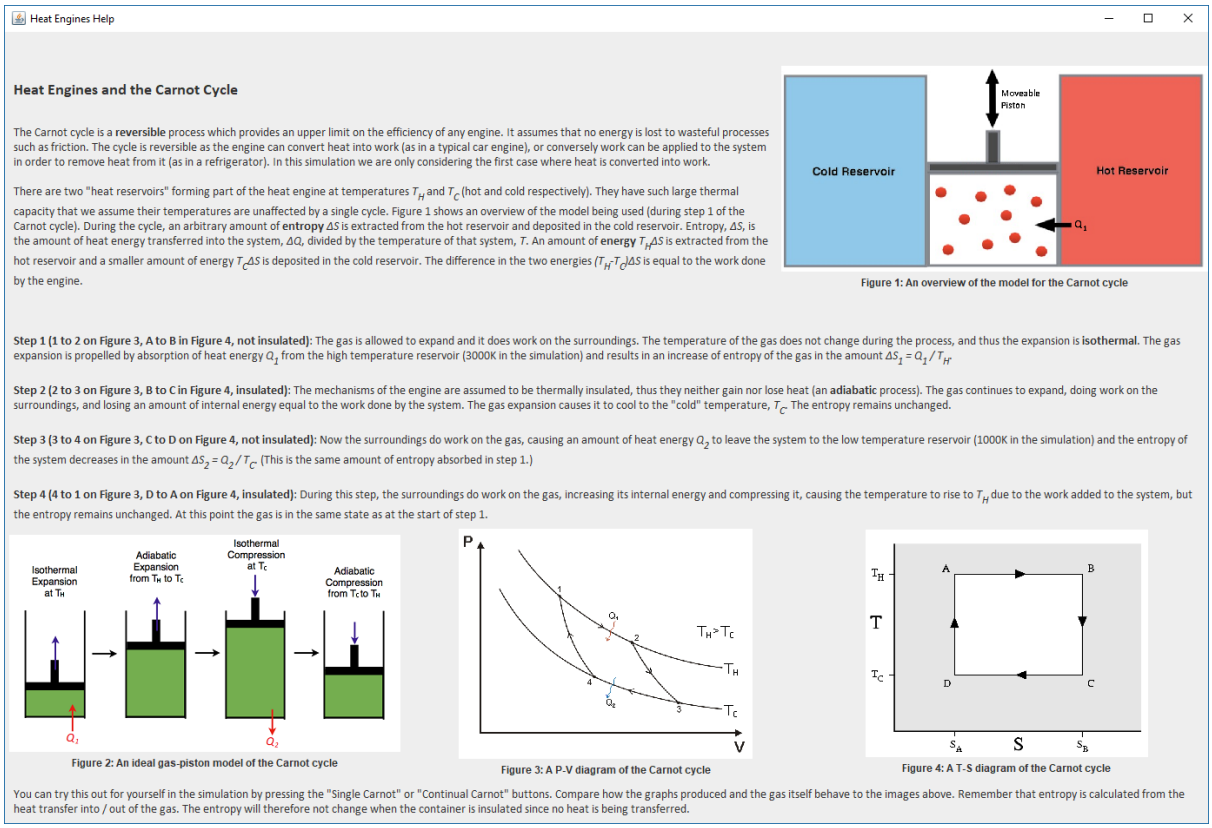


Figure 10.1.4: The help screen to provide some background information of the Carnot cycle and heat engines to the user.

The help screen above describes and explains the basics of heat engines and the Carnot cycle. This highlights the crucial information that a user needs to understand to experiment with the simulation in confidence. The diagrams included specify the model of the Carnot cycle as well as relate the description of the cycle to a more realistic representation and to the simulation itself.

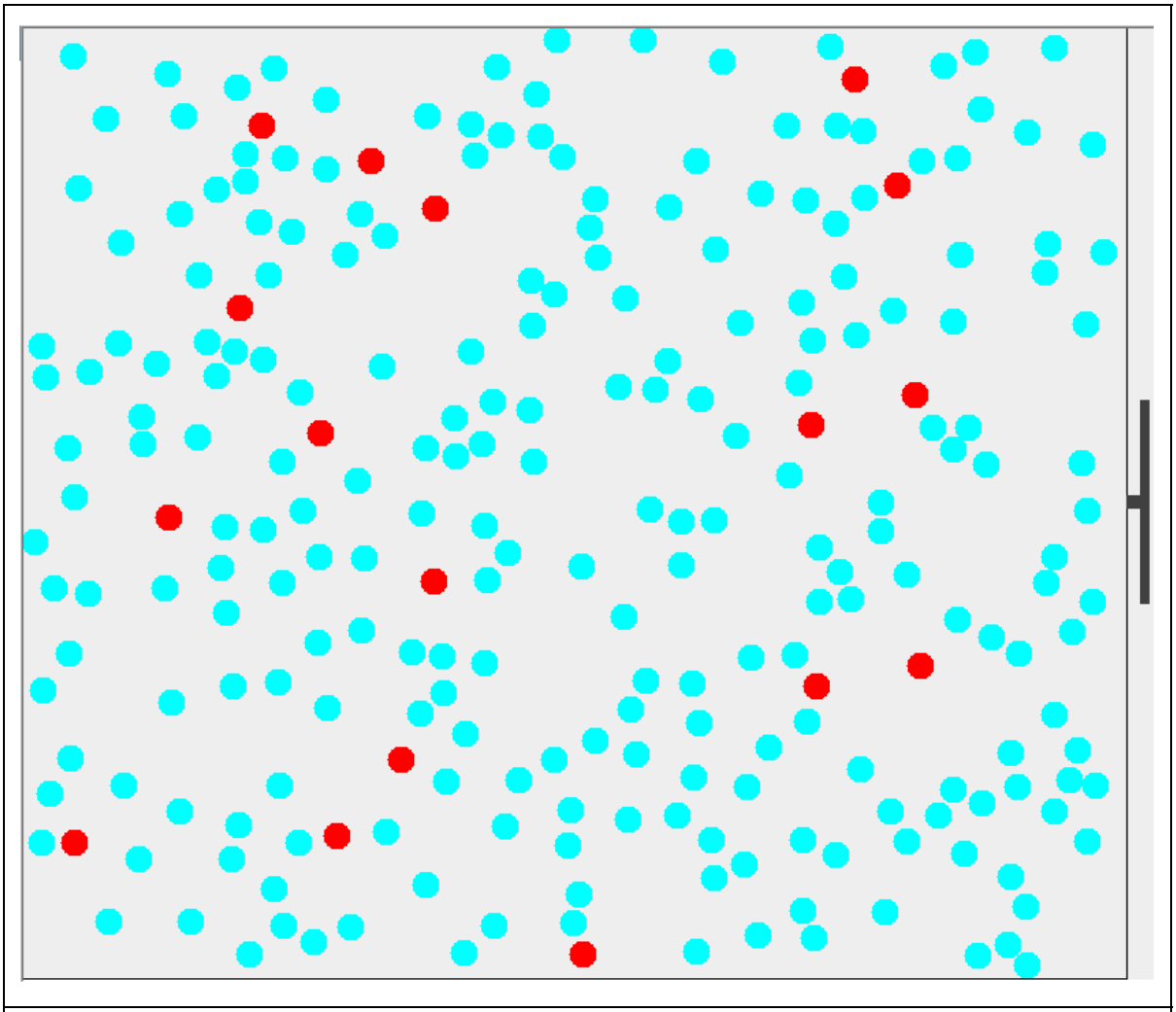


Figure 10.1.5: Screenshot of the container and the particles inside it while in activation energy mode.

When in activation energy mode, particles are by default coloured red when they have energies greater than the activation energy. The strong contrast of colour makes it easy for the user to quickly determine the proportion of particles above the specified activation energy. If the particles were set to disappear upon reaching activation energy, the highlighted particles would disappear.

Just as in the heat engines mode, the drop-down menu above the graphs allows the user to switch between the two modes. The help button next to it now opens a window offering background information to aid the user's understanding of activation energy (see figure 10.1.7).

In the activation energy mode there are also three graphs: a histogram showing the distribution of the speeds of the particles; a histogram showing the distribution of the energies of the particles; and a graph of Boltzmann factor (y-axis) against reactions per iteration (x-axis).

The energy histogram is included to observe the comparison between the distribution of the particles' energies and their speeds. The bars that are highlighted red represent particles that have energies greater than the activation energy.

The demonstration of activation energy shows much less variation than that of heat engines. When adjusting the temperature of the particles or the activation energy quickly, outliers can be produced because the number of reactions must be recalculated using data from the simulation, while the Boltzmann factor is updated immediately. Students using the simulation should be familiar with outliers and understand that they can be disregarded to focus on the major trend in the graph, and as such these outliers are not a problem.

Overall, the macroscopic properties of activation energy are accurately displayed very reliably, and thus by the original specification this demonstration has been successfully implemented.

See section 7.2 for more information on the accuracy of these results.

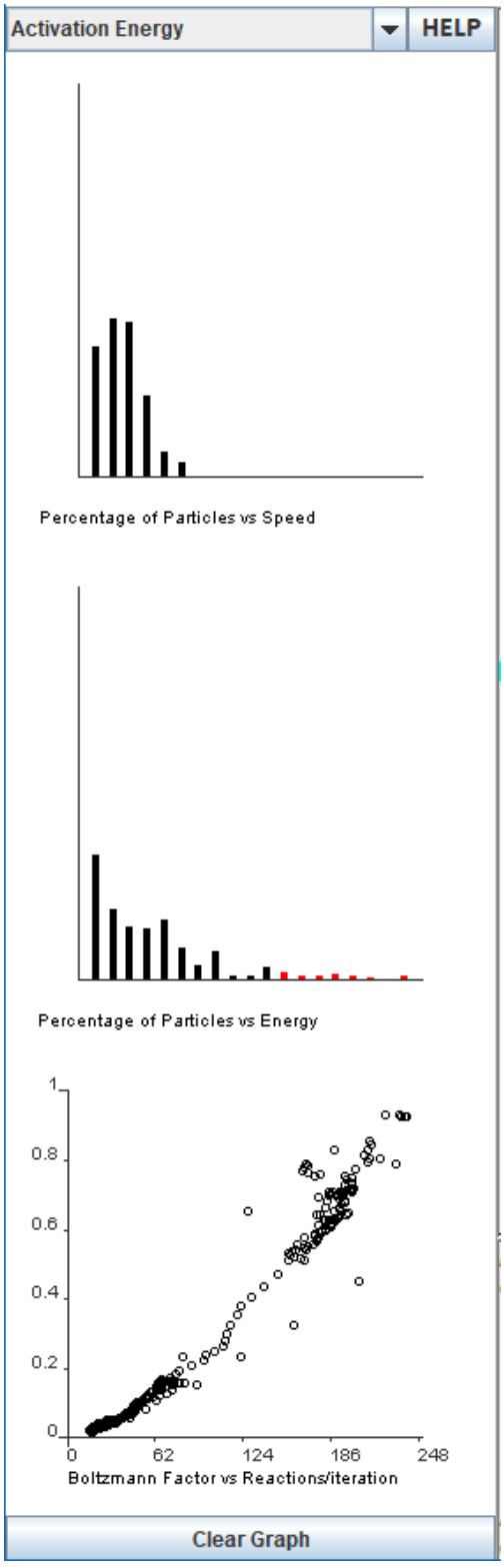


Figure 10.1.6: The left bar for the activation energy mode of the software.

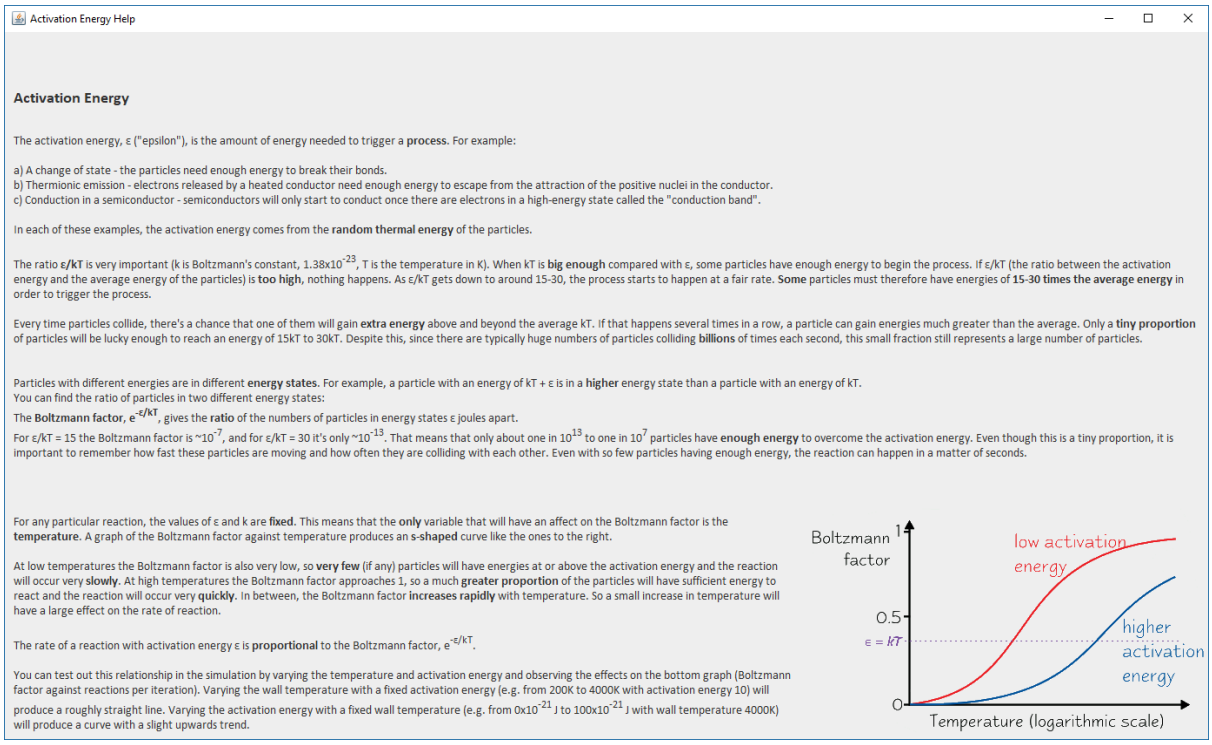


Figure 10.1.7: The help screen to provide some background information of activation energy to the user.

The help screen above describes and explains the basics of activation energy. Since the basic concept of activation energy is relatively simple, this help screen goes through some examples of processes that are triggered when particles reach the activation energy, in addition to providing an understanding of how the activation energy relates to the Boltzmann factor. This helps the user to understand the significance of the Boltzmann factor-reactions per iteration graph.

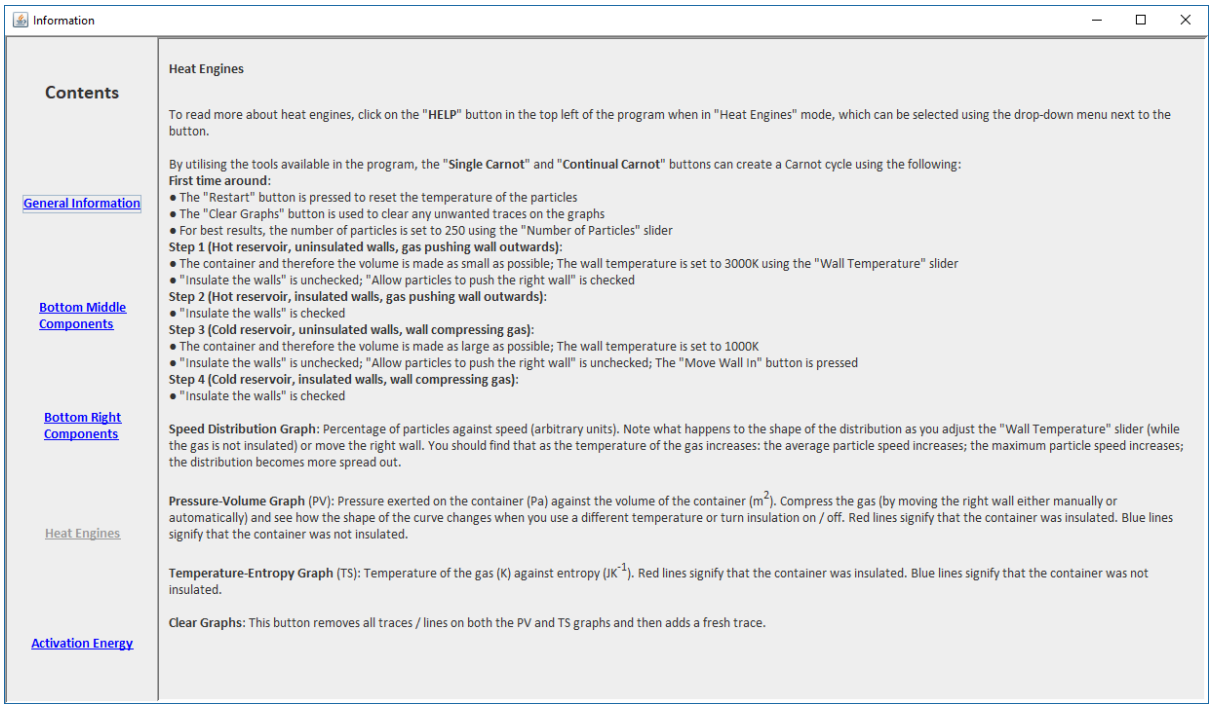


Figure 10.1.8: The information screen to provide the user with instructions on how to use and get the most out of the program.

To quickly find out about a component the user can hold their mouse cursor over it and a tooltip will pop up. To find more detailed information and some explanations of how best to use the software, an additional information screen is provided when pressing the “INFO” button in the top right of the simulation. The data here is organised simply and logically so that the user can rapidly locate the information that they are looking for.

10.2 Evaluation by potential users

Having potential users evaluate the software is hugely important for improving the usability of the software, and even more important for maximising the learning potential available.

10.2.1 The System Usability Scale (SUS)

The System Usability Scale (SUS) (Brooke, 1996) was used to evaluate the usability of the software because it is “a reliable, low-cost usability scale that can be used for global assessments of systems usability. “ The SUS can be easily used to measure usability through a ten-question questionnaire, and the results produced are reliable even with a small sample size. Above all else, the SUS is an industry standard which has been proven to yield valid results, differentiating effectively between usable and unusable systems.

The results of a survey can be interpreted as a score in the range of 0 to 100, although this score is not a percentage and should only be considered in terms of its percentile ranking. The average score according to this ranking is 68, thus a score below 68 reveals that there are some serious usability problems that need to be resolved.

10.2.2 Questionnaire usability results

The questionnaire was given to eight potential users and produced an average score of 62, indicating that the software has poor usability and many improvements must be made. The results showed quite a bit of variance, but one of the benefits of the SUS is that it produces reliable results even with a small sample size, so the results of this questionnaire are likely representative of the larger potential audience.

Despite the majority of users finding the usability of the software adequate, some users seemed overwhelmed by the user interface and were therefore not confident using the software. When users focus their attention on how to use the software, they put much less of their attention on actually learning from it, so this aspect of usability is crucial to the overall effectiveness of the teaching tool.

The average score of the software's usability was hurt by the first question of the SUS: "I think that I would like to use this software frequently." Users mostly disagreed with this statement, which is understandable since this software is not designed to be used regularly. Perhaps the response to this question would be different if the questionnaire was only given to teachers.

Overall it is clear that more work needs to be done and more feedback gathered to improve the usability of the software. This work must be done with close communication with both students and teachers to design it for their needs. Furthermore, a more practical approach must also be taken to gain a better understanding of why the users are not satisfied with the usability. This can be done by observing users perform pre-determined scenarios and tasks to understand how they are using the simulation. Measurements such as the numbers of errors and the time to complete each task can be used to objectively gauge the effectiveness of the user interface.

10.2.3 Questionnaire learning potential results

Another section was also included in the questionnaire with questions regarding the learning potential of the software, along with some miscellaneous questions about their overall experiences with the software. These were not industry standard questions and therefore their results do not have the reliability of the SUS results. However, they do give an indication of the overall impression that the users had on the learning potential of the software.

Most users stated that they learned quite a lot from the software, and that the software improved their understanding of heat engines and activation energy. This suggests that the software is useful for its intended use as a teaching tool. Just as for the software's usability, there is still room for improvements and more feedback must be gathered to ensure that the information offered in the software is clear and useful.

Users also gave very positive responses for the smoothness of the simulation and the accuracy of the physics. One of the goals of the project was to create accurate

demonstrations of heat engines and activation energy, and these positive responses suggest that this part of the project has been a success.

Finally, the users were invited to leave general comments at the end of the questionnaire. A few users expressed confusion about the user interface of the software, as well as some of the explanations of the physics, for instance the explanation of entropy.

One user commented on the versatility of the software:

> *"I believe this tool would serve as a great teaching tool within a classroom environment, as the tool has enough depth and features where a series of lessons could easily be planned around using the simulation with different parameters for various investigations/experiments."*

Although the primary use of the simulation is to demonstrate heat engines and activation energy, the various tools available in the software make it versatile enough to support investigations of other processes, for instance the gas laws for ideal gases. These tools exhibit the potential for the software to be used as a teaching tool.

In summary, while users recognised the potential for learning in the software, the usability of the software was seen as lacklustre. As this project is aimed to be used as a teaching tool, it is imperative that feedback in the future is received from both students and teachers to discover how the software performs in its intended environment. This will provide extremely valuable information which can be used to improve the design and usability of the software.

11 Discussion

11.1 Achievements

The largest achievement of this project is the accurate depiction of heat engines and activation energy. A large portion of the time spent developing the software for this project was spent experimenting with and improving the accuracy and consistency of the physics. Based on user feedback these processes are displayed convincingly in the simulation, and therefore the experimental aspect of the project resulted in a successful demonstration of the physics.

Using this underlying physics, several tools including high quality visual illustrations of the macroscopic properties of the particles are included which form a complete and feature-rich teaching tool. The feedback from potential users confirms that the software is a capable teaching tool with great potential.

11.2 Deficiencies

It is obvious from the results of the questionnaire that the usability of the software needs much improvement. While the simulation includes a number of useful tools, the user

interface does not have the friendly appeal or usability to make it a great teaching tool like the PhET simulation.

Users can also be overloaded by the amount of information in the program. To get the most out of the simulation, the user should be confident in the theory of heat engines and activation energy so that they may investigate these processes to further their understanding. Detailed background information is available in the software, but a teaching tool with better usability would not require the user to read this information before investigating these processes.

The many tools of the simulation are mostly devoted to explaining heat engines. More tools could be added to the simulation to encourage further exploration of activation energy. For example, the simulation has limited options for controlling what happens when a particles reaches the activation energy. A few extra processes could be added to provide further examples of what may occur when a particle reaches the activation energy, such as a demonstration of exothermic reactions (requirement 4.2.1.16).

Finally, the physics of the simulation are tuned to effectively demonstrate the heat engine cycle at the predefined reservoir temperatures and number of particles. Changing these parameters while the demonstration is running drastically affects the accuracy of the results. The simulation should be able to accurately simulate the cycle regardless of the temperature of the reservoirs and the number of particles.

11.3 Additional extensions

The Carnot cycle is a reversible process but the simulation only considers the heat engine cycle. The refrigeration cycle could also be demonstrated (requirement 4.2.1.9) to complete the user’s understanding of the Carnot cycle as a whole.

The software could also be converted into a web application so that the simulation may be more easily accessed and discovered, particularly in an educational environment, with the added benefit of making the software easier to update and improve. Furthermore, general information on how the simulation is used could be gathered to facilitate these improvements.

12 Conclusion

There were two primary aims of the software produced in this project. The first was to demonstrate heat engines and activation energy as accurately and consistently as possible given the available time. The user feedback is evidence that the demonstrations of these two process are in fact adequately accurate with reasonable consistency, resulting in a convincing simulation. The second aim was to create an effective and capable teaching tool. While the software is feature-rich and the potential of the software was recognised by potential users to be used as a versatile teaching tool, it was let down by the disappointed responses to the usability of the software.

The software therefore shows great potential, but the design and usability must be improved by working closely with teachers and students to make it fit for its purpose as a teaching tool.

13 References

Adams, W.K., Reid, S., LeMaster, R., McKagan, S.B., Perkins, K.K., Dubson, M. and Wieman, C.E. (2008). A study of educational simulations part II-interface design. *Journal of Interactive Learning Research, 19*(4), p.551.

Berchek, C. (2009). *2-Dimensional Elastic Collisions without Trigonometry*. Vobarian Software, pp.1-3. Available at: http://www.vobarian.com/collisions/2dcollisions2.pdf (Accessed: 8 April 2018).

BlyumJ (2017). *Carnot Cycle Figure - Step 1.jpg* [Graph]. Available at: https://en.wikipedia.org/wiki/File:Carnot_Cycle_Figure_-_Step_1.jpg (Accessed: 8 April 2018).

Brooke, J. (1996). SUS-A quick and dirty usability scale. *Usability evaluation in industry*, *189*(194), pp.4-7.

CGP Books (2009). *A2-Level Physics OCR B Complete Revision & Practice*. Cumbria: CGP Books, p.44, graph.

Hou, X. (2017). *An ideal gas-piston model of the Carnot cycle* [Graph]. Available at: https://chem.libretexts.org/Core/Physical_and_Theoretical_Chemistry/Thermodynamics/Thermodynamic_Cycles/Carnot_Cycle (Accessed: 8 April 2018).

Keta (2006). *A Carnot cycle illustrated on a PV diagram to illustrate the work done* [Graph]. Available at: https://en.wikipedia.org/wiki/File:Carnot_cycle_p-V_diagram.svg (Accessed: 8 April 2018).

Khan Academy (2018). *Sample of gas (with a constant amount of molecules) at different temperatures* [Graph]. Available at: https://www.khanacademy.org/science/physics/thermodynamics/temp-kinetic-theory-ideal-gas-law/a/what-is-the-maxwell-boltzmann-distribution (Accessed: 8 April 2018).

Nielsen, J. (1995). 10 usability heuristics for user interface design. *Nielsen Norman Group*, *1*(1).

PhET Interactive Simulations (2018). *Gas Properties* (Version 3.15) [Computer program]. Available at: https://phet.colorado.edu/en/simulation/gas-properties (Downloaded: 28 October 2017).

Wikimedia Commons contributors (2017). *A Carnot cycle acting as a heat engine, illustrated on a temperature-entropy diagram* [Graph]. Available at: https://commons.wikimedia.org/wiki/File:CarnotCycle1.png (Accessed: 8 April 2018).

14 Appendices

14.1 Structure of the .zip file

Report.pdf - This report
simulation.jar - The pre-compiled executable JAR file
src/ - Directory containing the source files of the software
- **code/** - Directory containing the source code of the software
  - Container.java
  - ControlPanel.java
  - GraphView.java
  - HelpScreens.java
  - IntFilter.java
  - Particle.java
  - SimBuffer.java
  - SimComponent.java
  - SimModel.java
  - Simulation.java
  - SimulationGUI.java
  - Vector.java
- **resources/** - Directory containing additional resources used in the software
  - jchart2d-3.2.2.jar - Third party library used to draw graphs
  - **helpscreens/** - Directory containing text files for the help screens (“HELP” and “INFO” buttons)
    - ActEnergy1.txt
    - ActEnergy2.txt
    - HeatEngines1.txt
    - HeatEngines2.txt
    - HeatEngines3.txt
    - HelpActEnergy.txt
    - HelpBottomMid.txt
    - HelpBottomRight.txt
    - HelpGeneralInfo.txt
    - HelpHeatEngines.txt
  - **images/** - Directory containing images used in the software
    - BoltzmannTemp.png
    - CarnotCyclePistons.png
    - CarnotCyclePV.png
    - CarnotCycleTS.png
    - Pause.png
    - Play.png

Reservoirs.png

tooltips/ - Directory containing text files for tooltips created when the user hovers their mouse over a component in the software

ActEnergySlider.txt
AddTracesButton.txt
AutoCarnotButton.txt
AutoCarnotConstButton.txt
AvgPressureLabel.txt
AvgTempLabel.txt
BFRChart.txt
BFRClearChart.txt
ColourParticlesCheckbox.txt
EnergyDist.txt
InsulateWallsCheckbox.txt
MoveWallInButton.txt
MoveWallOutButton.txt
NumParticlesSlider.txt
ParticlesDisappearCheckbox.txt
ParticlesPushCheckbox.txt
PauseResumeButton.txt
PVChart.txt
RemoveTracesButton.txt
ResetButton.txt
RestartButton.txt
SimSpeedSlider.txt
SpeedDist.txt
TSChart.txt
WallTempSlider.txt

14.2 How to run the software

14.2.1 UNIX

To execute the executable JAR file:

java -jar simulation.jar

To compile and run the source code in a terminal:

1. Navigate to /src/code/
2. *javac -cp ../resources/jchart2d-3.2.2.jar *.java*
3. *java -cp ../resources/jchart2d-3.2.2.jar:../ code.SimulationGUI*

14.2.2 Windows

To execute the executable JAR file:

java -jar simulation.jar

Or simply double click on *simulation.jar* in File Explorer.

To compile and run the source code in Windows Powershell or Command Prompt:

1. Navigate to /src/code/
2. *javac -cp ../resources/jchart2d-3.2.2.jar *.java*
3. *java -cp "../resources/jchart2d-3.2.2.jar;../" code.SimulationGUI*

14.3 Questionnaire and results

SUS Questionnaire Results		
Question		Normalised score out of 10 (higher is better)
1	I think that I would like to use this software frequently	4.0625
2	I found the software unnecessarily complex	6.25
3	I thought the software was easy to use	6.5625
4	I found that there was not enough support in the software for me to use it effectively	7.1875
5	I found the various functions in this software were well integrated	6.25
6	I thought there was too much inconsistency in this software	7.1875
7	I would imagine that most people would learn to use this software very quickly	6.25
8	I found the software very cumbersome to use	5.625
9	I felt very confident using the software	6.25
10	I needed to learn a lot of things before I could get going with this software	5.9375
Learning Potential of the Software Questionnaire Results		
Question		Normalised score out of 10 (higher is better)
1	I learned a lot from the software	6.875
2	I found it easy to learn using the software	6.5625

3	My understanding of heat engines has improved after using the software	7.1875
4	My understanding of activation energy has improved after using the software	6.875
5	The information provided by the software was clear and useful	6.25
6	I felt convinced by the physics in the simulation	8.4375
7	I was able to keep track of what was happening in the simulation	7.8125
8	I found the simulation ran smoothly	9.0625
9	I encountered many problems while using the software	8.125