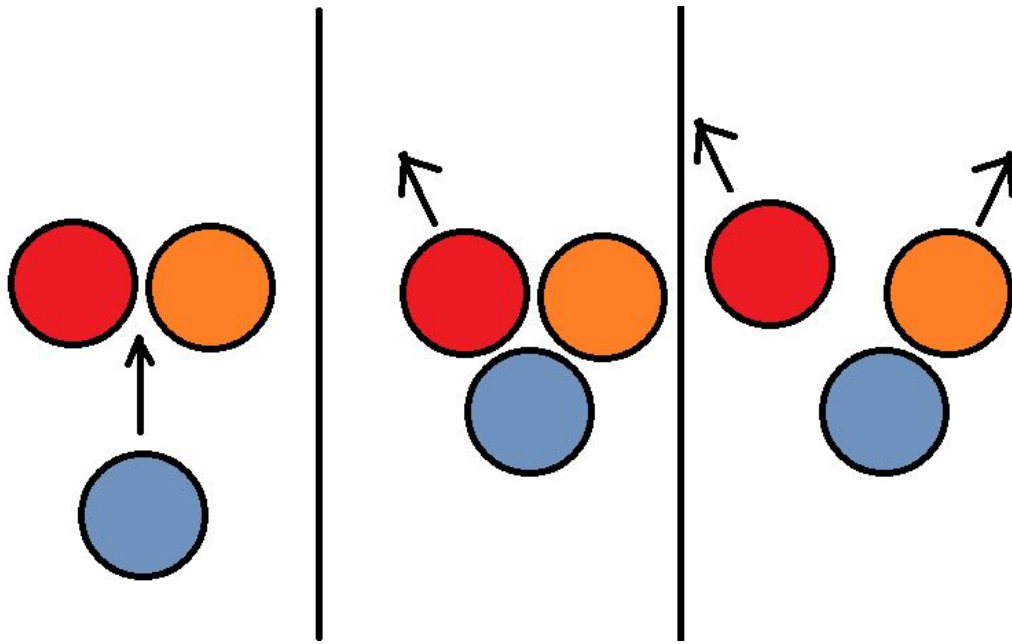


Physics Simulation



Analysis	5
❑ Description of the problem	5
❑ Stakeholders	6
❑ The target platform	7
❑ How is this problem suited for a computational methods	7
❑ Thinking abstractly	8
❑ Thinking Ahead	8
❑ Thinking logically	9
❑ Thinking concurrently	9
❑ Research about similar programs	9
❑ What I like about these examples	11
❑ Hardware and software requirements	16
❑ Communication with Stakeholders	17
Design	19
❑ Structure of the Solution	19
❑ Decomposition	20
❑ User Interface Design	21
❑ Algorithms / Decomposition	24
❑ Particle Class	24
❑ Input and Data Processing	31
❑ Decomposition:	31
❑ Collision and Splitting	36
❑ Table of Variables	38
❑ Global	38
❑ Particle	39
❑ Input	40
❑ Explanation of Modules	41
❑ Test Table	42
❑ Objective 1	42
❑ Objective 2	43
❑ Objective 3	44
❑ Objective 4	45
❑ Further Data	46
Development	47
❑ User Interface Development - Phase 1	47
❑ Basic HTML and Screen Layout	47
❑ Toolbar Menu	48
❑ Canvas	50
❑ User Interface - Phase 1 Review	53
❑ Particles & Rendering - Phase 1	54
❑ Particle Constructor	54
❑ Particle Draw Method	56
❑ Particle Update Method	57
❑ Create Particles Method	58
❑ Particle Boundary Collision	60
❑ Particle - Phase 1 Review	62
❑ Stakeholder feedback - Phase 1	62
❑ Testing to Inform Development - Phase 1	63
❑ User Interface Development (Phase 2)	66
❑ Particle and Window Options	68
❑ User Interface - Phase 2 Review	72
❑ Particles & Rendering Development (Phase 2)	73
❑ Switching between Simulations	73
❑ Particles and Rendering - Phase 2 Review	77
❑ Stakeholder feedback - Phase 2	77

❑ Testing to Inform Development - Phase 2	78
❑ User Interface Development (Final Phase)	82
❑ Canvas Manipulation - Play, Pause & Reset	82
❑ User Interface - Review (Final Phase)	86
❑ Particles & Rendering Development (Final Phase)	87
❑ Addressing Stakeholder Feedback	87
❑ Optimising and Bug Fixing	91
❑ Big O Notation & Efficiency	93
❑ Particles & Rendering - Review (Final Phase)	101
❑ Stakeholder Feedback - (Final Phase)	102
❑ Testing to Inform Development - Final Phase	103
Evaluation	105
❑ Testing to Inform Evaluation	105
❑ Objective 1	105
❑ Objective 2	107
❑ Objective 3	109
❑ Objective 4	111
❑ Evaluation of the Solution	113
❑ Phase 1	113
❑ Phase 2	115
❑ Final Phase	116
❑ Usability Features	118
❑ Limitations in Solution	120
❑ Solutions for Limitations	121
❑ Maintenance	122
❑ Further Development	123
❑ Conclusion	124
Code	125
Bibliography	137

Analysis

Description of the problem

The main problem that I am targeting my solution at is visualising physical interactions between objects, using realistic physics in order to aid with both student learning and teacher teaching.

Most students learn about physics concepts in a traditional manner of reading and writing, however this may not be the most effective way at portraying a concept, making it hard for a physics student to conceptualise the idea. Because of this, some teachers use online simulations for practical events such as momentum, collision, energy loss, etc. However, these tend to be somewhat lacking in features, and may not be applicable to most scenarios that teachers require them in. Additionally, most students will not use these simulations for further research as they tend to lack any 'experimental' features for students to play around with. Moreover, the learning environment has not encouraged the use of visual learning as much compared to reading and writing skills, which may have hindered both the teachers and the students desire to use these simulations. This is a problem because it may make some concepts seem harder to understand than they truly are as the way in which students learn is hindered.

Additionally, the use of creating a virtual sandbox for simulation events, allows for the solution to be portable, so students can have access to the solution anywhere. This can be a problem, especially outside of the classroom, when students may want to access these simulations and resources at any time.

Secondly, teachers may find these simulations to be lacking in features such as customisation, and data representation, in order to show how a concept is applied in realistic scenarios. Additionally, teachers may struggle to set visual based, or interactive homework due to a lack of available software to accomplish this. This is a problem as it does not allow or encourage students to experiment with physics concepts.

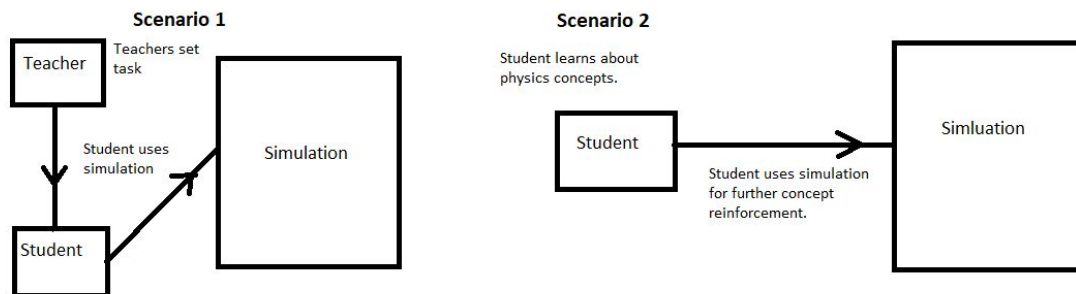
Finally, virtualizing this problem into a computational solution is suitable as having the solution accessible via a computer means that it is portable across a wide set of locations, and is cheap for users to have access as only a computer and access to the files is required. Additionally the use of a simulation means that a visual element is involved. This makes it very suitable for computational methods as many thousands of on screen entities can be rendered and updated every second allowing for it to simulate real life examples, even on lower end machines. Finally, as teachers will use this solution for displaying scenarios it can be used as either a replacement or used in conjunction with practicals and can therefore be accessed by most, if not all, schools.

Stakeholders

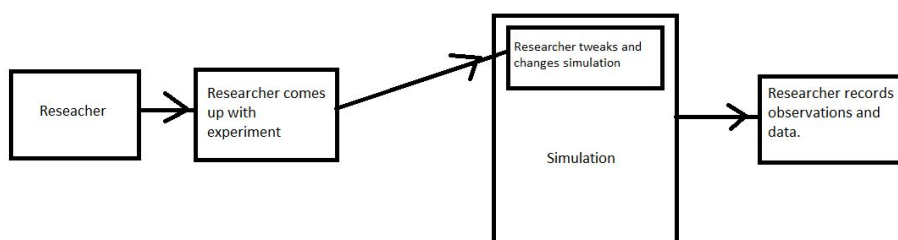
For this problem, the main stakeholders involved are physics students, physics teachers and for professional use.

For physics students, they would use this application for understanding and learning physics concepts such as particle collision, momentum, gravity, etc. This is because the solution allows for students to experiment and interact with a realistic simulation of learnt events. Moreover, physics students will be able to collect and study the data that is collected during a simulation such as how does the momentum of a particle change over the course of a collision, achieved from collecting and representing data in graphs. A physics student would first use this solution by first downloading the files and running them within a browser. Afterwards, the student will then be able to select a simulation scenario to play, forwarding the student to the simulation screen. Then the student can configure and adjust a variety of settings such as the mass of a particle, it's velocity, etc. Next, the student can then start the simulation and see it occur in real time on screen. Finally, the data collected from the simulation is then represented into a graph for the student to interact with.

Like physics students, physics teachers can use this software for personal investigation and for learning. However, physics teachers can use this mainly for setting casual and non traditional forms of homework such as 'experimenting with a particular scenario'. Moreover, physics teachers can use these simulations for further reinforcement of concepts, or as a replacement to physical practicals conducted in class.



The final stakeholder would be that of professional use by academics or researchers. A researcher may use this simulation to experiment with a realistic scenario (such as particle collision) and tweak the variables used (such as gravity) to note how the outcome of the event changes based on how these variables change. This is due to the fact that the simulation does not enforce a set of constants that cannot be changed, so therefore researchers and academics alike can experiment and use the simulation as a sandbox for testing ideas and proposals for how certain scenarios can be altered.



Stakeholder	Description of use
Physics student	A physics student would use this application for both academic and self investigation purposes. Additionally, students would use this to reinforce the knowledge about a particular concept or idea.
Physics teacher	Physics teachers would use this for both personal use and for setting casual homework tasks for physics students. Furthermore, they could use this application for replacing practicals performed in class, or for use in conjunction with them to help students understand an idea.
Researcher	A researcher would use this for personal and work related use, through investigating ideas and experiments. The researchers would experiment with different scenarios and record their findings and changes that occur during the simulation as a way of proving or disproving ideas.

The target platform

This solution will be written via web based languages such as JavaScript and HTML so would therefore require a user to have access to a web browser with suitable processing support for JavaScript. Additionally, the solution is mainly focused on Windows based operating systems such as Windows 7, 8 or 10, but may work on other operating systems such as Linux distributions or Macintosh however testing will not be done on these platforms due to their limited user usage. This solution will not target direct support for mobile users and therefore formatting of the simulation's elements will not be done to be correctly displayed on a mobile phone's display. Finally, this solution is mainly aimed at desktop or laptop users due to their better hardware specifications and easier accessibility for most stakeholders.

How is this problem suited for a computational methods

The problem of simulating lots of objects on screen and having each element act independent of each other makes it suitable for computational methods. The reason for this is because a processor can perform millions to billions of operations per second, meaning it can easily simulate realistic scenarios which contain many hundreds to thousands of pieces of changing data. For example, simulating a 2d collision between 2 particles would be suitable as most current processors would be able to easily update the two particle's behaviours and their characteristics (such as position) 60 times per frame, creating a fluid and somewhat realistic reflection of an actual physics scenario.

Additionally, the problem requires user input and response which can easily be achieved through the use of buttons and having a layer between the simulation and the user. Furthermore, there are a variety of input devices such as keyboards and mice suitable for allowing users to input data into the simulation. Moreover, the use of outputting information to the user is very suited for computational methods as the canvas for which the simulation is displayed on can easily be outputted to a monitor or similar output device.

Thinking abstractly

The main idea of the solution is that it is a physics simulation, that will be used in the browser. This solution will be written in JavaScript, HTML and CSS. The physics simulation will be hosted in a browser to allow for a user interface to be created. Before a simulation the user can tweak and enter data/settings about the simulation to adjust it's outcome.

When the simulation is run the data values entered into specific fields by the user will be processed and stored. These settings will then be applied to the simulation and then the main simulation loop will occur. This simulation loop will run at a set interval, say 60 times a second, in which it will update all entities on screen, performing collision and positional checks along with adjusting the x and y coordinates of the entity. Furthermore, the program will always be waiting for user input for either starting, stopping or resetting the simulation, so the simulation will adjust itself depending on what the user wants the simulation to do.

After the simulation has ended, a graph will appear displaying the lifetime of a chosen variable/entity and how it's value/values have changed over the course of the simulation. This graph will allow users to interact with it, download it, and investigate particular points of the simulation.

Thinking Ahead

For my knowledge based skills, I will need to improve myself in JavaScript, through learning about classes, inheritance and other OOP concepts and how they can be implemented in a program. Additionally, my HTML and CSS syntax knowledge will have to be improved by learning about specific tags for HTML elements, or by fully understanding how CSS style sheets can be linked across a multitude of pages/files.

I will have to do additional research on specific JavaScript libraries or derivations of JavaScript which can help in performing graphical functions inside the simulation or to offload a particular function such as rendering certain shapes and particles to the canvas. Additionally, I will need to research similar programs that accomplish the same output as my solution, and hopefully, read through and understand the source code. This will allow myself to learn new concepts and apply them to my own solution.

Further features could be implemented at a later phase in development, which could include additional customisation features for the simulation such as

allowing the user to change the canvas size, or allow the user to create their own particle with a custom shape/sprite.

Thinking logically

For the physics simulation, it will be split into multiple sections, including rendering, updating, user interface and the simulation. The user interface stage will involve myself creating a user interface with multiple input elements such as buttons, fields and dropdowns. Furthermore, sliders and limiters could be used for allowing the user to more easily adjust a value, but have limiters in place to avoid the user picking erroneous data.

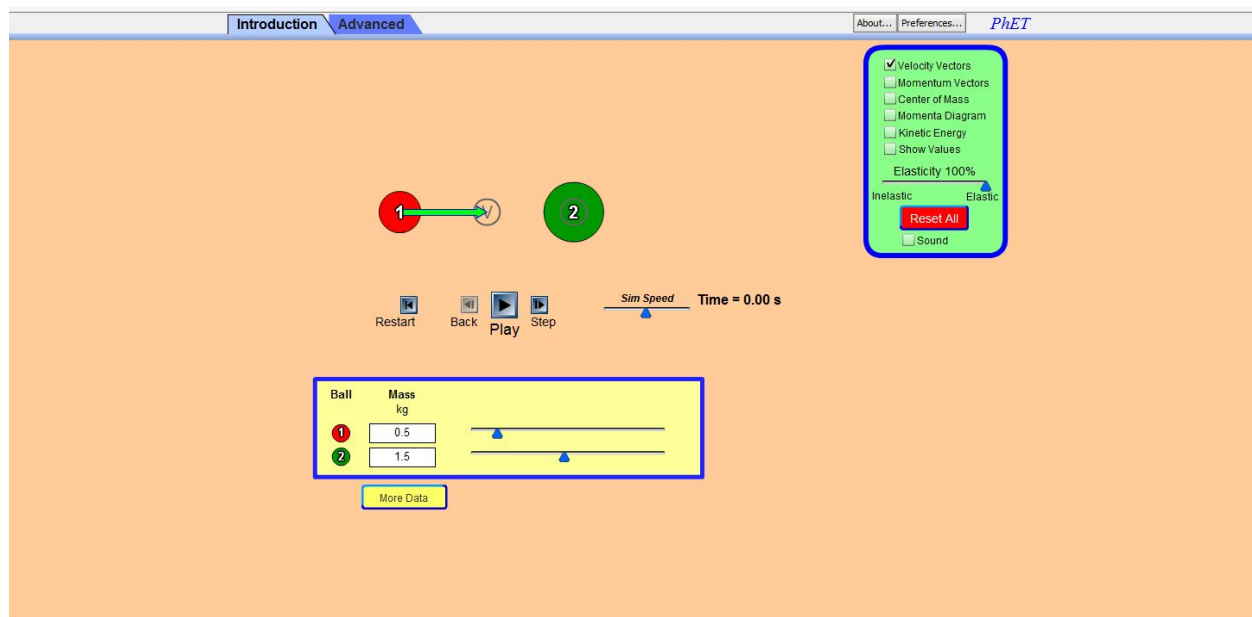
The next section is the actual simulation display. The simulation will be produced via having core files for particles and utility functions such as creating particles. In addition to this, these core files will need to be applicable to multiple different simulation scenarios to reduce redundancy. The core files will require a hierarchy for objects such as particles, so that different object types can be linked to a common class type.

Thinking concurrently

In the physics simulation I can make the user interface along with implementing the settings tab for each scenario. For example, as the user interface for the simulation is being developed, I can also start development on the options that will be available to the user. In addition to this, the main rendering aspect of the physics simulation can be created alongside the development of the behaviour and interaction of particles. This is because the rendering of the particles and how they react towards events are dependent upon each other. However, the rendering could be developed alongside the user interface instead. Other aspects of the project such as testing can be implemented during the end where i will perform a variety of testing methods on data.

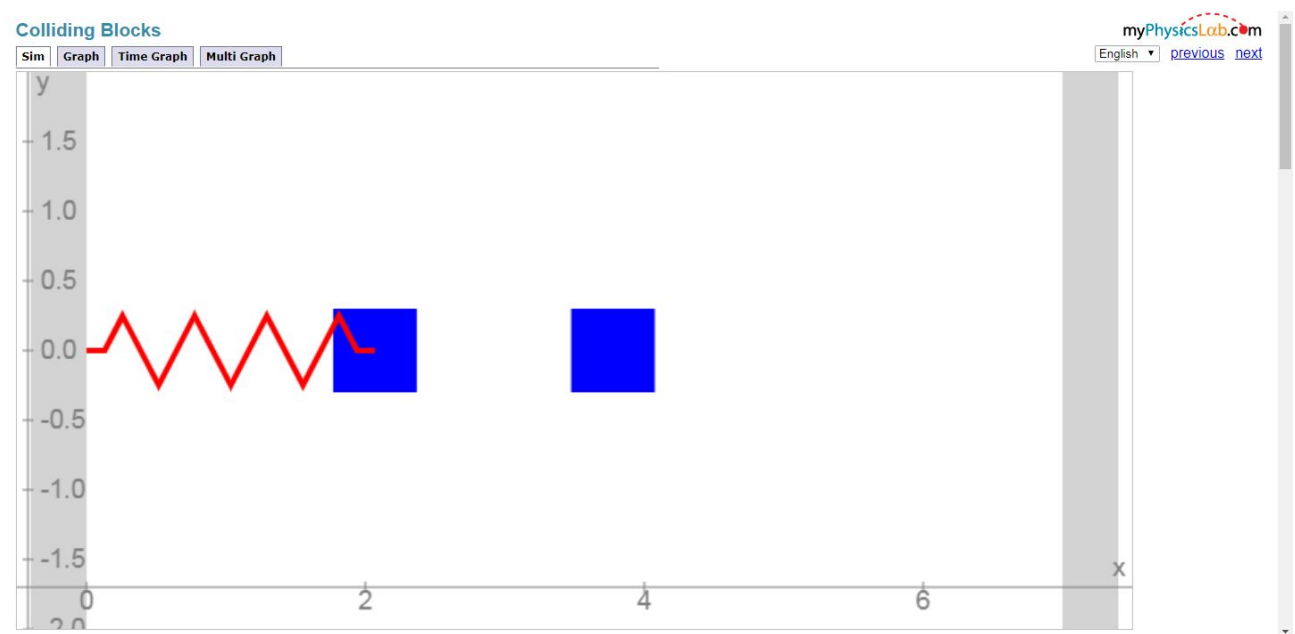
Research about similar programs

There are many similar programs such as 'phET simulations' which contain simulations for 2d physics scenarios. This program is hosted on a website and contains a comfortable and easy to use interface, helpful for navigation. Their approach is to have a main page in which all scenarios can be 'previewed' by the user, and users are able to download and run these scenarios through the use of a Java Virtual Machine.



Once downloaded, the user can run the program and a window pops up with a user friendly interface and a 'data' tab for configuring settings for the simulation. This approach allowed for the software to be more portable as opposed to the other example.

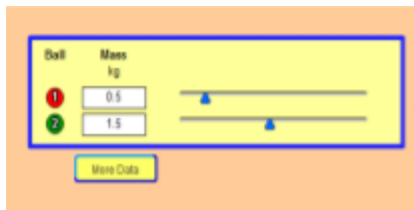
Another approach is by 'myphysicslab' which simulates 2d scenarios. The style and layout is similar to the approach made by the previous program, however users cannot download these scenarios and are instead run directly in the browser. Additionally, the program contains 4 tabs, 3 being related to graphing about the data of the simulation.



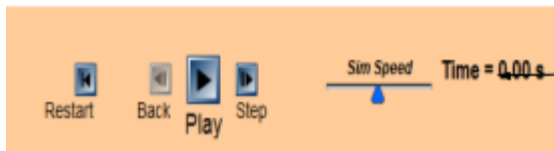
What I like about these examples

In the first program I like the overall layout of the website which is concise and easy to navigate through. Another feature is that users can download these scenarios and then run them directly on their machines. Additionally, the program also has a very user friendly interface accomplished via having a simple colour palette and using semi large icons. I may be able to involve this concept within my solution by using a simple and reduced pallet size, using only colours to highlight the user's attention to significant parts of the project. Furthermore, the use of colours in this example, also aided in guiding the user implicitly towards where the controls were, and at no point during navigation of the program did I feel confused or lost when navigating it.

Moreover, the program also allows for elements to be 'hidden' to reduce the amount of elements on screen at once, making the program much cleaner and easier to use. This is because it reduces the amount of objects that the user will have to look at, on screen, making the overall design be more robust and dynamic, adjusting its appearance according to what the user chooses.



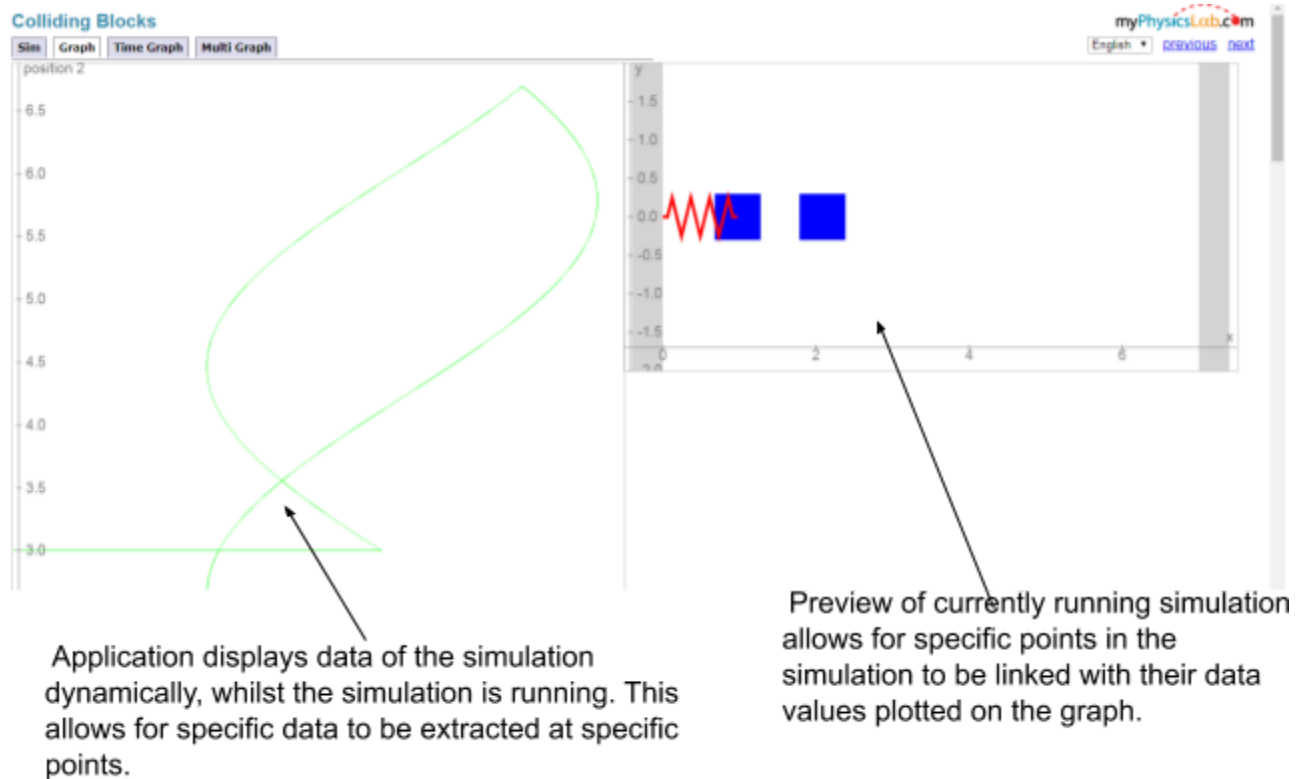
Program allows user to toggle certain elements on and off to reduce on screen clutter.



Program allows for users to change the speed at which the simulation is played, along with allowing the simulation to be paused and then restarted.

Additionally, the first program comes with the option for users to download specific scenarios useful for storing simulations offline or for physically transferring simulation files without the need for a user to access the specific website. Furthermore, the software uses a Java Virtual Machine in order to run its applications, where Java is very commonly installed on most devices, meaning the software has a larger accessibility pool compared to the second program.

In the second program I like the fact that the layout is very simple, and therefore reduces the clutter. In addition to this, the scenarios are run directly through the browser, so users can immediately see the simulation in effect, which was a drawback on the first, as it increases the time to setup the program for the user, decreasing the program's overall ease of use. Another thing is that these simulations come with a variety of tweaking and input that the user can use, along with data representation techniques such as graphing.



What I dislike about these examples

In the first example, what I disliked about it was the user was limited to having a Java Virtual Machine installed on the computer in order for any downloaded simulation scenario to be run, which would have limited the accessibility and portability of the simulation.

Furthermore, lots of simulations that I tried lacked many detailed features, such as changing specific particle types and their properties, or allowing for the environment size and shape to be changed. This struck me as a weakness, as it prevented further customisation, and also reduced the accuracy and control to which the user can place and setup the simulation in accordance with their specific needs.

The download feature required special software to be installed on the users machine - A Java Virtual Machine and the necessary Java dependencies. As a result of this, I may also hinder the portability and ease of use, as the user will have to download extra software to ensure the simulation works correctly. The lack of a web version for the simulation can hinder the portability, for users who cannot install the necessary dependencies on their system, and may deter future users from using it due to its longer startup requirements as opposed to a web based solution.

What I will adapt

For my solution, I will adapt it to contain the variety of settings available in example 2, such as particle and environment customisation. This is because this would be more important than having a download function because otherwise, the simulation may not be suited for users in depth needs. Additionally, without a variety of settings

to tweak, the user would not be able to customise the simulation to their requirements, hindering one of the objectives of this solution.

Furthermore, I will aim to include a data representation aspect to the solution, taken from example 2, as it can allow for easier data extraction from users, but additionally, can also be used for developer testing of the simulation to ensure variables and data are behaving correctly.

I have decided not to include the download feature present in the first example because it presents extra challenges that may not be appropriate to spend time on as it will have negligible effect on user experience. Moreover, the download feature would present some challenges such as what it will be compiled into, (.exe, .jar, etc) presenting compatibility and platform problems with devices that cannot run these files.

Proposed Solution

In my solution, I will develop a 2 dimensional simulation of a few common physics scenarios, aimed at students, professors and professional use in order to explore and examine physics concepts visually. My solution is addressing the need for visualised learning in a classroom environment, by having it being digitised and therefore accessible to most schools. Current approaches to physics simulations are very well developed, however my solution aims to be user friendly to students and professors, along with encouraging users to explore and manipulate the simulation to their needs.

The solution will revolve around the particle, which will be the main entity within my simulation, that the user can manipulate, control and observe. This particle will contain a variety of methods for collision checking, velocity, force and acceleration to mimic real life physics of how objects travel in motion. Moreover, the proposed solution aims to have the particles contained within a sandbox, where there are boundaries, with the particles bouncing off of both sides of the walls and other particles. This feature will aim to use OOP concepts and use classes, with functions to mimic the real life particle object. Moreover, the particles will be 'packaged' into a class using OOP concepts to more easily organise the development.

In the solution the main features are the settings area for which the users can change the properties of the simulation to thereby change the outcome of the physics simulation. My solution will focus on having customisation features for the user to address the issues found in the first research simulation. Additionally, the more features that are added for the user to interact with, the more control the user has over the simulation. These settings aim to be easy to navigate and use through the use of large texts, and a simple colour palette. Examples of these settings include sliders to adjust the number of particles to be created along with input fields like buttons for changing the states of the game - i.e. play, pause, restart, etc.

Finally, the users will access the website through the use of hosting the HTML pages on a domain, or by possibly using some free alternative methods such as GitHub pages. Moreover, as the code will be written in web technology - CSS, HTML and JavaScript, it will make it easier to create a website for it.

Limitations of Solution

My solution has some limitations, one of them being that the user will not be able to directly download a runnable file from my website, which means that my solution has reduced portability across computers. The reason for this limitation is that most users will want to just use the program directly, but will not feel the need to download the program directly to their computers. Additionally, a download feature is not present because it may require a large development time, which could have a knock on effect on the other more important features of the simulation such as usability features and developing the User Interface.

Another limitation of my proposed solution is that it will only cover a specific number of simulations of simulations within physics, such as collision and momentum, but not address gases, and liquids. Because of this, my solution cannot be used independently on a particular physics topic, and instead, it can be used in conjunction with other simulations.

Project Objectives

Objective 1
Create a display canvas for each simulation scenario.
Success Criteria
<ul style="list-style-type: none">- When a scenario page is opened, a blank canvas will be displayed.- The canvas must be able to connect to the JavaScript code.- The canvas must allow for entities to correctly be drawn to the canvas.- The canvas must be cleared after each update and new entities' positions updated.
Sub Tasks
<ol style="list-style-type: none">1. Design and implement a canvas for each scenario.2. Write appropriate canvas methods such as clear, reset, etc.3. Allow for entities to 'connect' to the display to draw appropriate shapes.4. Create buttons for changing width and height of the canvas dynamically.
Coding / Technical Work Required
<ol style="list-style-type: none">1. HTML is used for basic canvas structuring.2. CSS used for implementing more appropriate design and polish to the canvas.3. Research into how elements can be updated dynamically on a web page using JavaScript.

Objective 2
Create a variety of buttons and input fields for users to customise the simulation.
Success Criteria
<ul style="list-style-type: none">- Have a variety of buttons near the canvas which change values.- Connect the buttons via JavaScript so data can be processed.- Have settings for canvas properties.- Contain some time and canvas manipulation settings such as play,pause and reset.- Contain settings for entity creation and properties.
Sub Tasks
<ol style="list-style-type: none">1. Design a main frame for which the HTML elements, buttons, drop downs, etc, will be

placed.
2. Code and place buttons and elements onto the main frame.
3. Create JavaScript code for taking in the values that are inputted into the buttons and input fields so they can be stored and processed.
4. Allow for the collected data to then be applied to the variables that change the simulation.
Coding / Technical Work Required
1. HTML code used for creating the main frame and its elements.
2. CSS used for appropriate formatting.
3. JavaScript code used in collecting data from specific HTML elements.
4. Either linking external JavaScript files or by embedding the code within the HTML.

Objective 3
Create entities when the simulation has started, and include settings taken from objective 2.
Success Criteria
<ul style="list-style-type: none"> - Allow for the number of entities to be specified. - Allow for the entities' radius to be specified. - Include limiters to avoid large performance losses, and to ensure the solution performs relatively consistently without crashes.
Sub Tasks
1. Create a basic interface to allow users to create particles, done through buttons and input elements. 2. Develop back-end code for collecting the data of the entity properties, colour, size, etc, and how many to create. This is mostly performed via Objective 2. 3. Create back-end code to allow for entities to be created, performed via having an entity class which has custom sizes and colours, which are passed into constructors when they are created.
Coding / Technical Work Required
1. HTML for creating buttons and input fields. 2. Link this objective with objective 2, so that when entities are created, their properties are included in their construction. 2. Learn a variety of input fields that can be used to display information to the user such as sliders, buttons, dropdown lists, etc.

Objective 4
Create a simulation manager that will control the flow of the simulation.
Success Criteria
<ul style="list-style-type: none"> - Create one instantiation of the simulation manager at simulation startup. - Have a simulation 'loop' where the simulation continues until specific events happen. - Coordinate the updating of all entities in the simulation. - Coordinate the rendering of all entities in the simulation. - Check current conditions of simulation to determine if appropriate to end the simulation.
Sub Tasks
1. Create a basic class outline for the simulation manager – constructor, destructor, etc. 2. Implement 'update entities' method for looping through each entity and updating their positions and properties. (done within the entity class). 3. Implement 'render entities' method for looping through each entity and rendering their

current shape and position.

4. Create a basic data structure for storing the entities – preferably a dynamic array as elements have a Big $O(1)$ time complexity when indexing.

5. Create a loop method for calling both the 'render' and 'update' methods, along with checking specific simulation flags to determine if the simulation should end.

Coding / Technical Work Required

1. JavaScript code for class creation.

2. Setting global flags and determining their value and condition.

3. Link this file with the entity file so that entities can be updated correctly.

Objective 5

Create entity class to have individual entities be made.

Success Criteria

- Must be able to be instantiated in large quantities.
- Must be able to update it's position by applying forces.
- Must be able to have collision with other objects.
- Must be able to have collision with the boundaries of canvas.
- Must be able to have custom properties passed into it.

Sub Tasks

1. Create a basic JavaScript class for the entity.

2. Implement the update method which will call smaller methods that check the entity's position and others that apply force to the entity such as vectors (2 dimensional, x and y).

3. Implement a render method that will correctly render the specific entity and it's custom properties to the screen (shape, radius, etc.).

4. Implement public fields so their values and properties such as colour and radius can be easily modified.

Coding / Technical Work Required

1. Integrate entity class with the input section so that the input section can create entities based on specific user parameters.

2. Integrate entity class with simulation manager so that the entities can be updated and rendered from the simulation manager.

3. Research about JavaScript class syntax and implementation.

Hardware and software requirements

Requirements	Justification
2GB of RAM	2GB of RAM is the minimum requirement, this is because firstly, most operating systems require at least 2GB of RAM to operate, and the simulation may use up a few 10s of megabytes maximum due to the amount of particles rendered on screen, but this should be relatively low. Furthermore, the browser that the simulation is run in will also use up at least 500MB.
1 – 2GHZ Processor	The simulation will require at least a 1GHz processor in order for minimum performance loss. Any CPUs with lower clock speeds most likely have 2 or fewer cores,

	and a very slow clock speed that would not be able to process the operating system, browser and the simulation all at once. Furthermore, my simulation may be CPU intensive at times where lots of particles are colliding with each other, and therefore the simulation needs access to at least 400MHz.
Monitor	A monitor will be required to use the simulation, mostly in order to actually see the results and to see the particles when they are drawn to the screen.
Mouse	A mouse is required by the user in order for basic computer use and to allow the user to interact with the simulation's input features and user interface. Finally, there will be a substantial number of buttons that can only be activated with the use of a mouse.
Keyboard	A keyboard is also required in order for the user to input values into text fields that require it, although technically the user can get away with not using a keyboard, but it will limit the scope of what features they can use in the simulation.
A suitable mainstream Operating System.	This simulation will require a suitable operating system, preferably one of the main 3 - A Linux Distribution, MacOS and Windows. This is because testing can be done easier on these systems. The simulation will require this as the user will not be able to launch any applications without a suitable operating system underneath it that can support a web browser such as Firefox, or Google Chrome.
Browser	A browser is required in order for the simulation to work properly, as the browser will contain JavaScript processing and HTML/CSS support for .html web pages. Furthermore, the web browser is responsible for connecting to the Internet when using external libraries to connect to.
JavaScript Support	JavaScript support is required by a browser in order for the simulation to run. Additionally, scripts should not be blocked in the browser, to allow for my scripts to run correctly.

Communication with Stakeholders

Interview

The initial interview will consist of a few predefined questions that I will be asking the stakeholder, where their response is then noted.

Questions

1. In what ways are physics concepts taught?
2. How do you think physics concepts could be taught easier?

3. As a physics student, how do you learn material the easiest? (Visuals, reading, listening)
4. Do you think students would benefit from being able to control and visual specific physics concepts?
5. Would a physics simulation be able to replace or be used in conjunction with a physical practical conducted in the classroom?
6. What concepts would be most suitable to be simulated?
7. What features would be useful in a simulation?
8. Any other points to add?

Communication with Physics Student

1. In what ways are physics concepts taught?

"Physics concepts are mostly taught through question papers and practice papers, along with teachers occasionally showing helpful videos for practical uses."

2. How do you think physics concepts could be taught easier?

"Sometimes I struggle to understand certain topics as they are taught mostly with reading exercises. I feel as though this has hindered my learning of physics, and would need to be taught with more practical and visual elements. Finally, I think that slower approaches to teaching could also benefit my learning."

3. As a physics student, how do you learn materials the easiest?

"For myself, I learn physics easiest when it is taught slowly, and I can experiment with the ideas, thinking about how they can be applied realistically, but I also enjoy watching videos as they are more engaging."

4. Do you think students would benefit from being able to control and visualise specific physics concepts?"

"I think that being able to visualise a concept be it momentum, or particle collision can very much aid in the process of learning a concept and therefore would be beneficial. Additionally, being able to control a concept and how it plays out would make it easier to learn and understand difficult tasks."

5. Would a physics simulation be able to replace or be used in conjunction with a physical practical conducted in the classroom?

"A physics simulation I think would be a very useful tool to be used alongside having regular classroom practicals as the practicals stick in the student's minds, but a simulation could allow for more control and experimentation."

6. What concepts would be most suitable to be simulated?

"Any concepts that involve the use of particles or objects such as atoms interacting with other objects could be most suitable, alongside exploring physical values such as gravity and how it interacts on objects, etc."

7. What features would be useful in a simulation?

"In a simulation I would want to be able to control the features of the simulation and be able to create and destroy entities at will, along with customising their properties such as shape and size."

8. Any other points to add?

"No."

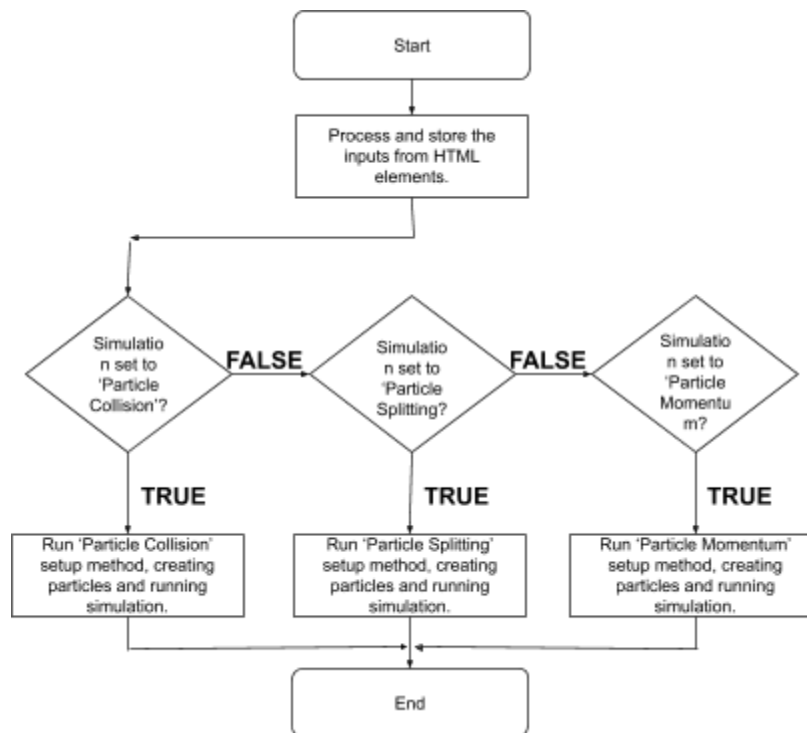
Design

Structure of the Solution

Breakdown of Solution:

Flow of Program: The basic flow of the program begins when the user has selected a simulation and has pressed the 'play' button. This begins by processing and storing all inputs from HTML elements such as buttons and data fields, which is done at the beginning as these values will be used when creating particles later on, and manipulating the settings of certain simulations. The program will then continue processing until the user either refreshes the page, or stops the simulation.

How each section is connected: This solution contains many 'sub' sections such as input, particles, and each simulation. There is a strong link between user input and the simulation as the user is responsible for controlling the flow of the simulation - when it begins, when it ends, what particles are created, how many, etc. Because of this, lots of HTML elements will be used, and the program will contain a variety of buttons that the user can click to change the properties of



the simulation. JavaScript code will be used to take in and process these values, ensuring they are valid and suitable to be used in the program - correct data types, correct value (non extreme), etc.

Decomposition

Another way of decomposing the physics simulation is via splitting it into 3 smaller problems;

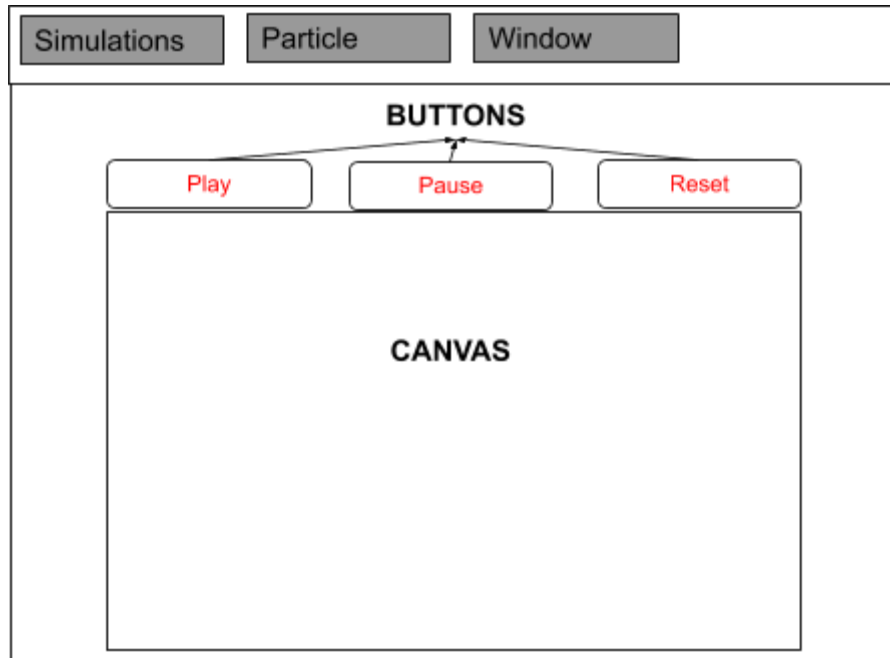
- Input Management
- Simulation Updating
- Simulation Rendering

Input Management: Firstly, the simulation will involve a variety of input boxes in order for users to change and select various settings about the simulation. This will be done via using HTML syntax for creating the placement for buttons, input boxes, check boxes, etc. Additionally, CSS can be used for formatting and configuring minute details about the input elements. Behind the scenes, JavaScript will allow for data to be captured and stored, through HTML linking of JavaScript files. The files will contain functions responsible for capturing the entered data values, then calling appropriate methods such as creating particles using the custom settings, or by changing possible global variables used throughout the simulation. This process of input handling is suitable for computational methods as multiple data boxes or input elements can be captured simultaneously and processed.

Simulation Updating: The updating of the simulation is separated into rendering and updating logic. Each particle will have both a 'render' and 'updatePosition' method which neatly separates the logic of moving particles, collision detection, applying forces, etc, whilst also allowing the particle to be rendered in whatever order is necessary. Additionally, the 'updatePosition' method will need to perform some collision checking as the particles will need to be contained within the rendering canvas, otherwise they will not be visible.

Simulation Rendering: The third problem is the use of rendering each particle to the screen. This will be performed via having a main rendering method which is called every frame after the update method has been called. The render method is solely responsible for looping through all particles within the simulation display and calling their rendering methods. These methods will involve connecting to the current canvas and redrawing the particle's shape (with the new coordinates) to the screen. The actual rendering code will be placed within a 'render' member function within each particle, making it easy to simply loop through a list of particles and call both the 'updatePosition' and 'render' methods on them.

User Interface Design

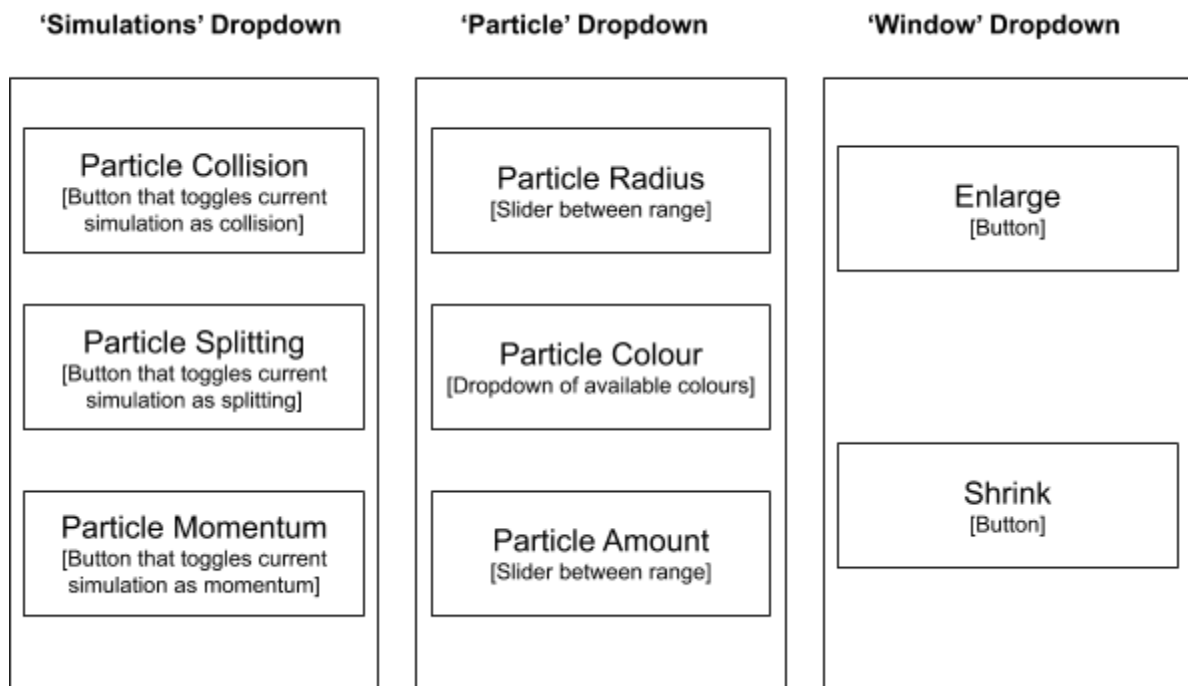


Breakdown of Features:

Canvas: The program's user interface will consist of a main rendering canvas situated at the centre of the screen, with time manipulation buttons present directly above it. This main canvas aims to take up most of the space, and is positioned in the centre of the window, as it is the main aspect of the simulation.

Buttons: The buttons are situated directly above the canvas rendering screen, and are semi large in size. The increased size means the button is easier to click for users, and additionally, medium sized spaces are present between each button to prevent overlap, and to avoid the user misclicking on buttons that they did not intend to click. Furthermore, the buttons are coloured red to stand out more easily.

Toolbar: The next section of the user interface is the toolbar which consists of 3 dropdown menus - Simulations, Particle, and Windows. These 3 dropdown menus consist of varying options, so for example, the 'Simulations' dropdown menu will display a list of all simulations that the user can choose from, whilst the 'Particle' dropdown menu will display a list of settings that change the properties of the particles.



Particle Collision: This is a button that when pressed, enables a flag that changes the current simulation to the collision simulation.

Particle Splitting: A button that will toggle the flag for running the 'Particle Splitting' simulation, which has its own draw and setup methods.

Particle Momentum: A button that will toggle the flag for running the 'Particle Momentum' simulation, with its own draw and setup methods.

Justification for Button and Toggle Use: A button is used rather than another HTML page because when using P5.js' it's draw and setup functions are restricted to one canvas page, so the way around this is to have toggles which are evaluated during these methods, with each branch dictating a different simulation outcome.

Particle Radius: This is a slider which has a value between 1 and a selected limit. The slider is easy to use as the user can drag and drop the slider to the desired value. This is combined with an output of the slider value, so the user knows what the value of the slider is.

Particle Colour: A dropdown list of available colours, which when selected, will change the current value of the 'Particle Colour' box to this specific colour.

Particle Amount: A slider which has a value between 1 and a selected limit, which will have an appropriate output box to display the value of the slider.

Justification for Slider Use: A slider is used because they are easy to use for users as a simple drag and drop is required. But additionally, the range that the user can enter between can be set using sliders, which can eliminate the need for input validation checking in the JavaScript code.

Justification for Drop-down List: A dropdown list is used as the available list of colours to choose from may be quite large, and the values of for the colour types are qualitative rather than quantitative

Algorithms / Decomposition

Particle Class

Particle Class
Constants: Global constant float damping Global constant float gravity Members: Public float x Public float y Public float radius Public float xVel Public float yVel Public string colour
Methods: Private procedure checkBoundaryCollision() Public procedure new (float x, float y, float xVel, float yVel, String colour) Public procedure updatePosition() Public procedure render() Public function splitsWith(Particle particle) Public function isOver(Particle particle) Public function checkCollision(Particle particle)

Decomposition:

Members:

Each particle class contains an x and y value, representing the 2 dimensional position that the particle is in. These values are manipulated when applying the 'xVel' and 'yVel' values to the x and y positions. Floating point types will be used to increase the precision, and allow for smaller increments, instead of whole integer number increments (1,2,3). Finally, the colour is a string type, and is used in rendering to render the particle object as the value stored in this member. String data type was chosen as HTML's colour options return a string, so when a user selects a colour, the selected colour will need to be stored as a string.

Methods:

CheckBoundaryCollision: Checks the x and y values to see if the particle has reached, or gone over, the boundaries of the window. If so applies a 'bounce' effect to the particles.

Constructor: The constructor is used when instantiating new particle objects and takes in an x,y, xVel, yVel and colour as parameters.

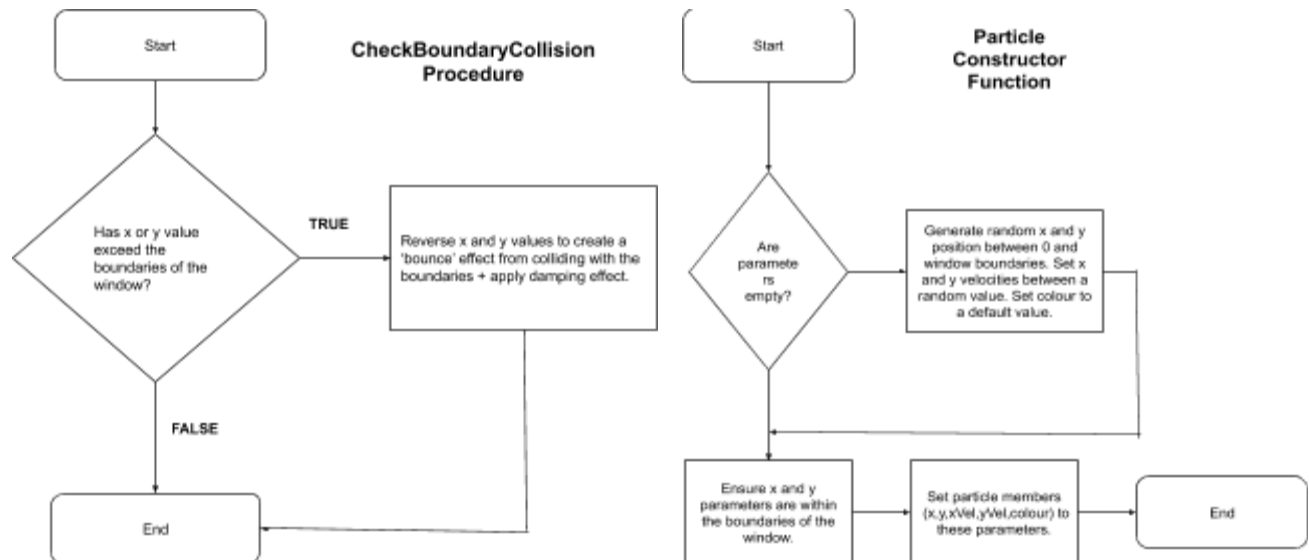
UpdatePosition: This method applies the velocities to the x and y coordinate, and calls collision checking methods.

Render: Renders the actual particle to the selected canvas.

CheckCollision: Checks if collision has occurred with the current particle and a passed particle.

CreateParticle: Not a member function, but will be bundled with the class, creates a specified number of particles, with the passed parameters.

SplitsWith: This function checks if a particle can split with another particle.



Decomposition:

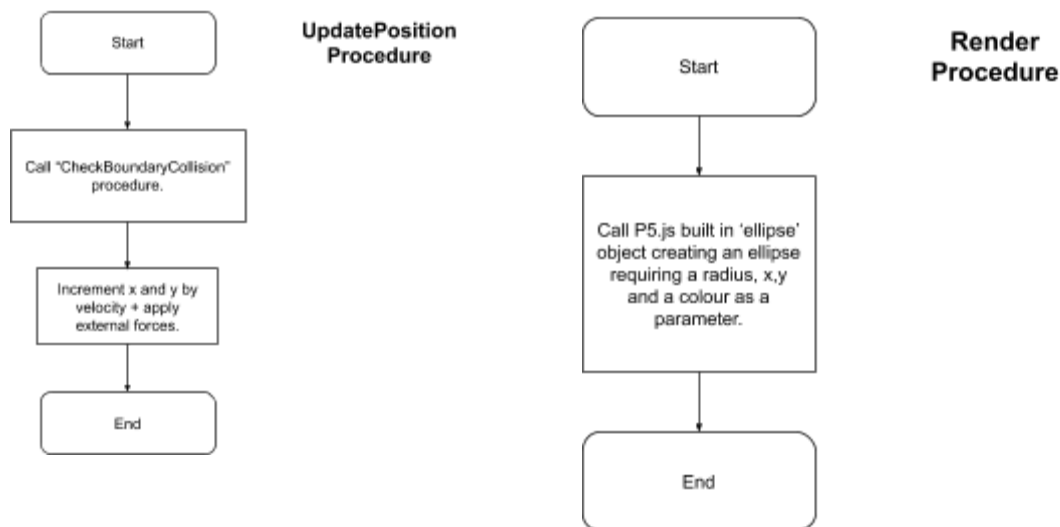
CheckBoundaryCollision: The 'CheckBoundaryCollision' method is responsible for checking the x and y value of the particle with the set window width and height boundaries, to ensure that if the particle goes off screen, it is correctly restrained, and certain collision effects are applied to it. The window width and height values are obtained from global variables. It accomplishes this via simple comparison operations of ' \leq ', ' \geq ', ' $=$ ', and checks if a given value, e.g. x, is \geq to the window's width, or ≤ 0 , meaning is the x value either too far to the right (positive), or too far to the left (negative).

Particle Constructor: For the particle constructor method, it is responsible for creating a particle object, whenever a particle object is instantiated within the code. It takes 5 parameters (x,y,xVel,yVel and colour), which are then used to set the particle member values. Default values will be used in the constructor to allow for randomly generated particles, and to reduce the number of parameters required to be passed to the function when a particle is instantiated.

Justification:

CheckBoundaryCollision: Boundary check involves checking the limits of the rendering window because if the particle's position goes over or under these values, then it's rendered position will appear incorrectly, resulting in for example, a semi circle being rendered at the edge of the screen, or incorrect collision with other particles. Furthermore, if the particles are not restrained to the window, then they will go offscreen and still be rendered and updated resulting in a loss of performance.

Particle Constructor: The particle constructor is designed to use default values to ensure that if no value is passed, a valid random value is generated. Additionally, validation and range checks are performed on the particle's x and y positions to ensure that they are within the window that will be rendered, otherwise these particles may not be viewable to the user when they are rendered.



Decomposition:

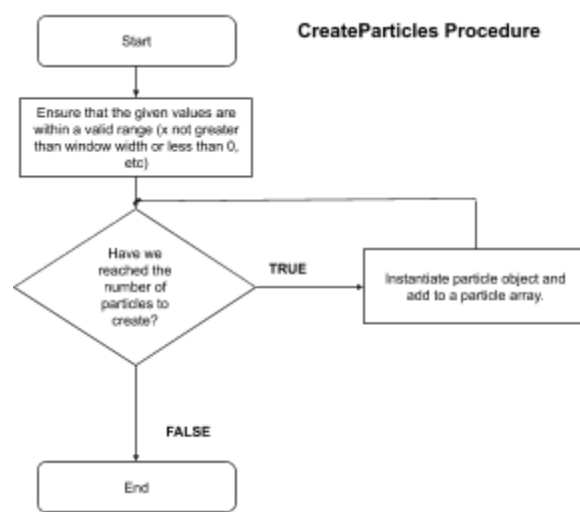
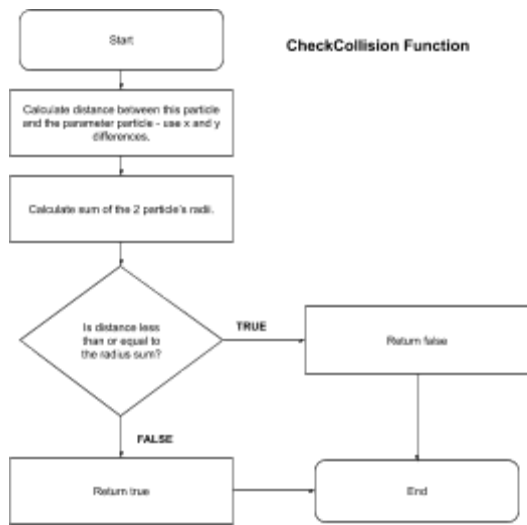
UpdatePosition: The 'updatePosition' procedure will first call the already defined 'CheckBoundaryCollision' procedure, which will check the boundaries of the particle and ensure proper collision occurs. This is done to ensure that if the particle is offscreen (or beginning to go off screen) it is restrained to the rendering window. After this, the procedure will apply the x and y velocities to the x and y positions, and then apply any external forces that are implemented such as gravity.

Render: The 'render' procedure is used for rendering the particle object to the screen. It uses a P5.js method of creating an ellipse object, and using a plotting method to plot it onto the window. Furthermore, the ellipse takes in a radius, x,y and colour as parameters. These values are then passed to the ellipse constructor.

Justification:

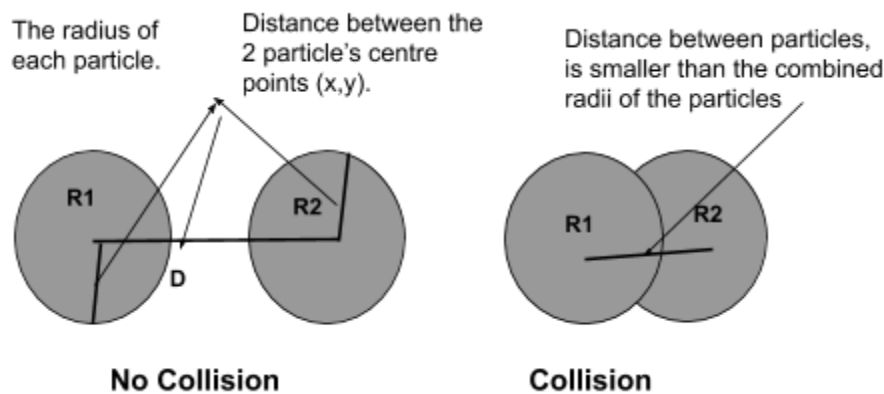
UpdatePosition: 'updatePosition' procedure runs the boundary collision before applying any velocities or external forces, because if the order was reversed, then the already out of bounds object would then have it's properties changed, and then collision will occur, rather than colliding with the particle with the boundary of the wall and then applying the velocities and forces, ensures that it stays on screen and behaves normally. Finally, the velocities are applied first to the x and y values, before any forces are applied, so that the initial acceleration does not decrease before it is applied.

Render: The 'render' procedure is separated from the 'updatePosition' procedure so that the logic of updating the particle's position and rendering the particle with these new positions can be separated, so the order of render or updating logic can be changed depending on when they are called.



Decomposition:

CheckCollision: The 'CheckCollision' function will return either true or false, depending on whether the current particle has collided with another particle (parameter). It does this by calculating the distance between the 2 circles, and seeing if this is less than or equal to the sum of the radii of the 2 particles.



Collision between particles is essentially (in 2D space) if one particle overlaps another particle. We can determine if one particle has overlapped another, if the distance between them is less than or equal to the sum of the 2 particle's radius. It can be calculated as follows:

$P1 = \text{Particle 1}$

$P2 = \text{Particle 2}$

$$\text{Distance } (d) = \sqrt{(P1.x - P2.x)^2 + (P1.y - P2.y)^2}$$

$$\text{Radius} = P1.\text{radius} + P2.\text{radius}$$

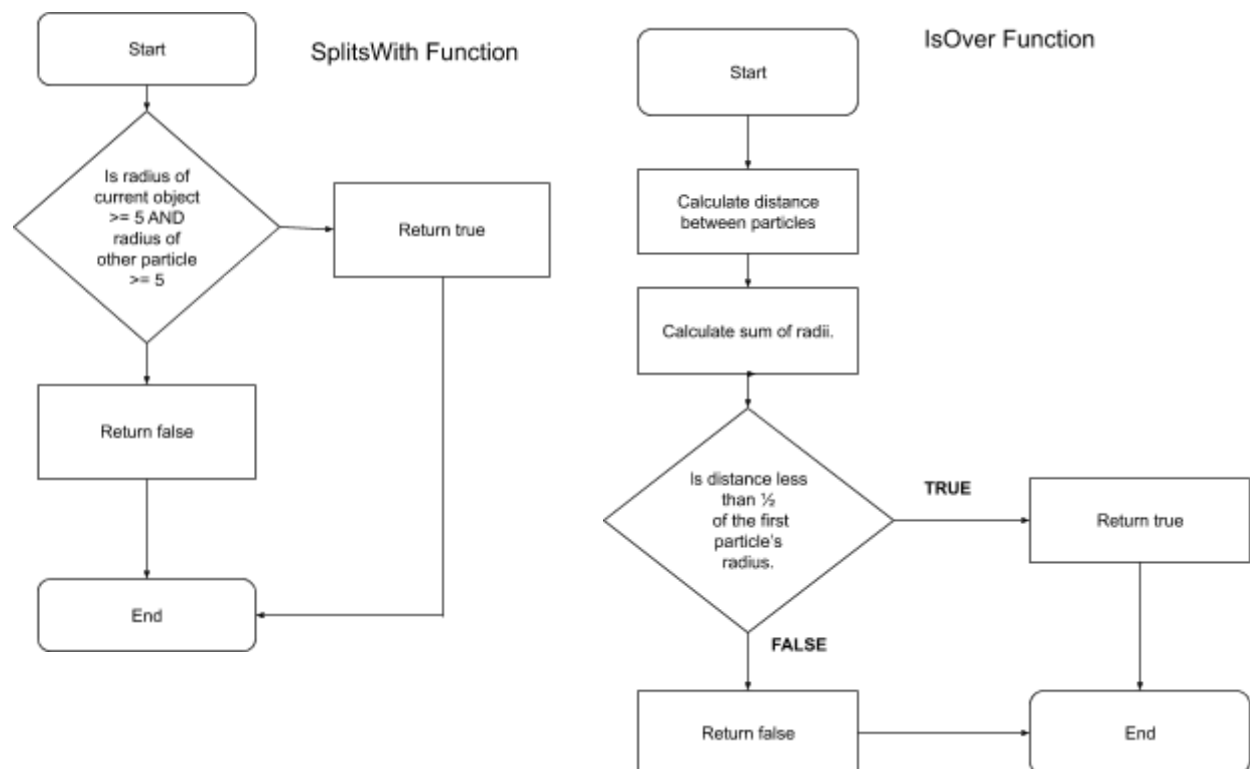
So if the calculated distance is less than or equal to the radius sum, then return true, otherwise we return false.

CreateParticles: This procedure's main role is to create a specified number of particles, with the specific parameters passed and add these particles to an array of particles available to all other source files / simulations. To ensure that the parameters are valid, their values are range checked to ensure that they are not extreme or erroneous, so for example, the x parameter is checked between the range of 0 and the window's width. Finally, default parameter values will be used to allow the function call to be more dynamic - particles can be created with just a colour passed in resulting in varying sized particles of the same colour, increasing the usage and randomisation of the particles created.

Justification:

CheckCollision: In this procedure, the distance between the points is calculated as the square root of the sum of the centre points of each particle, because we need to get the hypotenuse of the x and y of the particle to get a resultant or total distance between them. Furthermore, the radius of each particle is added, rather than one of the radii being squared, as the particles may be of varying sizes, so to account for this, each particle's radius is used, rather than squaring just one.

CreateParticles: Because the parameter values are unknown, validation is required. Validation performs range checking, because the function call could pass an invalid position for the particles to be created, such as the y value being greater than the height of the screen, meaning the particles will not be visible when rendered.

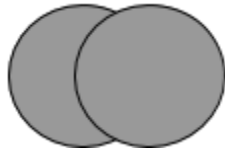


Decomposition:

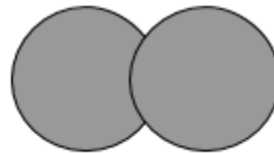
splitsWith: This function will check to see if the two particles are of a large enough size to split. The function will perform a comparison to see if the radius of the 2 particles is at least 5, if not it will return false.

isOver: This function will check whether the particle can split with another object. This is determined, if radii of both the first and second particles are greater than or equal to a preset value (in this case a radius of 5). The steps to determine this are similar to checking collision with another particle, except that we need to check if a majority of the particle is over the other particle. We can do this by comparing if the distances between the 2 particles is less than $\frac{1}{2}$ of the radii sum.

Second particle has overlapped by at least the radius of the first particle, meaning they will split



Second particle is only slightly overlapping the first particle, no split occurs.



$P1 = \text{particle 1 (left)}$

$P2 = \text{particle 2 (right)}$

$$\text{Distance } (d) = \sqrt{(P1.x - P2.x)^2 + (P1.y - P2.y)^2}$$

$$\text{Radius} = P1.\text{radius} * 0.5$$

If the distance is less than or equal to the calculated distance, then the particles can split.

Justification:

splitsWith: This function will only return true if both radii are greater than or equal to 5 as if one is smaller than the other, then that particular particle cannot split properly.

isOver: In this function, the distance between the particles is calculated as the square root of the sum of the centre points of each particle, because we need to get the hypotenuse of the x and y of the particle to get a resultant or total distance between them. Furthermore, the radius of the first particle is calculated, as we are seeing if the second particle is over the first particle.

Algorithms / Decomposition

Input and Data Processing

Input Processing
Functions: Public function <code>getAmount()</code> Public function <code>getRadius()</code> Public function <code>getColour()</code> <hr/>
Procedures Public procedure <code>togglePlay()</code> Public procedure <code>canCreateParticles()</code> Public procedure <code>removeParticles()</code> Public procedure <code>resizeWindow(int width, int height)</code>

Decomposition:

getAmount: Retrieves the value from the HTML element that contains the value for the amount of particles that are to be created.

getRadius: Retrieves the value from the HTML element that contains the value for the radius of the particles to be created.

getColour: Like the other 2 functions, this function retrieves the string value from the HTML input element that contains the dropdown list for the available particle colours.

togglePlay: This procedure will check the current state of the 'play/pause' button and determine based on its current value if it should either play or pause the simulation.

canCreateParticles: This procedure is used to see if any more particles can be created on screen, or if the maximum number of particles has been created already.

removeParticles: Removes all particles from the particle list, clearing it out.

resizeWindow: Resizes the canvas window to the new size given.

```
Function getRadius():  
    Int radius = document.getElementById("RadiusInput");  
    Return Radius;  
End function  
  
Function getAmount():  
    Int amount = document.getElementById("AmountInput");  
    Return amount;  
End function  
  
Function getColour():  
    String colour = document.getElementById("ColourInput");  
    Return colour;  
End function
```

Decomposition:

getRadius: This method will first create an integer (radius) to store the value. The variable is then assigned to the value at the element ID "RadiusInput". Afterwards, the value is returned from the function.

getAmount: This function will create an integer variable to store the value. After the value is successfully retrieved from the desired HTML element, the value is then returned from the function.

getRadius: This is a function that creates a string variable of colour. The colour variable is then assigned to the value stored in the HTML page, and then returned.

Justification:

getRadius: The function creates a variable of type int because the radius is of integer value as the slider goes from 1 to a set limit, incrementing by integer values.

getAmount: This function also creates a variable of type int because the amount of particles to be created is in whole numbers.

getColour: This function uses a string data type as the types of colours are of qualitative form, and are easier and clearer to represent using strings rather than numbers.

```

Int particles = 0;
Const int maxParticles = 5000;
List particles;

Function togglePlay():
    If the play button's state equals "play"
        Set play button state to pause;
        Re-enable the updating of simulation.
    Else if the play button's state equals "pause"
        Set play button state to play
    Endif
Endfunction

Function canCreateParticles():
    If particles <= maxParticles
        Return false
    Else
        Return true
    Endif
Endfunction

```

```

Function removeParticles():
    for(i = 0 to particles.length):
        particles.pop(i)
    endfor
Endfunction

Function resizeWindow(width,height):
    windowWidth += width;
    windowHeight += height;
Endfunction

```

Decomposition:

togglePlay: Used for either playing or pausing the simulation when the button is pressed. As one button will be used, it's HTML element will have a value that is evaluated to get the current state of the play button to determine whether to play or pause the simulation.

canCreateParticles: Determines if more particles can be added to the simulation, by comparing the current size of the list of particles, to a constant integer specified at a value of 5000. This is used to prevent large performance loss.

removeParticles: Pops all elements from the list of particles, and sets the size to 0. This method is used in resetting the canvas, or when switching between simulations.

Justification:

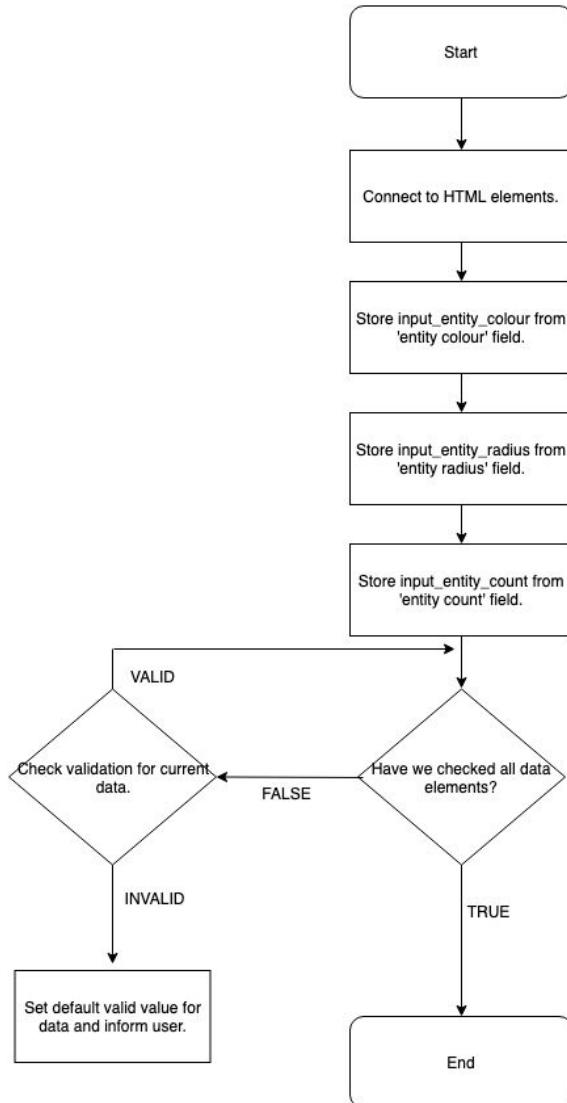
togglePlay: One button was used in an attempt to decrease the visual clutter on screen as instead of having play and pause buttons independent we just have the one. Additionally, when the simulation is paused, pressing pause will be useless as the program is already paused, so the buttons will be redundant 50% of the time, if they were separated. However as they are combined into one button, the buttons stays useful all the time.

canCreateParticles: A constant is used here, as the program will not want to create any more values beyond and given limit, and we do not want to accidentally modify this value, so therefore it is set to constant.

removeParticles: Removes all particles present within the particle's list. Manually loops through and pops each element from the list.

Input

As the simulation requires a strong connection between the user and the program, all inputted data must then be correctly handled and validated, to ensure that their values can be processed correctly by the simulation.



1. The algorithm for collecting input starts off with first linking the JavaScript code to the HTML page so that input elements such as buttons and fields can have their values retrieved.

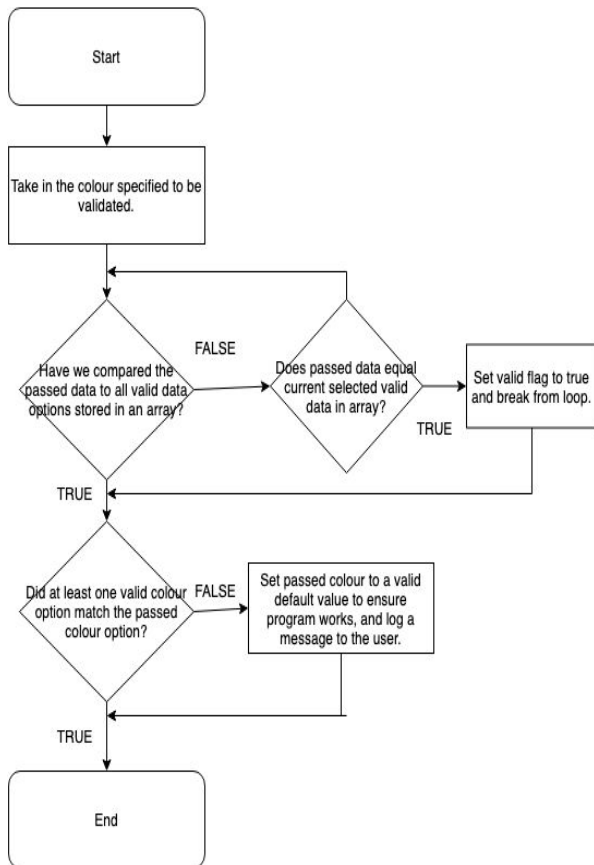
2. Once the JavaScript and HTML page are linked, a local variable is created for each input field e.g. 'var input_entity_colour' will be a local variable to store the entity's colour taken from the specific HTML element that possesses this value. This process is repeated for all required HTML fields that need to have their data values extracted from.

3. After all data has been collected, they will then be individually validated to ensure that their values are valid and will not cause problems to the simulation. The process works by looping through each data collected (stored in array), then running an appropriate method which will be specific for each data type, and compare it to values to ensure that it is valid. If a specific element is invalid, then it will have it's value set to a default valid value. After all elements have been checked and or validated, the algorithm will then end.

4. Finally, the flow of the simulation will move onto the entity creation algorithm which will take in these specified and collected values to then be used in creating the specified amount of entities with the specified entity properties.

Validation

Along with the main sections (input, entity and manager), specific sub routines will also be present in the project such as input validators, and data stores. These will have their logic displayed in this section.



```
//outside of function

colour_array =
["Blue", "Black", "Green", "White", "Yellow", "Red", "Orange", "Purple"]

procedure check_valid_colour(string colour):
    bool valid = false;
    for(i = 0 to colour_array.length() i++):
        if(colour_array[i] == colour):
            valid = true;
            break;
        else:
            continue;

    if(valid):
        return;
    else
        int position = random.randint(0,colour_array.length());
        colour = colour_array[position];
endprocedure

procedure check_valid_radius(float radius):
    if(radius >= canvas.width or radius >=
canvas.height):
        radius = rand.randfloat(0,canvas.width)
endprocedure
```

The main logic for the sub routine validation checks involves checking the passed value to a set of predefined valid values. For example the colour inputted into the application is validated via comparing the colour to an array of acceptable colours. If the passed value is not in the array, then the value is set to an appropriate value. Otherwise the validation method will return with no changes to the data. For other arithmetic data such as the radius it is compared to an appropriate maximum size set to the canvas width/2 and canvas height/2, so that if the radius exceeds this then it will not be created.

Algorithms / Decomposition

Collision and Splitting

```
Procedure collisionDraw():  
  If the game is not paused  
    Render the background screen as white.  
  
  While i < length of particles list  
    Run particle update at position i  
    Run particle draw at position i  
    While i < length of particles list  
      Store output from particle checkCollision  
      If output is true  
        Reverse x velocity of first particle.  
        Reverse y velocity of first particle.  
        Reverse x velocity of second particle.  
        Reverse y velocity of second particle.  
      Endif  
    endwhile  
  endwhile  
Endprocedure
```

```
Procedure collisionSetup():  
  Amount = output from 'getAmount'  
  function  
    Radius = output from 'getRadius'  
  function  
    Colour = output from 'getColour'  
  function  
    While i < amount  
      Instantiate  
        particle object with parameters.  
      Add this new  
        particle object to particle list.  
    endwhile  
  Endprocedure
```

Decomposition

collisionDraw: This procedure is the main 'update' method for the collision simulation. It will be called every frame, or at set intervals, and will be responsible for checking if the game is paused, and if so will not draw the white background used to clear the screen. Afterwards, the method will loop through the list of particles, and one by one, run both the update and render methods for each particle. Inside this loop is another loop, which is used for collision detection between each particle.

collisionSetup: This method is first called, before the draw method is and setups the appropriate entities and flags before the simulation runs. It first gathers the required data entered by the user via the previously mentioned 'get' methods for getting the radius, amount and colour selected by the user. Afterwards, a loop will be used from 0 to the amount of entities to be created. This loop will create a new particle on every iteration, with the specified values and add this particle to a list of particles, for future rendering in the collisionDraw procedure.

Justification

collisionDraw: This procedure will only draw a white background when the simulation is not paused, otherwise even if the simulation's state is set to pause, the draw update will still clear the screen and draw the new positions of the particles. Furthermore, iteration is used as we need to go through each particle in the list, and make sure that they have been updated properly. The inner loop for collision detection will loop through every other particle, but its index is started at $i + 1$ because the current index i will equal the index of the current particle, resulting in the 'checkCollision' method checking collision with itself returning true, and simulating the collision incorrectly.

collisionSetup: In this procedure it will retrieve the data via the get methods, and it gets these values beforehand, as they are only needed to be retrieved once per setup call. Particles are added to a list of particles, so that in the updating stage, we simply loop through the list and call the updates for each particle. A list is appropriate as the number of particles is unknown, although a cap of 5000 is known, the number on the screen can be anywhere in between. Because of this using an array is inefficient as memory will be allocated on the stack for the specified number of entities, being memory inefficient.

Table of Variables

Global

Name	Type	Use	Justification
current Simulation	String	The name of the current simulation, used in determining which simulation should be run.	A string is used, as this value is compared, and string values are easier to read and understand, whilst integer literals would be less understanding i.e. if it was int, reading code would be hard to understand the meaning of '1', whilst having 'Collision Simulation' is easier to understand. This variable is validated through the use of it only being one of n possibilities, with n being the number of simulations present.
Particles	List	Contains every particle that is currently on the render window. It is used for looping through and updating each particle, or for removing each particle from a simulation.	A list is used as it is dynamic and can change in size during runtime. As the simulation will be responding to input, the user is able to create entities at any given time. Because of this, if other structures such as an array were used, this would result in static memory, which would not be able to grow in size as efficiently. The number of particles present in this list are validated at all areas of the code that allow for new particles to be created, checking if the new amount of particles can fit in the new list.
window Width	integer	Stores the current width of the rendering window. Used in boundary calculations and when creating the particle's x and y positions.	The value of these is not marked as constant, because the values of these can change. This is due to the fact that the user can enlarge or shrink the render window, and having it constant would not allow for this. This value is validated, as there will be a set upper and lower limit as to the size that the window can be. Every time the user enlarges or shrinks the canvas, its new dimensions are checked with the lower and upper limit.
window Height	integer	Stores the current height of the rendering window. Used in boundary calculations and when creating the particle's x and y positions.	The value of these is not marked as constant, because the values of these can change. This is due to the fact that the user can enlarge or shrink the render window, and having it constant would not allow for this. This value is validated, as there will be a set upper and lower limit as to the size that the window can be. Every time the user enlarges or shrinks the canvas, its new dimensions are checked with the lower and upper limit.

Particle

Name	Type	Use	Justification
Colour	String	This variable is used in storing the particle's colour as a string. Used in the rendering of the particle to the render window.	The colour is in string form because firstly, the p5.js rendering methods, take in a colour parameter for creating and rendering objects, in the form of a string, so non string types may be incompatible. Secondly, numerical representations of colour are less clear and more easily mistaken, however string values such as 'red' or 'blue' are obvious in their meaning. - no ambiguity. Validation comes in the form having the UI element being a dropdown menu with constant values, preventing the user from selecting or entering an invalid colour.
X, Y	Floating Point	These variables are used in representing the x and y coordinates of the particle's position. This variable is used in rendering for telling the render method where it should render to, and forces applied to it such as velocity and acceleration.	A floating point type is used for these values, as it is more precise as opposed to an integer value. Precision is required as the particles will be changing in minute ways, so these changes must register. Integer values have low precision and therefore will not be affected by these small changes possibly resulting in incorrect simulation outputs such as a particle not moving due to a too small force. The x and y values are validated by comparing their values with the boundaries of the render canvas, if the x or y is less than or equal to 0, or greater than or equal to the window width / height.
xVel, yVel	Floating Point	These variables are used to store the velocities for both the x axis and y axis. These values are applied to the x and y positions on every update call. Additionally, if gravity or deceleration is present, the velocity will change on each update call.	A floating point type is used as small precise values are required. Additionally as these values are velocities, their value will be added to the x and y positions. Because of this, if they are not of the same precision, then when added, rounding errors could occur, and the expected outcome would not equal the actual outcome. These values are validated in the same way as the x and y positions, except limits are placed, so that the starting velocities are not too high or too low.
Radius	Floating Point	This variable is responsible for storing the radius of the particle. The radius is used in drawing an ellipse to the render	The radius is of type floating point because it needs to be precise as the radius will be varying in size. Validation of the radius value is performed by checking if the radius is either negative or too large, in which the radius is then set to the upper or lower limits.

		window.	
--	--	---------	--

Input

Name	Type	Use	Justification
radiusInput	Float	stores the input value taken from the HTML element containing the radius entered by the user.	The radius will need to be stored, for it to be passed to other methods and used in creating custom particles with varying diameters and circumferences.
amountInput	Integer	Stores the input value taken from the HTML element that the user has entered the number of particles to be created into.	The amount of particles will need to be stored, so that other processes such as collisionSetup or splittingSetup can use this value to create the specified number of particles.
Colour	String	Stores the input value from the user input that selected the particle's colour.	The objects used within p5.js such as 'ellipse' take in a string argument that will dictate the colour. Other data types could be used, but are not as efficient.

Explanation of Modules

Currently this project uses WebGL as a module for drawing and creating entities that can be connected and rendered to specific canvases and can perform basic graphical operations such as update and change. Further modules may be added in the future of development, and are included here.

WebGL: This module may be used in my program for creating simple entities and having these entities be correctly drawn to a specified canvas. Furthermore, this module allows for some specific graphical operations such as creating rectangles, circles and other shapes, along with including appropriate fill methods. Additionally, canvases can be created and have their context specified so my solution will involve a '2d' context.

CSS: Although not a module, style sheets can be used in my program to provide a layer of animation for any entities or objects that I see appropriate for, as it has good performance on the GPU and is more compatible with more target machines as opposed to an external 3rd party module.

P5.js: P5.js is a drawing library that focuses mainly on creating and drawing graphical objects to a canvas. It includes extensive documentation and includes extra features such as sound and device integration that could be useful for future unforeseen features to improve my application. However this module does not come with built-in rendering methods, so it may hinder my application and may not be as diverse in it's application.

Test Table

Objective 1

What is being tested - success criteria	Test Data / Methodology	Expected Result
'When scenario page opened, blank page is loaded'. Additionally, ensure that the window height and width of the canvas is reset back to its default value.	Refresh the HTML page for every scenario chosen - i.e. refresh page for collision simulation, and see if blank page is loaded. Additionally, the 'reset' button will need to be tested to ensure a fresh blank page is visible to the user with default settings.	When a page is refreshed or the simulation's canvas is reset, the render canvas should be blank.
'Canvas must allow for particles to be drawn correctly'. In this case, ensuring that the desired x and y positions of the particle are rendered on the correct position on the render canvas. Furthermore, ensuring that the specified number of particles created, are all drawn to the canvas.	To test this, a number of particles must be created, with random x and y values. When the particle's are drawn to the render canvas, the number of particles must match the inputted amount, which can be done by counting the number of particles that are on screen, with the desired number.	The specified number of particles should appear on the screen, at the designated positions. No clipping should occur.
'Canvas must be cleared after every update of particles' The canvas must be cleared after every update to prevent the previous position of the particle being visible on the canvas.	This will be tested by determining if the particle's previous position is still rendered onto the canvas, and if so, then the background has not been cleared on update. Furthermore, I will test for 1 particle first, and then multiple particles afterwards, to ensure the clearing works regardless of particle count.	The expected outcome is that each particle will move across the screen, and have it's new position be visible per frame, without leaving a trace of the previous frame when the particle was drawn.

Objective 2

What is being tested - success criteria	Test Data / Methodology	Expected Result
'Have a variety of input buttons and input fields which change values.'	This user interface criteria will be tested via ensuring that each of the particle's properties have a corresponding button or input field to allow for users to enter data into.	When the user loads the page, there should be a selection of input fields available to the user for configuring the particles.
'Have settings for canvas properties.'	Ensure that the window's dimensions (height and width) can be changed via user input. To ensure this, make sure that 2 buttons are available to the user for shrinking and enlarging, along with testing their outcomes - ensure limits are put in so enlarge and shrink does not create unusable simulation.	The expected outcome, should be that the simulation will contain at least 2 input fields for changing the dimensions of the window. When clicked they should either enlarge or shrink the window, and should not go over specified limits.
'Have settings for entity creation and properties.'	Similar to the first success criteria, in that I must ensure that there is a corresponding input box for each of the particle's properties. Additionally, for particle creation, this links in with objective 1, by ensuring that when the entered amount of particles to create is processed, that the expected amount matches the actual outcome.	The expected result is that the simulation will contain input fields for changing the particle's colour, radius, mass, and x/y positions and velocities, along with the number of these particles that are to be created.

Objective 3

What is being tested - success criteria	Test Data / Methodology	Expected Result
'Allow for the number of particles to be created to be specified by the user'	The methodology to testing this task is to first ensure that an input field exists to modify the particle count, and then enter extreme or erroneous data values such as strings or floating point numbers.	If the inputted data is not erroneous or extreme, then the specified number of particles (e.g. 10) should be created and displayed on the render canvas.
'Allow for the particle's radius to be specified by the user'	Ensure that an input field first exists for allowing the user to change the radius. Next, as the input for the radius will be in the form of a slider, input validation is already handled, then toggle the slider to the desired value.	The particles that are created should be of the specified radius. Furthermore, as erroneous data is handled, extreme data values such 100 or negative values are handled by setting these upper and lower limits in the HTML slider element.
'Include limits to avoid large performance losses.'	Attempt to create the upper limit of particles, with the largest available radius. Log the results using a browser's built in profiler tool equipped on Firefox and Google Chrome. Monitor for performance drops. Also keep creating particles and ensure that the particle limit works, and only a set number of particles can be on screen at once.	The simulation should stay at an acceptable FPS in order for it to be usable such as 20+ fps.

Objective 4

What is being tested - success criteria	Test Data / Methodology	Expected Result
'Simulation loop will be created, continuously updating and rendering the particles until the simulation has been flagged as ended.'	To test this, it can be visible checked by determining if the particles move more than once, and continuously move, until their velocities are virtually 0. Furthermore, ensure that the simulation will end or reset when either an input button is pressed - reset, pause, etc. Signalling the loop to end.	The particles should constantly be moving and updating, until the simulation is reset or paused, in which either the canvas is reset, or the updating and rendering of the particles is paused.
Ensure that the particle's positions get updated and logic is applied to them.	Run the update method for a particle, print out the result to the screen, if the new results after the update have changed and are correct - e.g. the x position has increased by the specified x velocity value, then the task is functioning correctly.	The particle's should have their x and y values changed, along with boundary and collision checking taking place, however the particle should not actually render itself to the canvas.
Ensure that the particle is rendered onto the canvas screen.	Run the render method, when it is run to determine if the particle has been rendered to the screen, along with checking the radius, and x/y position of the rendered object.	The particles should be rendered at their specified x and y positions, but these locations should not change on each render method call, as logic is applied only in the update method.

Further Data

What 'data' is required?

The only source of external data, comes from connecting with p5.js's online version of its library hosted by cloudflare for the definitions of its functions used within the program. The program uses the online version of the library rather than a local storage downloaded version of the library, as the local storage option would require that each user has the library installed on the machine. Furthermore this method would prevent the user from using the application online, and would force them to run the website and the scripts from the local storage, rather than allowing it to be hosted.

Why is no further data required?

No further data is required for this application because the application stores all of its required data such as physical constants, particle objects, and input data internally. This includes, from user input such as particle radius, colour, mass, etc. Data is stored internally, as data such as constants - Gravity, Formulae, etc, can be hardcoded easily within the application. Additionally, the only required data 'externally' is from the user, who would enter the required data in order for the simulation to function correctly.

To add to this, another reason for not including any external software, or outsourcing the data to other locations is to also increase the compatibility and portability of the software. If too many external libraries are used, then it represents too many points of failure and testing. For example, new updates to include libraries and tools may result in failure of the software, but if elements are hard coded and future proofed then it reduces the risk of the software. Even more so is that, not including many external data and libraries, reduces the total size of the application.

Development

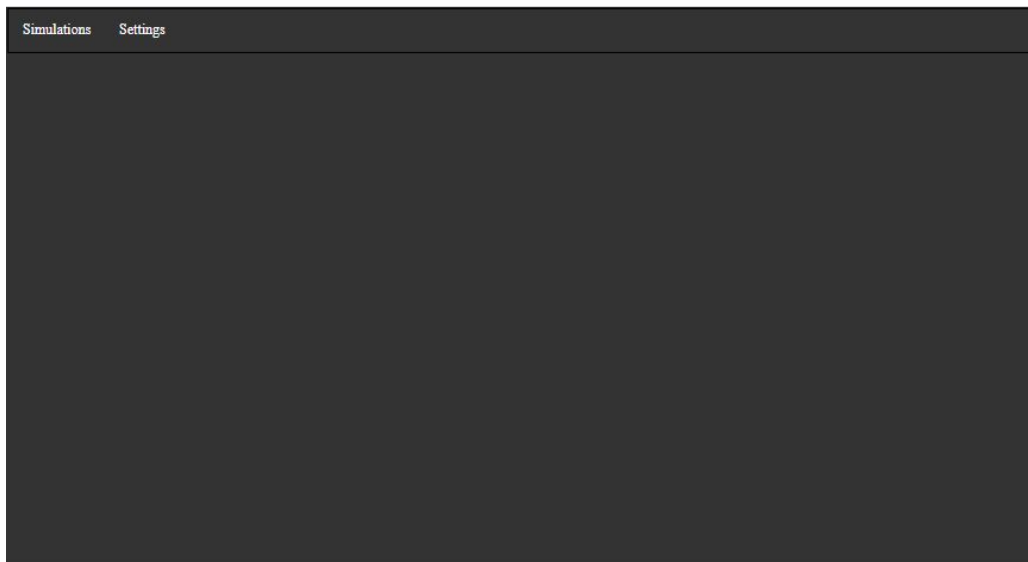
User Interface Development - Phase 1

Basic HTML and Screen Layout

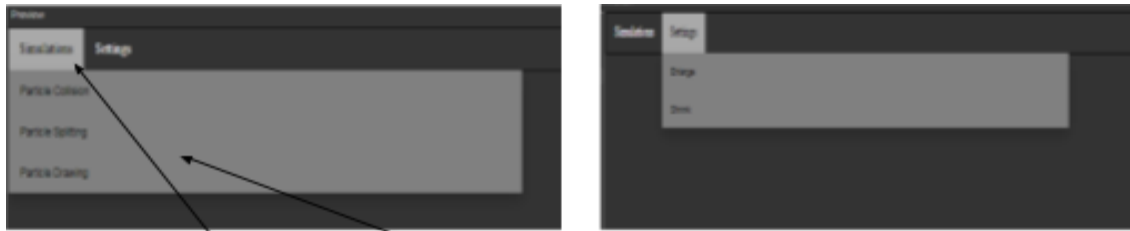
P5.js was chosen as the rendering module to use for rendering elements to the screen.

I first began by creating the basic HTML layout for my solution. This included creating a tool-bar menu to comply with my UI designs. This process used a combination of CSS and HTML code to create a menu bar that is displayed at the top of the HTML page. This menu bar would have to include elements for switching between simulations and for adjusting settings.

The main challenges were formatting challenges, such as when the html page was resized, ensuring that the elements stayed where they were before. This was tackled via using CSS properties such as 'position' and setting 'width' and 'height' to percentage based values, rather than pixels. Percentage values were used instead of using pixel values, to ensure that the size of the HTML elements and text would upscale and downscale appropriately, thereby not creating unreadable or mis sized elements/text.



Toolbar Menu



The drop down menu appears when the user's cursor is near it, displaying a list of options.

Simulations drop down a list of available simulations to choose from.

The first 'prototype' for this just included the tool-bar menu with the 'Simulations' and 'Settings' section, with a black border surrounding the two elements.

As you can see, at this stage, the page contains a menu bar, complete with dropdown menus for both settings and simulations that the user can select from. However, improvements can be made to the overall appearance and size of each element to make them more visible and user friendly.

The source code for these input elements consisted of some CSS and HTML manipulation. For example, below is the basic HTML code for creating the dropdown menu. I used a collection of CSS classes to achieve correct formatting across all similar elements, such as buttons having the same background colour, text placement, etc.

```

//The dropdown elements are contained within the HTML body section.
<body>
<li class="dropdown">
  <a href="javascript:void(0)" class="dropbtn">Simulations</a>
  //The classes are a CSS feature, used for assigning certain properties to
  Html elements such as their width, background colour, etc.
  //The 'dropdown-content' class has already been defined in a CSS file.
  <div class="dropdown-content">
    //The same process is applied for the buttons - a dropButton class is
    Used by all buttons for correct formatting.
    <button class = "dropButton"> Particle Collision </button> <br>
    //As of now, these buttons do not perform any actions - placeholders.
    <button class = "dropButton"> Particle Splitting </button> <br>
    <button class = "dropButton"> Particle Drawing </button> <br>
  </div>
  //Closing tags for the HTML interpreter to know when an element has ended.
</li>
</body>

```

I have decided to use CSS stylesheets alongside the use of classes to decrease the repetition of code in my project, along with allowing common properties such as the font size, background colour, etc to be kept consistent across all instances. For example, I use these classes mostly in ensuring that all the buttons are formatted correctly and look identical, without having to change the properties for the whole button element. These new HTML input elements do not require validation as they only perform one task when clicked on.

```

//This CSS code is applied to all <a> elements, with the 'dropbtn' specifier.
Li a .dropbtn
{
  Display: inline-block;
  Color: white;
  Text-align: center;
  Font-size: 34px;
  //Padding is applied on the outside of the element.
  Padding: 14px;
}
//This section changes the properties for the dr
Li.dropdown
{
  Display: inline-block;
}

```


Canvas

The next stage was creating the canvas element for rendering the simulation onto. This was done via using P5.js's 'createCanvas'. The canvas was created in a 'main.js' javascript file and included the methods draw and setup required by P5.js for proper identification of where the program starts.

```
function draw(){  
  // ...  
}  
  
function setup(){  
  createCanvas(400,400);  
}
```

The canvas was created in the setup function, as this is run once, or when the drawing loop is reset. It was not placed in the draw method as this method would create a canvas every time it is run.

```
//This CSS section applies to all elements that have their class set to the  
'dropButton' class. It makes the buttons have a grey fill colour, with a small  
1px border and aligns the text to the centre.  
.dropButton  
{  
  Display: inline-block;  
  Color: 'black';  
  Background-color: grey;  
  Border-style: solid;  
  Border-color: grey;  
  Border-width: 1px;  
  Text-align: center;  
  Padding: 14px;  
}  
  
//Applies to all HTML elements that are apart of the 'dropdown' menu.  
.dropdown-content  
{  
  Display: none;  
  Position: absolute;  
  Background-color: grey;  
  //Set a minimum width for this element, to ensure that this element can  
  Display the information correctly.  
  Min-width: 160px;  
}
```

From this, it has created a 400 pixel by 400 pixel canvas, displayed in the figure below (top left corner). However, one issue is that the canvas is not placed in an appropriate place, and instead should be placed nearer the centre of the screen, so that larger sized canvases can enlarge on all sides. This problem can be solved by centering the canvas by creating a function 'centerCanvas' that will adjust the width and height elements of the canvas.

```
//canvas object.  
Canvas cnv;  
//centreCanvas function responsible for centering the canvas back to the  
middle of the screen - width/2, height/2  
Function centreCanvas(width, height)  
{  
    //We are subtracting the parameter width from the current width and  
    //dividing by 2 to recentre the canvas to the middle.  
    Int x = (windowWidth - width)/2;  
    Int y = (windowHeight - height)/2;  
    //the canvas element has a built in function called 'position'  
    //responsible for placing the canvases centre point at the new  
    //x and y values.  
    cnv.position(x,y);  
}  
//windowResized is a P5.js function that when implemented, will be run  
whether the canvas or window has it's size adjusted. I am setting this  
Function to run my custom 'centreCanvas' method.  
Function windowResized()  
{  
    centreCanvas(windowWidth, windowHeight);  
}
```

Once this was implemented, when the simulation page is loaded, the canvas will now be placed in the centre. This method will also aim to be used when the user enlarges or shrinks the canvas, to keep the window centered even after adjustments.

Before



After



Sliders implemented for changing the number of particles. Sliders used for determining max and min values.

The drop down menu 'Particle' has been added along the top toolbar section.



I continued development of the UI by implementing options available to the user for manipulating particle properties including their radius, size, color, etc. These features as of yet will not have functionality to them and instead serve as a placeholder for future development. Because they are apart of the particle, I created another dropdown menu specifically for modifying particles. I added sliders for changing the values, although the values are not visible to the user yet, and will be added in the next iteration. Furthermore, the sliders are not formatted correctly, and will be fixed.

User Interface - Phase 1 Review

What was developed?

In the first phase of developing the user interface, I developed the toolbar menu consisting of dropdown selections, and options placeholders with their functionality being developed in either later stages, or a different phase. Furthermore, the user interface also consisted of developing the canvas for which future particles and elements will be rendered onto. This involved using p5.js's 'createCanvas' feature for creating a canvas to the screen. It had to be done via p5.js in order for p5.js's internal methods and setup processes to work correctly. This will become more prevalent when developing the particles and features of the simulations.

How will I improve this section in the next phase?

In the next phase of development on the UI, I will aim to implement input fields for enlarging and shrinking the canvas's size. Afterwards, I will aim to address some of the sub tasks in objective 2 including 'Create JavaScript code for taking in the values that are inputted into the buttons and input fields so they can be stored and processed.' which includes implementing input fields for changing particle settings, but this will need to be done after particles have been implemented into the game. Moreover, some additional validation and testing methods will have to be implemented to ensure that input buttons work correctly, such as buttons for adding in new particles, or inputs for adjusting certain particle properties.

Particles & Rendering - Phase 1

Particle Constructor

```
//constructor called when particle object instantiated.
constructor(radius, color, x, y, xVel, yVel)
{
    //all members are set to the constructor's arguments.
    this.radius = radius;
    this.color = color;
    this.x = x;
    this.y = y;
    this.xVel = xVel;
    this.yVel = yVel;
}
```

At this moment, the basic user interface and its features are mostly complete, leading onto the next stage - creating the particle class and rendering them to the screen. This will involve creating a particle class for which all particles have a draw and update method. The particle class contains the x and y coordinates, x and y velocities and other methods such as updating and drawing the particle to the canvas. A class is used as it allows for instances to be made of it, and neatly organises the data and operations that can be performed on particles.

At this stage, the constructor was very basic and one issue was identified. The constructor takes in too many parameters by default. One way to fix this is to use default values, and random number generation. We want to reduce the number of parameters to keep the code for particle instantiation clean, but to also allow for flexibility. For example, when instantiating a particle, I can decide whether to pass in values for specific properties (i.e the radius and colour), or let the constructor randomly generate one for me.

```

//constructor called when particle object instantiated.
constructor(
//uses default values.if no x or y value is passed, then valid x and y
//values are randomly created within the bounds of 0 - window Width - radius.
//same process applied to xVel and yVel variables.
radius=random(1,10),
colour="black",
x=random(wWidth),
y=random(wHeight),
xVel=random(-5,5),
yVel=random(-5,5))
{
    //all members are set to the constructor's arguments.
    this.radius = radius;
    this.color = color;
    this.x = x;
    this.y = y;
    this.xVel = xVel;
    this.yVel = yVel;
}

```

The constructor now generates random values between ranges. For example, if no radius is passed, then a random value between 1 and 10 is generated. As I can specify the range, it ensures that even if no parameters are passed to the particle, a valid particle is created. Moreover, the x and y positions of these values are within the boundaries of the window.

Particle Draw Method

Afterwards, I started development on the particle class's draw method. The draw method is responsible for drawing a circle shape to the screen, with said particle's x and y coordinate and radius. The algorithm for developing the draw method is simple - set the colour for drawing, and simply run p5.js built in render method to render it to the canvas. The main algorithm works

```
//The particle's draw method, used for rendering the particle to the canvas.  
Function draw()  
{  
  //Set the outline / stroke to white (255 as RGB)  
  stroke(255);  
  //Set the fill colour to the colour of the particle.  
  fill(this.color);  
  //Create an ellipse on the screen at the particle's x and y position,  
  //With the given radius.  
  ellipse(this.x, this.y, this.radius);  
}
```

No validation is required for the colour, x,y or radius as these values are only set if these values are valid. Furthermore, the values are validated in the constructor. Also as the rest of the parameters used in the other functions are constant, no validation is required.

Particle Update Method

After developing the draw method I then had to develop the update method for the particle. The update method worked simply by incrementing the accelerations to the x and y positions and then applying any external forces such as gravity to these x and y velocities to decrease them over time. This is used to act as a damping effect for the particles, making them slow over time.

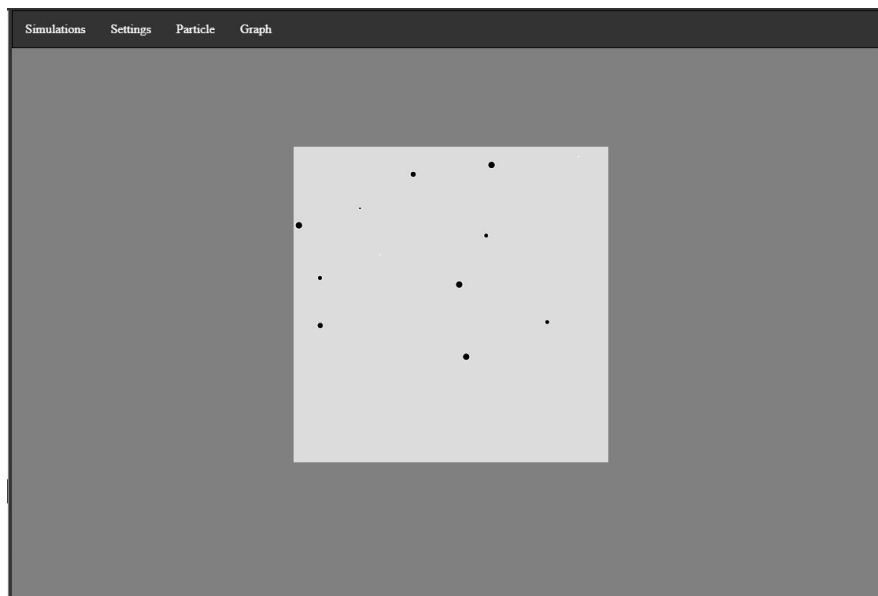
```
//Gravity set to a constant
Constant Int gravity = -0.0981;
//The particle's draw method, used for rendering the particle to the canvas.
Function update()
{
    //Apply the velocities to the x and y positions.
    This.x += this.xVel;
    This.y += this.yVel;
    //Apply gravity to the
    this.yVel += gravity;
}
```

Gravity is applied to the y velocity because gravity only acts in the vertical axis, and therefore does not affect the x velocity. Furthermore, gravity's value is set -0.0981 as the real value (9.81) would not work well, given the small size of both the particles and the canvas. As a regular HD screen is 1920 by 1080, having 9.81 would result in a very substantial effect, making the simulation end too quickly.

Create Particles Method

After this, I began implementing code to produce any number of particles, and rendering them to the screen, which will be added in for allowing the user to specify the number of entities to be created to the screen. The instantiation had to first be implemented. This was done by creating a 'createParticles' function, taking in the properties for the particles and the number of particles that we should create. The function then returns a list of particles. A list of particles is returned as a global particle list is not used, and therefore we need to share the created data with the ones created in the function, otherwise the created list will be removed from the stack, when it goes out of scope.

```
//This function will create the specified amount of particles, each having the same
properties as the values of the parameters.
Function createParticles(amount, radius, colour)
{
    //Create a temporary list, which will be returned at the end of the function.
    let particles = [];
    //We want to create a particle and push it to the list by the specified number of
    times.
    For (let i = 0 to amount; i++)
    {
        //Create a new particle, with the given radius and colour. This object will
        Be created and destroyed at the end of each iteration.
        Particle p = new Particle(radius, colour);
        //once we have instantiated the particle object, push this particle to the
        Particle list.
        particles.push(p);
    }
    Return particles;
}
```



This method involves using an array to store the particles, and in the setup method creating these particles and pushing them to the array. However, this raises some issues, for example, the code for creating an entity is quite messy and could be improved, by having the particle constructor generate these values instead. Furthermore, default values can be used on the constructor, so that specific values can be passed, but if none are passed, then a randomly generated valid value is created and assigned instead.

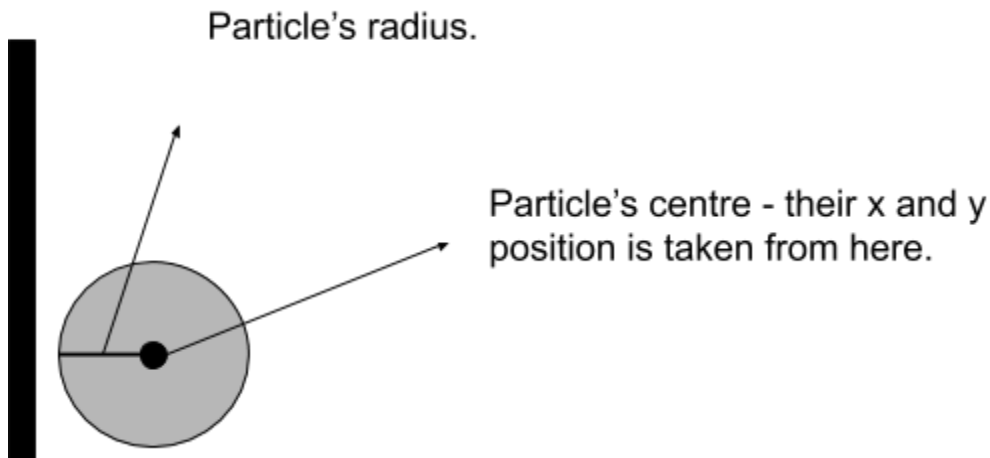
Particle Boundary Collision

One problem that I encountered during testing was that sometimes, if the particle's position was close to the boundaries, then half of it would be rendered within the canvas and the other half would be rendered outside of it, not being visible to the user.

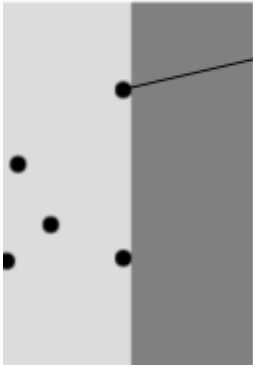


From the diagram, this particle has some of its area beyond the region of the render canvas. This could be down to 2 issues - firstly, a logical error to do with the boundary checking between particles and the canvases limits, or it could be that this particle's x and y positions were not validated, and therefore its position allowed it to be rendered outside of the canvas. Furthermore, this error seems apparent more often when the radius of the particles is increased.

I realised that the problem was due to the boundary checking between the particles. My proposed solution was to take into account the radius of the particle before determining whether it was beyond the boundaries.



The solution was not to just check if the particle's x and y were greater than or less than the window's limits, but to also account for the particle's radius. This was done by increasing or decreasing the x and y position by the radius. For example, when checking if the particle was too far to the left (less than 0), then we would do 'if (x - radius) <= 0)' which would see if the x value minus the radius would give us a value less than or equal to the x value of the boundary - (0). The other method, was then to add the radius to the x to check if the particle had gone too far to the right - 'if (x + radius) >= windowWidth)'. This logic was also applied for the window's height boundaries as well.



The particle's now collide properly with the boundaries of the canvas, done with the same settings for the particle as previously shown.

Particle - Phase 1 Review

What was developed?

In the first phase of 'Particles and Rendering' I began by developing the constructor for the particle class. This function is used in instantiating particle objects. I noticed some flaws such as too many parameters, and solved this by introducing default values and random number generation to the constructor class. Additionally, the draw and update methods were introduced and were tested to successfully draw an array of particles to the canvas, meeting some of the criteria for objective 4.

How will I improve this section in the next phase?

In the next phase I aim to work on particle collision, adding input fields that allow the user to change the properties of the particles. The changing of the particles will also involve completing success criteria for other objectives such as objective 1 and 2. Because of this, they may also be addressed in the next phase of the User Interface. Finally, I also aim to implement validation methods for the particle constructor and createParticles function.

Stakeholder feedback - Phase 1

Physics Student:

'So far the simulation contains some useful features like the basic layout and easy to navigate dropdown menu. Also, the simulation does allow for some particles to be created when the simulation starts, but the simulation is missing key particle features such as the ability to create a specified number of particles, and for the particles to react to the environment with collision. Moreover, the simulation does not allow for the canvas to resize in any way. Finally, it is unclear on the maximum number of particles that I can create in the simulation along with how strong gravity is, which could be useful information to be able to see while the simulation is running.'

Testing to Inform Development - Phase 1

Success Criteria

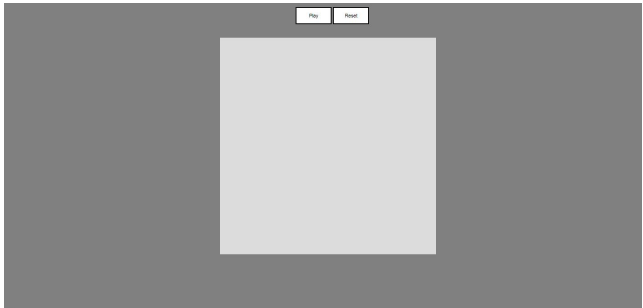
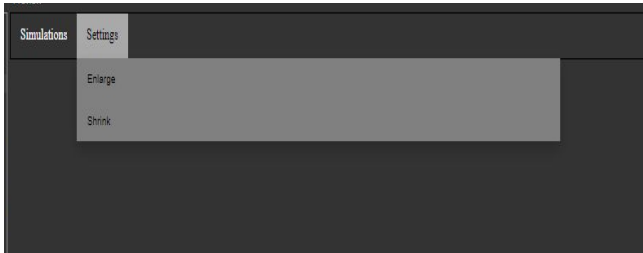
- When a scenario page is opened, a blank canvas will be displayed.
- Have settings for canvas properties.
- Screen is positioned in the centre, no matter the resolution.
- Have settings for entity creation and properties.

Objective

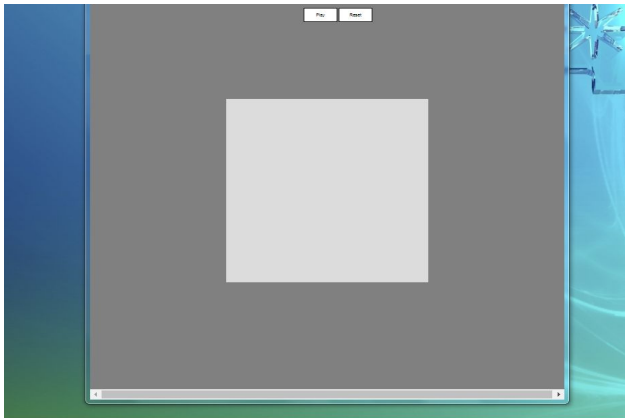
Create entity class to have individual entities be made.

Success Criteria

- Must be able to be instantiated in large quantities.
- Must be able to have collision with other objects.
- Must be able to have custom properties passed into it.
- Must be able to have collision with the boundaries of canvas.

Test	Evidence	Result
When a scenario page is opened, a blank canvas will be displayed.		There is clearly a blank canvas displayed, once the page has loaded PASS
Have settings for canvas properties.		There is a 'settings' dropdown menu including 'enlarge' and 'shrink' options. PASS

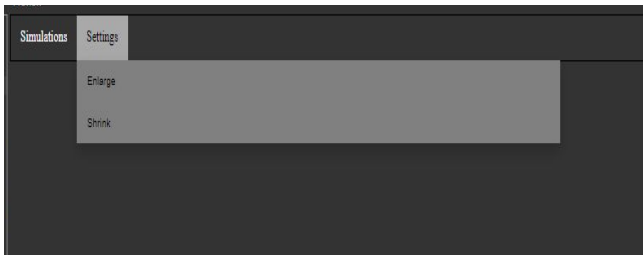
Screen is positioned in the centre, no matter the resolution.



The main rendering canvas is clearly positioned in the centre, even when the web browser's resolution is reduced.

PASS

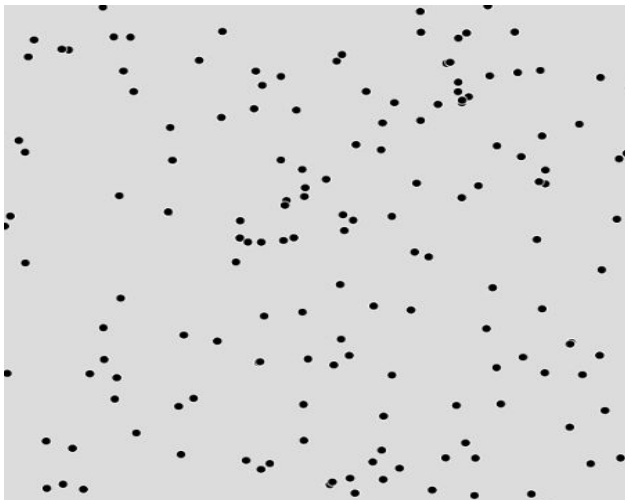
Have settings for entity creation and properties.



No 'entity' / 'particle' option available.

FAIL

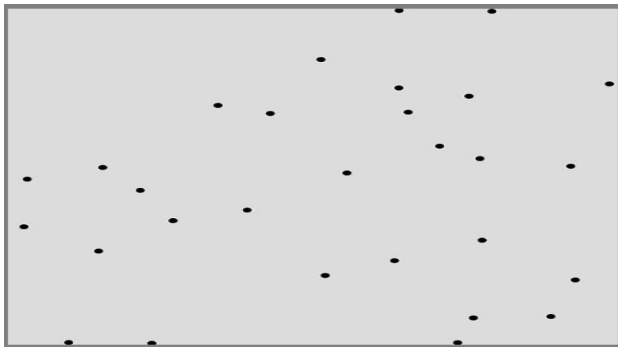
Must be able to be instantiated in large quantities.



I inputted a hard coded value into the creation loop and it resulted in each particle being rendered to a position on the screen.

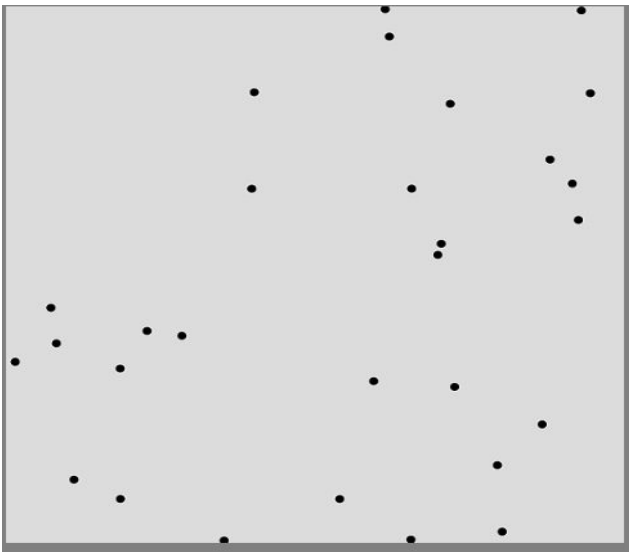
PASS

Must be able to have collision with other objects.



At this phase the particles do not collide with each other

FAIL

<p>Must be able to have custom properties passed into it.</p>	<pre> constructor(radius, color, x=random(width-radius),y=random(height-radius), xVel=random(-5,5), yVel=random(-5,5)){ this.radius = radius; this.color = color; this.x = x; this.y = y; this.xVel = xVel; this.yVel = yVel; } </pre>	<p>Currently, this criteria has been somewhat met as the constructor allows for radius and color to be passed to it. This allows for the radius and particle's colour to be changed. However, the program at this stage lacks the features to allow for users to specify it.</p> <p>PARTIAL</p>
<p>Must be able to have collision with the boundaries of canvas.</p>		<p>The particles do collide with and detect the edges of the canvas with particles bouncing off of both the vertical and horizontal axis.</p> <p>PASS</p>

Review:

In this phase , 2 tests failed, the settings for particle properties and collision with other particles. The custom properties test was a semi success, as custom values could be passed into the constructor, but no interface currently existed to allow the user to pass in custom properties. These failed tests will be addressed in the next phase of development for the user interface and the particles & rendering section.

User Interface Development (Phase 2)

In this phase of the user interface I aim to address the points that were previously mentioned such as the lack of functionality for input fields like buttons. I started out by first introducing key functions for each of the buttons - switching between simulations, and storing and processing the input values for the particle properties.

Input Processing

The first step was to develop the input processing methods for retrieving the values from HTML elements so that they can be used in creating particles and changing the simulation. I created 3 functions - 'getColour', 'getRadius', and 'getAmount', to retrieve their assigned values from the user. These functions did not require any validation checks because the HTML elements that were designated, had inbuilt 'validation' and range setting. For example, elements such as sliders, allow for upper and lower limits to be set, eliminating the possibility of extreme values. Furthermore, the type of data i.e. strings or integers, is chosen based on the values available - slider, dropdown etc. Additionally, these functions do not set global values, and instead return a copy of the local variable assigned, so that the values can be retrieved quickly .

```

//This function will retrieve the colour value from the appropriate HTML element and
return it.
Function getColour()
{
    //Creates a temporary variable which stores the particle's colour.
    //This colour value is retrieved from the HTML element with the ID 'entityColour'.
    Let colour = document.getElementById("entityColour").value;
    //Return the stored colour as the result.
    Return colour;
}

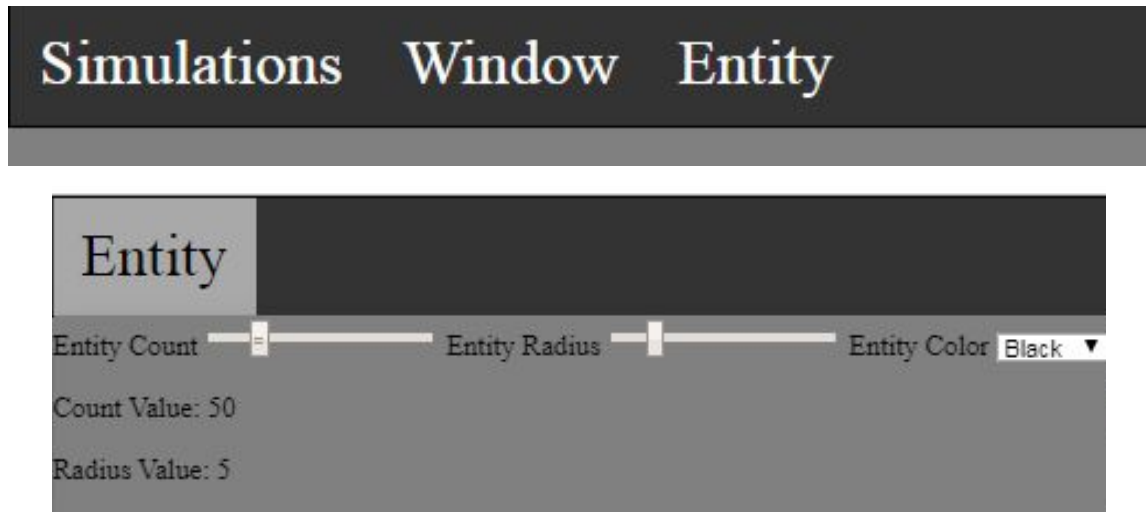
//This function will retrieve the radius value from the appropriate HTML element, and
return it.
Function getRadius()
{
    //Creates a temporary variable which stores the particle's radius.
    //This radius value is retrieved from the HTML element with the ID 'entityRadius'.
    Let radius = document.getElementById("entityRadius").value;
    //Return the stored radius as the result.
    Return radius;
}

//This function will retrieve the amount value from the appropriate HTML element and
return it.
Function getAmount()
{
    //Creates a temporary variable which stores the particle's amount.
    //This amount value is retrieved from the HTML element with the ID 'entityAmount'.
    Let radius = document.getElementById("entityAmount").value;
    //Return the stored amount as the result.
    Return radius;
}

```

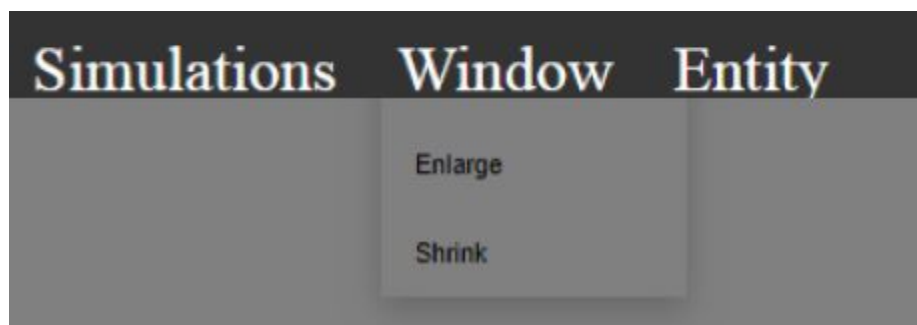
Particle and Window Options

The next feedback statement from the stakeholder was that some of the input fields were too small and their text was unable to be read clearly. This is a simple fix, and merely required changes to the CSS to produce larger width/height input elements. Furthermore, the text sizes for these input elements were also increased.



From the diagram above, I have made the dropdown selection icons much larger, as to aid in the usability of the program, and to address the stakeholder feedback from phase 1. Furthermore, I grouped the changing of the particle's properties into one section titled 'Entity' where the user can manipulate a variety of properties. However, some formatting and inconsistent use of the words particle and entity throughout the program are minor problems that can be addressed in the last phase.

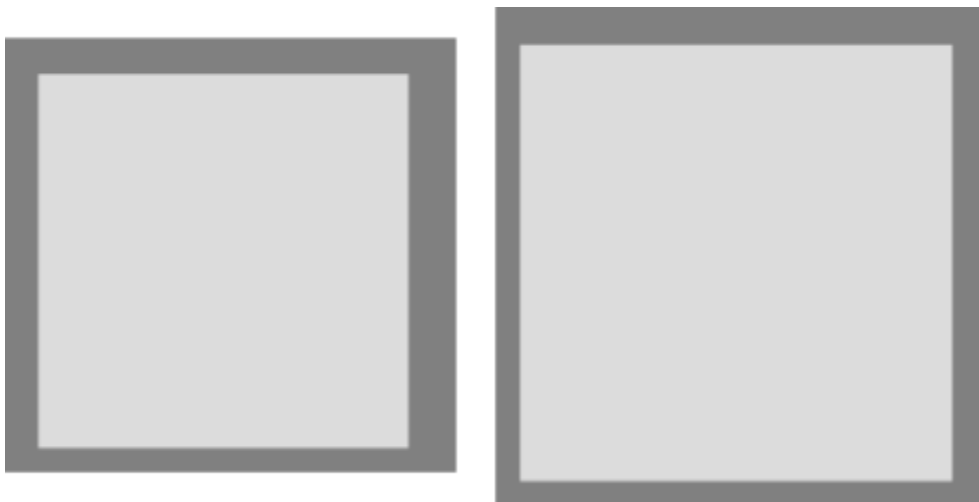
In the next phase of developing the UI i need to address objective 2's success criteria of 'Have settings for canvas properties'. In the design section I mentioned designing some input fields to allow the user to change the canvas size. To do this I will add in an 'Enlarge' and 'Shrink' button to increase the width and height, or decrease the width and height.



Right now these are just placeholders, so I will now need to implement the actual functions that will be called when these buttons are clicked. The basic steps for these are when the 'Enlarge' button is clicked, the canvas's width and height are increased by the same factor, and when the 'Shrink' button is clicked, the canvas's width and height are decreased by the same factor. However as both of these buttons to manipulate the canvas, I will instead create one function called 'resizeWindow' that will resize it based on what buttons are clicked.

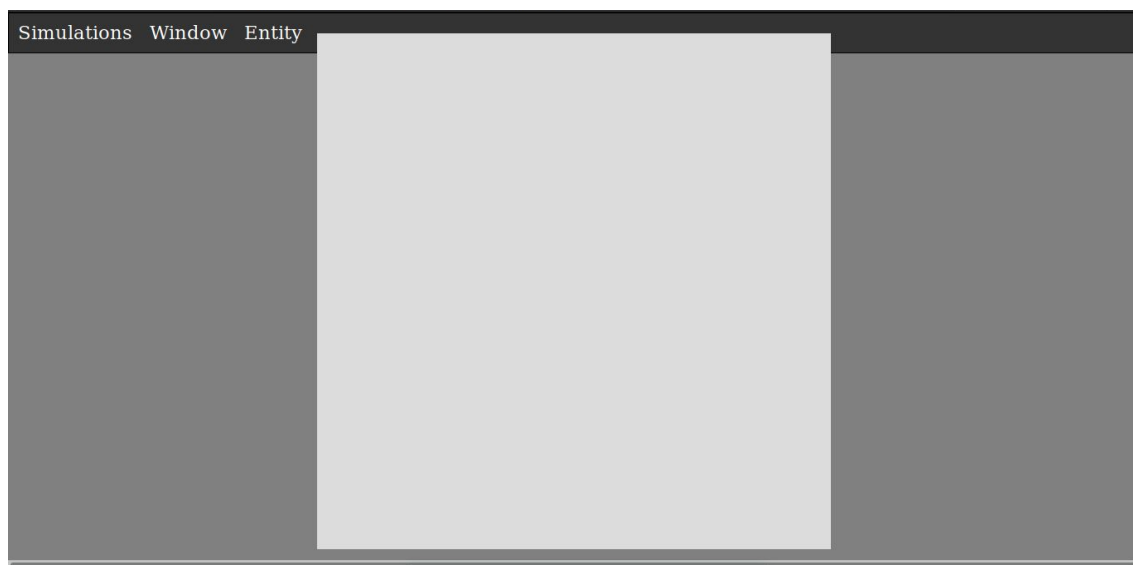
```
//This function will resize the window, it will take in the amount to decrease the
width and the amount to decrease the height by.
Function resizeWindow(width,height)
{
    //Increase the global window height and width values by the parameter amount. To
    shrink the window, negative values are passed to the function.
    windowWidth += width;
    windowHeight += height;
    //Use p5.js's in built resizeCanvas to change the size of the current canvas to
    the new values.
    resizeCanvas(wWidth,wHeight);
}
```

This created an issue however, that I could keep enlarging or shrinking the canvas forever and also introduced some non centering issues. (see below)



This issue happened because I was not recentering the canvas once I had increased its dimensions. To solve this I simply needed to add the 'centreCanvas' function that I had made in phase 1.

Even after adding this in, I now noticed that there was no limit to how big or how small the canvas could be, and therefore the user could set the canvas's dimensions to take up large amounts of space and ruin the UI. (See below). A solution to this problem is to check the current dimensions of the window and see if increasing the window by these dimensions will result in going over a specified window limit. As a limit is used we could define a constant value, however as this value will only ever be used in one function, we can just use it's raw values, rather than populating the global scope.



The coded solution now looks like this (See below)

```
//This function will resize the window, it will take in the amount to decrease the
width and the amount to decrease the height by.
Function resizeWindow(width,height)
{
    //Define the limits for a canvas being too large or too small.
    //The wWidth and wHeight are global values that keep track of the canvas's height
    and width.
    Let tooLarge = (wWidth > 650 && wHeight > 650);
    Let tooSmall = (wWidth < 150 && wHeight < 150);
    //Increase the global window height and width values by the parameter amount. To
    shrink the window, negative values are passed to the function.
    windowHeight += width;
    windowWidth += height;
    //Use p5.js's built in  resizeCanvas to change the size of the current canvas to
    the new values.
    resizeCanvas(wWidth,wHeight);
    //Recentre the canvas.
    centerCanvas();
}
```

User Interface - Phase 2 Review

What was developed?

In phase 2 of the user interface development, I addressed objective 3's sub criteria such as 'Allow for the number of entities to be specified' through the use of getter methods to retrieve the values from the HTML elements, along with addressing the criteria of 'Allow for the entities's radius to be specified'. Furthermore, limiters were implemented through the use of selecting specific HTML input fields such as sliders which can have their largest and smallest values set, to ensure that extreme values are dealt with appropriately. Moreover, this section addressed some previous aims from phase 1 to implement enlarging and shrinking input fields to change the size of the canvas dynamically.

How will I improve this section in the next phase?

Most of the designated features have been implemented, although some fixing and polish are required especially in the area of ensuring that each icon will work correctly when the web browser's display is adjusted, and also working on the final element of playing, pausing and resetting the canvas.

Particles & Rendering Development (Phase 2)

Switching between Simulations

I first had to implement the changing of simulations. To do this however, I cannot simply create 3 different HTML pages. This is because the p5.js libraries will then have to be imported for each one, and the canvas and input fields will then have to be copied over for each one. Additionally, p5.js uses both the 'setup' and 'draw' functions and looks for them when the JavaScript file containing them is loaded. Therefore my solution is to use a switch statement on a string value, holding the value of which program should be run. This allows for all simulations to occur on the same screen and canvas, without code duplication. One problem with this approach is the extra comparison checks that have to be performed on every frame, however as simple checks usually take around 1 cycle, this should not be very noticeable.

I had to create 2 variations of the setup and draw function - one for splitting and one for drawing, along with the main setup and draw functions.

```
//Global variable containing an array/list of particles.
let particles = [];
//This function will be called only once, when the collision simulation is selected.
function collisionSetup()
{
    //Retrieve the amount, radius and colour options from the inputs - previously
    mentioned functions.
    let amount = getAmount();
    let radius = getRadius();
    let colour = getColour();

    //Run the createParticles function that takes in the retrieved values to be
    processed, assign the particles to the return result of this function.
    particles = createParticles(amount, radius, colour);
}
//This function will be called only once, when the splitting simulation is selected.
function collisionSetup()
{
    //Retrieve the amount, radius and colour options from the inputs - previously
    mentioned functions.
    let amount = getAmount();
    let radius = getRadius();
    let colour = getColour();

    //Run the createParticles function that takes in the retrieved values to be
    processed, assign the particles to the return result of this function.
    particles = createParticles(amount, radius, colour);
}
```


This function simply first retrieves and locally stores the values from the desired input areas, using the previously defined 'get' methods. Additionally, no validation checks are required, as the HTML elements already use implicit validation. Finally, the function will then run the 'createParticles' function which will return a new array/list of particles, with the parameters specified and store this result in the global variable 'particles' are required, as the HTML elements already use implicit validation.

```
//Global variable containing an array/list of particles.
let particles = [];
//This function will be called once per update interval.
function collisionDraw()
{
    //Clear the canvas with a white background.
    background(255);
    //Loop through each particle in the 'particles' list.
    for(let i = 0; i < particles.length; i++)
    {
        //Call the update and draw functions.
        particles[i].update();
        particles[i].draw();
        //Loop through each element apart from the current particle.
        for(let j = 0; j < particles.length; j++)
        {
            //Check if particle[i] collides with particle[j].
            if(particles[i].collidesWith(particles[j]))
            {
                //For now reverse their velocities, no damping applied yet.
                particles[i].xVel *= -1;
                particles[i].yVel *= -1;
                particles[j].xVel *= -1;
                particles[j].yVel *= -1;
            }
        }
    }
}
```

This function is called at every update interval, and is responsible for looping through the particle array and updating/drawing each entity to the screen. The update method is applied before drawing the particle, as we want to apply forces and calculate things such as the particle's new position before it can then be drawn to the screen. Currently, no collision detection is present.

Before I can implement the splitting draw function, I will first have to implement specific checking functions for the particle class. This includes implementing the 'splitsWith' function mentioned in the design section. I first began by adding this new function to the particle class. The algorithm of the function is to see if one particle is over the top of the other. This is different to the 'collidesWith' function which checks for collision at the boundaries of beyond the particles. Along with checking the radii of these 2 particles, I must also ensure that there is enough room free within the 'Particles' list structure to add new particles to. This will involve a level of validation in ensuring that no more particles are created, once the cap is reached.

```
//This function is responsible for checking if one particle is ¾ over the other, to
initiate a split.
Function splitsWith()
{
    //See if both current particle's radius and other particle's radius are >= 5.
    Let canSplit = (this.radius >= 5) && (otherP.radius >= 5);
    Return canSplit;
}
```

Now that the 'splitsWith' function has been created, I need a way of determining if new particles can be created. To do this I created a simple 'canCreateParticles' method, which will check the current length of the particle list and take in an amount, to see if the added amount will not exceed the limit.

```
//Global variable containing an array/list of particles.
Let particles = [];
Const maxParticles = 300;
//This function is responsible for checking if the specified number of particles
Function canCreateParticles(amount)
{
    //Set the new size to the particle length + the amount we want to increase it by.
    Let newSize = particles.length + amount;
    Return (newSize < maxParticles);
}
```

This function works simply, by first getting the newSize of the amount of particles by taking in the parameter 'amount' and adding this onto the preexisting length value obtained from the 'particles' array structure.

I can now develop the 'splittingDraw' method. This method will be used in drawing the particles to the screen for the splitting simulation, but does not perform collision checks and instead performs checks for major overlap ($\frac{3}{4}$ overlap).

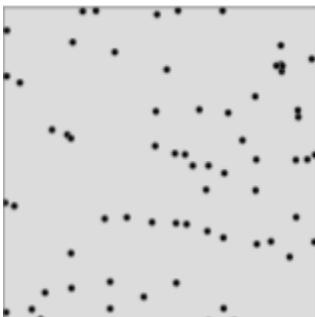
```

//Global variable containing an array/list of particles.
Let particles = [];
Const let particleMax = 300;
//This function will be called once per update interval.
Function splittingDraw()
{
    //Clear the canvas with a white background.
    background(255)
    for(let i = 0; i < particles.length; i++)
    {
        particles[i].update();
        particles[i].draw();
        for(let j = i + 1; j < particles.length; j++)
        {
            if(particles[i].splitsWith(particles[j]) &&
               canCreateParticles(1) )
            {
                //Give each particle easier to use names.
                Let p1 = particles[i];
                Let p2 = particles[j];
                //Set the new radius to be a quarter of each particle.
                Let radius = p1.radius * 0.25 + p2.radius * 0.25;
                //Set the new x value to the current particle's x + the second    particles radius
                make it appear as if it has split from the main one.
                Let x = p1.x + p2.radius;
                Let y = p1.y + p2.radius;
                particles.push(new Particle(radius, 'red', x, y));
            }
        }
    }
}

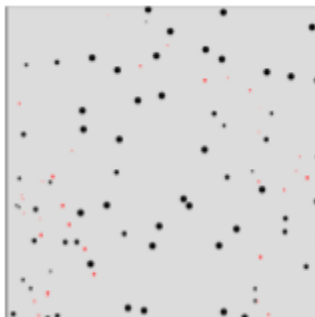
```

Evidence of switching between simulations

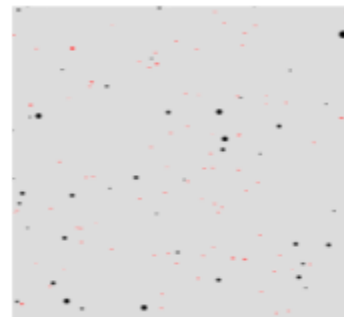
Collision



Start of splitting



End of splitting



From the diagrams above, the left side shows the outcome of the collision simulation, with the middle image showing the start of splitting, with some red particles that have appeared. Right now, the colour for the smaller created particles cannot be changed, but will aim to be added in the future.

Particles and Rendering - Phase 2 Review

What was developed?

In this phase development for particles and rendering, I mainly worked on implementing 2 simulations into the program - splitting and collision. These new simulations each worked independently of each with their own draw and setup functions. Because of this numerous success criteria were met for example Objective 4 was met as I implemented a main simulation loop, also implemented the setup and draw functions for each simulation which address the success criteria of 'Coordinate the rendering of entities in the simulation' and 'Coordinate the updating of entities in the simulation'. The splitting simulation managed to allow for smaller particles to be created once they bump into other particles - based on their size and mass, would determine the resulting particle's properties.

How will I improve this section in the next phase?

In the next section I will need to address Objective 4's success criteria of 'Check the current conditions of the simulation to determine if appropriate to end the simulation' I will also need to address the previous phase's goal of adding validation methods for the particle constructor and createParticle function. In the final phase for the user interface, I aim to address some of the objectives that have not been met, such as ensuring that the first success criteria in objective 5 is met and that large quantities of particles can be created and updated efficiently and effectively.

Stakeholder feedback - Phase 2

Physics Student:

'The new version of the simulation is now much easier to use, as the icons and buttons have become larger. Also, I like the fact that the toolbar text is much clearer and takes up a significantly larger portion of the screen, but not so much that it prevents myself from using the other features of the simulation. Moreover, the simulation now has the option to switch between simulations via a hyperlink button which is quite useful, and clear from the dropdown menu.


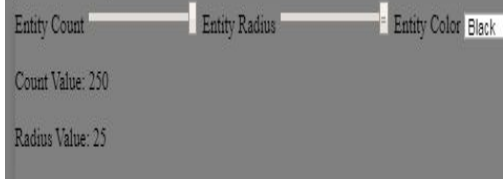
However, some features are still lacking, and I would like to see implemented in the next phase the ability for myself to be able to play and pause the simulation, essentially freezing the particles so that I can analyze them more closely. Additionally, I find that when I am using a web browser and I resize the window, it can cause some of the elements to change in size quite dramatically, to a point where the simulation is not usable, and therefore would like to see support for lower resolution windows. My final note is that I would like to be able to create particles at specified positions rather than having them be random."

Testing to Inform Development - Phase 2

Objective 3
Create entities when the simulation has started, and include settings taken from objective 2.
Success Criteria
<ul style="list-style-type: none"> - Allow for the number of entities to be specified. - Allow for the entities' radius to be specified. - Include limiters to avoid large performance losses, and to ensure the solution performs relatively consistently without crashes.

Objective 4
Create a simulation manager that will control the flow of the simulation.
Success Criteria
<ul style="list-style-type: none"> - Create one instantiation of the simulation manager at simulation startup. - Have a simulation 'loop' where the simulation continues until specific events happen. - Coordinate the updating of all entities in the simulation. - Coordinate the rendering of all entities in the simulation. - Check current conditions of simulation to determine if appropriate to end the simulation.

Objective 5
Create entity class to have individual entities be made.
Success Criteria
<ul style="list-style-type: none"> - Must be able to have collision with other objects.

Test	Evidence	Result
Allow for the number of entities to be specified.		<p>Number of entities can be specified by the user using the sliders, at a maximum of 250 values.</p> <p>PASS</p>
Allow for the entities' radius to be specified.		<p>Entities's radius can be set using the slider from the dropdown 'Entity' section.</p> <p>PASS</p>

Include limiters to avoid large performance losses, and to ensure the solution performs relatively consistently without crashes.



Each input slider has a maximum length in which it can be set to, these values are displayed as 250 for the count and 25 for the radius, clearly ensuring that no extreme values are able to be used by the user.

PASS

Create one instantiation of the simulation manager at simulation startup.

```
function setup() {  
  cnv = createCanvas(wWidth, wHeight);  
  centerCanvas();  
  if(!paused && particles.length == 0) {  
    if(start) {  
      switch(program) {  
        case "Collision":  
          collision_setup();  
          break;  
        case "Splitting":  
          splitting_setup();  
          break;  
        case "Drawing":  
          drawing_setup();  
          break;  
        default:  
          break;  
      }  
    }  
  }  
}
```

During development and design the use of a ‘simulation manager’ changed from a manager object to simply a function that coordinates which simulation is run. In this case a flag ‘program’ is used to determine which simulation is run. It successfully changes between simulations.

PASS

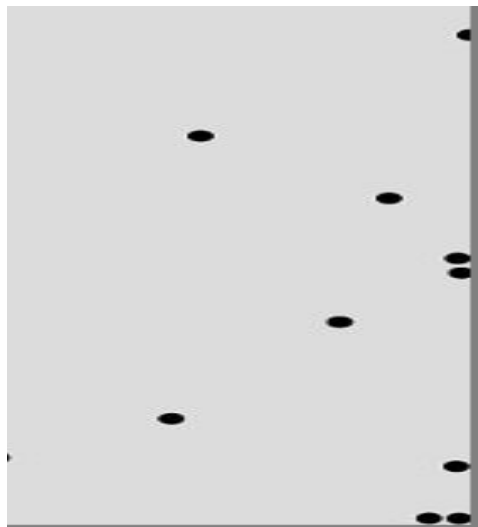
Have a simulation ‘loop’ where the simulation continues until specific events happen



This function is the collision simulation’s draw method which is run every frame or interval, it loops through the array of particles and runs the draw and update methods for each, in addition to checking collision.

	<pre>function collision_draw(){ background(220); for(let i = 0; i < particles.length; i++){ particles[i].update(); particles[i].draw(); for(let j = i+1; j < particles.length; j++){ if(particles[i].collidesWith(particles[j])){ particles[i].xVel *= -1; particles[i].yVel *= -1; particles[j].xVel *= -1; particles[j].yVel *= -1; } } } }</pre>	PASS
Coordinate the updating of all entities in the simulation.	<pre>function collision_draw(){ background(220); for(let i = 0; i < particles.length; i++){ particles[i].update(); particles[i].draw(); for(let j = i+1; j < particles.length; j++){ if(particles[i].collidesWith(particles[j])){ particles[i].xVel *= -1; particles[i].yVel *= -1; particles[j].xVel *= -1; particles[j].yVel *= -1; } } } }</pre>	<p>Updating is handled in the ‘draw’ methods for each of the simulations.</p> <p>PASS</p>
Coordinate the rendering of all entities in the simulation.	<pre>function collision_draw(){ background(220); for(let i = 0; i < particles.length; i++){ particles[i].update(); particles[i].draw(); for(let j = i+1; j < particles.length; j++){ if(particles[i].collidesWith(particles[j])){ particles[i].xVel *= -1; particles[i].yVel *= -1; particles[j].xVel *= -1; particles[j].yVel *= -1; } } } }</pre>	<p>Rendering Is handled in the ‘draw’ methods for each of the simulations.</p> <p>PASS</p>
Check current conditions of simulation to determine if appropriate to end the simulation	No play, pause or reset buttons present.	FAIL

Must be able to have collision with other objects.



Particles in this phase clearly interact with both the boundaries of the canvas and other particles on screen.

PASS

Review:

In this phase, I managed to pass mostly all of the tests set out, and the criteria that was aimed to be developed. The only test that failed was the 'Check current conditions of simulation to determine if appropriate to end the simulation'. This will be addressed in the next phase.

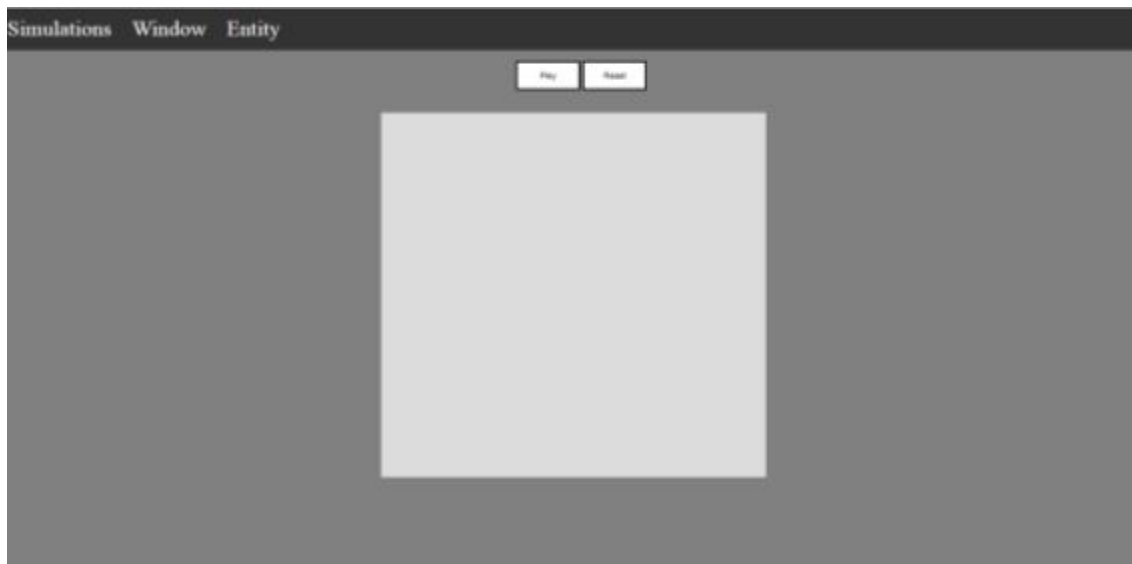
User Interface Development (Final Phase)

Canvas Manipulation - Play, Pause & Reset

In the final version of the user interface development, I aim to implement the 'play', 'pause' and 'reset' features addressed in the review section for phase 2, and to also address the success criteria of 'Contain time and canvas manipulation buttons such as play, pause and reset'. Therefore, I began by creating the HTML elements with their specific tags. I used buttons as my input field, so that I could bind an 'onClick' event to the function that I will create later. In order to keep track of what state the simulation is in, I needed to add another global variable to my list - paused. This is a boolean value that is used to signal whether the simulation has currently been paused, and will be useful in determining if certain lines of code can be run or not.

```
//Give both buttons an ID so that we can reference to them in the JavaScript functions.  
//I have linked the onclick event to 2 functions, that have not been defined/declared  
yet - enablePlay & resetSimulation.  
<button id = "playPauseButton" onclick="enablePlay();" value="Play">Play</button>  
<button id = "resetButton" onclick="resetSimulation();"> Reset </button>
```

Result of code included in the main .html file



When writing this code, I decided to instead of having both a play and pause button, that one button be used. Because of this, I will use 2 states - Play and Pause, and have this be the value of the button element. This value will be used to decide what operation, pause or play, to run when the button is pressed. This will also require my JavaScript code to change the text displayed in the boxes, when the button is clicked and its properties changed.

```

//This function is responsible for changing the state of the buttons and running
appropriate canvas methods for pausing or playing the canvases elements.
Function enablePlay()
{
    //Store the element at the playPauseButton id;
    Let elem = document.getElementById("playPauseButton");
    if(elem.value == "Play")
    {
        Start = true;
        //Change the value of the element to pause and pause.
        elem.value= "Pause";
        elem.innerHTML = "Pause";
        Paused = false;
        //Setup will run the main setup function again.
        setup();
        //Loop will re-enable the looping of the draw functions.
        loop();
    }
    Else
    {
        Start = false;
        //Change the values of the
        Elem.value = "Play";
        elem.innerHTML = "Play";
        Paused = true;
        //Will disable the looping of the draw functions.
        noLoop();
    }
}
}

```

Before I could start work on the reset button function, I first needed a function that would clear the particle array, and to do this would simply need to pop every element from the array. So I started by creating a 'removeParticles' utility function. It will utilise the 'pop' function which will remove the element at the front of the array.

```

//The global array storing the particles.
Let particles = [];
//This function works by looping through the 'particles' array and popping each element
from the array.
Function removeParticles()
{
    //Keep popping until the particle length is 0.
    while(particles.length > 0)
    {
        //run the pop function on the particles.
        particles.pop();
    }
}

```

For the reset button I used a variety of built in P5.js functions such as `noloop` and `setup` to change the flow of the program. The 'resetSimulation' code for that button simply works by first getting the reset button's element, then checking if it's value is set to 'Pause' or 'Play' and performing the necessary actions.

```

//This function is responsible for checking the status of the reset button element,
then removing all particles from the screen, and refreshing the simulation. - similar
to a refresh button.
Function resetSimulation()
{
    //Store the element at the playPauseButton id;
    Let elem = document.getElementById("playPauseButton");
    if(elem.value == "Pause")
    {
        enablePlay();
    }
    Paused = false;
    removeParticles();
    setup();
    loop();
}

```

As a result of this, when the user presses the play button the button will switch to the paused state. However, even though the flags have been set, I need to implement an if statement in the main file when the setup files are being checked.

```

//Global variables used in this function and a variety of other functions.
Let cnv;
Let wWidth = 300;
Let wHeight = 300;
//This function is responsible for checking if the specified number of particles
Function setup()
{
    //Assign canvas object o new object.
    Cnv = createCanvas(wWidth,wHeight);
    centerCanvas();
    //Only run the setup functions if the simulation is not paused and there are
    No particles.
    if(!paused && particles.length == 0)
    {
        if(start)
        {
            switch(program)
            {
                Case "Collision": collision_setup(); break;
                Case "Splitting": splitting_setup(); break;
                Default: break;
            }
        }
    }
}

```

The if statement is dependent on both the paused flag and the length of the particle array, as we don't want to rerun the setup functions which would rerun adding new particles to the particle array. Rerunning this function would create a performance cost for the function, potentially hindering the performance at startup. However this performance may not be so noticeable as this function is only run once, when a particular simulation is started, unlike the draw function which is run on every update.

User Interface - Review (Final Phase)

What was developed?

In the final phase for UI development, I added in play, pause and reset buttons for allowing the user to manipulate when the simulation can be run and when it can be paused. Because of this I had addressed the success criteria in objective 2 - 'Contain some time and canvas manipulation settings such as play, pause and reset' and to 'Contain settings for entity creation and properties'. Furthermore, I also addressed the issues raised in the previous phase such as increasing icon size, and ensuring that elements scale correctly, when the web browser's window was resized.

What targets were not met?

Most, if not all criteria relating to the user interface was successfully implemented, such as objective 2's criteria for input fields to change certain entity properties such as the amount of particles to create and what colour, and radius size they have. However some features may be missing, such as possibly adding more customisation to the particle creation - such as allowing the user to specify the coordinates at which each particle is rendered at. Additionally, the website's overall appearance may be slightly dull and lack a variety of colours.

Particles & Rendering Development (Final Phase)

Addressing Stakeholder Feedback

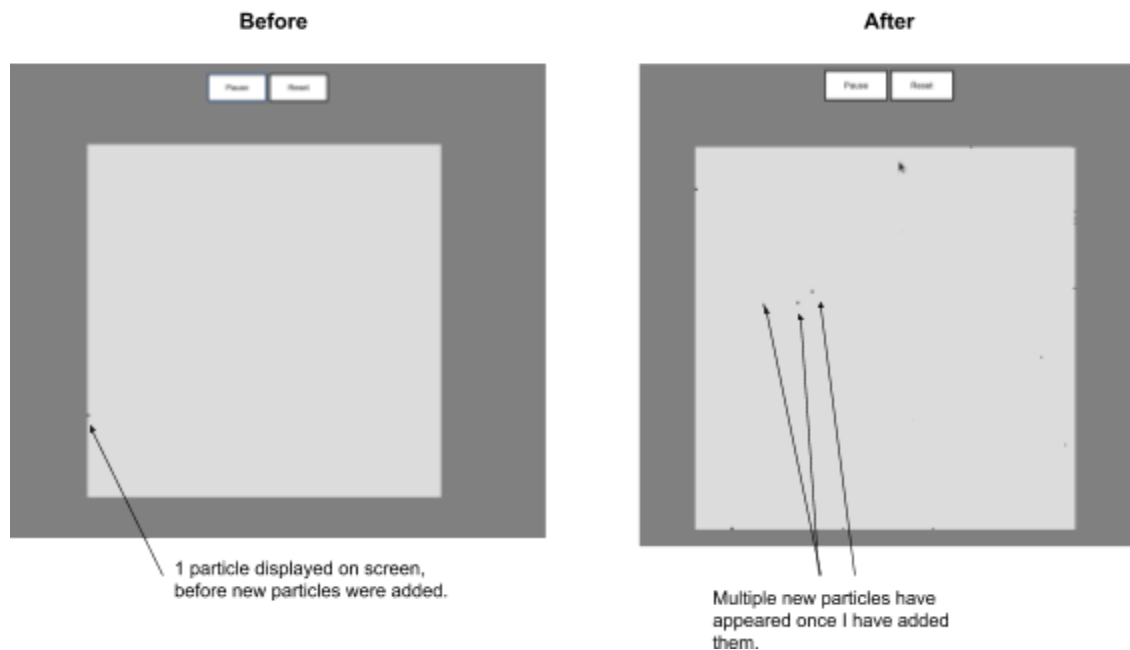
In the final phase of particle and rendering I aim to polish up the algorithms, and ensure that their performance is of an appropriate level. Furthermore, I aim to ensure that all simulations function correctly, and to check for logical errors such as to do with the collision detection between particles. This is going to be particularly addressed when I aim to address the success criteria (objective 5) - 'Must be able to be instantiated in large quantities efficiently'. Finally, the previous phases's improvement goal to address objective 4 has been successfully addressed in the 'User Interface Development (Final Phase)' section above.

The first step in the final phase was to take on board the feedback from the previous phase and the most notable was to add the ability for the user to place particles at particular places. Rather than using input fields, and having the user enter the x and y values of each particle individually, which would be of great annoyance for the user, I decided to use the mouse to 'draw' particles.

To implement, I used p5.js built in 'mouseX' and 'mouseY' variables, along with the boolean flag 'mouseIsPressed'. The flag variable will be set to true whenever the mouse is clicked onto the canvas. Because of this, along with the mouseX and mouseY variables I can create new particles at the position of the mouse, allowing for the user to create particles at custom locations (within the canvas of course).

```
//I only added the new lines to this existing function rather than rewriting the entire
Function again.
Function collision_draw()
{
    //Check if the mouse has been pressed - check the 'mouseIsPressed' flag
    if(mouseIsPressed)
    {
        //check if we can create any more particles - call 'canCreateParticles'
        //Passing in 1 as the parameter as we want to create one new particle.
        if(canCreateParticles(1))
        {
            particles.push(new Particle(5,"black",mouseX,mouseY));
        }
    }
    //Rest of 'collision_draw' code
}
```

This code was then repeated for the 'splitting_draw' function. The evidence for this is shown by first using only one particle to appear on the canvas, and then using the draw feature to then create new particles. We can visually test if their position is correct, by seeing if the particle is created at the cursor's position.



From the above diagram, before I only had one particle, but once I clicked on the canvas, multiple more particles appeared. From testing around, I identified a few problems. First, the user can draw particles onto the canvas, even when the simulation has not been started. Secondly, if I changed the particle's colour, or radius, then the actual values would not be seen to change when I next clicked onto the canvas. Finally, there is no way for the user to disable or enable this option.

Luckily these problems have simple solutions. The first problem is quite easy to solve as I will only have to implement a simple condition checking to see if the simulation has started in order for particles to be created. I will use condition checking, as flags have already been implemented from earlier stages of the development such as the 'paused' and 'start' flags. The second problem is also easily fixable as I will only have to call the 'getRadius' and 'getColour' functions on every draw update. The final problem will involve me adding a button to enable or disable the drawing mechanic. This feature will be done first.

The first step was to implement the button for enabling or disabling the option to draw particles onto the canvas. To do this I had to implement another button element into the HTML page.

```
//Create a button element in the page, and add the 'dropButton' as a class so that it's properties
such as colour, and text are formatted and consistent with the other input elements and buttons.
<button id = "enableDrawButton" class = "dropButton" onclick="enableDraw();" value = "False"> Enable Draw</button>
```

Now that the button element has been implemented I now need to implement the 'enableDraw' button that I have predefined as the function that will be called when the button is clicked. This function will work by first checking the value of the 'enableDrawButton' to see if it is 'False' or 'True' and then doing the appropriate action, similar to the 'enablePlay' function.

```
//This function is responsible for checking the value at the draw button and running the
Appropriate actions afterwards.
Function enableDraw()
{
    //Store an object copy of the user interface element with it's values.
    Let elem = document.getElementById("enableDrawButton");
    //Compare if the element's value is equal to True, if so then we disable it.
    if(elem.value == "True")
    {
        //This global variable determines if the draw ability is available or not.
        canDraw = false;
        //Set the value to false, so next time it is evaluated the other outcome will
        occur.
        Elem.value = "False";
        //Change the display text of the element so the user knows what the buttons now
        does.
        Elem.innerHTML = "Enable Draw";
    }
    //If elements value is false, do these operations.
    else
    {
        canDraw = true;
        Elem.value = "True";
        elem.innerHTML = "Disable Draw";
    }
}
```

I solved this problem by first having another flag called 'canDraw' which is similar to it's function with the 'paused' variable, in that it is used to determine if the user can draw particles to the canvas screen. Furthermore, the solution works by comparing the value at the button element. If its value is set to true, and then the button is clicked, then we know that the user wanted to disable the drawing. As the initial element's value is set to false, when it is first pressed, it will run the other code, therefore enable draw and set the text to 'Disable Draw'.

Although I have implemented the draw function, these values such as 'canDraw' will now have to be applied to each draw function for the collision and splitting simulation.

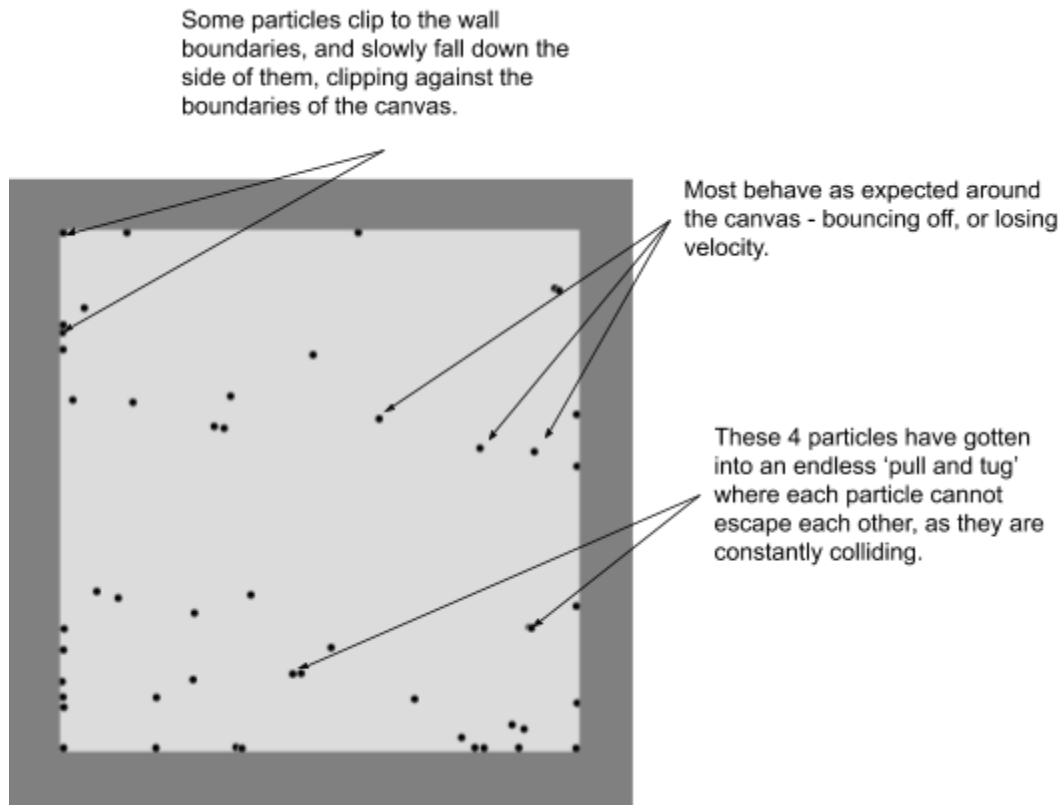
To do this I simply need to add these values to the condition checking when the mouse is pressed, so I need to remedy earlier code.

```
Function collision_draw()
{
    //Code from previous section - getter methods for dynamic retrieval.
    Let radius = getRadius();
    Let colour = getColour();
    If(mouseIsPressed)
    {
        //Only create new particles if we can create any more, and the 'canDraw' and
        'Start' global variables are set to true.
        if(canCreateParticle() && canDraw && start)
        {
            particles.push(new Particle(radius,colour,mouseX,mouseY));
        }
    }
    //Rest of 'collision_draw' function.
}
```

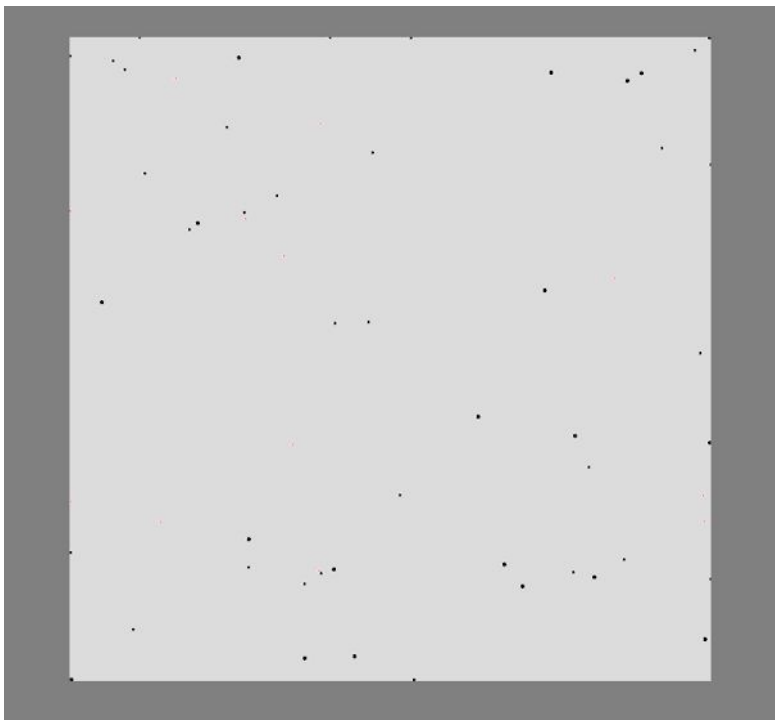
Optimising and Bug Fixing

Now that I have addressed the feedback from the stakeholder, I can now focus on decomposing the efficiencies of my main algorithms such as the collision checks, and splitting checks, along with other mathematical operations.

My first step is to ensure that the collision detection is working properly first of all, and that no bugs are present. When testing I found out that some of the particles may get 'stuck together' especially with each other and along the boundaries of the canvas (See below)



My first thoughts on what could be causing this problem, is the collision detection algorithm that I implemented, which only reverses the x and y velocities on collision, with no damping effect, so I first took a closer look at that.



The clipping issue is no longer as prevalent, this happened because of the way that I was applying forces to the particles, and the effect that I put on the particles when they collide.

Time Complexity

The collision detection algorithm has a Big O of N^2 , meaning it is quadratic and will scale tremendously based on the number of particles present. As my maximum number of particles is set to 300, $300^2 = 90,000$. This is a vast amount of operations, to be performed for every single particle, on every update, so this in reality is $90,000 * 300 = 27,000,000$ iterations done every update, with 300 particles on screen. This number is far too massive, and results in clear performance decreases when the simulation reaches a particle count above 100. One of my success criteria is to 'Have particles be produced in large quantities, efficiency', which clearly has not been addressed yet.

Before I attempt to optimise this algorithm, I must first set a reasonable goal for the maximum number of operations done per frame, with 300 particles.

From my research a very low end CPU, the 'Intel Pentium 967', has a clock speed of 1.3GHz. If I take this to be a model of what a low end computer's CPU will look like, then we can say that this processor can handle 1,300,000,000 CPU cycles per second, without factoring in the number of cores the CPU has, and the performance of the web browser that the simulation will be run in.

For my program, I would aim that at the most, 40% CPU usage, solely from my simulation will be seen as the maximum reasonable CPU amount. This means that our software will be limited to $0.40 * 1,300,000,000 = 520,000,000$ *cycles*

Now we must find out the number of clock cycles taken by my collision algorithm. To do this, I counted every operation as a single clock cycle. This included indexing, any mathematical operation, creating variables and function calls. Additionally, I counted the clock cycles from other methods such as the 'update' and 'draw' methods. However, the estimated number of clock cycles may be on the lower end, as it does not account for built in methods that are implicitly run, along with any built in P5.js methods such as 'fill' and 'stroke'.

```

//Global variable containing an array/list of particles.
Let particles = [];
//This function will be called once per update interval.
Function collisionDraw()
{
    //Clear the canvas with a white background.
    background(255);
    //Loop through each particle in the 'particles' list.
    for(let i = 0; i < particles.length; i++)
    {
        //Call the update and draw functions.
        particles[i].update();
        particles[i].draw();
        //Loop through each element apart from the current particle.
        for(let j = i + 1; j < particles.length; j++)
        {
            //Check if particle[i] collides with particle[j].
            if(particles[i].collidesWith(particles[j]))
            {
                //For now reverse their velocities, no damping applied yet.
                particles[i].xVel *= -1;
                particles[i].yVel *= -1;
                particles[j].xVel *= -1;
                particles[j].yVel *= -1;
            }
        }
    }
}

```

Each iteration is one clock cycle, from earlier we calculated that our collision algorithm does 27,000,000 iterations so 27,000,000 clock cycles.

A comparison function is 1 clock cycle, along with the 2 indexing, so 3 clock cycles (not including the 'collidesWith' function call)

4 mathematical operations, and 4 indexes of the particle list, so 8 cycles

This leaves this calculation with a total cycle count of:

$$\text{Cycles Per Iteration} = (4 + 4) + ((1 + 1 + 1) * 2) + (2 + 1) + (2) = 19$$

Cycles for External Methods =

$$\text{Update (Worst Case)} : (1 + 1 + 1 + 2) + ((2 + 1 + 1) * 2) + 2 = 13$$

$$\text{Draw} : 3 + 5 = 8 \text{ Cycles}$$

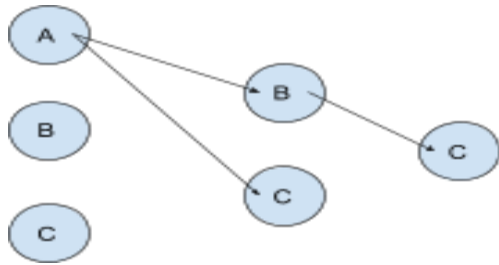
$$\text{CollidesWith} : 1 + (1 + 1 + 1 + 1) + 1 + (1 + 1 + 1) + 1 + 1 + 1 = 12 \text{ Cycles}$$

$$\text{Total Cycles Per Iteration} : 8 + 12 + 13 + 19 = 42 \text{ Cycles}$$

$$\text{Total Cycles Per Frame (Worst Case)} : 42 * (300 * 300 * 300) = 1.134 \times 10^9$$

$$\text{Fraction of CPU Cycles} : 1.134 \times 10^9 / 1.3 \times 10^9 = 0.872 / 87.2\%$$

From the calculations above, we can see that currently our collision detection algorithm will use up 87.2% of the clock cycles for the Intel Pentium 967, more than double the amount I want it to be at. So if we want to reduce it we will have to reduce it to Big O(N), but this is unreasonable, as we need to check every particle, so without any collision checking of other particles, we have a Big O of N. Therefore, our best way to optimise this algorithm is to reduce the checks with other particles. We can do this by not checking the particles that have already been checked. If we simply checked each particle with every other particle, then for 3 particles, (A B C) we would perform 3 x 3 comparisons - so 9 total.



One we can optimise this function is by reducing the number of comparisons. If we check A with B and C, we no longer need to check B with A, or C with A. Then when we check B we perform a check on B and C, therefore we can skip C. This leaves us with a total of 3 comparisons, reducing the comparison count by more than a $\frac{2}{3}$.

```

//Global variable containing an array/list of particles.
let particles = [];
//This function will be called once per update interval.
Function collisionDraw()
{
    //Clear the canvas with a white background.
    background(255);
    //Loop through each particle in the 'particles' list.
    for(let i = 0; i < particles.length; i++)
    {
        //Call the update and draw functions.
        particles[i].update();
        particles[i].draw();
        //Loop through each element apart from the current particle.
        for(let j = i + 1; j < particles.length; j++)
        {
            //Check if particle[i] collides with particle[j].
            if(particles[i].collidesWith(particles[j]))
            {
                //For now reverse their velocities, no damping applied yet.
                particles[i].xVel *= -1;
                particles[i].yVel *= -1;
                particles[j].xVel *= -1;
                particles[j].yVel *= -1;
            }
        }
    }
}

```

Now in the worst case scenario, it is now:

$$300(N) * (N - 1)$$

Where N is decreasing by 1 per iteration. (300,299,298,...)

For the worst case this will give us a total amount of 45,150, iterations, rather than 90,000. This is a 50% reduction (45,150 / 90,000) in the iteration amount, which will reduce our cycles by 50% - so total cycles now = :

$$\text{Total Cycles Per Frame (Worst Case)} : 1.134 \times 10^9 * 0.5017 = 5.689278e + 10^8$$

We are now only using 570MHz of the low end CPU, equal to (0.57 / 1.3) = 38.4%. We can now say that the collision algorithm has been properly optimised and I have met the initial goal of having the worst case run at 40% CPU usage.

A similar algorithm is also applied to the splitting algorithm, and can therefore be introduced to that as well. Similar results will be visible, as both the splitting and collision algorithm will loop through each untouched particle.

Space Complexity

Now that I have looked at the time complexity of my main algorithms, I must now determine the space complexity of the simulation, during a worst case scenario. To do I will have to estimate the memory size for one particle, storing its 6 properties, and 6 - 8 methods.

First, I will break down the collision algorithm, analysing it's space complexity this time. This will include any indexing, addition, and creation of variables.


```

//Global variable containing an array/list of particles.
Let particles = [];
//This function will be called once per update interval.
Function collisionDraw()
{
    //Clear the canvas with a white background.
    background(255);
    //Loop through each particle in the 'particles' list.
    for(let i = 0; i < particles.length; i++)
    {
        //Call the update and draw functions.
        particles[i].update();
        particles[i].draw();
        //Loop through each element apart from the current
        particle.
        for(let j = i + 1; j < particles.length; j++)
        {
            //Check if particle[i] collides with particle[j].
            if(particles[i].collidesWith(particles[j]))
            {
                //For now reverse their velocities, no damping
                applied yet.
                particles[i].xVel *= -1;
                particles[i].yVel *= -1;
                particles[j].xVel *= -1;
                particles[j].yVel *= -1;
            }
        }
    }
}

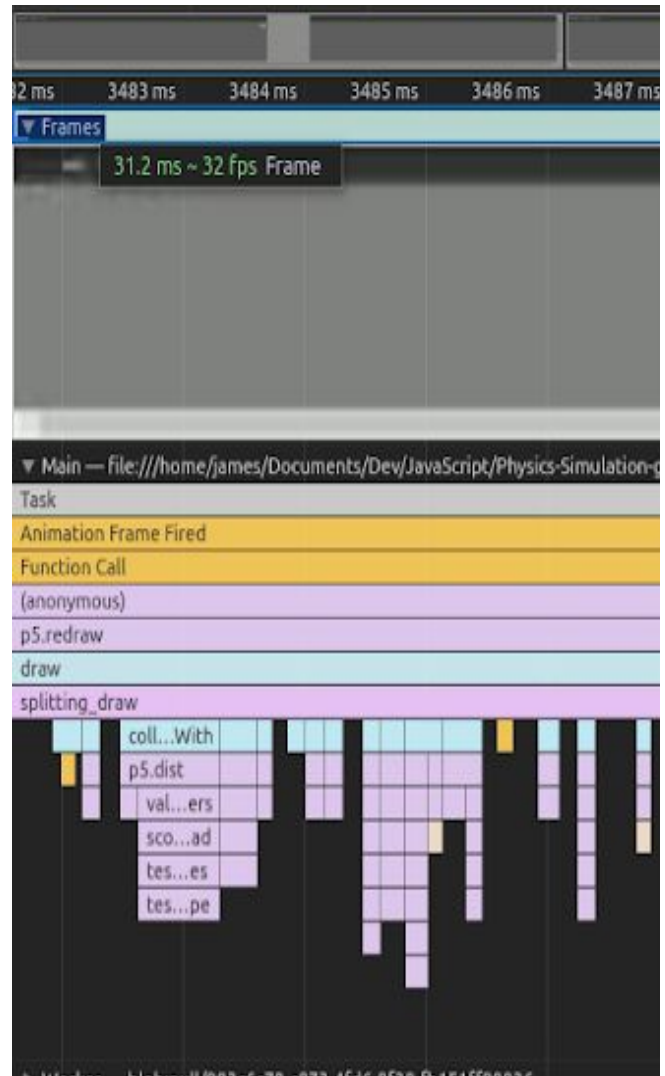
```

- 2 for the i and j variables.
- Each Particle has a 6 fields, so we can say that they have 6 space complexity.
- This is $12 + 2 = 14$ space complexity per iteration. But only 12 per iteration, as the i and j are fixed.

Performance Profiling

If we use a performance profiler tool in Firefox we can see the memory usage of this application. I will test this, like the others, using the worst case scenario possible of 300 particles. I will use a 'heap snapshot' which will take an image of what memory is being used at one second, which may cause invalid results, but it will still provide an accurate enough story on how my program uses memory

When testing the FPS for the simulation for the worst case scenarios, I got varying performances depending on the browser. For example, when testing in Firefox I got an average of 5 fps, but this was significantly different using a chromium based browser, where I achieved 32 FPS. One cause for this could be that the Firefox browser has been adapted more, with setting changes and extensions, which could result in this slowdown, whilst the chromium browser was freshly installed. Both browsers perform optimally for around 150 particles, but Firefox's performance begins to drop quite significantly, but upto 200, the frame rate still remains at about 20fps, which is still usable.



As a result of this, I will ignore this testing, and regard Firefox's performance as anomalous. Furthermore, Google Chrome has a market share of 66.64%, while Firefox has a market share of 8.36% (Market Share Reference 1)

Particles & Rendering - Review (Final Phase)

What was developed?

In the final phase of development for particles and rendering, I mainly focused on bug fixing the simulation, ensuring that collisions and intended effects worked correctly. I did this by fixing some clipping issues, and some logical errors to do with the ways in which I was simulating collision. After this, I mainly focused on addressing and analysing the performance of my solution. I first analysed my main algorithm of collision (also applies to other simulations), determining its time complexity and estimating the total clock cycle. Moreover, I chose a low end CPU and made an aim to utilise only around 40 - 50% of its total available clock cycles per second. I managed to reduce it from 1.13GHz usage. to 570MHz by simply changing the time complexity.

After this, I worked on the space complexity, and used performance profilers in both a Chromium browser, and Firefox. Both of these browsers showed a very acceptable result, with my program only using around 8 megabytes of total memory, and 2.1 megabytes for 300 particles. I finished this section off by looking at the average FPS shown. I got very bad results from the Firefox browser of 6 fps (worst case), but very good results in the Chromium browser of 32 fps (worst case). Because of this, I marked the Firefox results as being anomalous, and disregarded them.

What targets were not met?

Some features that did not get implemented in the final version of the software include, a wider range and variety of simulations, as this software is limited only to particle collision, and simulating particle splitting (loss of mass), which also provides a limitation to the software, only covering the essential features, and now delving into other territories of types of simulations such as fluids, or more in depth simulations of energy and the change of energy throughout an interaction. Finally, some features that I addressed in my analysis stage, such as a download button,

are not present. These limitations will be addressed and discussed in further detail in the evaluation stage - limitations, why they were left out, etc.

Stakeholder Feedback - (Final Phase)


Physics Student:

‘Overall, I am quite happy with the finished product, it allows me to visualise collision events, and a particle’s x and y velocities much more easily, improving my understanding of mechanical physics subjects such as momentum, and energy. Also, I am also pleased with the simplicity of the program, the UI is very easy to use and self explanatory, at no point when using it did I find it difficult to find out where a feature that I wanted was located, be that changing the particle’s colours, or size, or enlarging or shrinking the canvas size. Furthermore, I enjoy that the simulation is quite customisable with the canvas size being able to be enlarged or shrunk to fit my needs, kinda simulating a chamber, with the decreased chamber size resulting in a higher pressure.’

‘Although the simulation is very simplistic and easy to use, I find that it lacks a variety of simulations, and only specialise in one particular area, which isn’t done to a massive degree of detail. Because of this, I see this software as being more useful in combination with other software tools and websites. Additionally, there is no feature that allows me to track the properties of any of the particles, making taking notes and observations of the simulation difficult. Also, I think that the fact that I cannot download the simulation as a runnable executable a slight annoyance as I cannot access it offline, and have to instead result to the GitHub page that it is hosted on. Because of this, It has limited the portability of the software, but not to a great extent, and is still very useful nonetheless.’

Testing to Inform Development - Final Phase

Objective 2
Create a variety of buttons and input fields for users to customise the simulation.
Success Criteria
<ul style="list-style-type: none"> - Have a variety of buttons near the canvas which change values. - Connect the buttons via JavaScript so data can be processed. - Have settings for canvas properties. - Contain some time and canvas manipulation settings such as play,pause and reset. - Contain settings for entity creation and properties.

Test	Evidence	Result
Have a variety of buttons near the canvas which change values.		<p>The simulation contains a variety of input buttons present, enabling the user to manipulate particle properties, the windows size, and playing and pausing the simulation, along with switching between simulations.</p> <p>PASS</p>

Evaluation

Testing to Inform Evaluation

In the 'testing to inform development' sections I managed to provide evidence of testing at the end of each phase, with failed and passed tests both shown. These failed tests were addressed in the latter phases. During the evaluation section, I will perform post solution testing and address each objective in the test table labelled in the design section.

Objective 1

What is being tested - success criteria	Test Data / Methodology	Expected Result	Success Criteria Met?
'When scenario page opened, blank page is loaded'.	Refresh the HTML page for every scenario chosen.	When a page is refreshed or the simulation's canvas is reset, the render canvas should be blank.	This criteria was fully met, the page loaded with a blank canvas, with all HTML elements in their correct positions. PASS
'Canvas must allow for particles to be drawn correctly'.	To test this, a number of particles must be created, with random x and y values. When the particle's are drawn to the render canvas, the number of particles must match the inputted amount.	The specified number of particles should appear on the screen, at the designated positions. No clipping should occur.	This criteria was fully met as the particles were drawn successfully to the canvas, once the play button was pressed. PASS
'Canvas must be cleared after every update of particles'	This will be tested by determining if the particle's previous position is still rendered onto the canvas.	The expected outcome is that each particle will move across the screen, and have it's new position be visible per frame, without leaving a trace of the previous frame when the particle was drawn.	This criteria was fully met as the canvas was cleared after every update, removing any trails or remnants of the particle in it's last position in it's previous update. PASS

Objective 1 Testing - See 'NEA - Test Video 1' (attached)

Overview of Video

1. I firstly, read out the success criteria that I was going to test, and then began by refreshing the page. Refreshing the page and seeing if a blank canvas is displayed, is suitable enough evidence for the first criteria to be checked off.
2. Next I tested the second criteria by creating a random number of particles and then pressing the play button and observing if any circles appear on screen. In this, I used a variety of usability features including buttons and input fields, along with the slider, which were responsible for adjusting the value of the slider HTML element.
3. Finally, I then addressed the criteria that the canvas must be cleared on every update. I did this by first creating a specified number of particles, and then starting the simulation. If a trail of particles was visible, then it was clear that the simulation was not clearing the screen on every update. However, the simulation worked correctly.

Objective 2

What is being tested - success criteria	Test Data / Methodology	Expected Result	Success Criteria Met?
'Have a variety of input buttons and input fields which change values.'	This user interface criteria will be tested via ensuring that each of the particle's properties have a corresponding button or input field to allow for users to enter data into.	When the user loads the page, there should be a selection of input fields available to the user for configuring the particles.	This criteria was mostly met, as there was a corresponding input field for each of the particle's properties. One thing is that, the specific velocities could not be specified and were random, although the positions could be specified using the draw function. PARTIAL
'Have settings for canvas properties.'	Ensure that the window's dimensions (height and width) can be changed via user input.	The expected outcome, should be that the simulation will contain at least 2 input fields for changing the dimensions of the window.	This criteria was fully met, as there was an enlarge and shrink button to change the size of the canvas. PASS
'Have settings for entity creation and properties.'	Similar to the first success criteria, in that I must ensure that there is a corresponding input box for each of the particle's properties.	The expected result is that the simulation will contain input fields for changing the particle's colour, radius, mass, and x/y positions and velocities, along with the number of these particles that are to be created.	This criteria (like the first) was mostly met, as most of the particle's properties could be changed, apart from their x and y velocities which were random. PARTIAL

Objective 2 Testing - See 'NEA - Test Video 2' (attached)

Overview of Video

1. Firstly, I read out the success criteria that I was testing.
2. Next, I began by testing the first success criteria, by showing the buttons on screen that allowed for the particle size to change. These were visible once the 'Particle' button was clicked, displaying a list of options for it. The testing also included, showing off their functionality once the buttons were pressed.
3. Afterwards, I started to test the second objective of 'Have settings for canvas properties'. I began by displaying the settings, which were visible under the properties button on the toolbar.
4. Finally, I addressed the testing of the final objective by showing the dropdown list of input fields for the entity section. In the video, I clicked on each button, and visibly showed it's output on the program. For example, when I changed the particle's radius, I first showed the particle size before, then showed it after, with a clear size increase shown.

Objective 3

What is being tested - success criteria	Test Data / Methodology	Expected Result	Success Criteria Met?
'Allow for the number of particles to be created to be specified by the user'	The methodology to testing this task is to first ensure that an input field exists to modify the particle count, and then enter extreme or erroneous data values such as strings or floating point numbers.	If the inputted data is not erroneous or extreme, then the specified number of particles (e.g. 10) should be created and displayed on the render canvas.	This criteria was fully met, the user could adjust a slider which changed the total number of particles to be created. PASS
'Allow for the particle's radius to be specified by the user'	Ensure that an input field first exists for allowing the user to change the radius.	The particles that are created should be of the specified radius.	This criteria was fully met, the user could adjust a slider to change the radius of the particles to be created. PASS
'Include limits to avoid large performance losses.'	Attempt to create the upper limit of particles, with the largest available radius.	The simulation should stay at an acceptable FPS in order for it to be usable such as 20+ fps.	This criteria was mostly met, as the average FPS was very high in a Chromium browser - 62 FPS with 200 particles, but had quite anomalous results when using Firefox, the performance was stable until around 150 particles, and would then drop to around 8 FPS. PARTIAL

Objective 3 Testing - See 'NEA - Test Video 3' (attached)

Overview of Video

1. Firstly, I started testing by reading out the success criteria that were going to be tested in the video.
2. Secondly, I then addressed the first criteria for 'allowing a specified number of particles to be created'. In the video, I clicked on the entity dropdown field on the toolbar, displaying a list of particle properties. After this, I then proceeded to input a particle amount, and click the start button. Then the specified amount of particles appeared. To confirm I counted all of the particles present on the screen and compared it to the inputted amount.
3. The next part was addressing the second criteria of 'Allow for the particle's radius to be specified by the user'. This involved similar steps that were done in step 2, but the particle's radius value was changed to a higher value, with a noticeable difference. I then clicked start, and visually saw the particle's overall size change, and it's radius becoming larger.
4. Finally, the video then addressed the final success criteria of 'include limits to avoid large performance losses'. This was visually tested by going to firstly any sliders and putting them to the max values, which confirmed limiters implemented for sliders. Then I moved onto ensuring that any input fields were raw integer values can be typed were validated also. This was done, by inputting the largest value available, and pressing start. The particle count was set to a maximum value of 250, and would not allow any more particles to be created on screen.

Objective 4

What is being tested - success criteria	Test Data / Methodology	Expected Result	Success Criteria Met?
'Simulation loop will be created, continuously updating and rendering the particles until the simulation has been flagged as ended.'	To test this, it can be visible checked by determining if the particles move more than once, and continuously move, until their velocities are virtually 0.	The particles should constantly be moving and updating, until the simulation is reset or paused, in which either the canvas is reset, or the updating and rendering of the particles is paused.	This success criteria was fully met as the particles would have their positions updated and move whilst the simulation was playing, but if the pause button or reset button was pushed, the simulation would either stop, or completely clear the canvas. PASS
Ensure that the particle's positions get updated and logic is applied to them.	Run the update method for a particle, print out the result to the screen.	The particle's should have their x and y values changed, along with boundary and collision checking taking place, however the particle should not actually render itself to the canvas.	This criteria was fully met, as the particles would move around, each with varying directions and magnitudes for their velocities. They would eventually slow down, when damping was applied to them. PASS
Ensure that the particle is rendered onto the canvas screen.	Run the render method, when it is run to determine if the particle has been rendered to the screen, along with checking the radius, and x/y position of the rendered object.	The particles should be rendered at their specified x and y positions, but these locations should not change on each render method call, as logic is applied only in the update method.	This criteria was fully met as the circles were rendered to the screen, which reflected correctly both the specified colour and radius of the particles. PASS

Objective 4 Testing - See 'NEA - Test Video 4' (attached)

Objective Video

1. Firstly, I started off by listing what objectives I would be testing in the video.
2. Next, I began by testing the first criteria "Simulation loop will be created, continuously updating and rendering the particles until the simulation has been flagged as ended.'. This was shown by starting the simulation (pressing 'play'), and then letting the simulation run for a period of time. After a while I pressed the 'reset' button and the particles disappeared, with the canvas being cleared.
3. Afterwards, I began by testing the second objective 'Ensure that the particle's positions get updated and logic is applied to them.'. I tested this by first testing to see if when I pressed play that the particles would move on every frame continuously. Because the particles were visibly moving and colliding with each other, This test passed.
4. Finally, to ensure this worked across the board, I repeated the same test for the other simulations which also passed.

Evaluation of the Solution

On the whole, my solution did manage to address most, if not all, of the success criteria first set out in the design stage. I managed to meet all the success criteria used in the test tables to at least the 'mostly met' degree. No success criteria were completely ignored, or not attempted, and most were 'fully met'.

Phase 1

What did I do?

In the first phase, I managed to address lots of the core success criteria and features such as buttons and input fields, along with having a canvas displayed, and particles being rendered onto the screen. I fully accomplished this through creating a particle class, containing both an update and draw method, reducing code redundancy and promoting code reuse through object creation. This also allowed me to easily adjust the amount of particles that were on screen, by simply instantiating new particles or removing existing particles from an array containing all of these particles. An array was used in this section, as it contained a few built in methods such as push and pop, allowing for myself to easily add particles to the simulation, and easily remove them. Because of this, it created an easy way to process particles generically, increasing the robustness of this solution.

Additionally, in this phase I also developed some key UI features (briefly mentioned earlier) such as buttons, although these did not have any functionality, and acted as placeholders to be addressed in phase 2.

Objectives Set

- When a scenario page is opened, a blank canvas will be displayed.
- The canvas must allow for entities to correctly be drawn to the canvas.
- The canvas must be cleared after each update and new entities' positions updated.
- Must be able to have custom properties passed into the particle.

The first criteria was met, because once the page was loaded, the visible canvas was clear and not particles were visible on screen. Furthermore, the second objective 'connect HTML to JavaScript' was only partially met in this phase as only the canvas connected to the JavaScript functions, but the input elements were placeholders in this phase. This was not addressed in this phase, as it was merely used to get core functionality up and working. This included addressing some of the success criteria relating to particles, including the particle class.

Objectives Met

- Design a main frame for which the HTML elements will be placed.
- When a scenario page is loaded, blank canvas *will be displayed*.
- Must be able to update it's position by applying forces.

The first criteria was met through first creating a particle class. This particle class had a constructor, which was used when instantiating a particle object. The constructor took in parameter values which were assigned to the particle's members such as its radius, colour, velocities, etc. The second objective was met via the 3rd testing video, demonstrating the particles appearing on screen.

Phase 2

What did I do?

In the second phase of development, I first began by improving the user interface by adding in functionality for the placeholder elements, previously placed in the first phase. This included implementing functionality for specifying the number of entities to create, along with their radius and colour being able to be specified. Additionally, I implemented enlarging and shrinking abilities in this phase, creating 2 buttons to either increase or decrease the size of the canvas window when the buttons were pressed. Input validation was mostly added to this section implicitly through the use of using specific HTML elements such as sliders that allowed for me to set upper and lower limits.

For the 'particles & rendering' section, I implemented default constructor values, allowing for random values to be generated for the x,y xVel, and yVel member variables. Furthermore, this phase involved creating the 'splitting' and 'collision' simulations.

Objectives Set:

- Must be able to have custom properties passed to the particle.
- Coordinate updating of all entities in the simulation.
- Coordinate the rendering of all entities in the simulation.
- Must be able to have collision with other objects.
- Have a variety of buttons near the canvas that change values.
- The canvas must be cleared after each update and new entities' positions updated.

Objectives Met:

- Must be able to have custom properties passed to the particle.
- Coordinate updating of all entities in the simulation.
- Coordinate the rendering of all entities in the simulation.
- Must be able to have collision with other objects.
- Contain settings for entity creation and properties.
- Have a variety of buttons near the canvas that change values.

The first objective was met as the newly developed input buttons for the radius, amount and colour allowed for the user to change the properties of the particle. However, it may be noted that the user can not directly change the particle's starting position, but this was addressed in the final phase of development for the UI. Finally, this criteria could be interpreted as partially met due to the fact that the user cannot change the x or y velocities at all, unless they edit the code directly.

The second objective was met through the use of having a simple for loop that looped through an array of particles calling the update method for each of the

Final Phase

What did I do?

In the final phase of the development I worked mostly on implementing the play, pause and reset buttons which allow the user to start the simulation, pause the simulation, and reset the simulation back to its original state - clear the canvas. The pause button temporarily paused the updating of the particles by setting a boolean flag to true, preventing the drawing of particles when the draw function is run. Furthermore, the reset button was implemented which involved removing all current particles and then stopping and restarting the main setup function again. Finally, for the user interface I addressed stakeholder feedback mentioned in the previous phase about the icon sizes and the usability of the input buttons and features.

In the final phase for particles & rendering development I mainly focused on optimising the collision detection algorithms along with the algorithms for splitting. This included taking an in depth look at the Big O complexity of the algorithms. I found out that my collision algorithm was of Big O N^2 . To further add to the depth of my optimisation, I attempted to estimate the total number of clock cycles performed in the collision detection algorithm, and found out that it had a total of 1.134×10^9 cycles per frame. I introduced a slight fix with the amount of particles that I needed to check and managed to reduce this to a total of $5.689278e + 10^8$ clock cycles per frame, reducing it by nearly half.

Objectives Set:

- Write appropriate canvas methods for clear, reset, etc.
- Contain some time and canvas manipulation settings such as play, pause and reset.
- Check current conditions of simulations to determine if appropriate to end the simulation.
- Must be able to be instantiated in large quantities efficiently.

In the final stage of development, I managed to achieve numerous criteria for both the user interface and the development sections. For the user interface section I managed to successfully address the second objective for having input fields present that change certain entity properties like the amount of particles, colour, size, etc. Additionally, I managed to partially address the ability for the user to select a position to render the particle in by having a draw function, which draws a particle at the mouse pointer's location, once draw mode is enabled. However, this target has not been met fully, as this solution lacks precision, and is dependent upon the accuracy of the user and the mouse pointer.

Objectives Met:

Appropriate methods have been implemented for the reset and clear buttons. Additionally, this was accomplished through the use of creating input buttons to toggle these particular functions on and off.

Another objective that was met was implementing time and canvas manipulation buttons to the user. In this stage, I managed to implement the code for input buttons such as the play button. This play button toggled on and off a global variable, along with changing the text present in each input field.

Moreover, I managed to achieve better performance for the simulation, through a simple optimisation tweak in the collision iteration section of the code. This reduced the clock cycles that had to be performed by 50%, which saved lots of CPU time and cycles, increasing the overall FPS of the simulation. This was accomplished through a variety of performance monitoring tools such as firefox and chrome's inbuilt debugging tools. These tools allow for inspection of memory consumption, and CPU usage. Additionally, the use of memory allocation inspection displayed which javascript functions were using up large chunks of memory, along with displaying what their memory values were over time.

Furthermore, this stage also addressed the objective to ensure that the solution is cross browser compatible. During some testing, I ran into some issues with Firefox relating to the performance, and would see much better FPS and performance on chromium based browsers such as Google Chrome, Brave and Chromium, compared to Firefox.

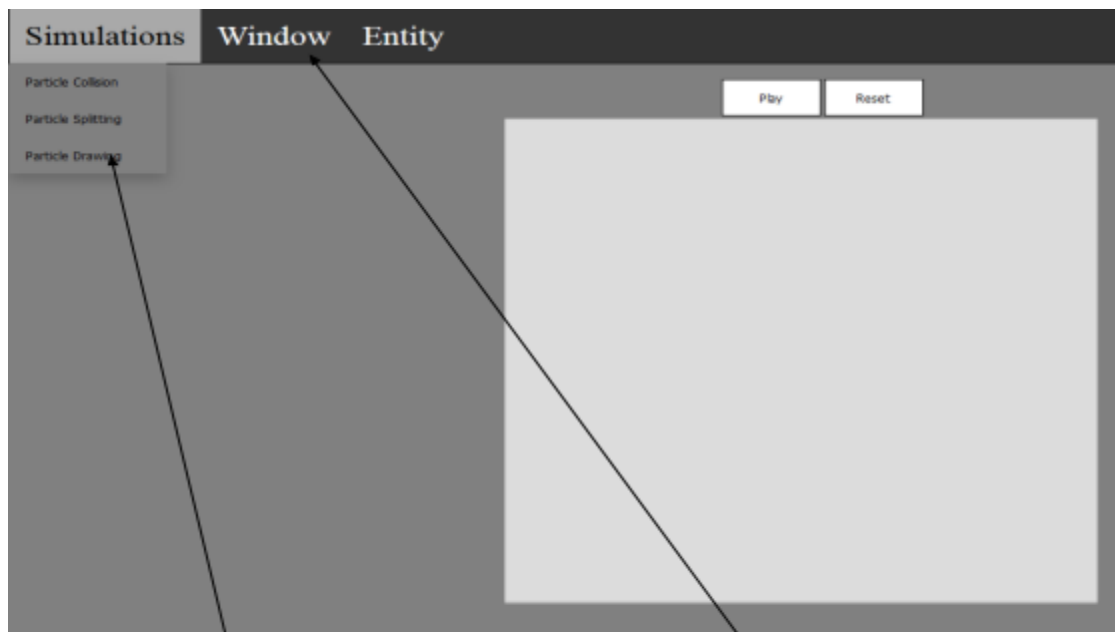
Usability Features



3 dropdown menus present using simple bold white text, clear to the user, but kept in the top right to keep them out of view when needed.

Play and Pause buttons are placed in the centre of the screen and are fairly large and easy to read.

A simple selection of 3 colours - grey, dark grey and white make for a simple and effective colour palette, that does not drown the user with colours, keeping it simple. Furthermore, the usability features are clearly identifiable with large bold white text, which draws the users to them preventing confusion.



A dropdown menu appears directly below the highlighted area.

All important UI elements have large white text, making them more readable.

My solution also has large boxes for buttons, to allow for easier pressing, and the buttons are separated reasonably apart. Furthermore they are positioned in the centre of the website as to draw attention to them more easily, and prevent confusion for the user.

From the above diagram, It is clear that my solution contains usability features such as large white text for UI elements, which aim to make searching for specific buttons and functionality easier for the user.

Limitations in Solution

Although the solution addresses most, if not all of the set out success criteria, the solution still has some flaws, and did not meet some set out goals from the beginning.

Lack of Simulations: One flaw with the software is it's lack of a range of simulations. Although the simulation does present collision and particle velocity well, along with having a simplistic and easy to use interface. The lack of simulations will come as a negative to users, as they will be limited to 2 simulations, which don't have much variety. Although what they lack in variety they make up for in customisation, and ease of within the simulation.

Download: As noted in the final feedback from the stakeholder, the simulation was also criticized for its lack of a download feature. I did mention that this will not be implemented in my analysis because of it's increase of development time, and it's limited in terms of its impact on the portability of the software.

Adding a download feature would not significantly increase the portability of the software because most of my end users will have access to the hosted website in order to use it rather than relying on a physical executable in order to run the

simulation. Although, this was expressed as a hindrance to the program, as mentioned by the stakeholder. Additionally, adding this feature would require testing on multiple OS platforms for Windows, MacOS and Linux distributions in order to ensure that it works correctly on these specified platforms.

Lack of Information: Another limitation to my solution is that there is a lack of information available to the user, such as being able to track the velocities and x/y positions of a particle over the course of the simulation. This limitation can be very prevalent to end users who would like to observe and analyse a simulation in more detail, over the course of the simulation's lifetime. Because of this lack of information the simulation may be more suited to be used as a tool alongside another simulation which displays more information to the users.

Somewhat Difficult to Improve: The final limitation identified in my final solution is that if stakeholders wish to add more simulations, or the project was to be continued in it's development stage, then there might be a slight learning curve as my simulation involves using an external library (P5.js) to perform the crucial step of rendering the particles to the screen. Because of this, the solution may be slightly harder to adapt, as compared to if it was built using only built in functions and pure JavaScript.

Does not work with specific script blocking: If any extensions are used that block 3rd party scripts, it will render the solution obsolete. This is because my solution uses cloudflare to retrieve the P5.js library components, and without these, even the canvas cannot be visible.

Solutions for Limitations

Lack of Simulations - Solution:

A solution to the first limitation may be to use this software in conjunction with another simulation tool which provides other simulations. For example, if a student or stakeholder would want to investigate fluids, they could use my solution as a way of understanding particle interaction involving velocities and collision, but use another simulation in order to understand the second part of the concept, which is understanding how these particles behave in the liquid environment.

Download - Solution:

For the second simulation, a solution to this limitation is to download the source code, along with the html page, and directly run this page with a browser to run the program offline. This solution is sub par and can result in errors and incorrect results if files are missing, or some functionality breaks down.

Information - Solution:

The third limitation can be addressed similarly to the first solution, which is to use this simulation in conjunction with another program that will display

information about particles during a collision. Although this is not a very effective solution, as the information will obviously be different from the information about the particles in my simulation. Furthermore, some further development could be done to simply 'print out' the particle positions onto the console, or have an HTML element that will be constantly updated with the new text - the particle's position.

Somewhat Difficult to Improve - Solution:

The final limitation is hard to solve. This limitation is based on the structure of my program. My program is very modular in nature - plenty of files, with code separation logically done - particle file, for particle class, collision file for collision simulation, etc. This ensures that the solution can be somewhat easily built upon. Although it uses P5.js, an external library, it has a relatively easy learning curve, therefore not being of much hindrance. Finally, the external libraries are very user friendly to learn, and have plenty of documentation and guides.

Does not work with specific script blocking - Solution: Go through any installed extensions and figure out if they involve 3rd party script blocking, some examples could be ad blocker, ublock origin, umatrix, or any other extension that prevents the cloudflare connection with my device.

Maintenance

Open Source:

My solution is hosted on GitHub, containing all of the source code. Because of this, when maintaining this solution through fixing bugs and ensuring it's functionality, other developers and myself can easily look back at the source code, without requiring physical access to it. Furthermore, hosting the software on GitHub allows for other developers and users to create comments, and issues, which I, or other developers, can then directly use when addressing these issues.

Modular:

My solution is modular in nature, with functions and data structures, like classes, being logically separated via files. The particle class is separated into its own function, HTML processing functions are put into their own files, along with each simulation getting their own files, with their 'draw' and 'setup' functions.

This modularity means that any new developers can easily comprehend the structure of my program, allowing them to quickly and effectively understand how each 'module' functions. Moreover, having the solution be modular in

nature, means that bugs and issues can be isolated more easily, because if all the functions were placed in one file, it would be much more difficult for developers to figure out where a particular function or line of code is.

Commented:

The solution's code is commented to a degree of detail that is not too overwhelming in that it dilutes the readability of the code, but just enough so that it ensures that if developers reading the code struggle to understand any complex logic, it can be clarified within the comments. Using comments, also allows for a mini breakdown to the problem, which can help with debugging to identify errors in decomposing the problem. Finally, the use of comments can provide extra insight into how I solved a problem, that cannot be expressed using the chosen programming language.

Further Development

Testing on Multiple Browsers:

Currently, this solution has some compatibility problems, especially the lack of support for Internet Explorer. Additionally, the Firefox browser suffers from large performance loss, when particles exceed around 150 in number. In future development, the solution will need to be tested more thoroughly on these browsers to ensure that their performance is consistent with the performance seen on Chromium based browsers.

Testing on Multiple Operating Systems:

The solution has been tested on the Windows operating system, and debian derived distributions for Linux. However, the solution has not been tested on the MacOS operating system. In order to test this, I will need to launch the simulation on the 3 browsers - Chromium based, Firefox and Internet Explorer.

Increasing the Number of Simulations

Further development could include increasing the number of simulations through diversifying the available simulations. Additionally, simulations revolving around fluids or masses and kinetic energy could be implemented, to increase the available

simulations, and to remove the first limitation, making this simulation more independent.

Adding in Graphing and Displaying Information

Another feature that could be implemented in future development could be to include graphs and information gathering features. For example, a graph could be used to display overall statistics in real time, about the simulation, such as the number of particles, total mass, arrows indicating velocity, etc. Additionally, a feature could be added to allow the user to select a particle, creating a pop up window of information about that particular particle - its velocity, mass, current position, etc. Although this would have to be carefully implemented, as it may cause slowdowns and noticeable drops in performance.

Finally, other ways to display information could be simply done through having a basic output HTML element, that will have its data constantly updated on every update call. But again, performance will have to be kept in mind.

Conclusion

Overall, I am pleased with the result of the solution as it managed to address most, if not all of the set criteria specified in the design section. Because of this, it managed to fully address and become what it was designed to be.

Throughout the 3 development phases, I separated the development into the user interface and the features section, with both sections being evaluated at the end of the section deciding on what objectives were and were not met during that particular phase. The first stage managed to address numerous base criteria as I managed to get a basic framework setup for where HTML input elements would be placed, along with connecting to, and using the P5.js framework. Deciding what libraries to use came down to ease of use and what it provided. When researching libraries I came across numerous physics based libraries, however most of these such as Matter.js, provided too many features such as gravity, collision, momentum, etc, that would reduce the amount of things that I would have to build.

The next development stage focused on working on physics features such as the particle class and it's methods along with collision and detection algorithms. This stage also highlighted numerous errors and some usability features that were

lacking. Moreover, in this stage, stakeholder feedback highlighted some key areas of improvement such as there not being a way to pause the simulation, or reset the canvas back to a clear slate.

The final stage involved addressing fixing slight bugs and errors, along with testing the performance of the solution within a select amount of browsers such as Google Chrome, Brave and Firefox. After testing, I realised that the current version of Internet Explorer was not able to support the external libraries and some JavaScript code used in my simulation.

Overall, my solution was able to fulfill a majority of the setout objectives in the analysis stage. Moreover the solution was iterative in its design, improving in each phase by addressing stakeholder feedback.