Mykyta Osovskyi
Dronfield Henry Fanshawe School

2d Physics Simulations

Mykyta Osovskyi

Dronfield Henry Fanshawe School

1 Content

2 Analysis

2.1 Introduction

An exciting subject of mechanics is often taught very theoretically based in schools. It leaves most students not captivated and bored. Pictures and drawings provide them only with a very basic understanding of what happens when you, for example, project an object or slide a rigid body down a slope. It would be much better to run the simulation once you performed calculations and see how the match to the model or how adding air resistance would affect the system. The main objective of the project will be to create a tool that enhances student experience of learning mechanics / physics at school.

The program will consist of a simple GUI that would allow users to choose simulation they want to run and modify it as they want. The program will provide a slightly different physics engine for some simulation with several similarities across different systems. Some models would require a higher level of accuracy, so more precise algorithms would be applied, others would be intended to run for a huge number of objects, so precision will be sacrificed for efficiency.
At its core, the program would use algorithms of numerical analysis to approximate solutions for differential equations that determine behaviour of the system. Some simulations would involve Event-driven simulation with continuous collision prediction.

2.2 Stakeholders and Users

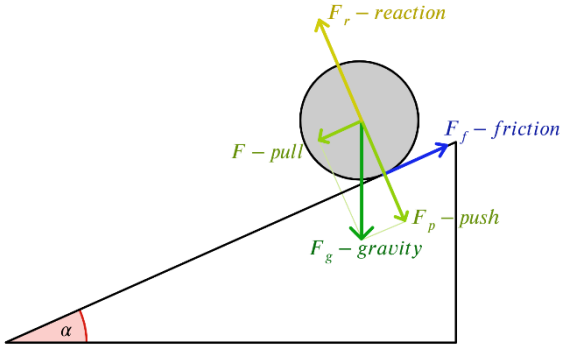
The program will consist of different systems that you can choose from to simulate. Target users will be physics/mathematics students and teachers. In this section I list main categories of people who might use the software, as well as the potential ways they can make use of it.

2.2.1 Physics / Maths Students

One of the areas of study in mathematics is mechanics (which is also studied in physics). The program will provide students with an environment to explore simulations of real systems of objects that are often subjects of mechanics problems, e.g., a projectile modelled as particle[1], a rigid body sliding a rough/smooth inclined plane. They will be able to modify conditions, run simulations multiple times to compare the outcomes with different input parameters.
It will allow students to contrast their expectations and calculations to the system modelled in the same or different conditions. Visual representation will make their learning experience better and will give them a better understanding of the processes that would otherwise have been learnt based only on theory.

[1] In mathematics, an object modelled as a particle does not have spatial extent and therefore cannot collide with other particles. However, some models might simulate multiple particles interacting with each other, so actual objects in the program will only partly inherit properties of a particle.

2.2.2 Teachers



Mechanics and physics are areas where it is crucial to apply theoretical concepts and get practical experience. However, real experiments do not give a teacher opportunity to explain and show all the forces involved, how they change over time and how they affect the state of the system. The program's goal is to solve this problem by providing teachers with a flexible way to deliver knowledge. This tool will allow teachers to explain behaviours in dynamics, which is not possible with traditional drawings. As an example, the drawing on the right is what students would commonly see on the board. The software will allow them to change an incline, weight of the cylinder and its dimensions with an easy-to-use GUI. And then run the simulation to see what happens to the cylinder over time.

("Students" and "teachers" sections are about the same concepts of improving learning experience by helping students to visualize and get a better feel of models inspected in mechanics)

2.2.3 Researchers

Computer simulations of multiparticle systems are often used in various areas of science. They are applied when it is too expensive in terms of time or resources, if even possible, to carry out an experiment.

One of the famous examples is a simulation of noble gas dynamics at a nanoscopic scale. The app would simulate atoms as rigid circles interacting with each other according to Newton's laws of motion and laws of molecular interactions. It allows chemists and physicists to investigate behaviour of the system under conditions they define, running simulation multiple times. Relatively computationally cheap numerical methods of integration produce quite accurate simulation that allows to investigate magnetic, electronic, chemical, and mechanical properties of the system.

The program could potentially help researchers to investigate complex biomolecular effects that emerge from molecular dynamics simulations and make reasoned conclusions.

(However, as discovered further in the research section, real-time simulations are much less accurate than it is required for any research).

2.2.4 User survey

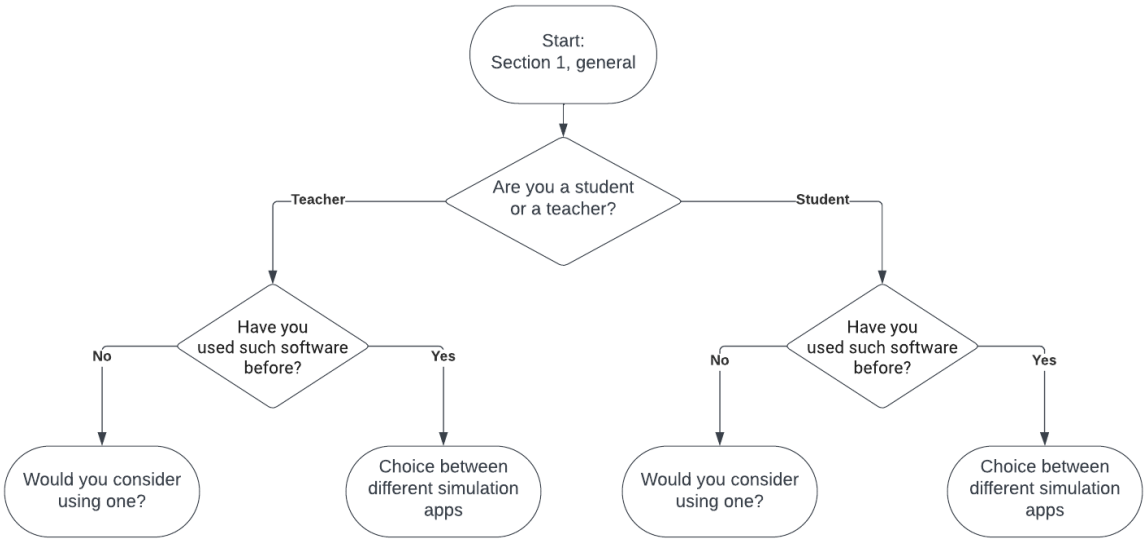
The survey was designed to gather feedback from a range of people to get a wider point of view on the potential features of the solution. It also includes a range of GUI mock-ups to make the survey more engaging and motivate people to think on each question a bit longer. They also test how the users respond to different decisions in design. These mock-ups might be used later in the GUI design section.

Title: "Your opinion on how physics simulation application can help you".

Description: "I am developing a physics simulation web application as a part of my A-level project. It will provide several interactive simulations where the user will be able to modify parameters and drag objects. It will potentially allow students to enhance their learning experience by engaging visual learning. Your opinion or ideas on it would largely benefit my project."

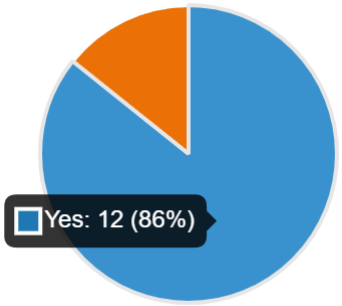
Mykyta Osovskyi
Dronfield Henry Fanshawe School

2.2.4.1 Section 1

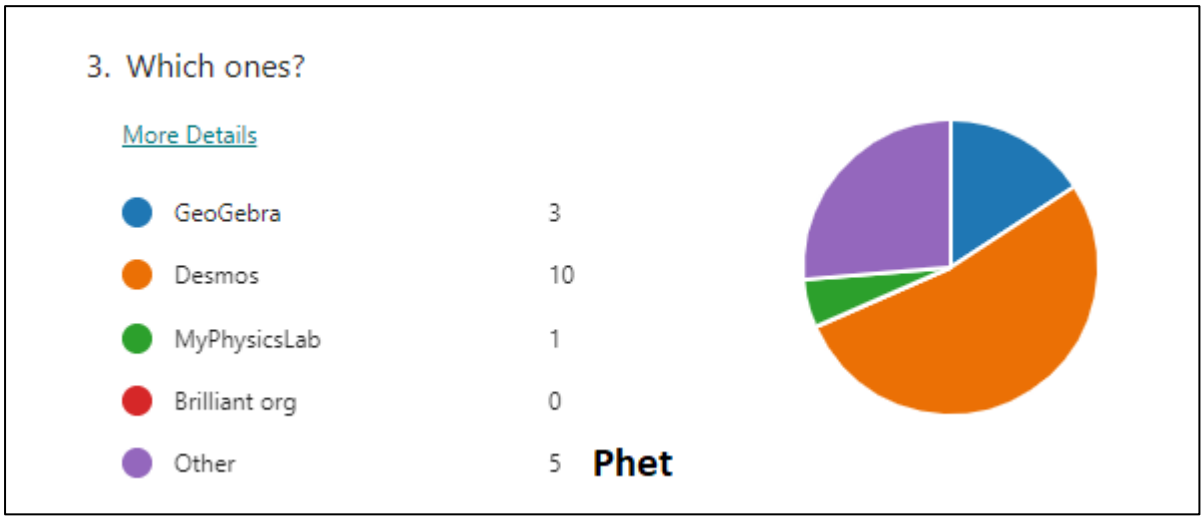


Structure of the first section of the survey

The first section of the survey gathers general information about the participants and their experience with software assisting their studying / teaching. 86% of respondents have been found to use some form of software. All of those who have not, claimed that they will consider using one.
The choice of different apps consisted of GeoGebra, Desmos, MyPhysicsLab, Brilliant.org and "other".



Piechart of responses on whether people had experience with such software



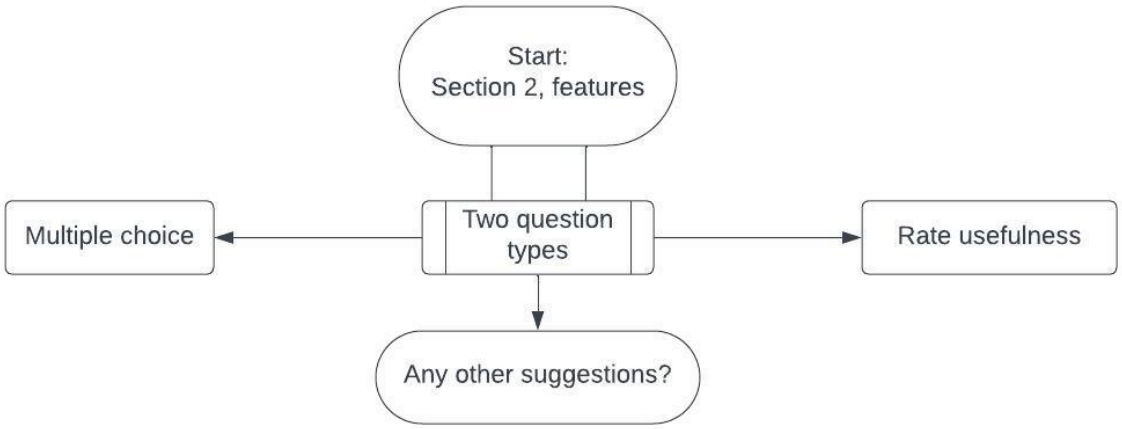
Question 3 response statistics

Question 3 intended to find out which solutions to this problem already exist. GeoGebra and Desmos are the most popular among others, however, they do not deal with physics simulation. Phet was the most relevant to the project and, as it turns out, it is often used by teachers in A-level physics lessons. MyPhysicsLab is the software that implements a general-purpose physics engine in

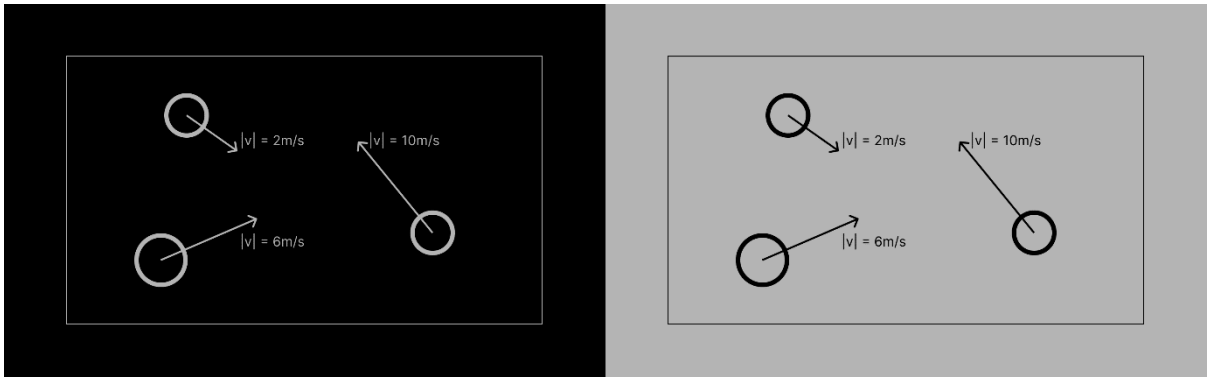
multiple layers of complexity. Simulations vary from no interactions between objects to collisions and contacts. In contrast to this project, myPhysicsLab simulations are not designed to be used by students to solve the problems “from the textbook”, however they can serve a role of a good demonstration of the physics concepts.
Phet and myPhysicsLab will form the “existing solutions” section of the research, myPhysicsLab from the technical point of view and Phet from the user interactions perspective.

2.2.4.2 Section 2

Description: “Following questions will give a description and provide a demo about potential features of the physics simulation application. Rate how useful for you would be the feature and answer any further questions please.”

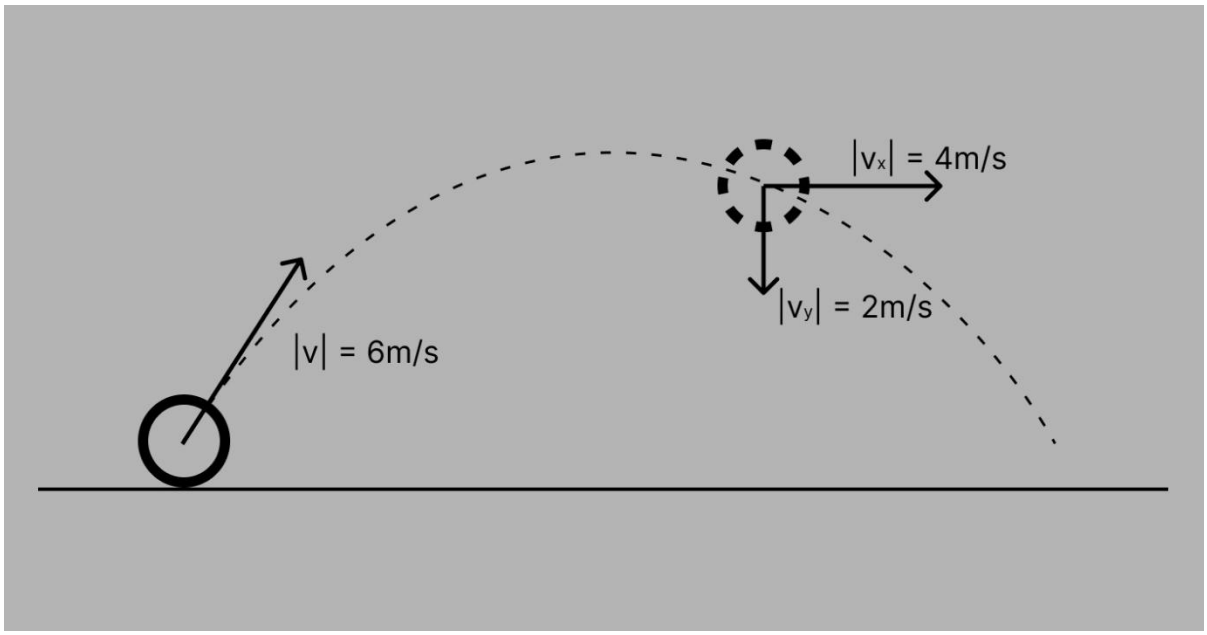


Structure of the second section of the survey



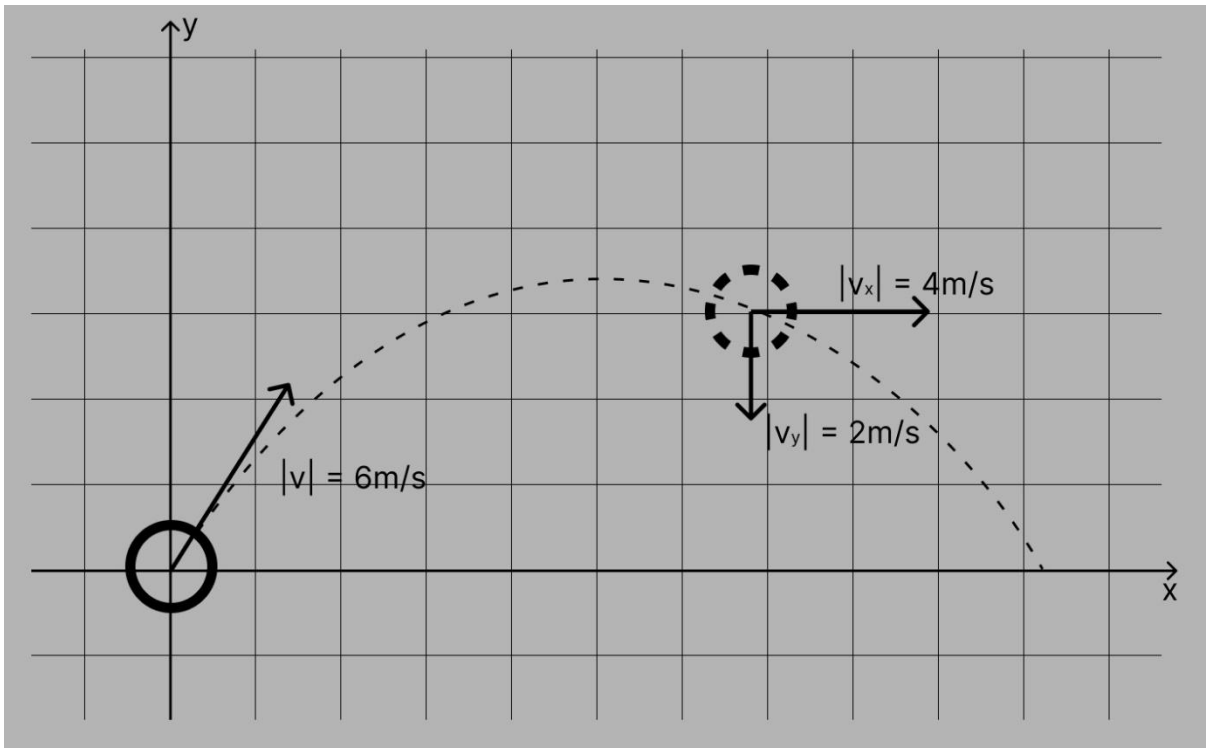
Q1. Black / light-grey theme

“Would you prefer an interface in black or light-grey theme?”
This question is in place to capture attention of the person completing the survey. There was a split between black and light grey. Some other suggestions are white and off-white. As it turns out, most people have a strict preference towards one or the other as nobody picked the option “Not important”. It suggests that it is worth paying attention to the colour theme.



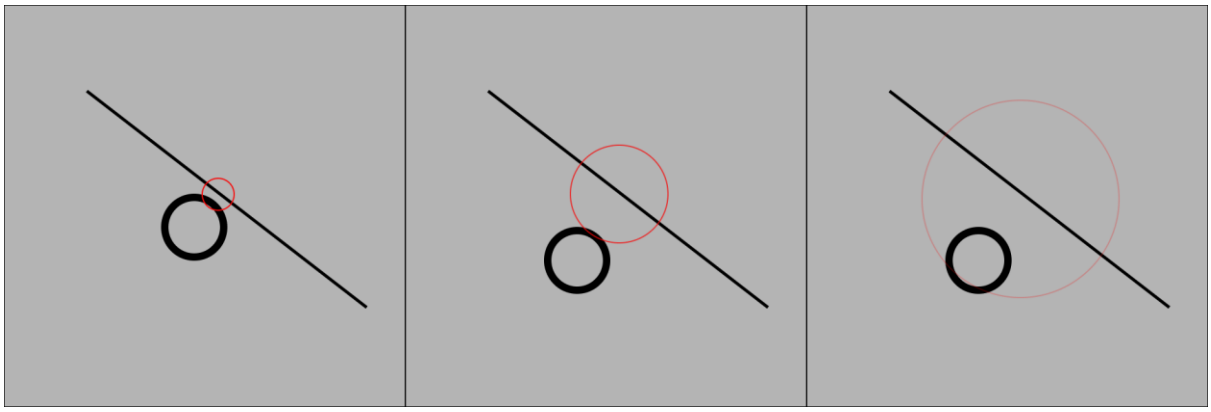
Q2. Information output

"A click on the object would result in various pieces of information, like its velocity, to appear on the interface around the object. How useful would this feature be on your opinion?"
Most people rated it positively, showing that it is a good way to present data.



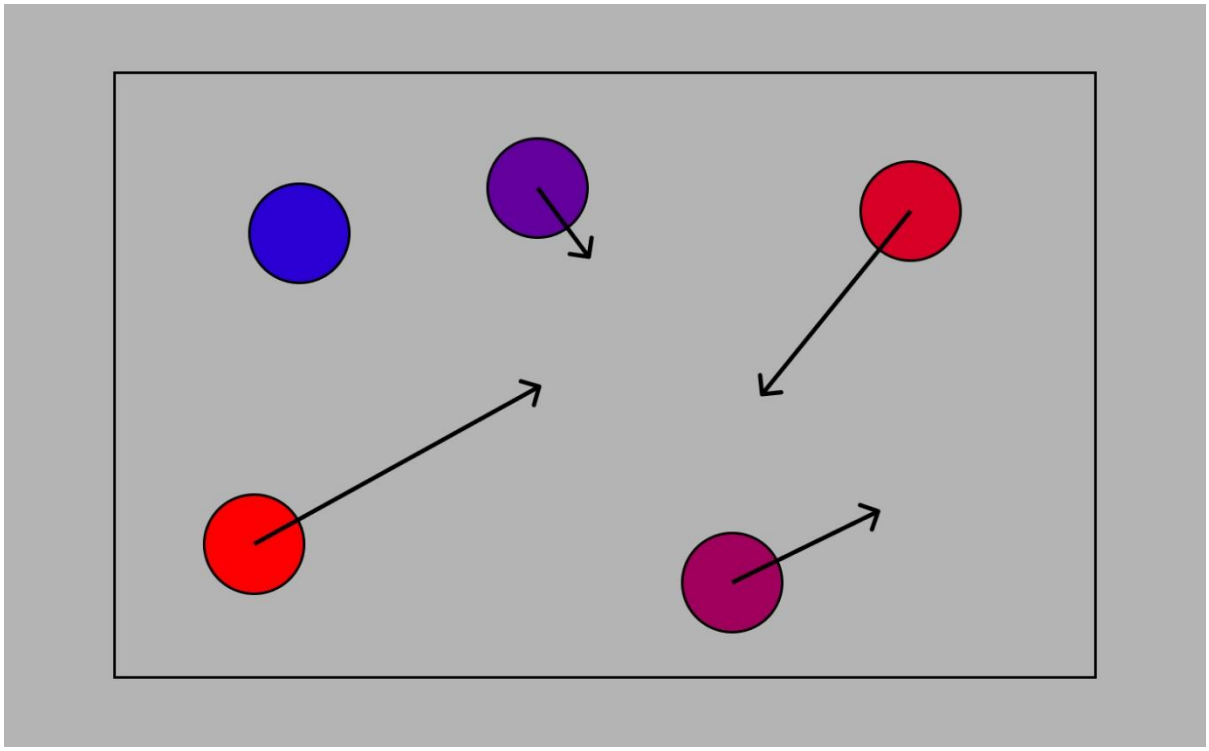
Q3. Coordinate grid

"Do you think coordinate grid on background will be useful?"
This feature was rated slightly negatively. It was suggested that it is better to allow users to switch the grid on and off, as it overloads the screen, but may be useful occasionally.



Q4. Collision indication

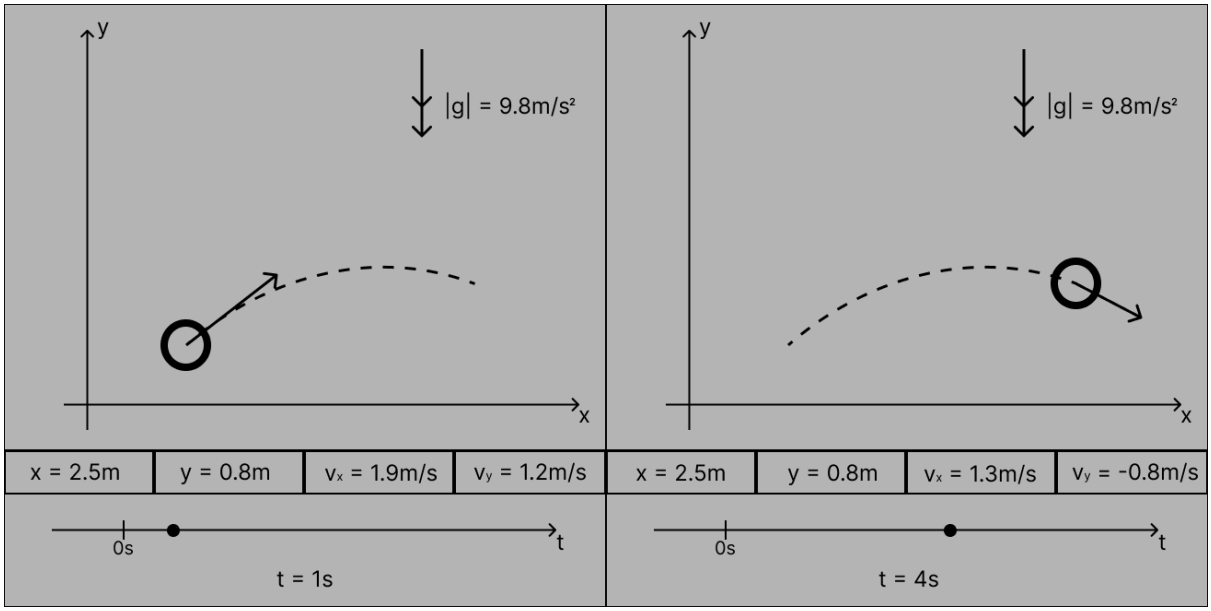
“In some simulations collisions between two objects will be indicated with some effect. What effects do you think will be most appropriate?”
The options suggested initially: a red circle expanding and fading from the point of collision (as on the diagram), Include a short sound effect, no indication. Most people did not mind and chose two first options. Some audience proposed no indication or a primitive sign like “X”. It suggests that some simple, perhaps not very noticeable, indication will be sufficient.



Q5. Colouring by speed

“Do you think colouring an object depending on its speed will be useful? (e.g., colour will go from red for fast objects to blue for static objects)”
This idea came from the fact that the temperature of any object is an average speed of molecules that make up the object (especially for liquid and gaseous). Some of them are faster, some slower, but as they are all the same mass, the total speed is not affected by internal collisions, hence the temperature changes only because of external interactions. This topic and its implications might be

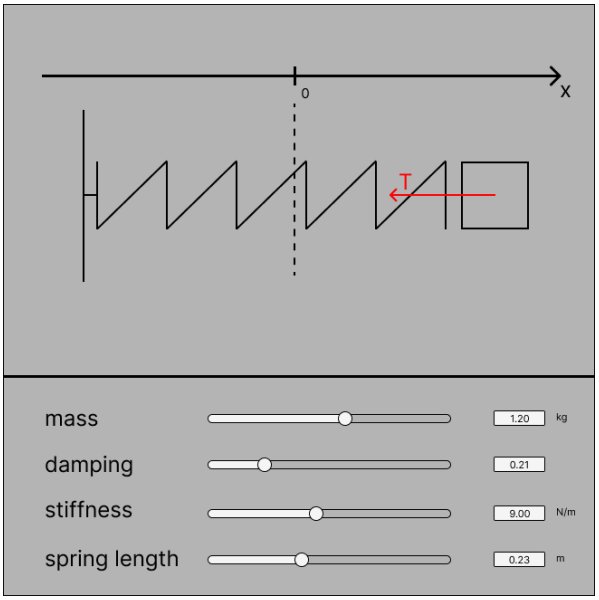
interesting to investigate, however an idea of colouring particles based on their speed was rated quite negatively overall as it does not have direct applications to an A-level program.



Q6. Time control

"The app might provide a feature to run simulation for a specific time interval after you've specified all the parameters of the system. It is potentially useful for problems where you need to find position/speed of a particle after t seconds. How likely are you to use this feature?"
A lot of common problems involve identifying how the system changes overtime. A slider that gives a user ability to change the time of the system can be useful to see how the user's predictions match with the simulation output. Students who completed the survey did not find it particularly useful, however responses were rather positive than negative. Teachers especially noted that it will give an opportunity to see how the system transfers from one state to the other in progress.

"How would you prefer to modify parameters of the system like velocity and mass of a particle?"
Suggested options: using sliders, dragging objects using graphical interface, typing specific values.
Most people picked first two options. They find it to be more precise way to input information. However, it does not omit the fact that making objects draggable makes simulation more interactive. Even though, it might not find application in some simulations that involve more precise modelling, it could still be useful when the system's behaviour is inspected in general. For example, in a setting of a teacher showing how the particles projected at different angles change the trajectory of their motion.



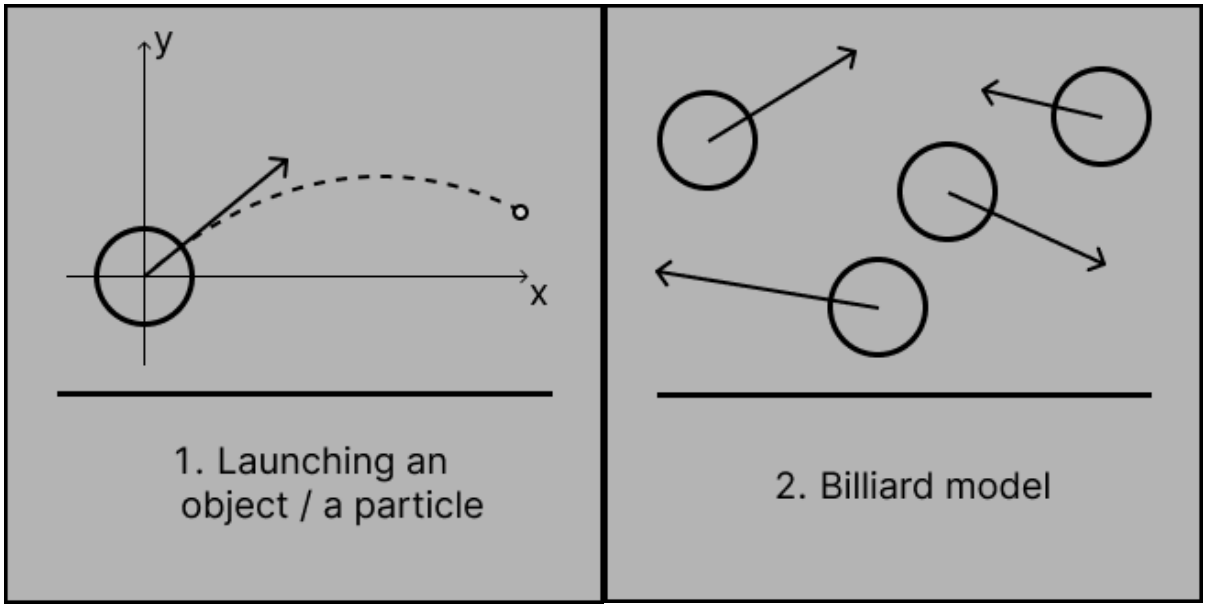
Q7. User input

Other suggestions:

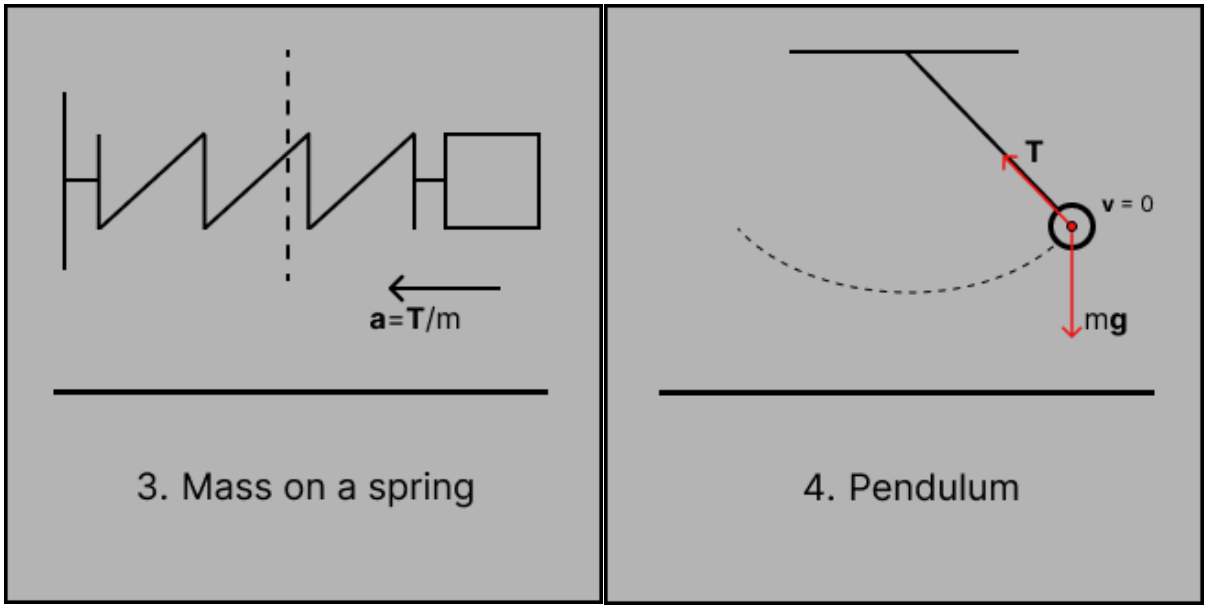
"Let people have a button to toggle the grid on and off in the background for question 6" – discussed above.
"This may be difficult to implement, but a useful feature in Blender and GIMP is the ability to type maths equations into input boxes (e.g., 15/3) and the answer would be calculated and used as the input" – It eliminates the need to use a calculator before inputting information into simulations.
"Expand the content"

2.2.4.3 Section 3

Description: "Rate how potentially useful to you appear following simulations."
This section provides a selection of different simulation and ask respondents to rate their usefulness.

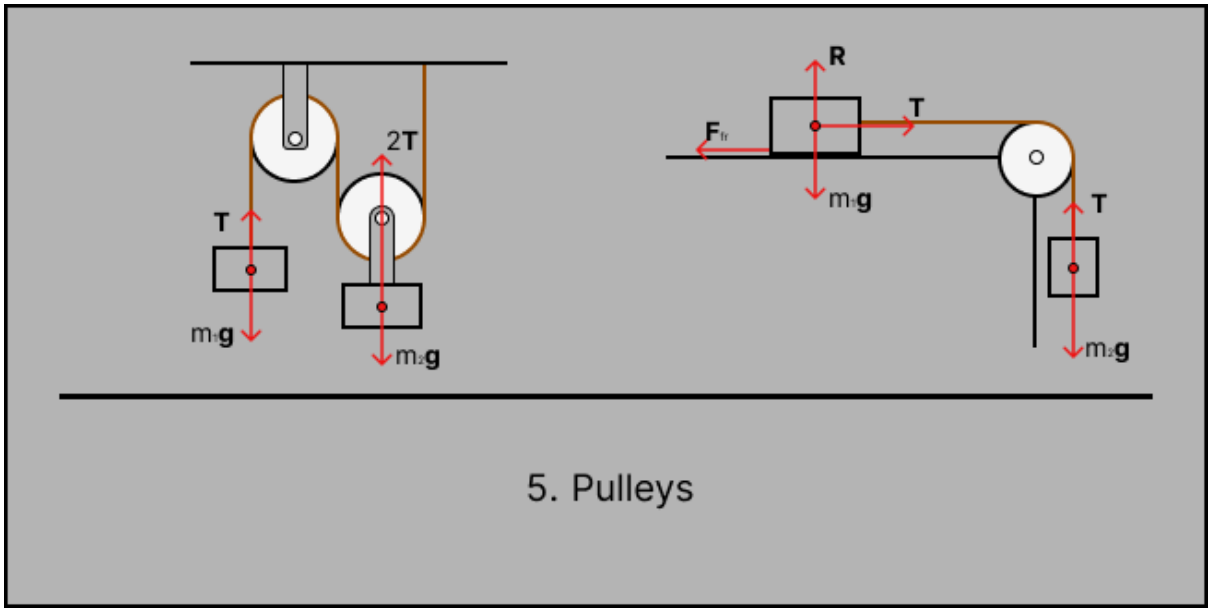


The first model received mostly positive responses. It can be easily explained by the fact that it takes up a big part of the A-level mechanics. Billiard model was not found to be useful at all.

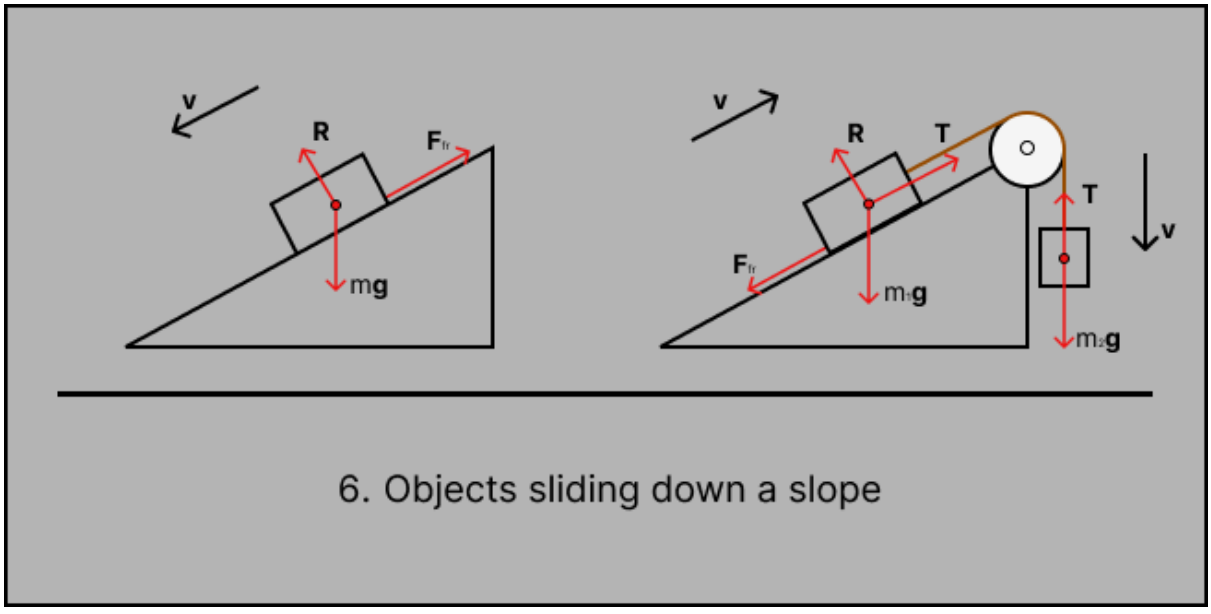


Mass on a spring and Pendulum are common examples of oscillating systems. These systems are considered at A-level physics; however, they did not receive much acceptance among respondents

as they are mostly A-level Math only students. These are valid expansions to the content of the simulations but not at a high priority.



This simulation received a largely positive response (50 by a Net Promoter Score), as students find this kind of problems to be the most difficult. The force modelling tool was generally considered to be very useful.



Again, a score of 50 by NPS. Students and teachers believe that this type of models is likely to find more users than others. These problems are what is typically found on actual exams, especially those that combine a range of skills learnt throughout the course making them the most difficult to solve. A specific suggestion of a feature on this model is to provide a support for events like a string snapping or one of the blocks hitting the floor.

Other suggestions:

“Moments” – other common topic at A-level mechanics. In contrast to other models, moments would not typically involve any dynamics. Most problems are based around the condition of the

system being in equilibrium with a rod on a point of tilting about a pivot. Once the rod starts pivoting, the behaviour of the system instantly becomes chaotic. This model will naturally become a part of the general-purpose simulation if it implements precise contact force resolution.
“Not sure if this would be too difficult, but a general-purpose physics simulation where rectangles and circles can be placed in the world and are affected by gravity.” – this will be the main application of the general-purpose physics simulation.
“Modelling elastic and in elastic collisions between different sized masses.” – a good suggestion for another type of simulations: elastic/inelastic collisions between objects. Like a billiard model but centred around tow rigid discs colliding with each other once.

2.2.4.4 Section 4

The end of the survey, to express gratitude to respondents.

“After completing the survey, do you think my project could be potentially useful in your teaching / studying?”
A 100% of respondents said yes. The survey was anonymous, which supports the truthfulness of answers.

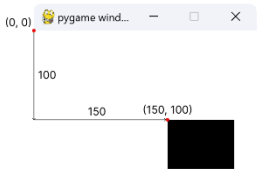
2.3 Research

In this section I will research common approaches for designing physics engines. It includes methods of representing two-dimensional space on the computer; updating position of objects over time, ways to detect object interactions and contact forces and resolve them (those can occur because of friction and collisions), etc.

2.3.1 Representation of a 2d Space

Several ways to graphically represent simulations will be discussed in this section.

“Pygame” module for python programming language provides a graphical engine with in-built tools for drawing surfaces, images, and text on them. To draw an object on the surface, the position is specified as the position of its top left corner relative to the top left corner of the surface it will be drawn on. Pygame also provides implemented Vector, Sprite, and other useful classes. It is highly portable as pygame supports all most popular operating systems.

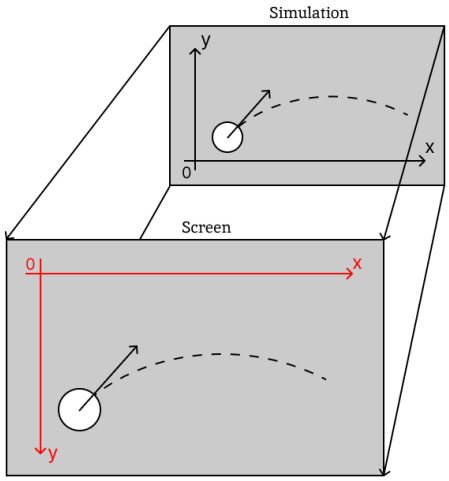


HTML5 canvas element provides 2d/3d graphics drawing space on the web page. The canvas itself is an HTML element `<canvas id="canvas"></canvas>`. JavaScript is used to perform any manipulations with it, such as drawing shapes and lines or clearing the canvas. A reference to the HTML canvas element can be retrieved using “`document.getElementById()`” method. Positioning system of the HTML5 canvas is the same as the pygame’s one. Canvas uses raster graphics which means that all the graphics is drawn at a pixel level where each drawing function defines which pixels are to be filled with a specific colour. JavaScript in conjunction with HTML provide efficient way to handle user mouse and keyboard inputs with a great support for event-driven applications. Animation loop can be implemented using a “`requestAnimationFrame()`” method which executed the callback function on the next available screen repaint.

C# MonoGame is an open-source framework used by game developers to make games that are available at a wide range of platforms. It is a popular choice for game developers especially with 2d games. It does not restrict developers as Unity or Unreal Engine with their specific highly abstracted

environments. It allows them to develop custom tools while also providing basic game functions for content processing, sprite drawing, music / audio, and input handling.

Simulation - Screen Coordinate system:



In mathematics, positive x direction is commonly taken as pointing rightwards and positive y direction upwards. This is different to many graphics' modules, specifically ones discussed above. This difference does not affect any calculations; however, it is much easier and more understandable for humans to imagine origin at left bottom corner with common x and y directions. Simulation coordinate system is not inherently linked to the coordinate system of the graphics module. Therefore, it can be abstracted to be represented in any desired way. This coordinate system can be referred to as “Simulation coordinate system”. Mapping of the simulation to “Screen coordinate system” may be accomplished during drawing routines that are separate from the simulation. Some function would convert the coordinates of the simulation objects to coordinates of the place on the screen they must be drawn at and pass them to the graphics module. It largely assists debugging and testing. Furthermore, if the user wants to move the camera around simulation and zoom in or out, an offset and scaling can be directly applied to the coordinate conversion function.

2.3.2 Vectors

Primary building blocks of any physics engine are vectors. In mechanics, a vector is a quantity that has both a direction and a magnitude. As my project is based in a two-dimensional space, the term vector would refer to a 2dVector with first component as x and the second component as y. Vectors are commonly emphasised in bold to distinguish them from scalars (quantity described only be magnitude). Velocity, distance, acceleration, force are some examples of quantities that need to be represented as vectors, because they are directed space.

If a vector is just a pair of numbers, then why use a different class for that? An array would have accomplished the task of storing two pieces of data perfectly with no overhead for keeping two distinct x and y variables. However, vectors as a separate class become convenient data structure when you consider vector specific operations that find application everywhere in calculations behind physics processes. Different approach to Cartesian coordinates is discussed below in a Polar coordinates section.

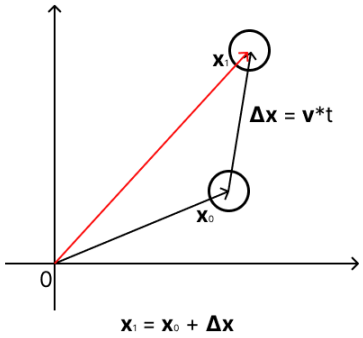
Vectors are mostly used to represent positions of objects relative to a certain point. All games/applications that normally incorporate some physics would have a fixed point, referred to as the origin. Implementation of a vector structure would extremely simplify coding process, making it easier to understand code as well. Common operations with vectors would be frequently used, so organising them into a separate module would benefit the project. Some vector operations are discussed below in this section, others will be reviewed during the design.

2.3.2.1 Addition / Subtraction

Addition and subtraction are similar operations as subtraction can be represented as addition of the subtrahend vector multiplied by –1.

Addition: $(x1, y1) + (x2, y2) = (x1 + x1, y1 + y2)$

Subtraction: $(x1, y1) - (x2, y2) = (x1 - x1, y1 - y2)$



Translation is the most common application of addition. When an object or a vector needs to be moved a certain number of pixels in some direction, it can be done by adding the translation vector to the object's position vector. It finds applications everywhere in the computer simulations.

Advancing an object in time requires translation by the vector of the distance the object is expected to travel (which is commonly a velocity vector multiplied by the time elapsed since the last update).

In some cases, it is required to shift the objects in coordinate systems to draw simulation in appropriate place on the screen.

Some tasks require finding velocity of one object relative to the other, e.g., when considering non-accelerating movement of two balls.

2.3.2.2 Scalar Multiplication / Division

Multiplication of a vector by a scalar is a multiplication of each component of the vector by this scalar. Similarly, to subtraction, division can be substituted with multiplication by the divisor scalar over one. However, it may still be beneficial to implement this procedure for the sake of readability (it may not always be clear to the reader why this ambiguous multiplication occurred).

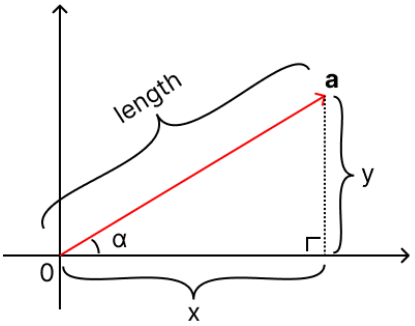
$$\mu * (x1, y1) = (\mu * x1, \mu * y1)$$

Scalar multiplication is often referred to as "stretching" because every multiplication operation results in a vector as many times shorter or longer as it was multiplied by. A common application is multiplication of a velocity vector by the time elapsed to get a vector of the distance travelled (showed on the diagram earlier).

It is important to note that scalar multiplication does not change the angle the vector makes with horizontal but may still change its direction if the scalar is negative. It may be done intentionally, to reverse the direction of the vector.

2.3.2.3 Length and Direction

Even though vectors represent magnitude and direction, they do not explicitly store these values. Both direction and magnitude can be derived from the x and y components. Length equals $|\vec{a}| = \sqrt{x^2 + y^2}$ (length of a vector is often denoted by a modulus of the vector). Direction can be represented as an angle the vector makes with horizontal (with positive direction of x specifically). An angle can be found by performing one of the inverse trigonometric functions, e.g., $\tan^{-1}\frac{y}{x}$. However, computation of inverse trigonometric functions takes a lot of CPU cycles and does not uniquely determine the direction (nearly for every value of a



trigonometric function there is two angles in the range from $0°$ to $360°$). Luckily, it is not usually required to determine a direction as even to perform a rotation, you do not need to know current angle of rotation.

2.3.2.4 Normalisation

Normalisation of a vector is a process of finding a unit vector, a vector of length one in the same direction as the given vector. Unit vectors are commonly used to represent some direction in space. To find a vector of a specific length in some direction, you just need to multiply a unit vector by the required length.

$\hat{\boldsymbol{a}} = \frac{\vec{\boldsymbol{a}}}{|\vec{\boldsymbol{a}}|}$ where $\hat{\boldsymbol{a}}$ is a unit vector in the direction of $\vec{\boldsymbol{a}}$.

A primitive usage example: when you want to launch a ball in a user specified direction. The user will choose the point on the interface. The vector from the launching point to the specified point will be built. To find required velocity vector, the user specified vector should be normalised and multiplied by the desired speed.

2.3.2.5 Dot Product

Dot product can be found as the sum of the products of corresponding components. The same result is achieved by multiplying the product of magnitudes of two vectors by the cosine of the angle between them.

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = x_a x_b + y_a y_b = |\vec{\boldsymbol{a}}|\left|\vec{\boldsymbol{b}}\right| \cos(\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}})$$

The usefulness of this operation comes from the fact that the dot product is computationally cheap task if components of both vectors are known. At the same time, finding an angle between the vectors, its cosine, and the magnitudes of two vectors takes a lot more CPU cycles. Dot product is mostly used to find a projection or a perpendicular vector to the other vector.

Some notable properties of a dot product:
The dot product of two perpendicular vectors is 0 as $\cos 90° = 0$.
Dot product of the vector with itself equals to its magnitude squared.

2.3.2.6 Cross Product

Cross product is primarily a 3d vector operation which allows to find the vector perpendicular to the plain created by two vectors. However, it still finds its use in 2d, in the same way as dot product, taking advantage of the vector components to save the effort of calculating an angle.

As such, the cross product of two vectors $\vec{\boldsymbol{a}}(x_a, y_a)$ and $\vec{\boldsymbol{b}}(x_b, y_b)$ can be found as follows: $x_a y_b - y_a x_b$. The same result is achieved as the product of the two vectors with the sine of the angle between them.

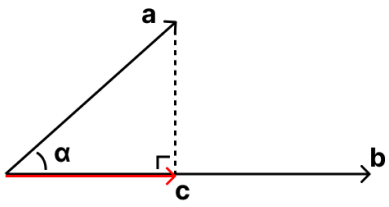
$$\vec{\boldsymbol{a}} \times \vec{\boldsymbol{b}} = x_a y_b - y_a x_b = |\vec{\boldsymbol{a}}|\left|\vec{\boldsymbol{b}}\right| \sin(\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}})$$

The cross product appears in lots of operations such as finding segment intersections or areas of polygons and others, where it is unexpected to be seen. However, it significantly simplifies code as it is easier to deal with several cross-product operation rather than with twice as much component-wise subtractions/additions and multiplications.
Some properties of the cross product: the cross product of two parallel vectors is equal to zero as $\sin 180° = 0$.
The cross product is not commutative: $\vec{\boldsymbol{a}} \times \vec{\boldsymbol{b}} \neq \vec{\boldsymbol{b}} \times \vec{\boldsymbol{a}}$, however $\vec{\boldsymbol{a}} \times \vec{\boldsymbol{b}} = (-\mathbf{1}) * (\vec{\boldsymbol{b}} \times \vec{\boldsymbol{a}})$
Result of the cross product in 2d space is a scalar.

2.3.2.7 Projection

Projection is used a lot when resolving collisions between objects to find components of vectors along one or the other axis.

This projection function will return a vector (call it $\vec{c}$) that is a projection of the instance vector (call it $\vec{a}$) on the vector $\vec{b}$. Vector $\vec{c}$ will extend in the direction of $\vec{b}$: $\boldsymbol{c} = k\vec{b}$. The scalar k can be expressed as $\frac{|\vec{c}|}{|\vec{b}|}$.



From the right-angled triangle formed by $\vec{a}$ and $\vec{c}$.

$$|\vec{c}| = |\vec{a}| \cos\alpha$$

$$k = \frac{|\vec{a}| \cos\alpha}{|\vec{b}|} = \frac{|\vec{a}||\vec{b}| \cos\alpha}{|\vec{b}|^2} = \frac{\vec{a} \cdot \vec{b}}{|\vec{b}|^2}$$

Once the scalar k is found, vector $\vec{c}$ can be found multiplying vector $\vec{b}$ by k.

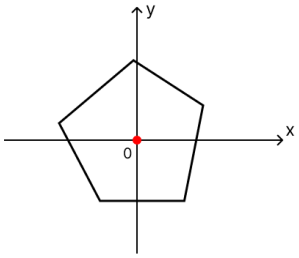
$$\vec{c} = k\vec{b} = \frac{\vec{a} \cdot \vec{b}}{|\vec{b}|^2} * \vec{b}$$

The dot product and length squared functions can be used to largely optimise the process: 2 multiplications for dot product, 2 multiplications for length squared, 2 additions, 1 division and two final multiplications to find vector $\vec{c}$.

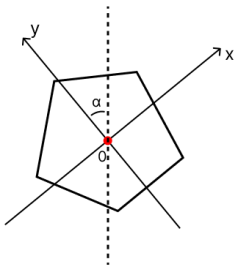
2.3.2.8 Vector Alternative

An alternative to cartesian vectors would be storing magnitude and direction (in the same format of an angle as discussed above), also known as polar coordinate system. This approach might save some time as length will no longer take any calculations to get as it can be directly accessed from memory. Even though all the essential operations could be replaced with their alternatives that use magnitudes and trigonometric functions, it results in much higher resource demanding task as trigonometric functions are more computationally expensive than addition / subtraction or multiplication. At the same time, they become irrelevant as most of the time their whole purpose is to overcome expenses of calculating an angle and its trigonometric functions. Vectors can as well be optimised by caching some variables if they are used frequently. E.g., if the vector's length must be accessed frequently and it is not subject to change, for instance storing a stationary wall as a vector. Overall, vectors are widely used across the computer simulations of physics either as an array of parameters (as in myPhysicsLab which is discussed below) or as a separate vector class.

2.3.3 Simulation Objects



Real world objects are of different shapes and materials. The project's scope will cover only 2d rigid bodies, specifically rigid discs, and convex polygons. These two types of objects are most appropriate to the purpose of the simulations: assist teaching or studying of school level mechanics generally and specifically in an A-level course.
Convex polygon is a collection of points (vertices) with edges linking neighbouring ones. As it moves and rotates, coordinates of each point on the polygon change.



An idea of a local space is common for simulation software. It is based on keeping a separate coordinate system for each object. As the object moves, positions of its points remain unaffected relative to each other. An arbitrary point can be chosen to be the origin for the object abstract coordinate system. This point is often chosen as the centre of mass for the object as it simplifies computations for rotations. When the object moves and rotates, only its local coordinate system gets updated whereas the object remains still within its coordinate system. The position of the object in simulation coordinates is the position of its local space's origin and rotation is the rotation of its coordinate axes. The rotation can be represented in various ways, for example, the angle that the positive y-axis of the local space makes with the vertical line in an anti-clockwise direction. Position of every point of the object can be calculated easily when it is required. In this way, every time the object moves or rotates, only its local space is updated. It results in less calculations when the system is being advanced in time. However, to draw the object on the screen, the position of each vertex must be calculated. This exchange is reasonable as for most simulation programs, the system is advanced in time more frequently than redrawn on the screen.
Discs are simpler objects than convex polygons. A radius of the disc fully describes its spatial extent. Nevertheless, discs also can rotate. The concept of a local space can be applied to any rigid body.

More complex objects can be comprised of multiple convex polygons and discs rigidly hinged to each other. Correctly maintaining these connections is the task of a contact force resolution mechanism. The primary reason of restricting polygons to be convex is that collision detection algorithms cannot efficiently work for concave polygons. They need to be split up into multiple convex shapes and checks need to be made against each of these shapes. This task is not impossible, but not a priority to implement.

2.3.4 Advancing Simulation in Time

In contrast to real world, computer simulation state cannot be updated every instance in time as computers operate in a finite number of computations per a unit of time. It leads to the primary reason computer simulation cannot entirely replicate what would happen in real world. Not depending on the algorithms, the system state will be updated[2] once in every time step so the next frame can be represented on the screen. The lower the time step, the more accurate results are achieved which also increases the time required to complete the step. Consecutively, attempts to achieve more accurate results lead to increasing number of computations required per frame.

Computer physics simulations can be split into two main categories: research and real-time. The research simulations are intended to get to a real-world behaviour as close as possible while taking a

[2] The word "update" refers to recalculation of the state of the system based on the previous state.

lot of time to be generated and, in some cases, omitting the step of drawing anything on the screen. Real-time simulations' goal is to run synchronously to real time while maintaining reasonable accuracy and stability[3]. In the case of my project, the simulation falls into category of a real-time one and tries to achieve the goal of representing behaviour of systems explored by students in school.

The real-time simulations tend to be updated more frequently than redrawn on the screen depending on their complexity. The main time constraint of the game loop is to finish all computations by the time the next frame is to be drawn. So that if the simulation is to be updated every 0.016 seconds (which corresponds to ≈ 60 fps), the game loop ideally needs to take less than this amount of time.

The physics engine of the general-purpose simulation is responsible for three main tasks: move objects, detect collisions, and resolve them. A lot of physical systems do not have a closed[4] form solution, they require numerical methods to approximate their behaviour. One of the famous examples is a three-body problem, the problem of solving subsequent motion of three massive objects (point-masses or objects with a spatial extent such as Earth, Moon, and Sun) moving under gravitational forces (exerted by each object on other two) in space. This is also an example of a chaotic system, meaning that even small derivation in the initial state of the system will result in significant difference in their trajectories over time. There are several numerical techniques that can be used to approximate such systems (which are also applicable for systems that have a closed form solution). The need for numerical solutions might as well appear when the system involves collisions. They do not directly make finding general form solution impossible, however if these solutions exist, they are largely affected by collisions and most likely will have to be recalculated after every single collision.

Some physics systems do not even need to be simulated in a stepwise manner, meaning that they have analytical solution which can exactly tell the state of the system given time and initial state. Examples of these are the ball falling freely under gravity, simple pendulum, or mass on a spring oscillating in a frictionless system. These systems, however, are of the main scope of the school mechanics problems. They will have a more efficient ways of solving/simulating them that does not require numerical techniques. These are likely to be tailored towards a specific topic and not involve collisions. One such system will possibly be implemented in the project. However, the main physics engine will use a more general approach.

2.3.5 Dynamics of Movement

Moving logic bases itself on Newton's two laws of motion (the third one finds its application in collisions).
The first law states that a body will continue to move with constant speed (move in a straight line) unless an external force is applied.
The second law states that acceleration of the body (change of the velocity over time) is directly proportional to the resultant force acting on the body and inversely proportional to mass of the body $a \sim F \quad a \sim \frac{1}{m} \;\leftrightarrow\; a = \frac{F}{m} \;\leftrightarrow\; F = ma$. The equation as well holds for vector $\vec{\boldsymbol{F}}$ and $\vec{\boldsymbol{a}}$ as the direction of the acceleration matches to the direction of the force in space.
These laws reflect in the simulation in a way that objects continue their movement in space in straight lines unless acted upon by a force, in which case, their velocity changes according to the

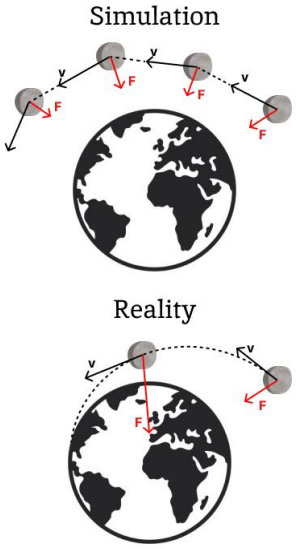
[3] The more stable simulation is, the longer it maintains accurate behaviour to the real-world system.
[4] Closed form solution is the same as analytical solution, i.e., a set of formulas that can be used to find exact results for the state of the system at any given moment in time.

equation provided by the second law. As discussed before, these laws act continuously which is not possible to replicate by a computer.

2.3.6 Force Handling

One of the ways to support second Newton's law is to calculate all the forces acting on the object at a given state of the system / frame. Afterwards, the resultant force can be directly converted into acceleration according to $F = ma$ which is then handled by an appropriate integration method. The main drawback of this system is that if the force changes too fast, the accuracy will fall drastically. Therefore,
For example, when considering the motion of the force dependent on the position of the object in space such as the motion of an asteroid next to the massive body.

When an asteroid gets close to the massive body (say the Earth in this example), the force which acts on the asteroid changes drastically in small time intervals. The simulation assumes the force and velocity to be the same throughout the time step. In real situation, the force and velocity of the asteroid will continue to change throughout the time step. The asteroid might have even fallen on the Earth whereas in simulation, because of the discrepancy caused by discrete updates, the asteroid will escape its orbit or get accelerated too much. This type of error is present in any scenario; however, it is particularly apparent when variables are being changed too fast throughout the time step. (The diagram is exaggerated, but it conveys an idea of inaccuracy introduced by discrete updates)
More advanced numerical techniques aim to resolve this issue getting more accurate approximations by considering the change of the variable in previous iterations or predicting how it might change in future.



This trivial algorithm allows to easily handle forces that act continuously and do not experience significant changes between consecutive frames. Other forces, such as ones occurring in collisions will require a separate mechanism.

2.3.7 Numerical Integration

The term numerical integration has been mentioned a lot previously. In mathematics, numerical integration is a family of technique that are used to approximate values of definite integrals. They are used when the integral of a function cannot be expressed easily or even found at all.

Equations solved are usually of the form:

$$\frac{dy}{dt} = f(t, y)$$

Where y represents the state of the system and t is the moment in time. Initial state and time must be defined: y_0 and t_0. State of the system for simulations is comprised of all the positions and velocities of the objects in the system. The function $f(t, y)$ determines the value of the derivative, i.e., how the state of the system changes in this exact moment in time. In the case of physics simulations, functions for these derivatives are not unknown. Forces are recalculated each iteration and velocities can change unexpectedly because of collisions. For velocities, the function f resolves all the forces acting on the object into a single resultant force and calculates the acceleration. For the position of a certain object, it is just its velocity at that moment in time. These relationships are represented as:

$$s = \int v\,dt \quad \leftrightarrow \quad \frac{ds}{dt} = v$$

$$v = \int a\,dt \quad \leftrightarrow \quad \frac{dv}{dt} = a$$

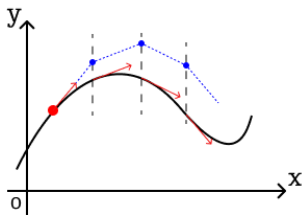
2.3.7.1 Forward Euler Integration

Forward Euler is often called Explicit Euler integration because the state of the system at the next time step is calculated entirely from the current state. Call s_n and v_n position and velocity at n-th iteration respectively. Δt is the size of the time step. Forward Euler integration uses following equations to calculate position for successive frames:

$$v_{n+1} = v_n + a_n \Delta t$$

$$s_{n+1} = s_n + v_n \Delta t$$

Euler integration basically follows the vector field formed by the derivative of the function. Displayed on the diagram:
It is the simplest and most intuitive numerical integration technique. However, it introduces large inaccuracies. To compensate for that, the time step must be very small. The number of iterations per frame outweighs a relatively cheap cost of computations per iteration.



2.3.7.2 Backward Euler Integration

Also referred to as Implicit-Euler integration as it uses a prediction of the future state of the system:

$$v_{n+1} = v_n + a_{n+1} \Delta t$$

$$s_{n+1} = s_n + v_{n+1} \Delta t$$

Methods for predicting the future acceleration exist but are unreasonably computationally expensive for using this method in the context of physics simulation and this method.

Semi-Implicit Euler Integration combines benefits from both methods keeping calculations simple and increasing stability of the system by using the future velocity while calculating it with the current value of acceleration:

$$v_{n+1} = v_n + a_n \Delta t$$

$$s_{n+1} = s_n + v_{n+1} \Delta t$$

This is a very common choice in games technologies for its balance between performance and accuracy.

2.3.7.3 Runge-Kutta methods

Runge-Kutta methods are a family of numerical integration methods, which also include Euler methods. Higher order methods provide larger accuracy at a cost of more computations.

The most common method of the family is RK4. Given an equation for the derivative of a function, it starts by evaluating the derivative at current state of the system, k_1. Using this value, system is advanced in time by a half of the time step to evaluate a second approximation for the derivative k_2

It then uses this approximation to predict derivative half step size in future. In this way, 4 approximations are obtained k_1 at current time, k_2 and k_3 half step size ahead and k_4 a full step size ahead. Final approximation is an average of these values (k_2 and k_3 are taken two times)

$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} * \Delta t$$
$$t_{n+1} = t_n + \Delta t$$

*y_n represents state of the system on the nth time step (i.e., at time t_n).

The method requires a lot more computations than other discussed above, however provides much greater accuracy at this cost.

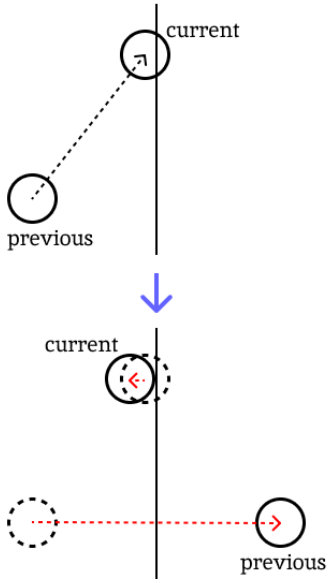
2.3.7.4 Verlet integration

Verlet integration uses entirely different approach to other methods. It does not involve storing the velocities of objects. Information about the velocity is conveyed by the position of the object at the current frame and the one before. Using equations for the semi-implicit Euler method: substitution of $v_{n+1} = v_n + a_n \Delta t$ and $v_n = \frac{s_n - s_{n-1}}{\Delta t}$ into $s_{n+1} = s_n + v_{n+1} \Delta t$ gives the basic Verlet formula:

$$s_{n+1} = s_n + (s_n - s_{n-1}) + a_n \Delta t^2$$

Benefits of using Verlet integration to Euler methods is the fact that the result of integration is reversible which is a large help when the game allows to go back in time or uses replays. Verlet also easier solves systems of multiple particles with constraints, e.g., two particles fixed at the end of the rigid rod or the spring with specific stiffness. Systems like chains of particles or pieces of cloth are therefore simulated much easier than with other method.

Furthermore, Verlet method automatically resolves velocities after collisions of particles. Two positions are modified directly in a smaller number of computations. In the simplest case, the current position will be set at the closest point such that there no overlaps and previous position will be reflected by the tangent line (diagram on the right).



It is worth noting that this simple collision resolution method does not follow the laws of conservation of momentum. However, the results it yields are satisfactory for most games where the Verlet integration is used.

However, Verlet restricts the system to use a fixed time step as the formula for the position on the next frame is derived with the assumption that the time step remains unchanged. It may lead to simulation going out of sync if the time between two consecutive frames is not enough for all required computations. It also leads to complications when the system gets more complex, e.g., when bodies can rotate and take different shapes. Therefore, it is rarely used for simulations where accuracy is more important.

2.3.8 Collision Handling

By its nature, rigid bodies are not deformable, and they cannot overlap with walls or other rigid bodies. Therefore, a mechanism of detecting and resolving collisions is required.

2.3.8.1 Physics Behind the Collisions

The third Newton's law states that for every action (force) there is an equal action in the opposite direction. This law establishes the key principle by which objects interact with each other. Whenever the Earth pulls an object with the force mg, there is the same value force acting on Earth in the

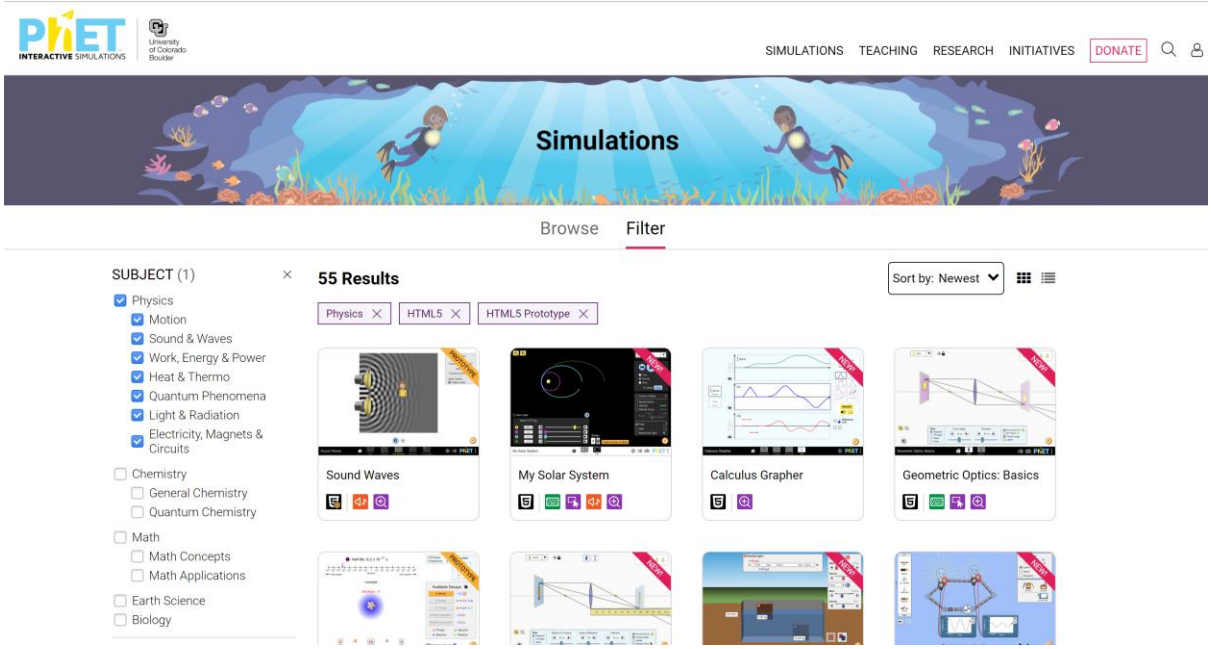
opposite direction, which however does not show any noticeable impact on Earth because of its extremely high mass.

The collision example is two billiard balls hitting each other. Over the brief time of being in contact they experience an action of the powerful force that changes their velocities to moving apart. However, the time balls are in contact is much lower than the simulation time step. So, it is not an option to apply force to the particles while they are not in collision. Typical way simulations resolve collisions is by manually adjusting speed and position of two colliding objects such that they do not overlap and move in directions that do not result in an immediate collision afterwards.

2.4 Existing Solutions

With the physics part of the engine being researched in previous sections, this section will focus on the architecture or user experience side.

2.4.1 PhET

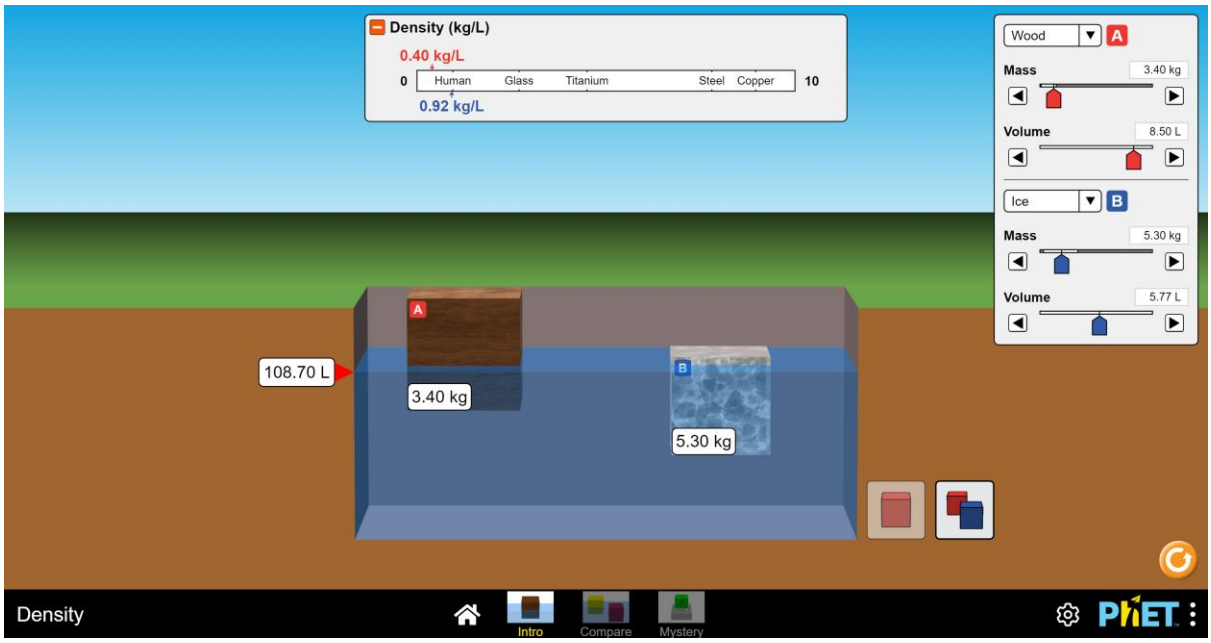


Phet is an online platform which provides a range of interactive simulations. Available subjects vary from physics to math. Each of them comprises of its own features and functionality. All simulations share some common features of the interface like an appearance of tick boxes and slider, ruler and timer tools. The simulations are designed for high school / university students to explore specific concepts of science or math.
Teachers use PhET to demonstrate dynamic systems in action where simulations play role of an equipment for the experiment.
In the next sections I will research some simulations on PhET.

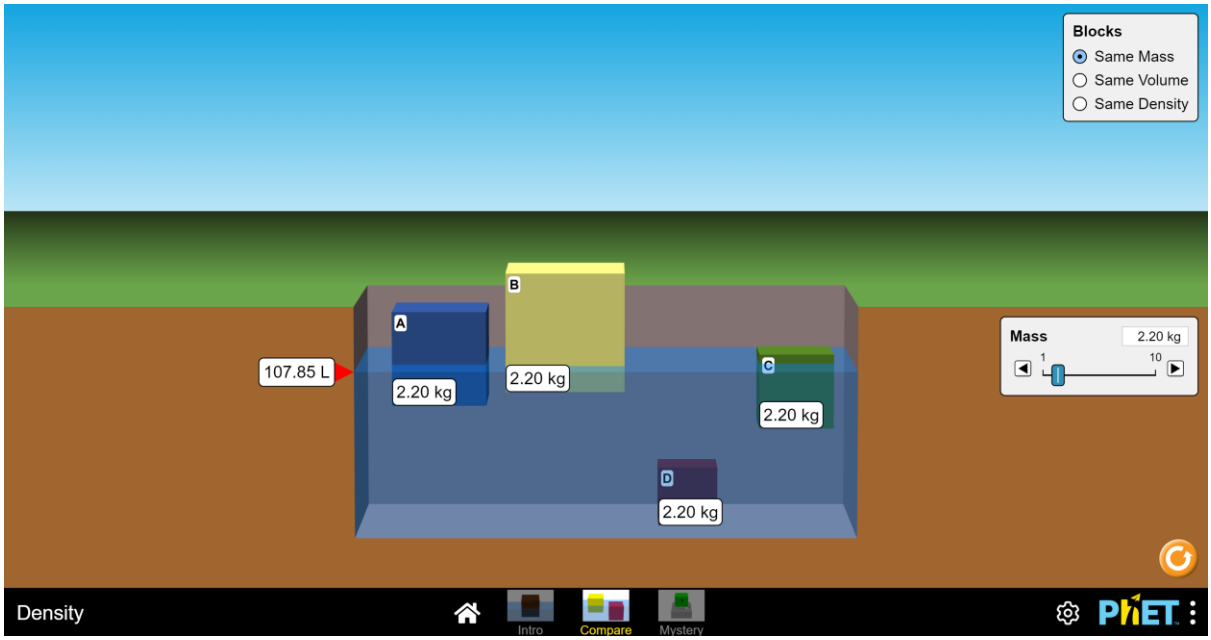
2.4.1.1 Density

This simulation allows the user to explore buoyant force and see how Archimedes law reflects ($F_b = \rho * g * V$) in dynamics. There are three available setups: Intro, Compare and Mystery. All of them consist of a pool of water and several blocks of specific mass and volume (mass and volume together give density, $\rho_x = \frac{m}{V}$, which determines whether an object will float or sink, float if $\rho_x \leq \rho_{liquid}$, otherwise sink).

Mykyta Osovskyi
Dronfield Henry Fanshawe School

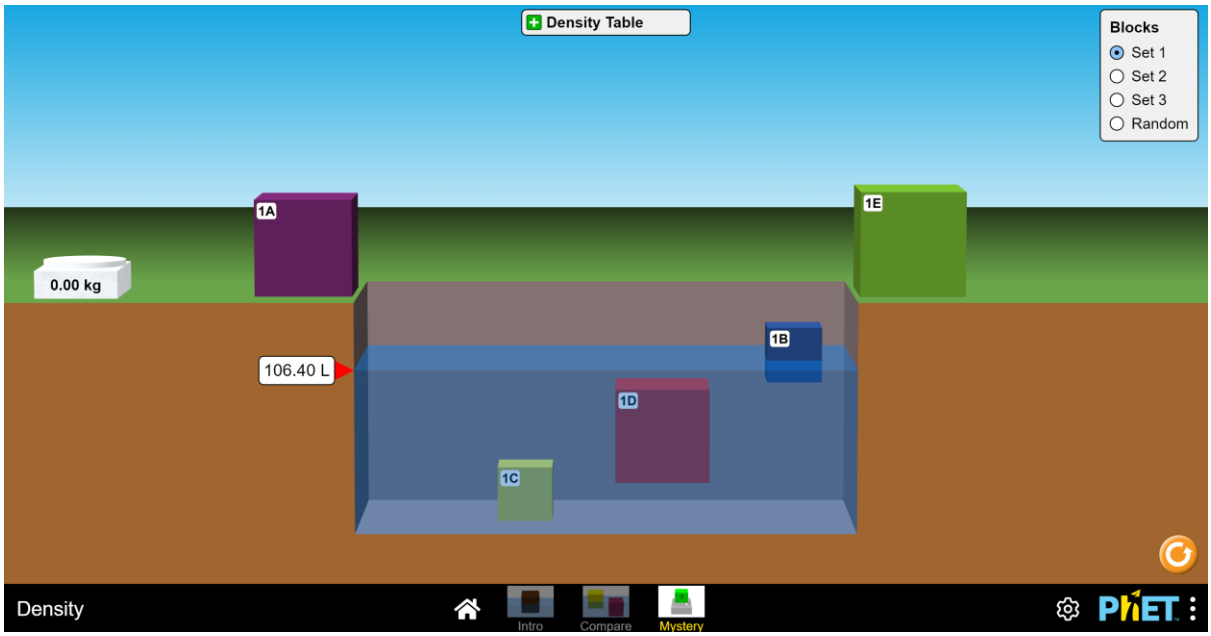


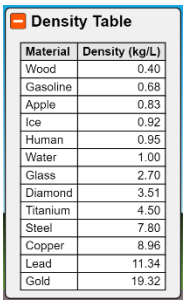
"Intro" setup gives a user a choice between one and two blocks. The user interface allows to drag them, once the object is released all the system forces start to apply, so the object will fall under gravity and float in water depending on its properties. The user can dynamically change the material from which the object is made (material directly corresponds to density) or specify mass and volume of blocks using sliders or arrows on the right. Changes apply instantly. Density is calculated internally and displayed on the scale where students can compare it with density of human and some common materials.



"Compare" setup gives an option of choosing blocks of the same mass / volume or density, so the user can explore how differently they float in a pool of water. As such, smaller blocks of the same mass sink as their densities are higher. Properties can be modified in the same way as for other simulations.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

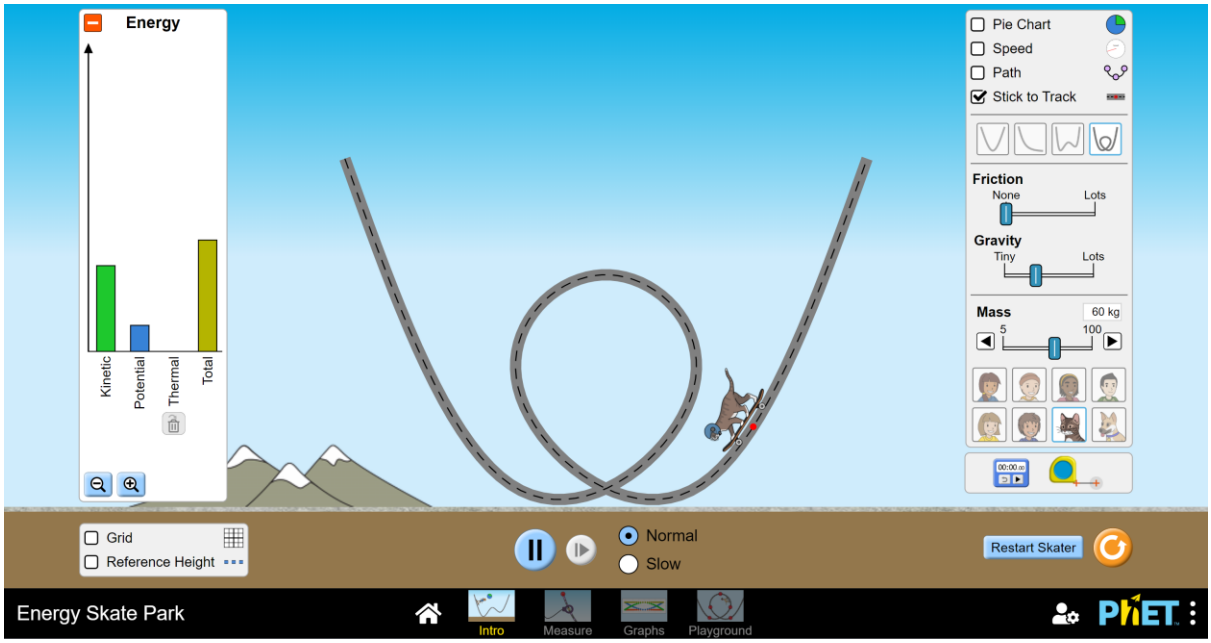




"Mystery" setup gives the user scales and a choice between 3 sets of blocks or a random selection of blocks. This setup gives a range of possibilities for teachers to design practice tasks on the topic. One of the examples would be to determine materials of blocks. They will need to go through multiple steps of measuring their masses on scales, putting blocks in the water, measuring volume of the water displaced by the block which will correspond to the volume of the block (however students will need to separately consider the case when the block floats), divide the mass by the volume, and match obtained density to the material using provided table. In this way, students can get some practical experience and therefore deepen their understanding by learning different aspects of the topic.

2.4.1.2 Energy skate park

This simulation is designed to teach students of the concept of energy.



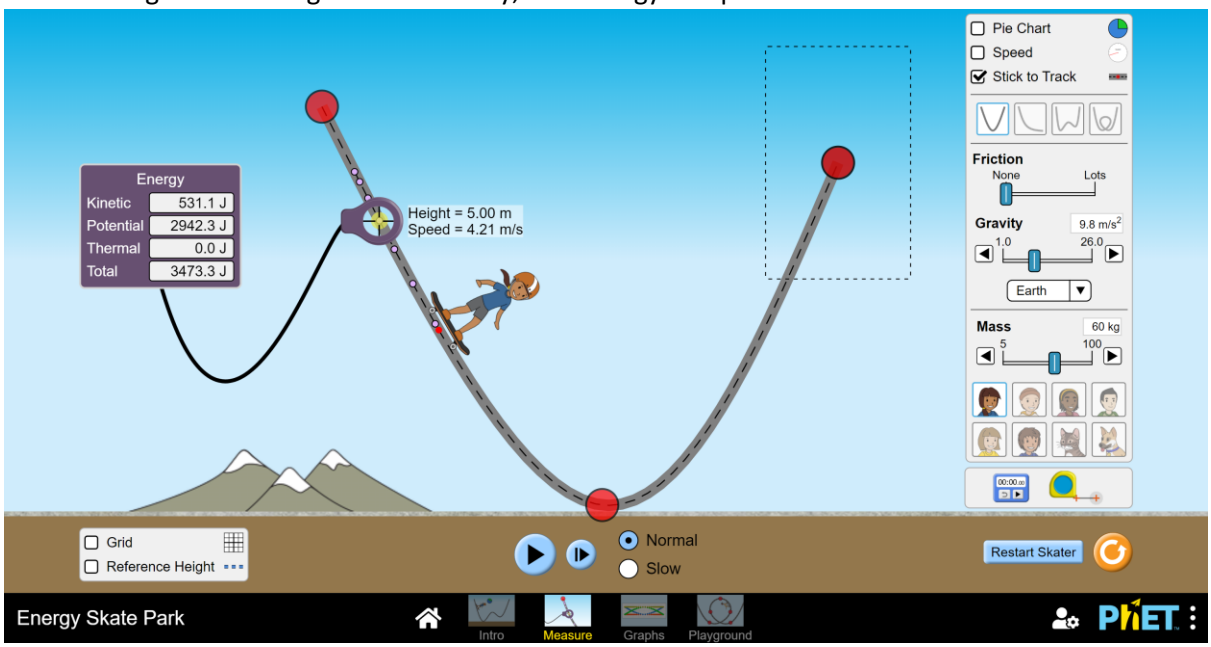
Mykyta Osovskyi
Dronfield Henry Fanshawe School

The first “Intro” setup provides a track and a character which will ride it (even though the character is some sprite, it acts as if it was a point mass at the red point under the sprite. The same abstraction applies to the track, its sprite is irrelevant to the functional part of the simulation). The user has a choice between different tracks as well as sprites for skaters. They can dynamically change mass, gravity and amount of friction between the skater and the track. The flow of energy is represented in bar charts on the left.

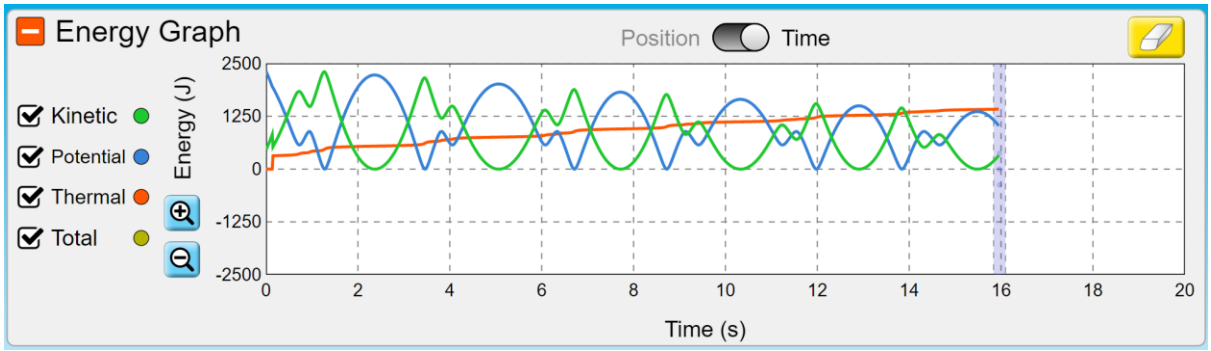
The user can define total energy of the system by dragging the skater upwards changing its potential energy ($E_p = mgh$). Once the skater is released, its energy is being converted into kinetic energy ($E_k = \frac{1}{2}mv^2$) under gravity until the skater hits some surface. In every moment, total energy of the system is defined as

$$E_{total} = E_p + E_k + E_{thermal}$$

Conversion to thermal energy can happen only as a result of rough collision (e.g., falling on the ground) as all collisions are inelastic or as a result of friction. Energy can’t go back from thermal and can be thought of as being lost. In this way, total enegy is kept constant unless redefined bt the user.



“Measure” setup extends simulation giving more freedom and information to users. Now they can modify position of three points that will determine curvature of the track. The skater leaves points on the track every time interval, and students can use a measuring tool to get information about the skater at that moment on the track.



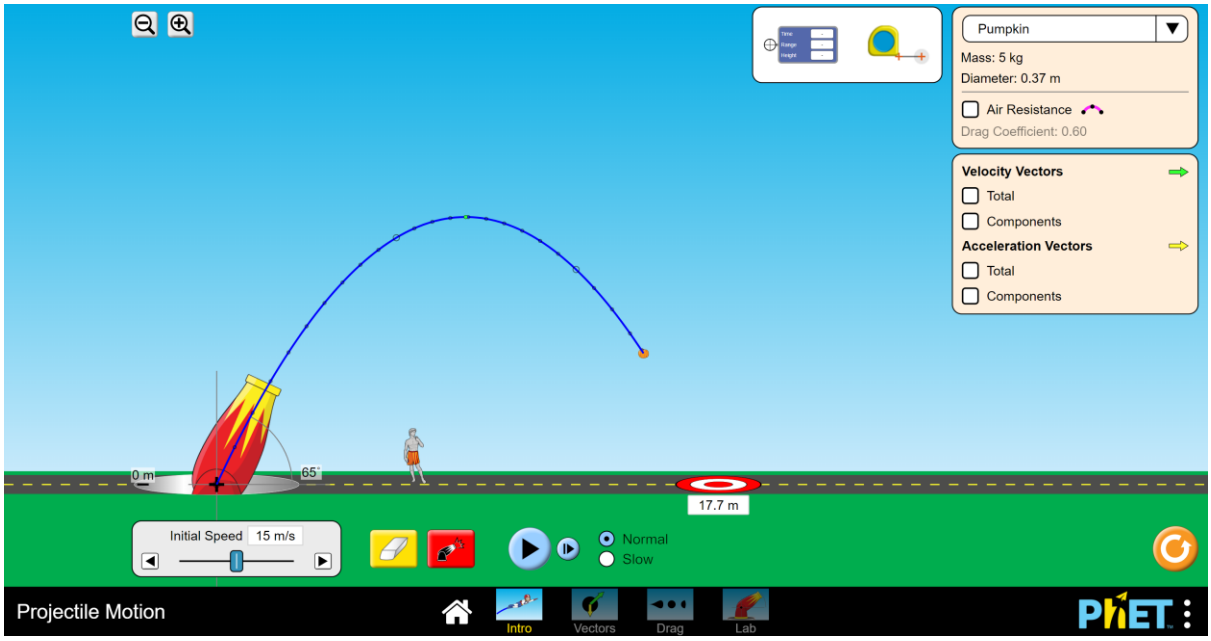
The next setup adds graphs of energies plotted against time / position. Students will experience how kinetic and potential energy are “reflections” of one another as a result of total energy being constant (disregarding small changes in thermal energy).

$$E_k = E_{total} - E_p \qquad <-> \qquad E_p = E_{total} - E_k$$

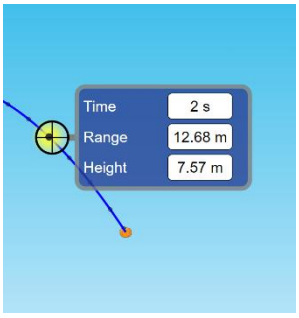
The last setup allows to design a custom track with all the functionalities of other setups.

2.4.1.3 Projectile Motion

This simulation gives basic overview on a motion of projectile launched at some angle above horizontal and moving under gravity.



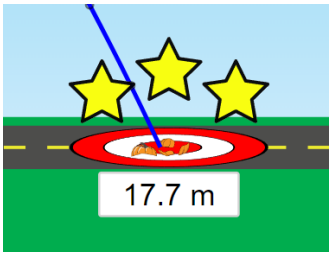
It has common to other simulation interface and a ruler tool. The intro setup allows the user to choose initial speed and the angle at which the projectile is launched. The user can change the height of the canon (canon is again just a sprite, it only determines the point from which the projectile is launched). Air resistance has constant drag coefficient and can be turned on/off by the user. The projectile leaves points on its trajectory every $0.1\ seconds$. They can be examined with the provided tool.

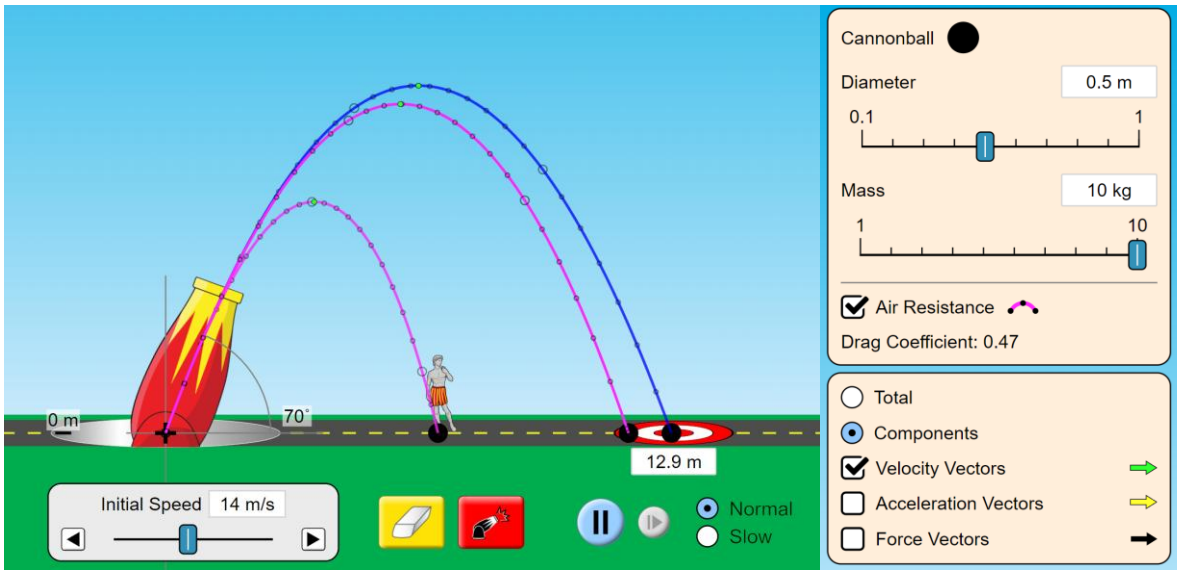


$$s_x = U\cos\alpha * t \qquad s_y = U\sin\alpha * t - \frac{gt^2}{2}$$

$$15\cos 65° * 2 \approx 12.68\ m \qquad 15\sin 65° * 2 - \frac{9.8 * 2^2}{2} \approx 7.57\ m$$

Users can change position of the target on the ground. It represents the place where the projectile is expected to land. Once the projectile lands, students are awarded with 1-3 stars depending on how accurate their estimation was. It again opens possibilities for tasks like: “Estimate how far from the canon the projectile will land given specified conditions of the system”.

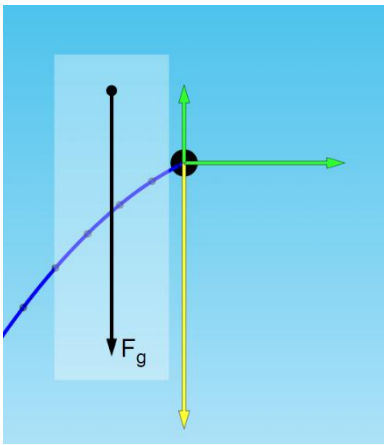


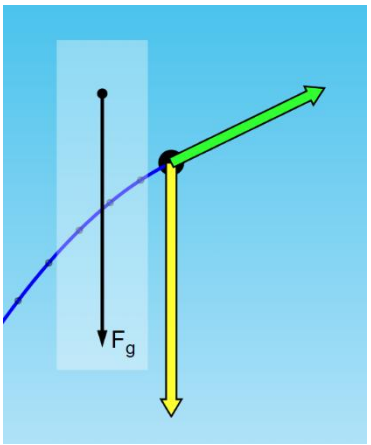


Other setups extend possible ways to modify system parameters. Now the user can specify mass and diameter of the projectile which make any difference only when the air resistance is turned on. Bigger objects have larger surface area and therefore have more collisions with particles in the air. Objects with bigger mass experience smaller change in velocity when acted upon the same force (consequences of the 2nd Newton's law). All these concepts can be easier explained with an aid of physics simulation. Simulations can also be used by students to practice their knowledge / check answers to similar problems.

The simulation also provides an option to draw vectors of velocity, acceleration and force on the screen either as total vectors or component-vise.

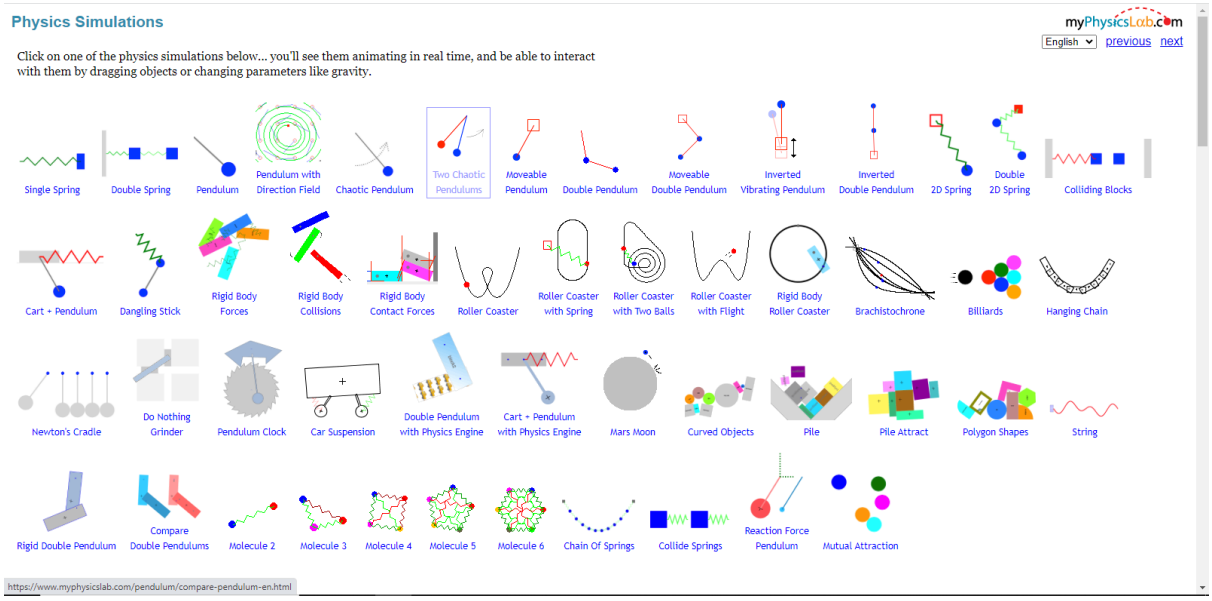
In the "Lab" setup users can choose the sprite of the projectile varying from piano to a car adding a bit of fun for students.





2.4.2 myPhysicsLab

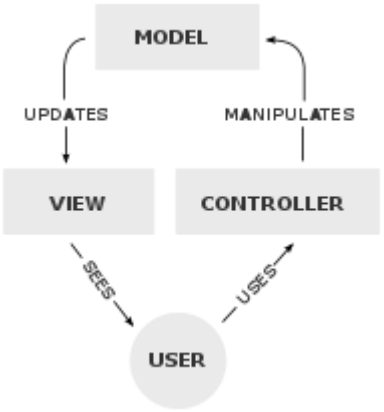
myPhysicsLab is an open-source web application by Erik Neumann. It provides a range of classes that can be used to build real-time interactive physics simulation. The code is mainly written in JavaScript.



The main page consists of several previews to pre-built simulations. Once the web page of the simulation loads, the application is being created. Some HTML elements that the application uses to load various UI elements like canvas/control panel are passed to an application constructor through the “elem_ids” object. Then the application start function is being called and all the other procedures happen from the application code.

2.4.2.1 Model-view-controller pattern

Model-view-controller pattern is a software design pattern commonly used in development of the web applications that involve some user interface. Its main goal is to separate the internal structure from what is represented on the screen. It allows to make the application scalable and easier to manage.

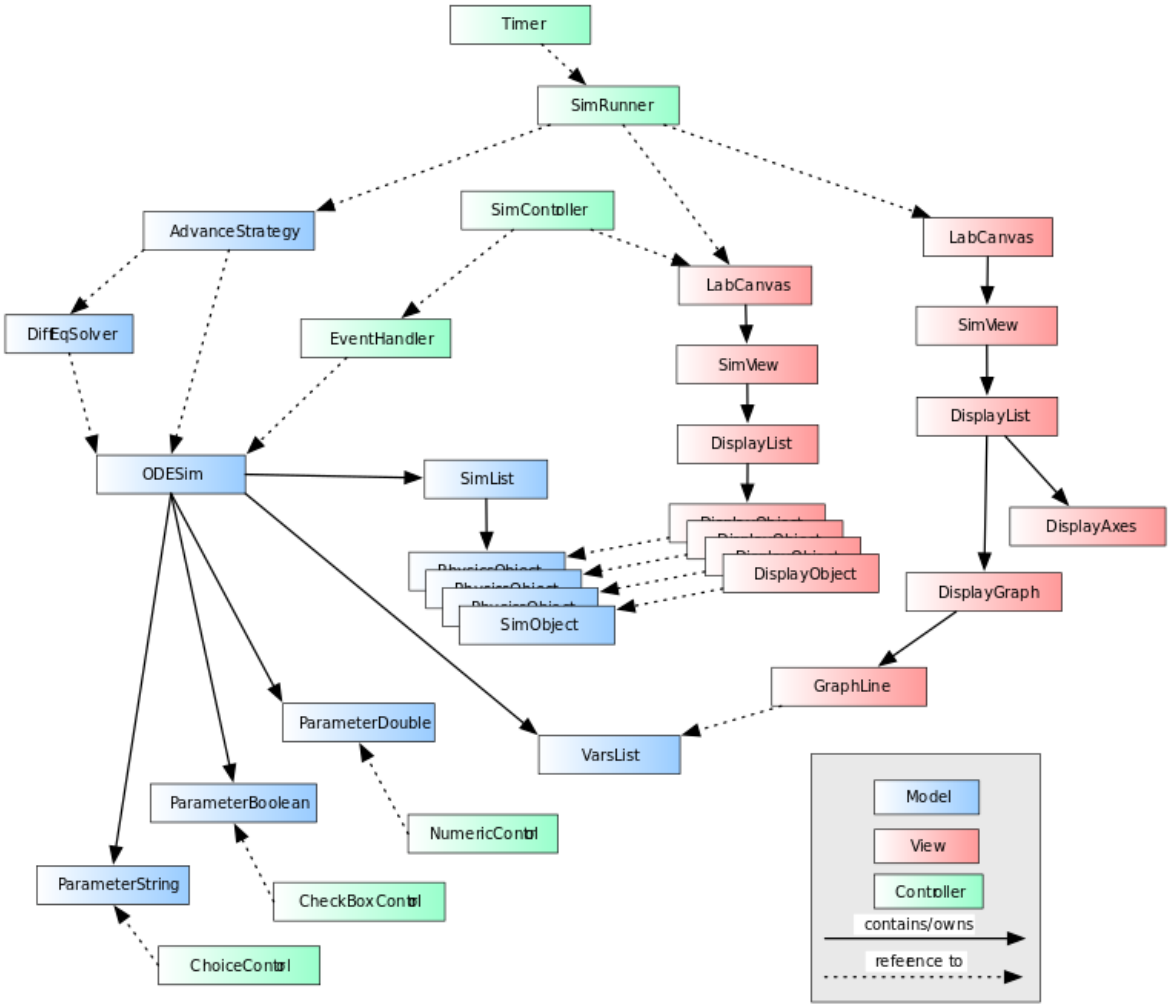


Wikipedia (2010) MVC-process. Available at: https://en.wikipedia.org/wiki/Model–view–controller

Model refers to the program’s data structure, i.e., objects and variables and can be thought of as a database with data logic. In some applications it can be static, receiving changes only as a result of direct user input. In the case of a physics engine, data will change overtime not requiring constant user input. (However, simulation time advancing can still be thought of as user input)

View is the representation of this data on the user interface. It is being synchronised with changes in model and decides how to represent the data on the screen.

Controller is responsible for handling user input as well as linking model to view (updating view according to changes in model).



Erik Neumann Overview Design. Available at:
https://www.myphysicslab.com/develop/docs/Architecture.html#overallarchitecture

The diagram depicts the architecture of the myPhysicsLab running. It corresponds to the network of objects created on the application start-up. Once an application as JavaScript object is created, model, view and control objects are instantiated within it. Every Display Object has a reference to the corresponding Model object. Controllers corresponding to dynamically changing parameters of the system are linked to the model and placed on the screen so the user can change these parameters in run-time.

2.4.2.2 Model

Most simulations of myPhysicsLab implement ODESim interface, a sub interface of the Simulation interface. The Simulation interface defines the mathematical model, i.e., set of differential equations that are to be solved to advance simulation in time. The task of solving these equations is left to CollisionAdvance or SimpleAdvance, implementations of AdvanceStrategy interface which do take collisions into account or do not, respectively. AdvanceStrategy. In its turn, AdvanceStrategy uses one of the implementations of DiffEqSolver which changes the variables that represent state of the system by numerically solving set of the simulation Ordinary Differential Equations (having multiple implementations allows to switch to a different one in run time. Extends possible user interactions with the system). The state of the simulation is kept as several SimObjects (objects that are the main subject of the simulation like rigid bodies, strings, etc.) in a SimList.

2.4.2.3 View

myPhysicsLab uses HTML5 canvas to graphically represent the system which is managed by the LabCanvas containing a list of LabViews which are what is being drawn on the canvas (multiple LabViews are drawn overlapping in such a way that the required one is drawn last to be visible). Each LabView has its own simulation and screen rectangles as well as simulation and screen coordinates. A CoordMap provides a mapping between them. It allows to easily change translation and scaling of the view. It can be done using pan and zoom tools on a GUI. The simulation creates its DisplayObjects corresponding to SimObjects at start up deciding what graphical parameters, such as colour, thickness, to use.

2.4.2.4 Controller

The controller is implemented through the SimController class which acts as a link between the model (ODESim) and view (LabCanvas). In this way, the SimController receives information about the user mouse/keyboard interactions with the canvas, applies some pre-processing (like converting mouse coordinates from the screen coordinate system to simulations coordinates) and sends to an Event Handler, usually implemented in the ODESim.
The user can modify system using sliders or typing specific values for them. It is implemented using Observer design pattern. An access to the value is provided by the parameter object of the subject which is linked to an input HTML element.

2.4.3 Conclusions

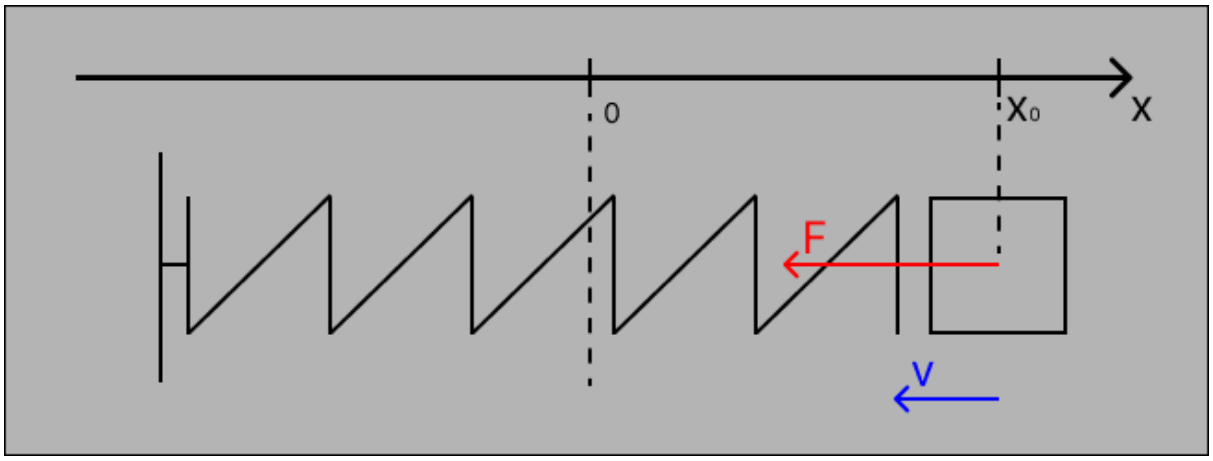
- It is better to design simulations with further possibilities to incorporate challenges, so students can complete tasks tailored to test different aspects of understanding the topic. The challenge discussed in overview of “Mystery” setup of density simulation is a good example. It might be beneficial to create different levels of progression moving students to harder challenges as they go further.
- A reward system for completing challenges could give students a sense of accomplishment making their studies more engaging. 3-star system from the projectile motion simulation is a simple example of one such reward.
- Bar or pie charts evolving through time can give a good idea of energy conversion in the process.
- It could be a good practice to abstract the sprite to a simpler form of object, e.g., a sprite of a moving object to a point mass with no spatial extent or a track to a curve with no width.
- Secondary information about the system is better to be left for users to turn on whenever they need it (as with the table of densities or graphs / charts). It will simplify the interface, enhancing the user experience.
- Simplicity in the user interface is a key to retaining users.

2.5 Computational Methods

2.5.1 Heuristics

The core of the strategy that advances simulations through time is a set of ODEs derived from the mathematical model of the system. Every new frame reflects the state of the system updated a certain number of times between frames using some numerical integration method. Numerical integration is a heuristic technique that allows to simulate systems that do not have an analytical (closed form) solution. Consider an example of a block oscillating on a spring.

Mykyta Osovskyi
Dronfield Henry Fanshawe School



The state of this system can be defined by two variables v and x_0 representing velocity[5] and position of the block (in this case v and x are just numbers as the block is moving in just one dimension). To advance the system, we must now how these variables change overtime, i.e., $\frac{dx}{dt}$ and $\frac{dv}{dt}$. By the laws of mechanics, the derivative of position is just the speed of the block, and the derivative of the speed is acceleration.

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = a$$

The Hooke's law dictates that the force acting on the block in any moment is the stiffness of the spring, k, multiplied by the deformation of the spring.

$$F = -k\Delta x = -kx_0$$

According to Newton's second law:

$$F = ma \;\rightarrow\; a = \frac{F}{m}$$

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = -\frac{kx_0}{m}$$

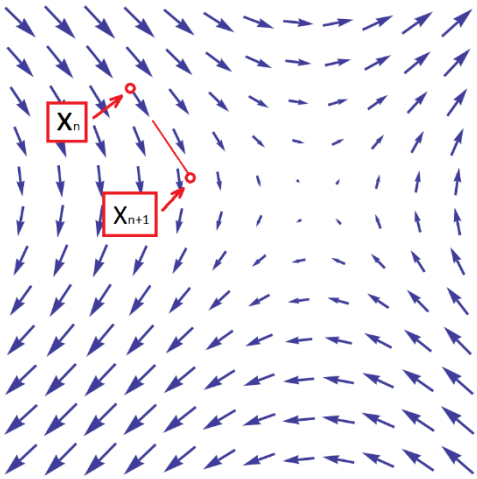
Using these two equations, the system can be updated every time step using one of the integration techniques. A choice of an integration method is itself a heuristic. All methods vary in accuracy and stability. More accurate method will require more time to compute the next state. If the time is too large, the simulation will not run in real time breaking experience of the user. At the same time, the simulation needs to be enough accurate to produce results accurate enough for students using the simulation to rely on them. Consider an Explicit Euler method:

$$v_{n+1} = v_n + \frac{dv}{dt}(x_n, t_n) * \Delta t$$

$$x_{n+1} = x_n + \frac{dx}{dt}(x_n, t_n) * \Delta t$$

[5] Velocity normally refers to the directed speed, i.e., the vector of the speed. As velocity in this case is one-dimensional, its direction (leftwards or rightwards) is represented by its sign.

Modelling the state of the system as a 2d vector x in a plane, calculation of the derivative of x gives a vector field. Euler method simply assumes that the vector (the change in x) is constant throughout the whole timestep which is not the case. The derivative at that point changes continuously and may change not just in space but in time as well. Smaller time steps follow the actual values of x more closely. It will take more time to compute the next state of the system for smaller time steps.
The choice of a time step and integration method is a heuristic that makes computer simulations possible. We may allow the time step to be bigger for smoother simulation in a cost of accuracy.



2.5.2 OOP approach

Object oriented approach is the most suitable paradigm for implementing a physics simulation application. The system consists of a range of objects with different and similar properties. It makes sense to gather common properties into super-classes (interfaces) and implement objects as their subclasses.

2.5.2.1 Abstraction

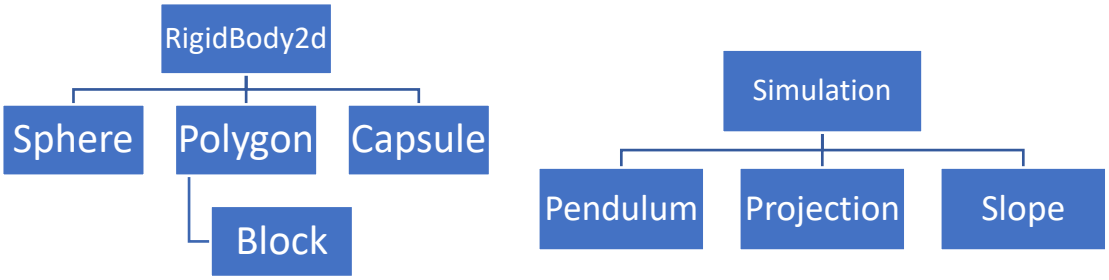
Every system is valid only under a certain set of assumptions. For the system discussed in a "heuristic" section, the only force taken into consideration is the tension in the spring. The Hooke's Law works only for small deformations which is disregarded in the model. However, for the sake of learning the law and aspects involved in it, this abstraction is sufficient.

The main purpose of abstraction in the code is to separate specific processes from one another, so code can be fully tested in isolation and then be used to interact with other pieces of code. Major examples are utility classes like vectors. Of course, it is possible to reimplement the dot/cross product using x and y components separately, but it will make code extremely harder to test and debug because of the vast number of points where a mistake could happen.

Other widespread practice in computer simulations is to separate the simulation from the objects and the environment where they are drawn following the model-view-controller architecture discussed above. It allows the view to interact with the model through a controller. The model in its turn takes care of advancing state of the system, keeping simulation time coordinated with real time, handling collisions, impulses, contact forces, etc. In this way, the code for graphical representation does not depend on the model which again assists debugging and readability.

2.5.2.2 Inheritance and Polymorphism

Objects of the simulation are all going to share common properties and methods. In my specific case, they are all going to be 2d rigid bodies meaning that all of them will be assigned mass, moment of inertia, coordinates of the centre of mass, etc. In other words, they share common attributes and methods. It makes sense to implement them to the 2d rigid body interface which other objects will inherit. Following the concept of polymorphism, they will have their own implementations of these functions as well as some additional, not common to the interface, properties. In the same way, multiple implementations of the "Simulation" interface will have different properties while other classes will be able to recognise their features common to interface.



Inheritance and polymorphism diagram example.

These concepts are particularly useful in the context of a physics engine as they promote reusability. All simulations will be using same classes of rigid bodies but the way they process them will differ.

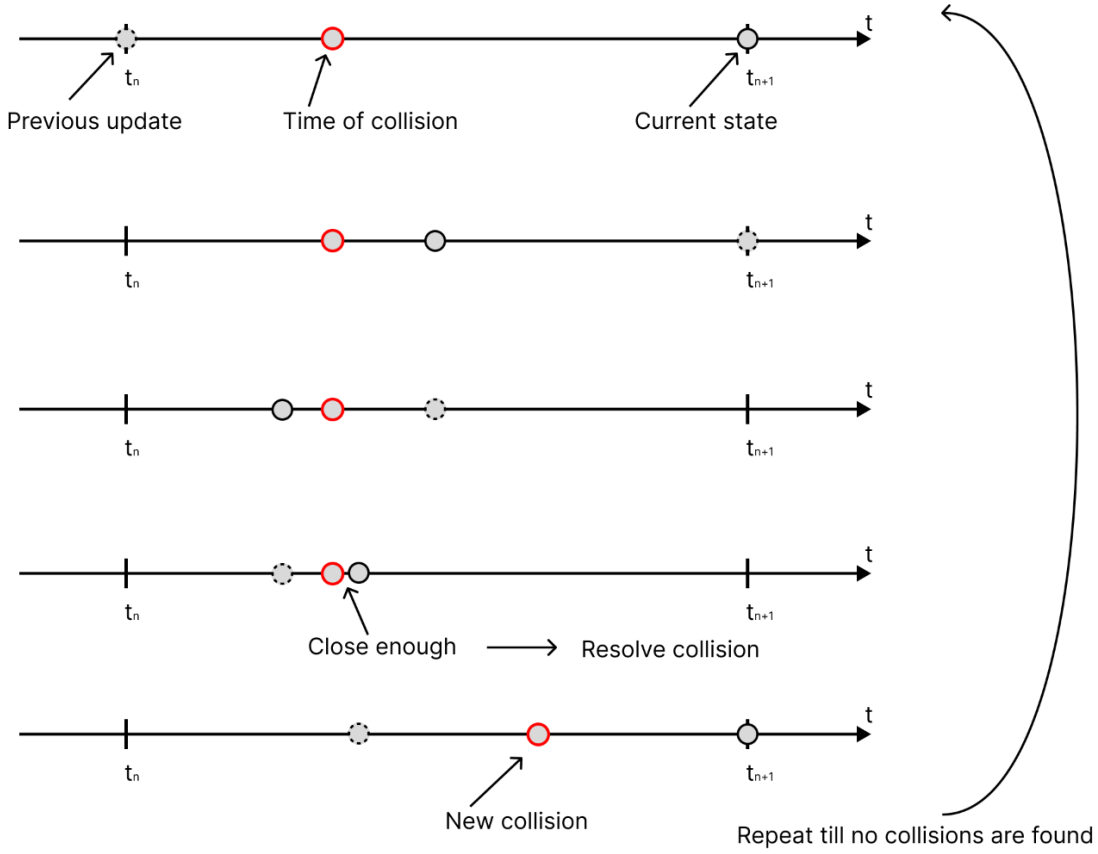
2.5.2.3 Encapsulation

Encapsulation in my project will be reflected in a usage of getters and setters that can potentially validate input parameters. It may become helpful when the user input is to be processed, so appropriate validation will be applied to protect the program from degenerate inputs and maintain robustness.

2.5.3 Divide and Conquer

Every time the simulation is advanced by a certain amount of time, there is a chance that a certain number of objects end up overlapping. There might be a lot of pairs of overlapping objects and resolving every pair of collisions at time might result in other collisions. One of the techniques to avoid it is to use binary search when resolving collisions, leading to a divide and conquer computational method. This approach is utilised in myPhysicsLab application reviewed at existing solutions section.

Whenever colliding bodies are found because of advancing simulation, the simulation will “go back” in time by a half of the simulation time between last update and current state. If no collisions are found, then the next iteration will view the state in a half of the time forwards, otherwise backwards. The process will be repeated until the time, when 2 objects are about to collide or are already overlapping by a tiny amount, is found. In this point, collision is resolved updating velocities of these objects.



Collision detection using binary search

2.5.4 Performance modelling

Some simulations will be flexible in terms of the number of objects that comprise the system (for instance when the user can add new objects). It inevitably results in more computations per update. If no measures are taken, simulation will get stuck at some point. To tackle this issue, performance modelling can be used during development / testing stage.
In regular circumstances, the internal clock of the system will aim to remain synchronous to real time. In this way, the program will always have some amount of time waiting the next iteration.

Example: say it takes 7 milliseconds to compute the next state of the system. The simulation runs in 60 frames per second, meaning there is 16 milliseconds between consecutive frames. Depending on implementation, the program will idle for 9 milliseconds before drawing the next state.
In a testing scenario, the internal clock can be used on its own to calculate each step of a simulation, not waiting before the 16-millisecond frame will be finished.
It can be used to save time to model stability of the simulation.

2.5.5 Visualisation

I will need to derive mathematical equations in many parts of my project. Some examples are collision handling and collision prediction, vector operations such as finding perpendicular vector or projecting a vector. Visualisation will help with understanding of each of them to correctly implement in the code. The same applies to differential equations that define the system and how it is updated overtime. (Diagrams will be coming up in various parts of the projects)

2.6 Features

2.6.1 Overview of the solution

This section establishes the direction of further stages as any decisions are made to meet one or the other success criteria or to implement a feature. A brief overview of how the solution is going to look like, acknowledging all the research done, will be given here.

A model-view-controller design pattern[6] will be used to split the program into more manageable parts.
The model will represent a physics engine, simulations, and their objects (also called bodies). It will be totally independent from other parts of the solution, whereas the View and certain parts of the Controller will be tightly linked to the model.
The view will consist of the classes that draw simulation on the screen as the graphical output. Separate classes for simulations and objects with links to their "model" reflections will be created. They will manage all the drawing routines and will store unrelated to the simulation parameters like colours or layers (if some objects are allowed to overlap).
The Controller will handle user inputs, communicating changes caused by them to the View and the Model. It will instantiate new simulations whenever it is required.

The solution will be developed as a static website. While no server-side processing is required, it will be initially hosted on "GitHub pages". Hence, all the code will be executed on the user side. The code will be mainly written in JavaScript. HTML5 canvas will be used to output the state of the simulation on the screen. HTML input tags for taking user inputs will be managed and placed on the HTML page by the Controller. Some simulations may support mouse or keyboard input.

2.6.2 Essential features

Essential features altogether will aim to be sufficient for the solution to be usable (meaning that potential users will find it useful and worth making effort of loading the application) by students or teachers in their studying or teaching. Features are derived from the research, discussions with the stakeholders, results of the survey and most common A-level mechanics problems.

Feature	Justification / Explanation
General features	
Simple to use interface. *when accessed from a desktop/laptop.	As the focus of the project is on the functionality, less time would be allocated to user interface and accessibility. However, making interface be easy to understand and not overwhelming for users is essential to provide good experience and motivate them to use the software.
Multiple simulations at a time.	This feature would allow users to run simulations under different configurations, e.g., altering velocity of the projected particle or turning air resistance on. May be beneficial for users to look at how different properties affect the system.
Web application as a static website. (Technical feature)	Web application hosted online makes it extremely easy for any user to have access to the program as everybody has some web browser. Furthermore, the

[6] In this section, the words: "view", "model" and "controller" will refer to the abstract view, model and controller parts of the project rather than to their meaning in colloquial English.

	program will be accessible from a wide range of devices right from the start with no additions to code. It also allows to extend to mobile devices in future.
Model-view-controller pattern maintained to an extent throughout the project. (Technical feature)	This approach enables centralised handling of user inputs and assists further development when a new feature is to be implemented.
Visual representation of simulation on the canvas. (Technical feature)	Simulations will have their visual representations that reflect current state of given simulation and draw it on the screen using canvas element associated with that simulation.
Input / Output area. (Technical feature)	Simulations will receive input and output from input boxes, sliders and buttons located on a certain area of the screen (most likely to the right of the canvas). Separating this form of input from input directly on GUI makes it easier to navigate the page.
Objects' position / velocity input and output.	Each simulation will have a list of objects with properties associated to them displayed on corresponding sections of input output area. Users can enter numbers into input area or use other input methods to change properties of different objects.
Typing maths equations into input boxes.	It eliminates the need to use a calculator before inputting information into simulations. In most mechanics' problems, vector quantities (such as velocity or force) are given in the form of magnitude and angle. They will need to perform extra calculations and round results before being able to use them in simulations. It gives inaccurate simulations and adds extra work to users. This feature was also requested by the stakeholders from the survey.
Text time input.	Allows users to observe state of the simulation at any given moment in time. Helpful for problems that ask you to evaluate position or velocity of an object after some amount of time.
Time slider.	A slider that controls current time of the simulation. Allows users to manually control time, come back to earlier moments or go forward as they wish. It is not a precise way to input time (unlike text time input), but rather a tool to interact with simulation.
Pause and continue buttons.	Two buttons to pause the simulation moving forward in time and continue it. A user will be able to inspect the state of the system at any given moment by pausing it and looking at all the objects of the system and their properties.
Reset time button.	Time slider and text time input can give the same result if the user inputs 0. However, in some cases, it is more convenient to just press the button and come back. May be helpful if users want to come back and try changing conditions of the system, while leaving the same objects.

Moving camera around simulation.	It allows students to choose certain areas of simulation that they are more interested in or follow the object as it moves in simulation.
Coordinate axes.	Coordinate axes give the user reference points on the scale and position of the camera to simulation. A user will be able to see how one unit of simulation space is mapped on the screen.
General-purpose simulation	
Placing a rigid body of a specific shape in the simulation.	All the objects of the general-purpose simulation are rigid bodies, either convex polygons or discs. A user must be able to choose an object they want to place on the screen and specify the position where it must be located.
Immovable objects.	Objects such as walls or obstacles. Increases the number of scenarios when the user might use the simulation.
Semi-Implicit Euler integration.	An easy to implement method which performs reasonably good for the computational resources it demands. Implementing numerical integration in a modular manner will allow me to switch between different methods and compare their performance in different conditions to pick the best one.
SAT collision detection method implemented.	The most appropriate and accurate collision method for convex polygons and discs.
Collision resolution.	Once detected, collision must be resolved by moving two objects apart from each other and applying impulses to change their velocities because of collision.
Gravity control.	An input box for the acceleration caused by gravity.
Particle projection simulation	
Placing a particle on the canvas.	Input menu for users to specify position and velocity of a new particle. It will be starting point of any particle projection simulation. Once the particle is in the system, users can look at its trajectory and see how its position and velocity changes overtime.
Manual simulation scale adjustment.	Depending on the numbers that the user inputs, the distance travelled by a particle may vary from being as small as ten pixels (e.g., for models of throwing an apple up in the air) to hundreds and thousands of pixels (e.g., for launching a projectile from the canon). If the scale of a simulation is constant, the user will not be able to see entire trajectory of the particle. Large distances will result in a particle going beyond the canvas, whereas small distances will be impractical to use. Therefore, it is essential to make scale adjustable.
Trajectory.	A particle can leave a dotted trail as it moves to form a curve of the trajectory dynamically. Or a full trajectory be created once the particle is on the screen. It might help students to learn how to draw correct diagrams to some problems.

Event system: run the simulation till an event has occurred.	A user will be able to specify an event. Some examples of events: the particle is 50 metres above the ground, the particle is 100 metres from the starting point, the particle returns to the same x/y coordinate, t seconds have elapsed from the start, the greatest height reached, etc. The simulation will pause when the event occurs allowing a user to look at position/velocity of particles and other properties of the system. This system provides a way to extract helpful information from simulation that has direct application to mechanics problems.
Simulation speed adjustment.	Different events will take up different time to reach. The simulation must adjust its speed maintaining the time to reach the event reasonable. So, the event does not occur instantly, and the user does not need to wait long for other events. Might consider a speed up button to give the user an option.
Finding the greatest height of a particle above the ground.	One of the examples of "events" discussed above. Stop when particle reaches its maximum height, look at its position and velocity.
Finding the total time of the flight from one point to the other.	One of the outputs from the event outcomes. The total time between the start (or a previous event) and the current event.
Identifying the time interval when a particle is a certain height above the ground.	Again, the time interval between two events: entering some height and leaving it.
Stop and inspect simulation after t seconds from the start.	Some problems ask students to find velocity or position of the particle after t seconds elapsed from the moment the particle was launched.

2.6.3 Less important features

Feature	Justification / Explanation
General features	
Mouse input.	It increases interactiveness of simulations. Allows to create objects much faster. On the other side, it does not substitute text input as you cannot get exactly accurate input with the mouse pointer.
Appealing user interface.	It would enhance user experience making the process of interacting with simulations more enjoyable (switch between white and black themes is desirable, as discovered from the survey).
Support for other devices.	A lot of students might not have access to a desktop/laptop while the software for such simulations does not require processing powers of a desktop computer. So, providing access for mobile devices would largely increase accessibility.
Explanations to simulations.	Formulas and explanations popping up when a certain event occurs to explain how the system evolved in time and how the user can use these formulas in their calculations. Explaining where the formulas come from,

	etc. It will make the software a better independent learning tool.
General-purpose simulation	
A polygon building tool.	An option for the user to place vertices of the polygon as they wish to form a polygon instead of choosing one of the already made shapes.
Advanced collision handling.	One advanced collision method was discussed in the "Divide and Conquer" part of the computational method section. It gives greater accuracy, however most of the simulations are not likely to run for long periods of time or to have many collisions.
More accurate numerical integration method implemented.	As discussed in the Research section, the main strategy for advancing general simulations in time is numerical integration. Different integration methods have different benefits depending on the system. RK4 is more computationally expensive than others, however, simulations are unlikely to involve large number of objects. Semi-Implicit Euler method on the other side is much faster and easier to implement, while not so accurate.
Particle projection simulation	
Automatic simulation scale adjustment.	An importance of adjustability was discussed above, this feature would make it easier for users. The simulation will estimate the space required to show the motion of the particle from start to finish and will adjust the scale accordingly.
Multiple particles.	It opens a wider range of problems that can be modelled with the software. Modelling two particles colliding with each other (from the kinematic point of view), finding the shortest distance between two moving particles, etc.
Velocity-time graphs.	Velocity-time graphs for the motion of a particle are studied in A-level Mechanics. Can also be put into a separate output window forming dynamically as the particle moves.
Zoomed in view on the particle.	A separate window zoomed on the particle with detailed information about it (e.g., a velocity vector projected on x and y axes). This is a useful way to output information graphically not overloading the main simulation view.

2.7 Success criteria

The task of creating a physics engine is vast and will take up a major part of the time to develop the solution. However, when it comes to needs of the actual users, a general-purpose physics simulation does not provide much help with the common problems of A-level Physics or Mechanics courses. All these problems have analytical solutions i.e., there exists a set of equations that determine a state of the system at a particular instance of time. They also do not involve random collisions. These factors eliminate the need to use numerical integration and complex techniques to detect and resolve collisions for such simulations.

As discovered from the conversations with stakeholders and potential users, it will really be useful if they could get a quantitative data about the system as an output. It will help them to understand where the numbers, they get from using the formulas, come from. It would also allow students to check the results of their own calculations against the output of the system by easily inputting the conditions of the problem.

Therefore, simulations in the program will be of two types: general-purpose simulations based on numerical integration and specific simulations that provide a range of tools to extract information about the system alongside the simulation. Some tools, such as zoom or coordinate grid, will be shared across multiple simulations.

To fit into time limits, the main success criteria will be to implement one specific simulation (particle projection simulation) including all the tools and functionality required for students and teachers to be able to productively use the program.

Success criteria will be based on features. Following table gives more specific details on what functionality one or the other success criteria requires rather than justification for why the feature is required (it is explained in previous section on features).

Success criteria	**Functionality details**
Simple to use interface. *when accessed from a desktop/laptop.	A user must be able to open the program and spend under two minutes, reading instructions before being able to navigate the page. 1) The user must understand how to create new simulation and close it; 2) How to open a menu for adding new objects; how to pass input to the program and how to add the particle; 3) Intuitively understand what each shape on the canvas represents: e.g., coordinate axes, particle, polygon, trajectory; The criterion is measured by how many of the points above are met when a user is presented with the program.
Multiple simulations at a time.	A user must be able to add new simulations on the page, while using each of them *independently*. Controls for each simulation must work separately and no input must affect other simulations than the one being used at that moment.
Web application as a static website. (Technical feature)	The web page will be accessible via a URL from any desktop browser. The code will be loaded on the user's computer and be processed client-side.
Object storage, update and drawing system	Simulations will need to be able to store objects (particles / polygons / discs). When simulation is active, they will need to be updated in accordance with how much time elapses between frames regardless of frame rate. A user will fill appropriate text boxes on the input output area. If input is invalid, an error message will be displayed, explaining how correct input must look like. Whenever the particle is successfully added to simulation, it will be displayed on the screen right

	away. Each particle must have its output block where the user can get current position and velocity of the particle When the object is deleted, simulation must clear out the space for the object and do the same to all other objects dependant on this one (for instance trajectory).
Typing maths equations into input boxes.	Alternatively, to numbers, a user will be able to input a maths equation in the format specified in instructions. The equation will be processed and resulting number returned into input box. If the format is not valid, the user will receive an error message to help with the format. Error message must be removed once the user passed correct input.
Time control system.	As mentioned earlier, simulations will advance in time when they are active and stop when paused. Each simulation will have to keep track of its internal time. Each simulation will need to have multiple ways for a user to control time. <u>Buttons:</u> Pause, continue, and reset buttons. A pause button will stop simulation from advancing in time while leaving it on the screen at its state. Continue button will put simulation in the active state, allowing it to advance in time. Reset button will reset simulation time to 0. <u>Text input:</u> A user will be able to input a number representing a moment in time since the start of simulation in seconds. Simulation will be paused, and the time of simulation will be changed to the user input value. An error message is displayed if input is incorrect. <u>Time slider:</u> Simulation time will be changed as the user moves a thumb along the slider. Numerical value representing current time will be displayed below the slider. The slider will have two points labelled with the value they represent on the slider, so the user can understand how fat along the slider they can move. Units on a slider represent seconds in simulation. The thumb must never go beyond edges of the slider. <u>Simulation speed adjustment:</u> A user must be able to choose one of the modes for simulation speed. Default value will correspond to one simulation second per real second. Once the mode is chosen, simulation rate of advancing in time will be adjusted.
Moving camera around simulation.	A user will be able to use WASD or arrow keys to move camera around simulation. With each key pressed, simulation will be translated by some value. Each object of the simulation will be translated.
Simulation scale adjustment.	Scale defines how many pixels on the screen correspond to 1 unit of space in simulation. Altering

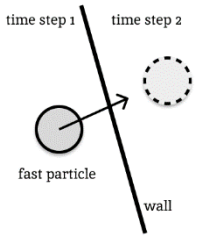
	scale allows zoom in and zoom out of certain areas of simulation. A user will be able to change the scale of the simulation using keys. To zoom in, the scale value will increase, so 1 unit of simulation space corresponds to more pixels on the screen. To zoom out, the scale will decrease. Each object on the screen will be moved accordingly. Objects with spatial extent will also appear to be of a different size when scaled. If the scale or translation are changed not by the user, the transition must happen such that the user understands why the image on the screen have changed.
Trajectory.	Once the particle is on the screen, the trajectory object will be created for the particle as well. The trajectory must be of a different colour on the screen and must be narrower than the radius of a particle. When the particle moves in simulation, it must perfectly follow the trajectory. The trajectory must be removed from simulation together with the particle.
Event system. Using events for maths problems.	Using a menu on the Input Output area, the user will be able to choose desired event type. Once the user has chosen the type of event, event condition input will appear. The user will need to choose the particle they work with and input conditions for the event. An error message must be displayed in the case of invalid input. When the event is added, the program must check if event is valid. If the event chosen by the user is invalid (for instance, it will never occur), no event is added, and the error message is displayed. Otherwise, the time before the event occurs is calculated and the event is enqueued to the simulation. Using one of the event types a user must be able to: - Retrieve the time when the particle reaches its highest point on the trajectory / reaches the ground / hits the wall; - Retrieve the time when the particle's velocity position attains certain values; - Set stopping point after specific amount of time (in other words stop simulation after user-specified amount of time). - Observe position and velocity of the particle at the moment event has occurred.
Coordinate axes.	The object is created when simulation starts. It must show direction of x and y axes as perpendicular arrows that intersect at one point, representing the origin of the simulation space. Each axis must have numerical labels that represent 1 unit of simulation space from the origin. The coordinate axes must not change when

	the scale of simulation changes, however labels will change their positions.

2.8 Limitations

Main limitation is that users will be able to solve problems that investigate the system from its initial state. For example, projecting a particle, given its initial position and velocity. Most real problems, on the other hand, would ask to find initial velocity given some other information about the particle. Nevertheless, students will still be able to validate their solutions with my software by inputting results of their calculations into my program and observing behaviour of the particle.
Also, some problems involve calculations around the fact that two or more particles collide at a certain point. With current set of features, it is not possible to model collisions of two particles, as they are considered as objects with no spatial extent. On the other hand, it would be possible to model collision of two objects in a general-purpose simulation. The closest approximation for two particles will be a model of two discs. However, they are not likely to give accurate results over large periods of time.

Updating simulation in fixed time steps means that we do not consider any changes happened in between updates. The model acts only as an approximation of reality, so two objects in real world in the same system with the same initial conditions can have a completely different state over time.



It not only introduces inaccuracy in updating positions of objects, but detection of collisions between two objects is also carried out every time step. In this way, there is always a risk that fast objects going in colliding trajectories might end up passing through the wall by the end of the time step, so the whole collision will be missed (demonstrated on diagram on the right).

The effect of this problem can be reduced with techniques like continuous collision detection. Continuous collision detection is a computationally expensive task, but it can reduce such errors leading to more accurate simulations. Other technique for more accurate collision detection was discussed in this section.

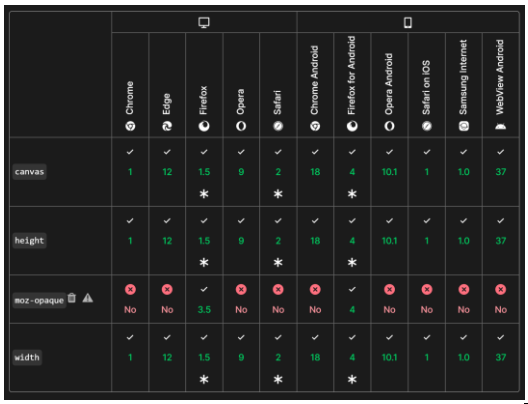
Because of this problem, small objects on high speeds cannot be modelled with real-time simulations. It is acceptable for a playground simulation that will not be used for research purposes. But for models like Brownian motion, accuracy of my simulation may be insufficient.

2.9 Requirements

Because I intend on making my program to be on a webpage, the only thing users are required to have is a computer and a browser installed on it. Different browsers vary on HTML elements they support. I will research browser compatibility of some HTML elements and list them down below.

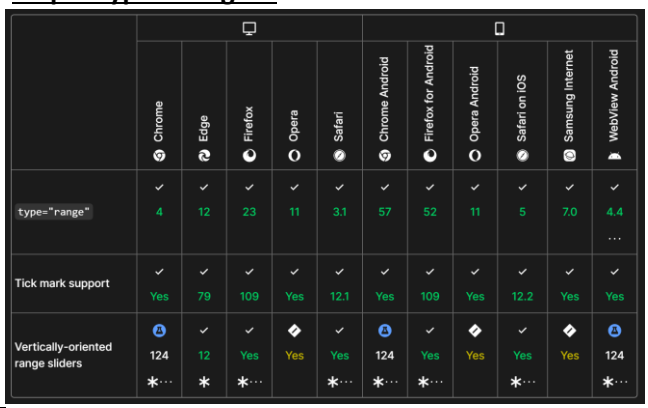
Mykyta Osovskyi
Dronfield Henry Fanshawe School

<canvas>



	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
canvas	✓ 1	✓ 12	✓ 1.5 *	✓ 9	✓ 2 *	✓ 18	✓ 4 *	✓ 10.1	✓ 1	✓ 1.0	✓ 37
height	✓ 1	✓ 12	✓ 1.5 *	✓ 9	✓ 2 *	✓ 18	✓ 4 *	✓ 10.1	✓ 1	✓ 1.0	✓ 37
moz-opaque	✗ No	✗ No	✓ 3.5	✗ No	✗ No	✗ No	✓ 4	✗ No	✗ No	✗ No	✗ No
width	✓ 1	✓ 12	✓ 1.5 *	✓ 9	✓ 2 *	✓ 18	✓ 4 *	✓ 10.1	✓ 1	✓ 1.0	✓ 37

<input type=”range”>



	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
type="range"	✓ 4	✓ 12	✓ 23	✓ 11	✓ 3.1	✓ 57	✓ 52	✓ 11	✓ 5	✓ 7.0	✓ 4.4 …
Tick mark support	✓ Yes	✓ 79	✓ 109	✓ Yes	✓ 12.1	✓ Yes	✓ 109	✓ Yes	✓ 12.2	✓ Yes	✓ Yes
Vertically-oriented range sliders	124 *…	12 *	Yes *…	Yes	Yes *…	124 *…	Yes *…	Yes	Yes *…	Yes	124 *…

Canvas and range input (for a time slider) are supported on most popular browsers, including mobile devices. And I am not going to use an unsupported “moz-opaque” property. Other than that, all elements of the webpage are more than common.

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android
requestAnimationFrame	✓ 24 …	✓ 12 *	✓ 23 *…	✓ 15 …	✓ 7 …	✓ 25 …	✓ 23 …	✓ 14 …	✓ 7 …	✓ 1.5 …	✓ 4.4 …

The “requestAnimationFrame()” is going to be the main function for updating simulations in time. It is also widely supported, so there should be no issues.

Simulations are intended to be real time, i.e., a user’s computer needs to be able to update simulation state in 16 to 32 milliseconds, which corresponds to 60 to 30 frames per second. This is a desirable frame rate for an average user. To reach a larger number of users, the software is going to be as light as possible, with little computational requirements.

As my program will run in a browser, I will use the system requirements for the most popular one, Google Chrome:

Chrome browser of version 109 or earlier (perfectly acceptable for my project) can be used on Windows 7 / 8 and 8.1. Windows 10 and 11 support the latest version of Chrome. Recommended processor is Intel Pentium 4 or later that’s SSE3 capable (these are all Intel CPUs of series Celeron, Pentium, Core, Xeon, and many others).

For Mac users: macOS Catalina 10.15 or later.

For Linux users: 64-bit Ubuntu 18.04+, Debian 10+, openSUSE 15.5+, or Fedora Linux 38+. Processor requirements are the same as the ones for windows.

If the project has sufficient support for mobile devices, the users will need to have Android 8.0 Oreo OS or later.

Information is taken from https://support.google.com/chrome/a/answer/7100626.

In terms of RAM, I believe that 4 gigabytes will be sufficient, even though more RAM is desirable.

3 Design

3.1 Defining I/O data

The most important task is to properly identify input and output data because it defines all possible interactions the user will have with the system. All inputs and outputs will be defined in this section. Type "text" will refer to any text box input. Some data may be labelled as both input and output, the program will accept input from the user and will change the value in the text box as simulation evolves over time.
Any text input that accepts real numbers may also accept a mathematical expression that evaluates into the number in specified boundaries.

General			
In or Out	**Type**	**Purpose**	**Validation**
In	Dropdown	Dropdown list of all the simulations that the user can instantiate on the page.	The number of simulations opened at the same time on the page will be limited, so they all can run in real-time. If no option is selected, nothing will happen.
In	Button	Adds selected simulation on the page.	
Out	Canvas	Graphical representation of the simulation state as an output on the canvas.	Scale of the simulation will be adjusted to fit the canvas.
In	Button	Button that terminates simulation it was linked to.	Click will be detected once the user releases the mouse button (default for any button).
In	Button	Button that erases everything from the simulation. It is more convenient for the user to restart the simulation without having to create a new instance of it.	None.
In	Button	Pause / continue button. Changes the simulation state from inactive to active and vice versa.	None.
In	Button	Add an object (depends on the simulation which objects can be added).	Possibly limiting number of objects at a time, blocking the callback function of the button to be called if the limit is exceeded.
In	Button	Delete an object from the simulation.	None.
In	Keys	Moving "camera" around the simulation using arrow / WASD keys.	The distance the user can move for may be bounded to

			prevent them from losing the main area of the simulation.
In	Keys	Zooming in and out using I/O keys on the keyboard.	Again, the minimum and maximum zoom will be bounded by some value to prevent degenerate input to affect the program.
Both	Text	x and y positions of the object.	Real number between -10000 and 10000. (may be changed further). As an output, the values will be rounded to 2 decimal places.
Both	Text	x and y components of the object velocity.	Again, real number between -10000 and 10000 (velocity is represented in m/s, any velocity over 10000 m/s is unreasonably large and might not be handled correctly by the simulation). As an output, the values will be rounded to 2 decimal places.
Both	Text	Alternative way of inputting velocity: angle above horizontal and magnitude.	Magnitude will be bounded by 1000 and clamped if the user inputs any larger number. An angle will be converted to be in the range of 0 to 360 degrees.

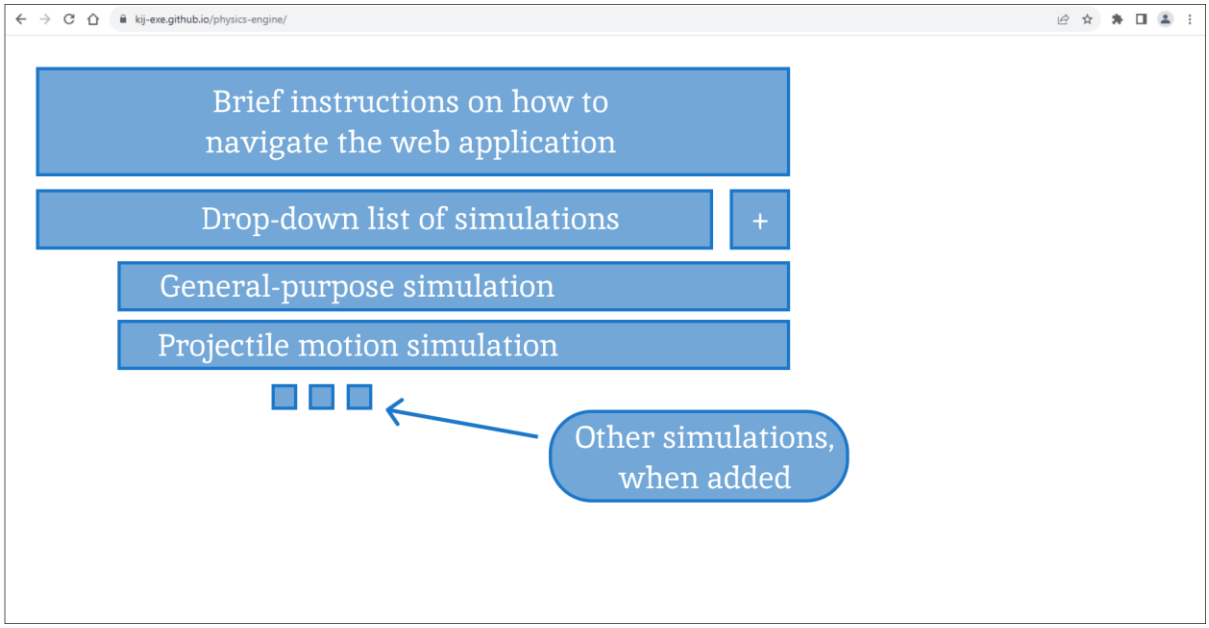
Particle Projection Simulation			
In or Out	**Type**	**Purpose**	**Validation**
In	Dropdown	Dropdown list of all allowed events.	The number of events in the queue will be bounded.
In	Button	Add an event to the queue of events.	
In	Text	Event-specific input, such as the time before the event, the height the particle needs to enter for the event to occur, etc.	Dependent on the event, generally accepts only real numbers and prohibits too large numbers.
Out	Text	Details about the particle once the event has occurred.	Output only essential data to not overload the area. More details could potentially be accessed by the user choosing to do so. For example, hovering mouse pointer over certain area will put more information on the screen.
In	Slider	Time slider.	The slider will be bounded by the minimum and maximum values and the user will be able to return to the time they started with

General-Purpose Simulation			
In or Out	**Type**	**Purpose**	**Validation**
In	Dropdown	Choosing a type of the shape to add to the simulation.	If the type is not chosen, the program will not add anything to the simulation if user clicks on the add button.
In	Button	Adding the body of a chosen shape to the simulation.	The number of bodies at a time in one simulation will be limited and the button will not do anything if the limit is reached.
In	Text	Mass of a body input.	A number greater than 0.1 (Better boundaries will be identified as the result of testing).
In	Button	Make the mass of a body be infinite (make the body immovable).	Default.
In	Slider	The size of the shape control.	Bounded by maximum and minimum values.

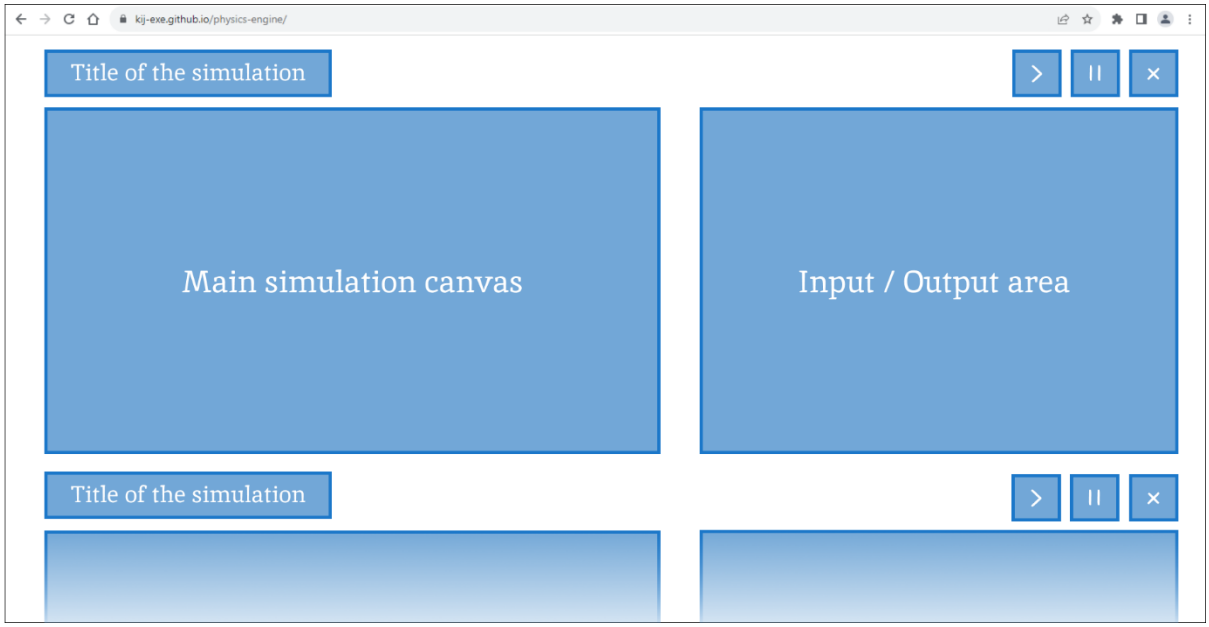
3.2 Decomposition

3.3 GUI Mock-ups

The blue colour is used mainly to highlight different areas (not actual interface colours).

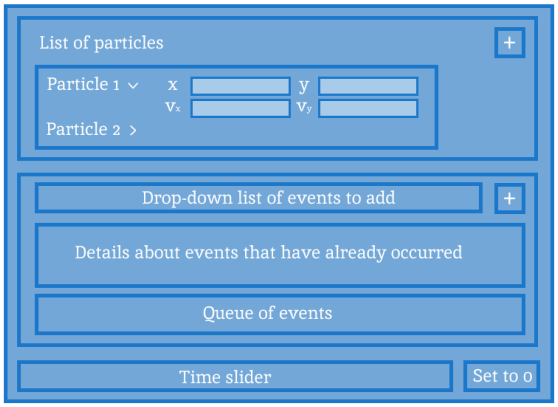


Entering the main page of the application, the user will be presented with brief instructions on how to use the software. They will be able to choose one of the simulations and press “+” button to add a new simulation on the page.

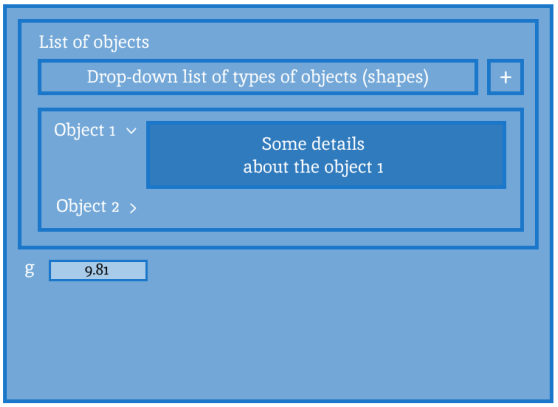


Simulations will be added with the title, close, stop and continue buttons and Input / output area attached to them. The simulation size is set to 800x450 pixels maintaining a common 16:9 aspect ratio. The size of the simulation area is a subject to change. The best configuration for the screen size is yet unclear and will be discovered throughout tests with stakeholders. It does not affect the flow of the project and this type of amendment easily fits at any stage. Taking a closer look at the Input / Output area:

Particle projection sim



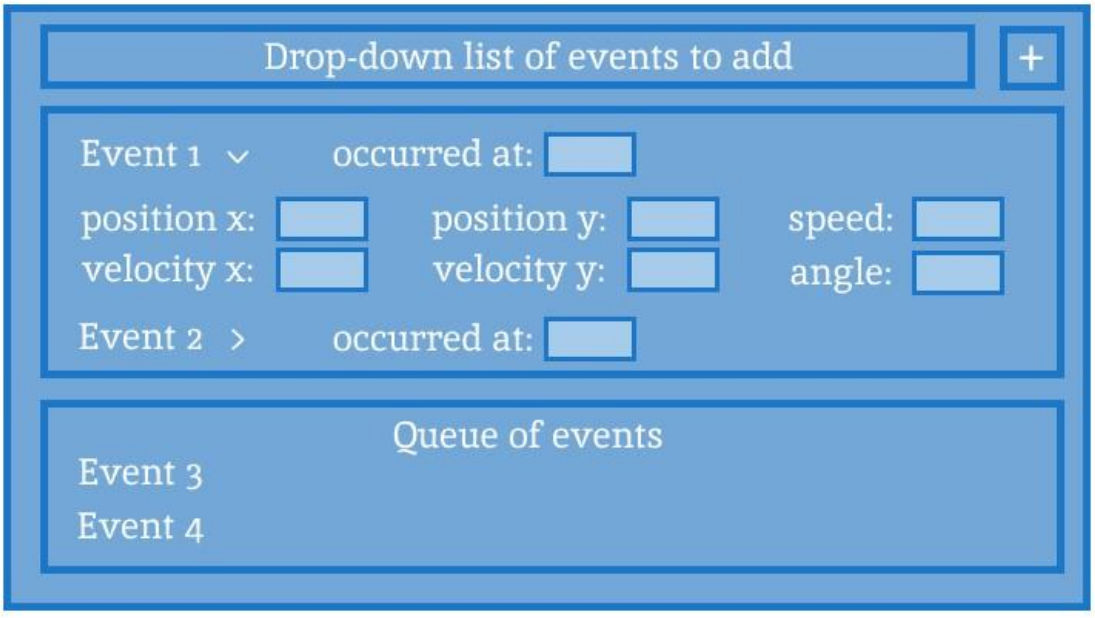
General-purpose sim



The user will be able to add new particles to the system and specify their positions and velocities. A drop-down list of events to add provides a choice between different types of events. It will also put a menu to specify conditions for a specific event, if required. It will vary from one event to the other and it will be designed down below in the Structure section. The queue of events will be displayed below the list of events that have occurred with some information about each one. On the bottom of the input output area will be a time slider that the user will be able to use to return to earlier points in time or jump forward. The button to set the time to 0, i.e., return to the start will be on the right of the time slider. The initial value for the gravitational constant (g) input will be set to 9.81 (measured in ms^{-2})

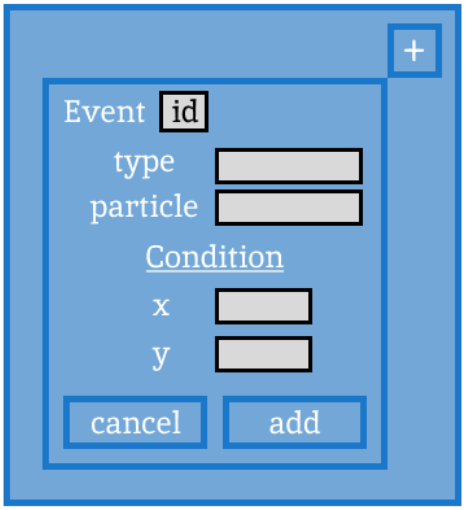
The size of the input output area will vary depending on the number of particles / events added by the user. So, it is essential to allow this area to be flexible either by growing as there appears more information or by scrolling down in specific areas (e.g., hover a mouse cursor over the area for a list of particles and scroll down to see other particles).

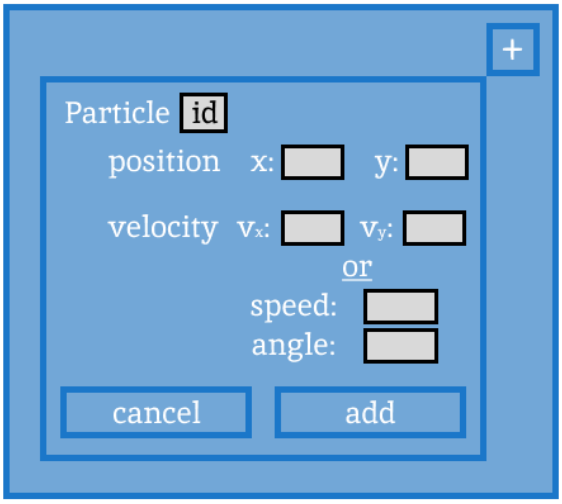
Taking a closer look at the event area:



Information about the event and the particle(s) that the user might need will be showed as output once the event has occurred. Queue of upcoming events might also come in handy for more complex problems. This area will be contained within the main input / output area.

Click on the plus button will put a pop-up menu on top of the I/O area. First designs for these menus:





The user will be able to specify all the required parameters for the event/object they want to add. The menu for adding objects to a general-purpose simulation is identical to the one for a particle, however, the user will also be able to specify the shape, mass, and size of the shape with the slider.

3.4 Pseudocode / Code conventions

- Pseudocode will mostly follow the OCR AS and A level Pseudocode Guide.
- Class names must be written in PascalCase, whereas functions must use camelCase.
- Attribute/parameter identifiers must be written in lower case. If multiple words form an identifier, they must be separated by an underscore.
- Identifiers for methods and attributes must be descriptive if it does not make them too long or tedious to use and remember (this definition is quite vague; the rule is in place just as a reminder to keep the code more readable). The exception to this rule is local variables or parameters that exist only within a function to break down calculations into multiple lines assisting readability. These will often be taken from the formulae (all explanations are provided right before pseudocode or as comments to pseudocode). For example, using “x0” / “y0” / “m2” as variables for equation.
  Some identifiers might be short and still descriptive enough, e.g., the gravitational constant G, it will not make sense to name it $gravitational\_constant$ as this constant is not likely to be used anywhere outside of the function calculating gravitational forces.
- Using American spelling for identifiers (e.g., “color” rather than “colour”, “normalize” instead of “normalise”). It will maintain integrity with HTML and CSS spellings. This rule does not apply to documentation or comments.
- Pseudocode primitive data types: real (any number), int (integer number), string, char, bool.
- List data structure will be used as a dynamic array.
- Every class will have a “`constructor()`” method (unless it is a library of static methods). This method will be called every time a new instance of the class is created (basically instantiation).

- "***this***" variable, borrowed from the JavaScript syntax, will denote a self-reference. Whenever a method of the object needs to call another method or refer to the attribute of the same object, its identifier will be used preceded by "`this.`". E.g., accessing the "x" attribute inside the Vector class will look like "this.x".
- Every parameter is assumed to be passed by reference, unless it is of a primitive data type, such as integer, real or char.
- Inheritance is denoted by the "`extends`" keyword. It refers to the class inheriting methods and attributes of the other class or an abstract class. Methods and attributes that are not overridden by the child class will not be reiterated in the child class pseudocode. Superclass constructor shall be called with the "`super`" keyword.

3.5 Structure

Each separate part of solution will have its own section which describes its design.

UML class diagrams:

Along explanations of decomposition and design of different parts of the solution, UML class diagrams will be constructed. Entire class diagram will be presented at the end.

Pseudocode implementations:

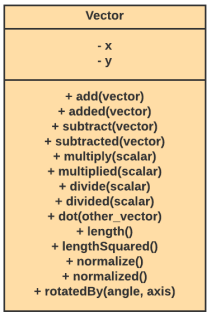
Pseudocode implementation will be provided for any algorithm. Diagrams and formulas will sometimes explain how the algorithm was derived and therefore how it works.
"Sim" will be widely used as an abbreviation for Simulation.
HTML input and canvas tags alongside event listeners will be extensively used for inputs and outputs. Their use will be either highlighted and explained in plain English stating the action that needs to be performed, e.g., "create a button" or a foreign method will be used with explanations in comments. Unchanged inherited attributes and methods will not be repeated in the class diagram of the child class.

3.6 Start-up page

The start-up page will be a simple HTML document with the basic welcoming text and instructions on how to start. It will have containers that will be used by the Controller to place the drop-down list of options for simulations and container that will consist of simulation areas: a simulation canvas and input / output area. In this way the JavaScript code as a Controller will define what simulations can be created on the page without having to change any HTML code. The style of the page will be managed by another CSS document. The style is a secondary consideration.
The webpage itself will be hosted on GitHub pages, as discussed earlier.
The webpage will contain a script that instantiates a controller and stores it in a global variable, so it does not get deleted by the garbage collector. All the other functionality is managed by the JavaScript code from the controller.

3.7 Utility Package

3.7.1 Vector



Vectors will be used extensively across the project for various tasks, from storing velocity and position to finding a projection of the point on the line. The vector object will have two attributes, x and y representing its two components.

3.7.1.1 Arithmetic operations

Arithmetic operations such as addition, multiplication, etc. of two vectors or a vector and a scalar will be implemented in two methods, one that returns a new vector as a result and the other one that directly modifies the vector on which the method was invoked.

```
public procedure add(vector: Vector)
      this.x += vector.x
      this.y += vector.y
      // accessing x and y directly
endprocedure
```

```
public function added(vector: Vector) -> Vector
      x = this.x + vector.getX()
      y = this.y + vector.getY()
      //   creating temporary variables for new x and y variables,
      //   original vector is unchanged
      return new Vector(x, y)
endfunction
```

```
public procedure multiply(scalar: real)
      this.x *= scalar
      this.y *= scalar
endprocedure
```

```
public function multiplied(scalar: real) -> Vector
      x = this.x * scalar
      y = this.y * scalar
      return new Vector(x, y)
endfunction
```

Similarly for divide - divided, subtract - subtracted. Important to note that subtraction methods will subtract the parameter vector from the vector on which the method was applied.

3.7.1.2 Dot product

Important vector operation on two vectors extensively used for physics simulations. It is based on the result of the following formula discussed in analysis section:

$$\vec{a} \cdot \vec{b} = x_a x_b + y_a y_b = |\vec{a}||\vec{b}| \cos(\vec{a}, \vec{b})$$

```
public function dot(other_vector: Vector)
      return this.x * other_vector.x + this.y * other_vector.y
endfunction
```

3.7.1.3 Length

Length of the vector is computed as the square root of the sum of its squared components. The length of the vector is often required in its squared form (without a square root) which is easier to

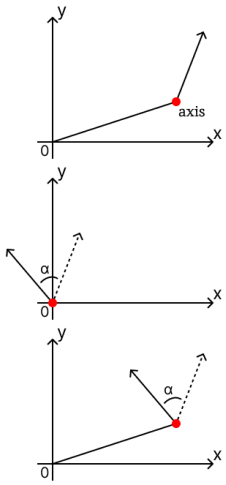
compute, so the vector class will have two methods, “lengthSquared()” which computes the sum of squared x and y components. And “length()” which computes the square root of the square length.

3.7.1.4 Normalisation

The process of creating a unit vector in the same direction as given vector. The vector components must be divided by its length. As well will come in two methods. “normalize()” and “normalized()”. Must watch out with validation as zero vector cannot be normalised.

3.7.1.5 Rotation

Also, a common operation for vectors. Vector class will provide “rotatedBy(angle, axes)” method that will return new Vector as the base vector, on which the method was applied, rotated anticlockwise around the axes. The “axes” parameter is a point in 2d plane around which the base vector is rotated (In this case, it is more of a point being rotated around the other point).



Rotation around the origin is commonly accomplished using the rotation matrix multiplied by the vector in the form of 2 by 1 matrix:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\alpha - y\sin\alpha \\ x\sin\alpha + y\cos\alpha \end{pmatrix}$$

The right-hand side of the equation is the vector $\begin{pmatrix} x \\ y \end{pmatrix}$ rotated by the angle α around the origin.

To accomplish rotation around any point (axis), the vector needs to be first translated such that the point about which the vector is to be rotated is at the origin. After that, rotate the vector and translate it back.

```
public function rotatedBy(angle: real, axis: Vector)
      rotated_vector = this.subtracted(axis)
      //   first tranlate the axis to the origin
      rotated_vector = new Vector(
            rotated_vector.x*cosine(angle)-rotated_vector.y*sine(angle),
            rotated_vector.x*sine(angle)+rotated_vector.y*cosine(angle)
      )
      //   rotate vector like that
      rotated_vector.add(axis)
      //   translate the axis back
      return rotated_vector
endfunction
```

The default value for the axis can be set to “new Vector(0, 0)” (the origin of 2d plane)

3.7.1.6 Testing

Because nearly all the methods are applied to some vector, the input will consist of the “base vector” to which the method is applied and the “parameter vectors”. As methods for vector arithmetic operations have two implementations each, only one that creates a new vector will be tested (the same logic is applied to both implementations so if there is an error in one of them, it will be the same for the second one).

Function being tested	Input	Expected Output
added()	Base vector: (2, 9)	(6, 17)
subtracted()	Parameter: (4, 8)	(-2, 1)
added()	Base vector: (-10, -11)	(-7, -8)

subtracted()	Parameter: (3, 3)	(-13, -14)
added()	Base vector: (3, 3)	(-7, -8)
subtracted()	Parameter: (-10, -11) Base and parameter vectors swapped, addition gives the same results, subtraction must give opposite signs.	(13, 14)
multiplied()	Base vector: (6, 12)	(24, 36)
divided()	Parameter: 3	(2, 4)
multiplied()	Base vector: (-7, 200)	(-28, 800)
divided()	Parameter: 4	(-1.75, 50)
multiplied()	Base vector: (0, 90)	(0, -900)
divided()	Parameter: -10	(0, -9)
dot()	Base vector: (0, 10) Parameter: (-7, 9)	90
	Base vector: (2, 2) Parameter: (-1, 2)	2
	Base vector: (0, 10) Parameter: (-7, 0)	0
length()	Base vector: (3, 4)	5
	Base vector: (12, -5)	13
lengthSquared()	Base vector: (2, 3)	13
	Base vector: (6, -10)	136
normalized()	Base vector: (3, -4)	(0.6, -0.8)
	Base vector: (80, -60)	(0.8, -0.6)
rotatedBy()	Base vector: (5, 8) Angle: 1 Axis: (0, 0)	(-4.030, 8.529) to 3dp rounded down
rotatedBy()	Base vector: (-20, 8.1) Angle: 13.8 Axis: (0, 0)	(-14.260, -16.194) to 3dp rounded down
rotatedBy()	Base vector: (4, 7) Angle: $\frac{-\pi}{2}$ Axis: (0, 0)	(7, -4)
rotatedBy()	Base vector: (3, 3) Angle: $\frac{-\pi}{2}$ Axis: (2, 1)	(4, 0)
rotatedBy()	Base vector: (5, 2) Angle: $\frac{\pi}{4}$ Axis: (3, 4)	(5.828, 4.000) to 3dp rounded down

Other useful methods:

Reflecting the vector in x and y axes is a useful operation, for example, used for coordinate conversion.

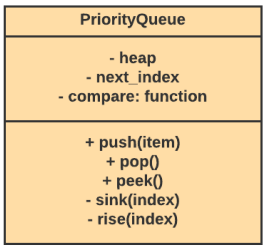
```
public procedure reflectedInX()
      new_y = this.y * (-1)
      return new Vector(this.x, new_y)
endfunction
```

```
public procedure reflectedInY()
      new_x = this.x * (-1)
      return new Vector(new_x, this.y)
endfunction
```

These methods would represent reflection in x or y axes. So, if the vector is reflected in x axis, its y component is changed.

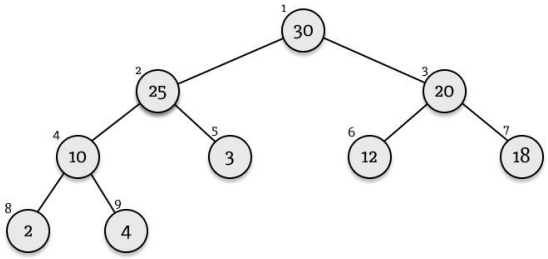
3.7.2 Priority queue

A priority queue data structure will be the main element for storing and retrieving events in the order they occur in time. Priority queue is a dynamic data structure that provides an efficient way to store elements in a sorted order with efficient insertion and retrieval of elements. It also can be used to store collision predictions when advancing simulations in time, store objects in the order they must appear on the screen, etc. Priority queue implemented for the program will be based on a concept of heap maintained in the list.



3.7.2.1 Heap data structure

Heap is a tree-based data structure that satisfies a property for every node. For a max heap, a parent node is always greater than or equal to the child node. For a min heap, a parent node is always less than or equal to the child node. A heap can be represented in memory as a dynamic array, where the heap starts at index 1 and for every kth element, two child nodes are (2k)th and (2k + 1)th elements and the parent node index is obtained as an integer division of k by 2.



The order of the priority queue as well as the parameter by which to compare two elements is defined by the "`compare()`" function passed to the constructor of the priority queue. The comparator function must take two parameters and return a number. If the number is negative, the first parameter is thought to be less than the second; if the number is 0, the elements are equal and if the number is positive, the first one is larger than the second one. This function allows to reuse the same implementation of the priority queue for any type of objects just by specifying a comparator function.

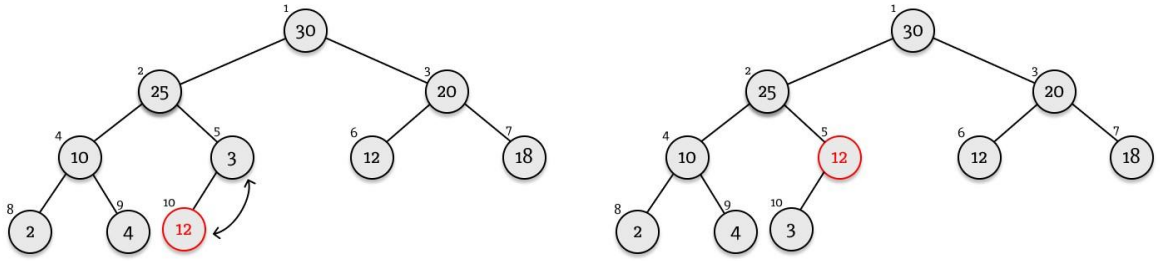
```
public procedure constructor(compare: function)
      this.heap = [null]
      //   a list that is used to store the heap, the first element is
      //   null because the heap starts at index 1
      this.next_index = 1
      //   index of the next free space in the list
      this.compare = compare
endprocedure
```

The main functionality of the priority queue is represented in "`push(item)`", "`pop()`" and "`peek()`" methods that push a new item into the queue, return the first priority item from the

queue deleting it from the queue and return the first priority item not deleting it. These methods will be assisted by "`sink(index)`" and "`rise(index)`" methods that will maintain the property of the heap.

3.7.2.2 Pushing

Whenever a new item is pushed to the queue, it is placed at the end of the heap. However, this operation might break the property of the heap if the new element is larger[7] than its parent. To maintain a heap, the "`rise(index)`" method will be called on the index of the new element in the queue. It will make the new element rise the heap until the property is satisfied.



Adding 12 to the depicted heap will require one swap between 3 and 12 to return the heap to its correct order. "`rise(index)`" method works by repeatedly checking if the item exceeds its parent and performing a swap in this case.

```
private procedure rise(index: integer)
      while index > 1
            //   while the current item is not in the first node
            //   (because the first node cannot rise anymore)
            parent_index = index div 2
            //   getting index of the parent element
            item = this.heap[index]
            parent_item = this.heap[parent_index]
            if this.compare(item, parent_item) > 0
                  //   if the item is larger than its parent, perform a
                  //   swap
                  this.heap[index] = parent_item
                  this.heap[parent_index] = item

                  index = parent_index
                  //   the next node to consider is now a parent of
                  //   the previous node
            else
                  //   if the item is less than its parent, then the
                  //   property is satisfied
                  break
                  //   exit the loop
            endif
      endwhile
endprocedure
```

Based on this method, the push method will be implemented in the following way.

[7] The priority queue follows the order defined by the "compare" function, so the word "larger" is correct for a max heap, but the same rules apply for any ordering.

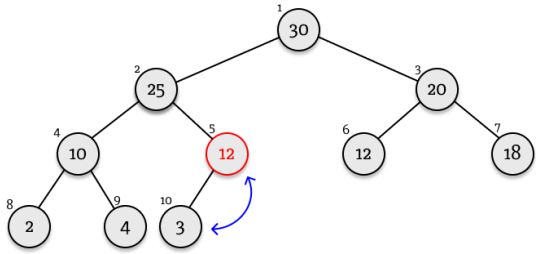
```
public procedure push(item)
        this.heap[this.next_index] = item
        //   add item to the heap
        this.rise(this.next_index)
        //   fix the property of the heap
        this.next_index += 1
        //   increment the index for the next element
endprocedure
```

3.7.2.3 Testing pushing

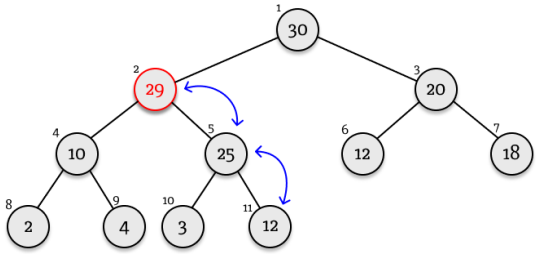
First initialise a heap that has f(a, b) = a – b function as a comparator. It must make a queue be max oriented. Next build a heap from the diagram above that explains how new elements are added. Push 30, 25, 20, 10, 3, 12, 18, 2, 4 in this order. Further tests are applied to this priority queue one by one.

The first "push(12)" operation was already described above. Exploring further pushes diagrammatically:

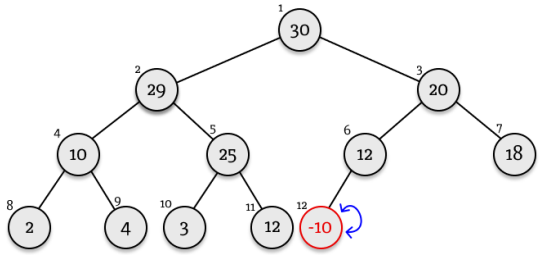
1) push(12)



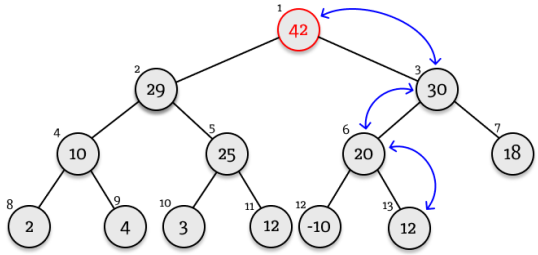
2) push(29)



3) push(-10)



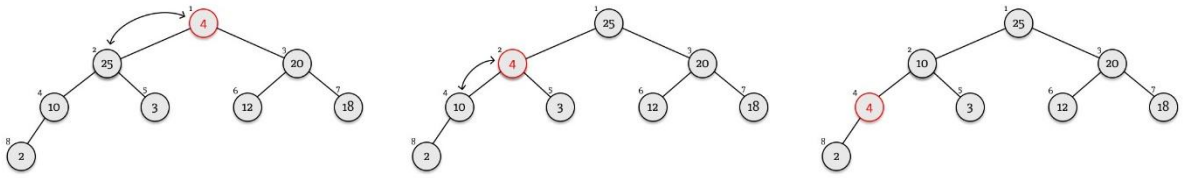
4) push(42)



Input	Expected Output
(0) Initial queue	Heap: [30, 25, 20, 10, 3, 12, 18, 2, 4]
(1) Push 12	Must result in only one swap with 3. Making a heap list look like: [null, 30, 25, 20, 10, **12**, 12, 18, 2, 4, **3**]
(2) Push 29	29 swapped with 12 29 swapped with 25 Heap: [null, 30, **29**, 20, 10, **25**, 12, 18, 2, 4, 3, **12**]
(3) Push -10	No swaps [null, 30, 29, 20, 10, 25, 12, 18, 2, 4, 3, 12, -**10**]
(4) Push 42	42 swapped with 12 42 swapped with 20 42 swapped with 30 [null, **42**, 29, **30**, 10, **25**, 20, 18, 2, 4, 3, 12, -10, **12**]

3.7.2.4 Popping

Retrieving the first element of the queue results in an empty space at the first node. The space is filled with the last element, which again breaks the property. To fix it, the element must "sink" down the heap.



When the "`pop()`" method is invoked, the last element in the queue (4 in this case) is placed at the top. Sinking down the heap, it gets swapped with 25 and 10 before it reaches its correct place. "`sink(index)`" method works by repeatedly swapping the element with the largest of its two children until it has reached its correct place in the heap.

```
private procedure sink(index)
      while 2 * index < this.next_index
            //   index of the left child of the node is 2 * index
            //   therfore this condition is satisfied as long as there
            //   at least one child of the node with this index
            //   if there are no children, then the node is already
            //   on the correct place

            if 2 * index + 1 < this.next_index
                  //    if there is right child
                  left_item = this.heap[2 * index]
                  right_item = this.heap[2 * index + 1]

                  if this.compare(right_item, left_item) > 0
                  //   if the right child is larger
                        largest_child_index = 2 * index + 1
                  else
                        largest_child_index = 2 * index
                  endif
            else
                  largest_child_index = 2 * index
            endif
            //   previous block of code finds the largest child of
            //   the current node

            parent_item = this.heap[index]
            largest_child_item = this.heap[largest_child_index]

            if this.compare(largest_child_item, parent_item) > 0
                  //   if the largest child is larger than its parent
                  //   perform a swap
                  this.heap[largest_child_index] = parent_item
                  this.heap[index] = largest_child_item

                  index = largest_child_index
                  //   the next node to consider is now at this index
            else
                  break
                  //   exit the loop otherwise
            endif
      endwhile
endprocedure
```

The "`pop()`" method, in its turn, returns the required element, places the last element at the top of the heap and invokes "`sink(index)`" method for the new top element of the heap.

```
public function pop()
      if this.next_index == 1
            //   if the queue is empty
            return null
      endif

      item = this.heap[1]
      //   required item temporarily stored

      this.heap[1] = this.heap[this.next_index - 1]
      //   place the last elememt at the top
      this.next_index -= 1
      //   decrement the next index attribute

      this.sink(1)
      return item
endfunction
```

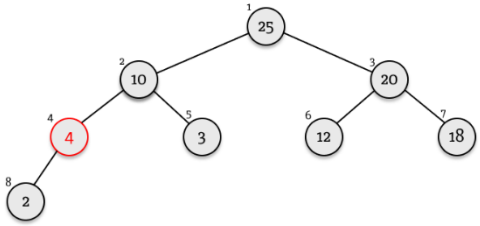
The "`peek()`" method will simply return the top element.

3.7.2.5 Testing popping

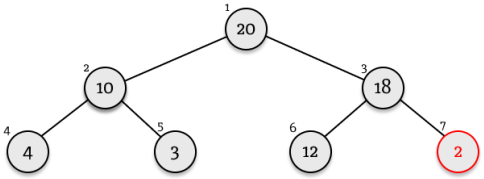
Starting with the same queue as was formed for the "push()" method test, pop elements sequentially and check if the value received as expected and if the heap contains elements in expected order.

Exploring visually what must happen after each "pop()" method called:

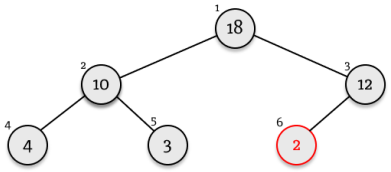
1) First example was discussed before, diagram on the right represents structure of the heap after the first "pop()" method call.



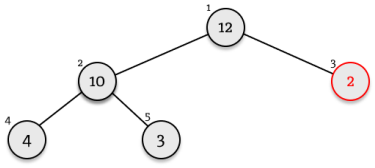
2) 2 gets to the top of queue and "sink()" method is called to fix the property of the queue. 2 gets swapped with 20 and then with 18.



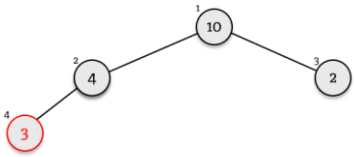
3) 20 is replaced with 2, 2 is swapped with 18 and with 12.



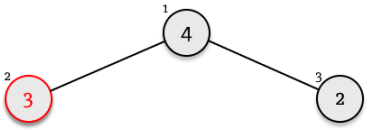
4) 18 is replaced with 2, 2 is swapped with 12.



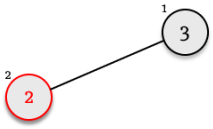
5) 12 is replaced with 3. 3 is swapped with 10 and 4.



6) 10 is replaced with 3. 3 is swapped with 4.



7) 4 is replaced with 2. 2 is swapped with 3.



8) 3 is replaced with 2.

9) the only element 2 is popped from the queue.

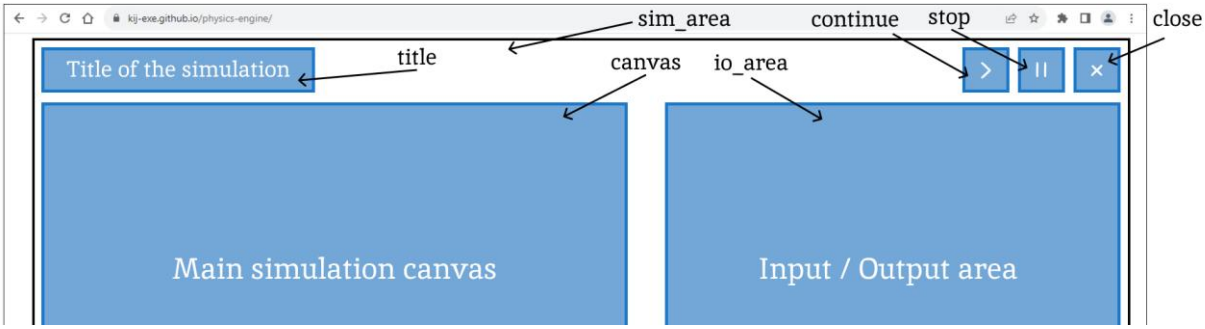
10) null returned as queue is empty now.

Input	Expected Output
Start with the same queue as the one for the "push()" method tests: **[null, 30, 25, 20, 10, 3, 12, 18, 2, 4]**	
(1) pop() **peek()**	30 heap: [null, 25, 10, 20, 4, 3, 12, 18, 2] 25
(2) pop() **peek()**	25 heap: [null, 20, 10, 18, 4, 3, 12, 2] 20
(3) pop() **peek()**	20 heap: [null, 18, 10, 12, 4, 3, 2] 18
(4) pop() **peek()**	18 heap: [null, 12, 10, 2, 4, 3] 12
(5) pop() **peek()**	12 heap: [null, 10, 4, 2, 3] 10
(6) pop() **peek()**	10 heap: [null, 4, 3, 2] 4
(7) pop() **peek()**	4 heap: [null, 3, 2] 3
(8) pop() **peek()**	3 heap: [null, 2] 2
(9) pop() **peek()**	2 heap: [null] null

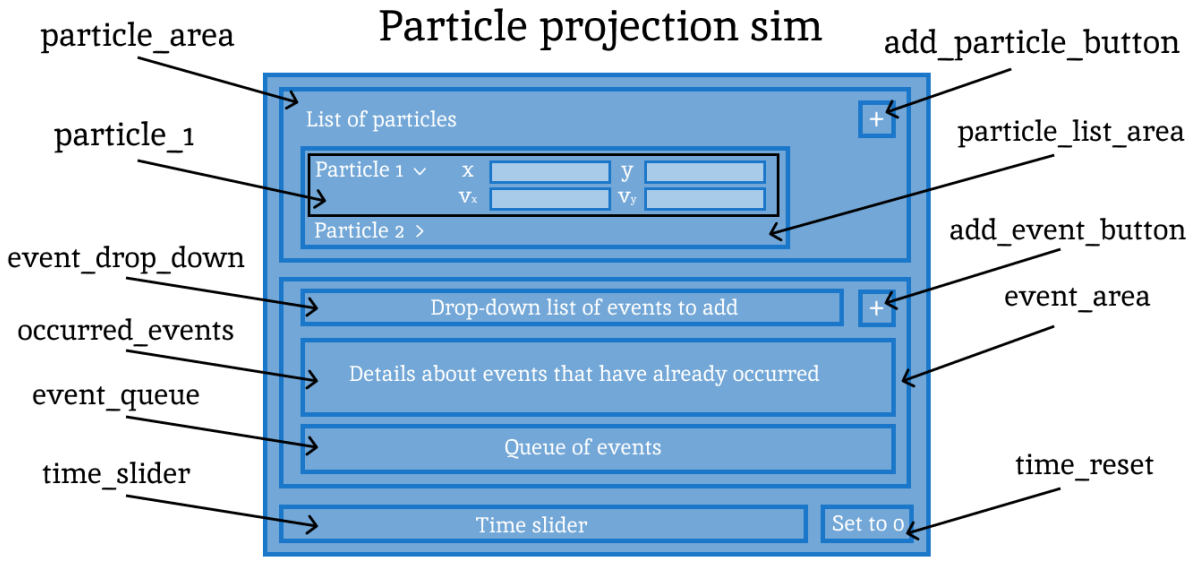
(10) pop() **peek()**	null heap: [null] null

3.8 ID system

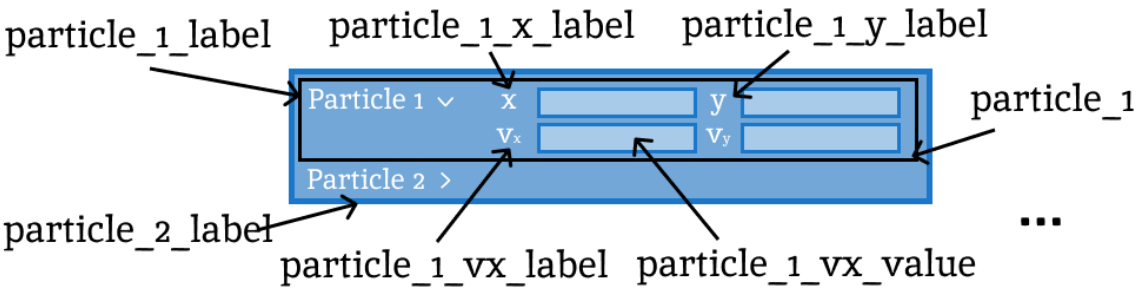
Every simulation will be assigned a unique id that specifies the group of identifiers for the html elements that form this simulation. Each area of the simulation will have its name, for example “io_area”, “canvas”, “sim_area”, etc. As there will be many simulations on the page at any time, the program needs a way to distinguish these elements. Every part of the program that interacts with html elements will know what identifier they have (for example, as one of the attributes of the object). So, every time they interact with html element they will use “name + ‘_’ + id” (where name is the default identifier of an element) as an html id of the element (e.g., “io_area_1”, “canvas_1”). Following diagrams represent a structure of default html identifiers assigned to different parts of the interface. In the actual program, these elements will exist only with an id assigned to them.



Simulation area id design

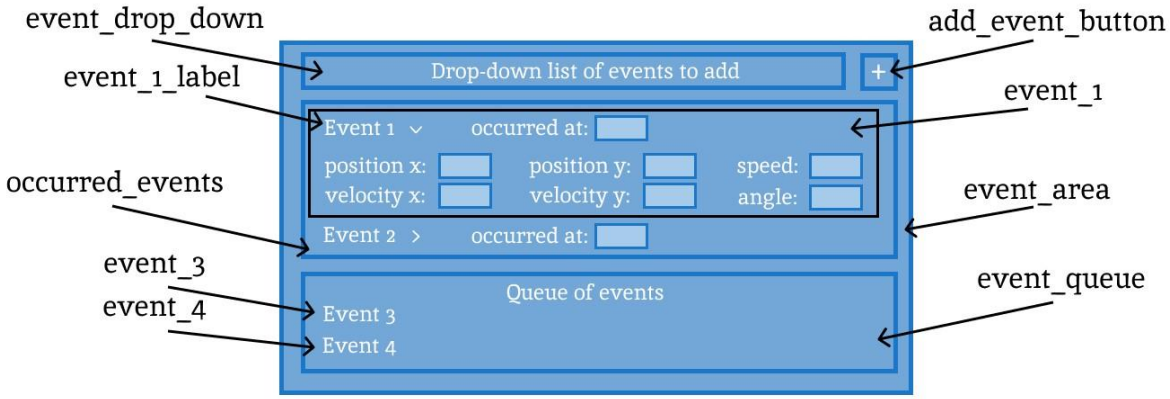


Input output area id design



List of bodies id design

Some inner elements might not even need an id as they will be placed once and deleted together with the element they are contained within. For example, “particle_1_vx_label”, once placed, it will not be accessed by the program and therefore does not need an id. Elements like “particle_1_vx_value” still may require an identifier as their content will be dynamically updated as the particle state changes over time.



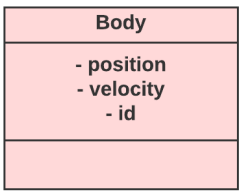
Event area id design

The same principle applies to an event area where “event_1_label” is a redundant identifier, however elements that hold values for “position x”, “position y”, etc. also do not require an identifier as they simply mock these values from the state of the system at the time event occurred.

This is an initial design which might change at later stages of design or development.

3.9 Model

As discussed earlier, the Model comprises of the classes of mainly Simulations and 2d objects. Every object will store its position and velocity as Vectors alongside mass and inverse mass as real numbers. The mass will be used to calculate acceleration from the second Newton’s law, division is a more expensive operation than multiplication. Inverse mass makes use of caching to optimise recalculation of inverse mass.
It provides methods to change position and velocity either by setting it to specified value directly or by adding the change to the current value.

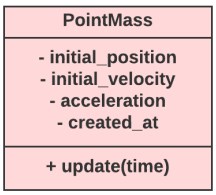


Body superclass

The controller works in abstraction from simulations so that a new type of simulations can be additionally implemented into the system with little to no changes to the Controller. To achieve this, the Controller will only "be aware" of the fact that simulations implement a Simulation abstract class the has an update method that accepts the time by which the simulation is to be updated.

3.9.1 Point mass

Point mass is an implementation of the abstract class "Body". It has additional attributes of an initial position and initial velocity that are used in its additional "update" method. Position and velocities of these objects will be determined by formulas:



$$\vec{s} = \vec{s}_0 + \vec{v}_0 * t + \frac{\vec{a} * t^2}{2}$$

$$\vec{v} = \vec{v}_0 + \vec{a} * t$$

These are the main formulas for a particle projected with certain velocity from some point moving freely under gravity. The point mass objects will be used by the Particle projection simulation. The particle will get its initial velocity and position directly from the validated user input. Acceleration affects the particle from the start of its motion and therefore needs to be one of its initial parameters. Apart from these parameters, the object will need to know the time it was created at to calculate time elapsed from that point. Representing the update algorithm in pseudocode:

```
public procedure update(time: real)
      //   parameter "time" represents the moment in time the simulation
      //   is at
      t = time - this.created_at
      //   calculate time from the creation of the object
      ds = this.initial_velocity.multiplied(t)
      //   initialising ds variable that represents change in position
      //   and calculating the first part of the equation

      ds.add(this.acceleration.multiplied(t * t / 2))
      //   adding the change caused by acceleration

      this.position = this.initial_position.added(ds)
      //   updating position

      dv = a.multiplied(t)
      //   initialising change in velocity variable
      this.velocity = this.initial_velocity.added(dv)
      //   updating velocity
endprocedure
```

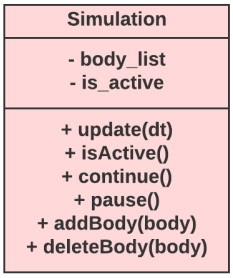
Calculations in pseudocode are broken down in several steps to make the code more understandable, however the same result can be achieved in one line for each position and velocity. Letter "d" at the start of the variable identifier denotes the change. E.g., dv is the change in velocity.

Testing:

Add new PointMass objects to simulations at 0s, 5s. Define their initial state and observe their state in 5, 10 and 60 seconds. Calculate expected state (using formulas given above) and compare results. (x, y) notation will represent x and y components of position, velocity and acceleration given in pixels or pixels per second and pixels per second squared respectively.

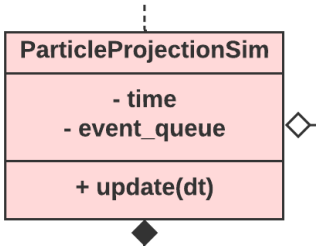
1) First PointMass object start at (100, 100) with velocity of (20, 100) and acceleration (0, -10).
   In 5 seconds: position – (200, 475); velocity – (20, 50)
   In 10 seconds: position – (300, 600); velocity – (20, 0)
   In 60 seconds: position - (1300, -11900); velocity – (20, -500)
2) Initial: position – (0, 400), velocity – (10, 0), acceleration – (0, -9.8).
   In 5 seconds: position – (50, 277.5); velocity – (10, -49)
   In 10 seconds: position – (100, -90); velocity – (10, -98)
   In 60 seconds: position - (600, -17240); velocity – (10, -588)

3.9.2 Simulation abstract class



Defines methods "`update(dt)`", "`isActive()`", "`pause()`", "`continue()`", "`addBody()`", "`deleteBody()`", and attributes "`body_list`" and "`is_active`" as they are shared across all simulations, so they can be used by the controller without any knowledge about how the specific implementation of the method works. Update method advances the simulation in time by the specified dt value. "`isActive()`" returns true if the simulation is active (in other words, does the simulation need to be advanced in time?), pause() and continue() toggle "is_active" Boolean attribute.

3.9.3 Particle Projection Simulation



A child class of the Simulation abstract class. Its main subject is the motion of a projected particle in free fall. It stores all its objects (point masses) in a list and events that are scheduled to happen in a priority queue are stored in the order that events occur.

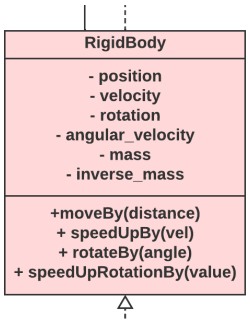
The update loop will take the change in time as a parameter. It will start by changing its current time to the time of the updated state. The time the event from the top of the queue occurs will be compared to the time of the simulation. If the event occurred between the previous and current updates, the time of the simulation will be changed to the time of the event occurring and simulation will be stopped.

Afterwards, every point mass, i.e., particles, will get their positions and velocities updated based on the current time of the simulation. It happens after the event queue is checked as regardless of the outcome, the correct instant in time will be specified and particles will be placed correctly.

```
public procedure update(dt: real)
      //   parameter dt denotes the change in time from the previous
      //   frame

      this.time += dt

      event = this.event_queue.peek()
      //   peek the first event from the queue

      if event.getTime() < this.time then
            //   if the time the event occurs is greater than the
            //   current time the simulation is at
            event.execute()
            //   execute the procedure defined when event was created
            this.time = event.getTime()
            //   set the time of the simulation to the moment when
            //   event has occured
            this.event_queue.pop()
            //   remove event from the queue
      endif
      for i=0 to body_list.length - 1
            body_list[i].update(this.time)
      next i
endprocedure
```

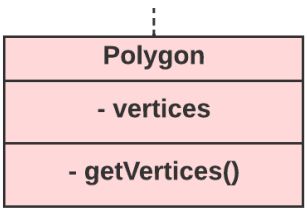
3.9.4 Rigid body

The RigidBody abstract class will be a base of the Polygon and Disc. The position of the shape will represent its centre of mass and the origin of its local space at the same time. The RigidBody class will extend Body by adding velocity, rotation, angular velocity, mass, and inverse_mass attributes and corresponding to them methods like, “moveBy()”, “speedUpBy()”, “rotateBy()” and “speedUpRotationBy()” which will be used to access the object’s attributes and change them. There will be direct getters and setters for each attribute as well.

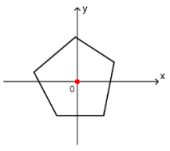


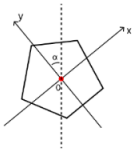
3.9.5 Polygon

Inheriting from the Rigid Body class all attributes, the Polygon class will also store a list of its vertices given as vector coordinates relative to its centre (i.e., in the polygon’s local space). The polygon will need a way to go from its local space to simulation space to correctly draw the polygon on the screen, detect and resolve collisions, etc. It will be accomplished with the “getVertices()” method that will return the list of vector coordinates that represent the polygon vertices in simulation space.



Before designing this method, the rotation must be clearly defined to avoid ambiguity. As the vector “rotatedBy(angle, axis)” method is defined as anticlockwise rotation around the axis for positive values of the angle, the polygon class will follow the same convention. Positive values for its “rotation” attribute will represent an anticlockwise rotation of the shape around its local space origin. In other words, it will represent an angle in anticlockwise direction that the local space y-axis makes with the vertical line in simulation space.





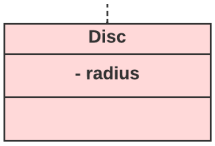
To go from the polygon’s local space to simulation space, its vertex coordinates first

will need to be rotated around local space origin and then translated by the "position" vector to move the shape to its correct position in simulation.

```
public function getVertices()
      converted_vertices = new List()
      for i = 0 to this.vertices.length - 1
            converted_vertex = this.vertices.rotatedBy(this.rotation)
            //   first rotate
            converted_vertex = converted_vertex.add(this.position)
            //   then move to the correct position
            converted_vertices.push(converted_vertex)
      next i
      return converted_vertices
endfunction
```

Vertices of the polygon will be given such that two adjacent elements of the list represent two adjacent vertices.

3.9.6 Disc



Disc's rotation and angular velocity attributes inherited from the RigidBody superclass can be seen as redundant, as the disc cannot rotate if the force acts along the line that goes through the centre of the disc (which will always be the case for all interactions with the objects). Nevertheless, it sets up opportunity to extend the project without changing a lot of the code to allow the rotation of the disc object (for example when friction forces are added).

Position attribute of the disc class will represent position of its centre. It will additionally need radius attribute to define its dimensions.

3.10 Events

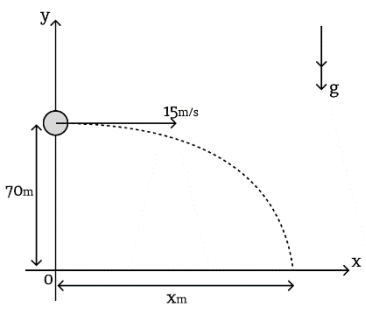
The main way for students to use the particle projection simulation will be through the events. Events occur singly in time once the condition specified by the user is met. The simulation is stopped whenever an event occurs to display to the user the state of the system at that moment in time.

3.10.1 Typical mechanics problems

Designing events, it is worth considering what do typical particle projection problems involve. Some examples are:

1. A particle is projected horizontally at $15\ ms^{-1}$ from a point $70\ m$ above the ground.
   (a) how long does it take for the particle to hit the ground?
   (b) the horizontal distance travelled in that time?

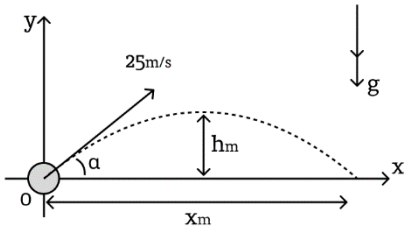
   Initial conditions are given in the question; the trajectory is defined by the formulas in the update method. Part (a) represents an event of "reaching the height of 0 metres". Part (b) will be discovered from the details of the event when it has occurred.



Model 1

Mykyta Osovskyi
Dronfield Henry Fanshawe School

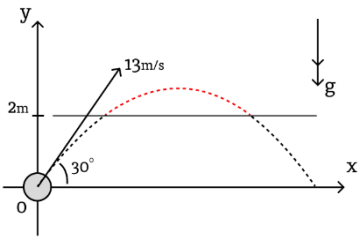
2. A particle is projected from a point on aa horizontal plane with initial velocity of $25\ ms^{-1}$ at an angle α above the horizontal where $\tan\alpha = \frac{3}{4}$.
   (a) what are horizontal and vertical components of the initial velocity vector?
   (b) what is the greatest height reached by the particle?
   (c) what is the range of the particle?



Model 2

   The question suggests an alternative way of inputting velocity of the particle. For part (a), it may be useful to provide an option to switch projections of the velocity vectors on axis on and off or a separate window with a close look at the particle at that moment with projections and other little but helpful details, as suggested in one of the "Less Important" features. Part (b) suggests another type of event: reaching the greatest height. It also can be imagined as the point where vertical component of the velocity is equal to zero. Part (c) is the same for the previous problem: the condition of hitting the ground or reaching 0 metres above the ground.

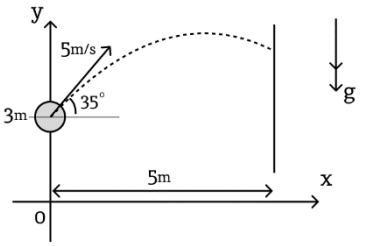
3. A particle is projected from the point O with speed $13\ ms^{-1}$ at an angle of elevation of $30°$. The particle moves freely under gravity.
   (a) what is the velocity and position of the particle after $3\ s$?
   (b) for what range of time is the particle above 2 metres?



Model 3

   Part (a) suggests a primitive but important event: t seconds pass after the start of the simulation. The simulation stops at that moment and the user can look at all the details about the particle: its velocity, position, etc.
   Part (b) can be accomplished either by two events of reaching the height of 2 metres or a separate structure that traces the particle across its way between entering the height of 2 metres and leaving it. The second approach may be easier for students to use and will give an opportunity to provide the user with more extensive output on the screen (it will be easier to draw a differently coloured trajectory with this implementation).

4. A ball is projected from the height of $3\ m$ with the speed $5\ ms^{-1}$ at an angle of $35°$ above the horizontal. After $t\ seconds$ the ball hits the wall at a distance of $5\ m$.
   (a) What is the value of t?
   (b) At what height does the ball hit the wall?
   (c) What is the speed of the ball as it hits the wall?



Model 4

   Part (a) can fit in the frame of an event that puts a condition on x or y value of the position of the particle. In this case: stop when x > 5 metres. Alternatively, a separate event of "reaching the wall" can be introduced. Parts (b) and (c) again emphasise importance of the ability to retrieve information about the system at specific time.

3.10.2 Event superclass

Two main types of events will put a condition on position or velocity of the particle. Every other event (except the event that stops simulation after user-specified amount of time) is secondary to these two and can be expressed as one of them. In this way, an event of reaching the greatest height is the same as horizontal component of velocity being equal to zero. Event of reaching the ground is the same as the event of vertical component of position being equal to zero. The user will be able to specify conditions themselves or choose one of the prebuilt events. Events will share the same constructor inherited from the "Event" superclass:

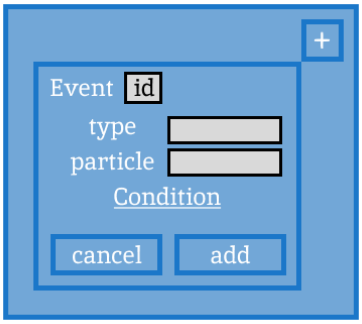
```
public procedure constructor(body: Body, io_handler:
ParticleProjectionIO, event_id: string)
      this.body = body
      //   assigning a body to the event
      this.occurs_at = -1
      //   an attribute representing the time that event occurs at
      this.io_handler = io_handler
      //   reference to the input output handler of this simulation
      this.event_id = event_id
      //   an id of this particular event (will be assigned by the
      //   IO Handler)
endprocedure
```

An attribute "occurs_at" represents the simulation time that the event is meant to occur. It is set to "-1" initially. The value of "-1" represents the fact that event is invalid or not yet specified (it will remain -1 if the condition is invalid). "`occursAt()`" getter corresponds to this attribute.

```
public function isValid()
      return this.occurs_at > 0
      //   return true if the time the event occurs is greater than
      //   0 (returns false if occurs_at is -1 or 0, meaning that event
      //   is not valid or not yet specified)
endfunction
```

Subclasses "`PositionEvent`", "`VelocityEvent`" and "`TimeEvent`" will provide methods that specify when exactly will event occur.

Once the user clicks on the add event button, a pop-up menu will appear on the I/O area. The type of input will be received in the form of a choice on a drop-down menu as well as the particle that is subject to the event. The condition will be defined by the type of event. Input will be received by input output handler whereas event will calculate the time it is meant to occur.

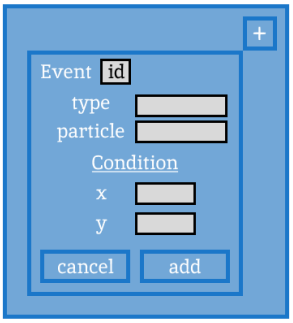


Initial "add event" pop-up menu design.

Once event has reached its time, it needs to be executed. Execution of the event is followed by putting an output block, so it must be completed by the I/O handler of this simulation by invoking its "`executeEvent()`" method which is designed in its section. As simulation can communicate with the I/O handler only through events and not directly, the event's execute method will invoke the associated to it I/O handler's "`executeEvent()`" method.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

3.10.3 Position Event



This implementation of Event represents condition on position of the particle. The event will stop simulation when position of the particle is equal to the user-specified value by x or y.

This implementation provides two separate methods for x and y. They will derive time before the event occurs by the main formulas for particle motion: (1) $\vec{s} = \vec{s}_0 + \vec{v}_0 * t + \frac{\vec{a} * t^2}{2}$ and (2) $\vec{v} = \vec{v}_0 + \vec{a} * t$
In order to get the time when the particle hits a vertical wall, the user will put a condition on x and for a specific height, a condition on y. Resolving vectors on specific set of axes involves computing a dot product with a unit vector in the positive direction of that axis. Therefore, computation of the time the particle reaches some value by x or y, i.e., (hitting the wall or reaching some height) is handled in the same way for both x and y axes. This approach also provides the ability to check for the particle hitting any type of wall. Using the equation (1) resolved on some axis to obtain a quadratic equation of the form $ax^2 + bx + c = 0$

$\frac{a * t^2}{2} + v_0 * t + s_0 - s = 0$ where s represents the x or y coordinate of the wall from the user input condition. The coefficients for the quadratic equation are as follows:

$$a` = \frac{a}{2} \qquad b = v_0 \qquad c = s_0 - s$$

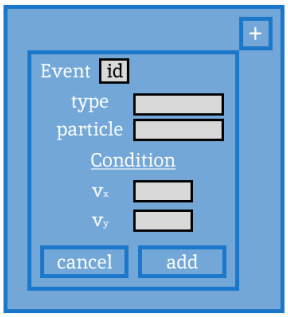
(the first $a`$ represents the a coefficient of the quadratic equation, the second a stands for acceleration)
The discriminant to this equation is given as $D = b^2 - 4ac$. At this point, if D is negative, then the particle never reaches this value by x or y and the $occurs\_at$ attribute can be set to -1.

The solutions to the equation are given as $x_1 = \frac{-b-\sqrt{D}}{2a}$ and $x_2 = \frac{-b+\sqrt{D}}{2a}$ where the first solution represents the time since the particle was created till it reaches given value.
This algorithm is different when the coefficient a is equal to 0. In this case, the equation looks like $bx + c = 0$ with the only solution of $x = \frac{-c}{b}$. And in the case when b also equals to zero, the particle is either already in that position not moving and not accelerating or will never reach it.

```
public procedure calculateTime(value: real, axis: Vector)
      a = this.body.getAcceleration().dot(axis) / 2
      b = this.body.getInitialVelocity().dot(axis)
      c = this.body.getInitialPosition().dot(axis) - value
      //   calculating coefficients for the quadratic equation

      if a == 0 and b == 0 then
            this.occurs_at = -1
            return
            //   invalid if both a and b are zero
      endif

      if a == 0 then
            //   if only a is zero, then there might be a solution
            x = (-1) * c / b
            if x <= 0 then
                  this.occurs_at = -1
            else
                  this.occurs_at = x
            endif
            return
      endif

      D = b * b - 4 * a * c
      //   calculating discriminant

      if D < 0 then
            this.occurs_at = -1
            return
            //   end function
      endif

      x1 = ((-1) * b - sqrt(D)) / (2 * a)
      x2 = ((-1) * b + sqrt(D)) / (2 * a)

      if x2 <= 0 then
            this.occurs_at = -1
            return
            //   if the second solution is negative, then the particle
            //   never reaches the value (the first one is always
            //   negative in this case)
      endif

      if x1 <= 0 then
            this.occurs_at = x2
      else
            this.occurs_at = x1
            //   if the first value is positive, then the particle first
            //   reaches the value after x1 seconds and the second time
            //   after x2 seconds
      endif
endprocedure
```

The input output handler in its turn invokes either "`setXtime()`" or "`setYtime()`" which invoke the event's "`calculateTime()`" method with the axis as the "`new Vector(1, 0)`" or "`new Vector(0, 1)`" respectively.

3.10.4 Velocity Event



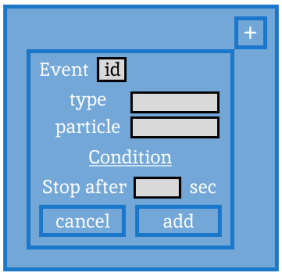
Similarly, the Velocity event gets either condition on the vertical (y) or horizontal (x) component of the velocity and calculates the time till the respective component of the velocity is equal to this value. The equation used this time is (in vector form):

$$\vec{v} = \vec{v}_0 + \vec{a} * t$$

Calculating projections on specific axis, the equation gets the form of $bx + c = 0$ where $b = a$ and $c = v_0 - v$ where v is given as user input "`value`". The only solution is $x = \frac{-c}{b}$ given non-zero b. If b is zero, the event is invalid. And similarly to the position event, the I/O handler will invoke either "`setXtime()`" or "`setYtime()`" which in its turn will define the axis on which the vectors are projected for the "`calculateTime()`" method.

```
public procedure calculateTime(value: real, axis: Vector)
      b = this.body.getAcceleration().dot(axis)
      c = this.body.getInitialVelocity().dot(axis) - value
      //   calculating coefficients for the equation

      if b ==0 then
            this.occurs_at = -1
            return
            //   invalid if b is zero
      endif

      x = (-1) * c / b
      if x <= 0 then
            this.occurs_at = -1
      else
            this.occurs_at = x
      endif
endprocedure
```

3.10.5Time Event



This is the simplest type of event that is defined by the amount of time after the start of the simulation till the moment simulation stops and outputs information about the chosen particle. All necessary validations of the user input are carried out by the I/O handler code.

```
public procedure setTime(value: real)
      this.occurs_at = value
endprocedure
```

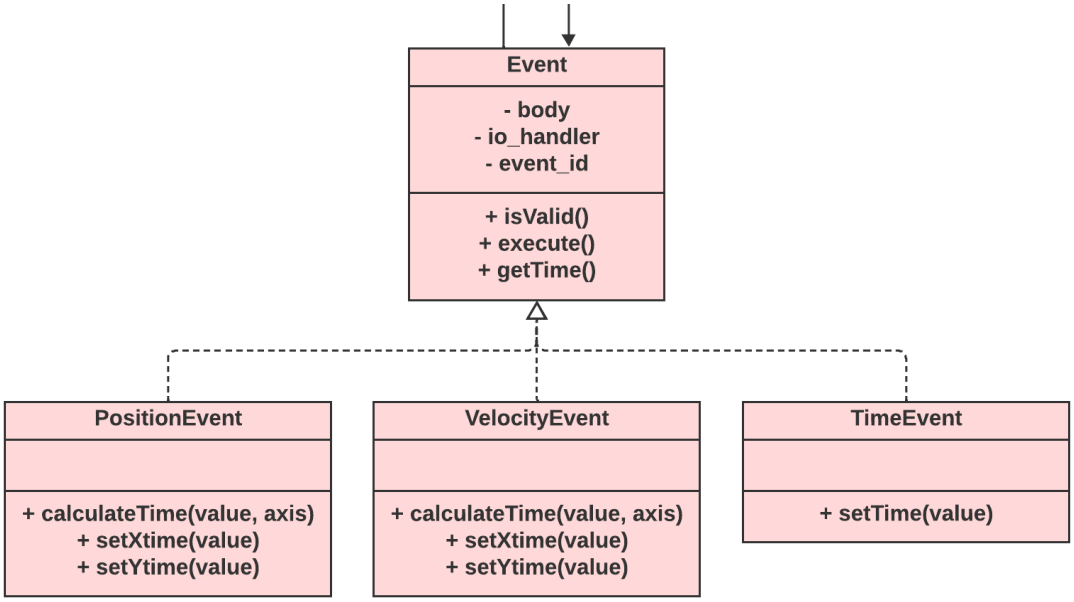
3.10.6Comparing Events

For events to be stored in a priority queue, there must be a way to compare them. The comparator method "`compare(event1, event2)`" can be defined in such a way that the priority queue will prioritize events that are due to occur earlier. That is:

If event1.occursAt < event2.occurs_at, then compare(event1, event2) > 0
if event1.occursAt == event2.occurs_at, then compare(event1, event2) == 0
if event1.occursAt > event2.occurs_at, then compare(event1, event2) < 0

```
public static function compare(event1, event2)
      return event2.occurs_at - event1.occurs_at
      //   will return a positive number whenever the second event
      //   occurs later, in this way priorotising the earlier events
endfunction
```

This function is passed to the priority queue constructor when it is instantiated. The function is made static so it can be accessed directly from the Event class.

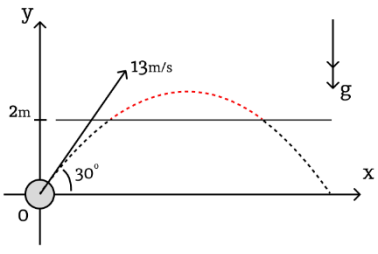


3.10.7Testing

Start by testing how does Particle Projection Simulation handles events using the TimeEvent class as it does not require any tests on its own.
First create events that are due to occur in 5, 10, 12 and 20 seconds. Output an absolute time when they are all created. Run simulation till the first event occurs, recording an absolute time. The time difference must match.

Now that the system of events works as intended once the time event occurs is known, to test PoisitonEvent and VelocityEvent, only methods that calculate time when the event is due to occur must be tested.
Considering two set-ups from typical mechanics problems, different events will be applied to each of them with different conditions. Precalculated results (the time event is due to occur) will be matched to received output.
Assume all set-ups use acceleration of (0, -9.8) m/s^2.

1) Starting with the wall at 5 m above the ground. It corresponds to the PositionEvent and “setYtime()” method call with the parameter of 2. The calculated time must be equal to 0.485059s for the first time and 0.841472s (to 6sf). If the event is created at the start of simulation together with the body, it must output the first value. When it is created for the second time after

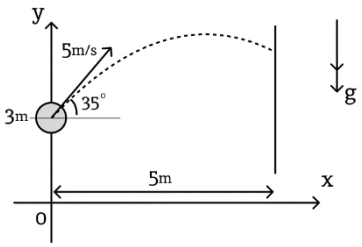


particle crossed 2m only once, the time output must give second value.

2) Continuing with the same set-up. Apply “setXtime()” method for the PositionEvent with a parameter of 7 must give the time of 0.621762s. For value of 12, output must be 1.06588s.
3) Testing for boundary inputs.
   setXtime(-2) -> -1 / invalid
   setXtime(0) -> -1 / 0 / invalid

   setYtime(5) -> -1 / invalid
   setYtime(0) -> 0 / invalid
4) VelocityEvent test:
   setXtime(13*cos(30^0)) -> 0 / -1 / invalid
   setXtime(2) -> 0 / -1 / invalid
   setXtime(-200) -> 0 / -1 / invalid
   (velocity does not change by x axis, so any value must invalidate the event)

   setYtime(4) -> 0.255102s
   setYtime(-10) -> 1.68367s
   setYtime(6.5) -> 0 / -1 / invalid
   setYtime(10) -> -1 / invalid
5) Proceeding to second setup:
   setXtime(5) -> 1.22077s
   setXtime(-1) -> -1 / invalid
   setYtime(3.1) -> first: 0.0372382s, second: 0.548044s
   setYtime(3) -> 0.585282s (boundary data as the particle is already at 3 by y)
   setYtime(-10) -> 1.94754s
   setYtime(5) -> -1 / invalid

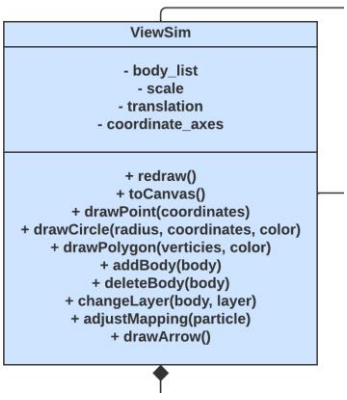


   VelocityEvent:
   setYtime(2.5) -> 0.037539s
   setYtime(-12) -> 1.51713s
   setYtime(0) -> 0.292641s

The events must also be tested for the case when the new event is added not at the start of the simulation.

3.11 View

View reflection of the simulation separates the process of redrawing simulation on the screen from input and output control and model processing. It will deal with all unrelated to the model details, such as the order in which objects are represented on the screen (for example, in case objects can overlap, they may be assigned a layer). It will contain all view representations of objects with colours and layers associated to them. Provides functionality to shift an object up or down the layers if necessary. It is responsible for all the drawing routines. The same implementation of the View will be shared for all simulation types.
It starts with initialising set of attributes and creating the canvas that will be used for visual output on the screen.



Update method pseudocode:

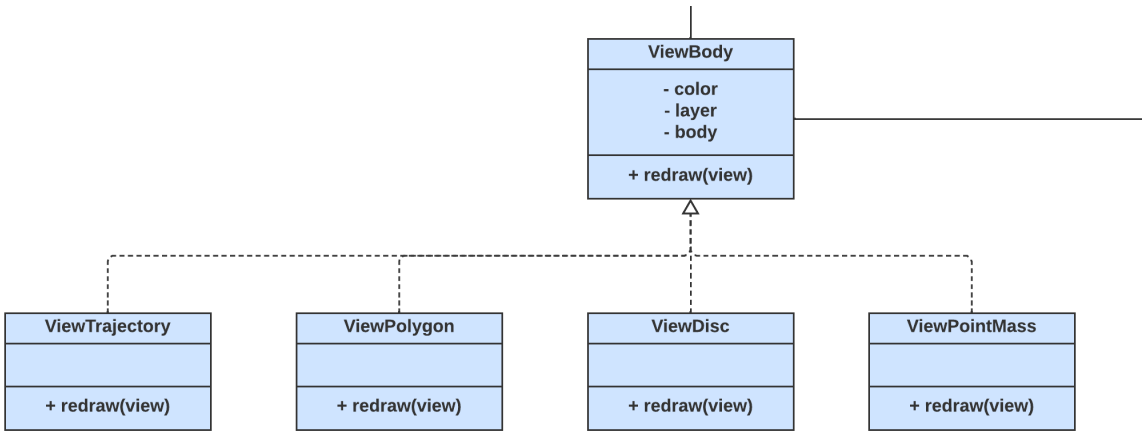
```
public procedure redraw()
      erase everything from the screen
      for i = 0 to body_list.length - 1
            body_list[i].redraw(this)
            //   passing the View object itself to the update method of
            //   each body
      next i
endprocedure
```

The View will provide routines that work directly with html canvas. These methods will draw primitives on the screen. Every View-Object will refer to them whenever its “redraw()” method is invoked.

3.11.1View bodies

View body abstract class will define attributes and the constructor as they are shared across all bodies. Drawing routines that the object uses will be specified by the implementation of the abstract method “redraw(view)” that receives simulation View as a parameter.

The need for this class arises from the fact that objects will have different drawing routines depending on their type. It also provides a convenient way of storing colours and other object specific (however unrelated to the simulation) properties. Position is accessed from the body whenever the object needs to be redrawn.

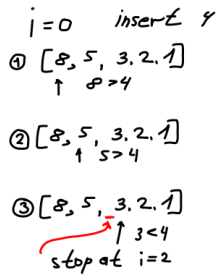


3.11.2Adding a body

The attribute “body_list” implies that it stores a list data structure. However, it is not the only and not the best way to approach this. The body list stores all the view representations of simulation objects in the order of layers, in this way the update function redraws object in the correct order (first object on the back layers). A primitive algorithm to implement this is to traverse the list every time a new object is added until the correct place to insert the object is determined. This option is simple to implement and unless the simulation has a lot of objects, it is satisfactory by the performance. An optimisation can be made by using a priority queue to efficiently insert objects (priority queue insert operation is $O(\log(n))$ order of time where n is the size of the priority queue). This optimisation is insignificant in the context of particle projection simulation and general-purpose simulation; however, it is worth considering if the program extends in future.

```
public procedure addBody(body: Body)
      i = 0       //   loop control variable
      while body_list[i].getLayer() > body.getLayer()
            and i < body_list.length
            i += 1
            //   increment i while the layer of the element at i is
            //   bigger than the layer of the new body
      endwhile
      body_list.insert(i, body)
      //   insert body on the position i
endprocedure
```

The algorithm for adding new body is to loop through the list of bodies, comparing the layer of the new body to the body in the list at position "i". If the body at position "i" is at a higher layer, keep moving, otherwise, the position to insert the body is exactly. The comparison "`i < body_list.length`" is in place to prevent "i" from exceeding body list length. An example of how algorithm works is shown on the diagram on the right. Number 4 is to be inserted. On first iteration, "i" points to 8, which is greater than 4, then 5 which is also greater than 4. On the third iteration, "i" points to 3, which is smaller than 4. Therefore, the correct position for number 4 is at "i = 2".



3.11.3 Deleting a body

Simple linear search through the body list.

```
public procedure deleteBody(body: Body)
      for i = 0 to body_list.length - 1
            if body_list[i] == body
                  body_list.delete(i)
                  //   delete element by the index i
            endif
      next i
endprocedure
```

Not the most efficient implementation as linear search takes a lot of operations as well as deletion of an element from the list. Possible optimisation is to use a Hash Map to store a body associated to its unique index. It will require a different implementation of an "addBody(body)" function. The same optimisation can be applied to Simulation body_list as well. Object comparison in JavaScript is carried out by comparing the memory locations associated with two variables. It happens automatically with "==" operator.

3.11.4 Moving up / down the layers

Reusing code from previous functions, the function to move a body up or down the layers can be achieved by changing a layer of the body with "`body.setLayer()`", removing the object from the body list with "`this.deleteBody(body)`" and adding back to the right place "`this.addBody(body)`". At the moment, functionality of layers will not find application and therefore can be enhanced in future with various functions to move objects to the top or to the bottom of layers.

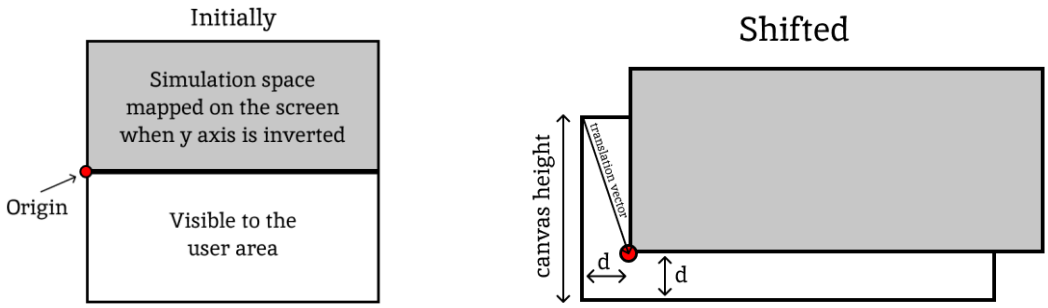
3.11.5 Coordinate conversion

As the simulation coordinate system follows a regular axis orientation which is different from what is found in the html canvas, the program will need a way to convert coordinates given in the simulation

space to the coordinates that correspond to the point on the screen, i.e., canvas. And the other way, whenever a user input is received as a click on the canvas, it must be converted to the simulation system to correctly process it.

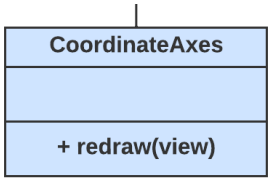
One of the ways to accomplish that is by keeping the translation and scale of the simulation in the corresponding view object. So, every time the program draws something on the screen, the view will directly apply translation and scaling to the coordinates.
In cases when the user wants to move the camera around the simulation space, zoom in or zoom out, the change in scale and translation can easily be applied to the view scale and translation attributes. This functionality is implemented in input output handlers.

The change in axes orientation is accomplished by taking a “y” coordinate as negative (in other words, reflecting the position vector in x axis) and then shifting and applying scaling. Initial translation will move the simulation to the point just above and to the right of the bottom left corner of the screen. In this way, the user will be able to see the coordinate axes more clearly.



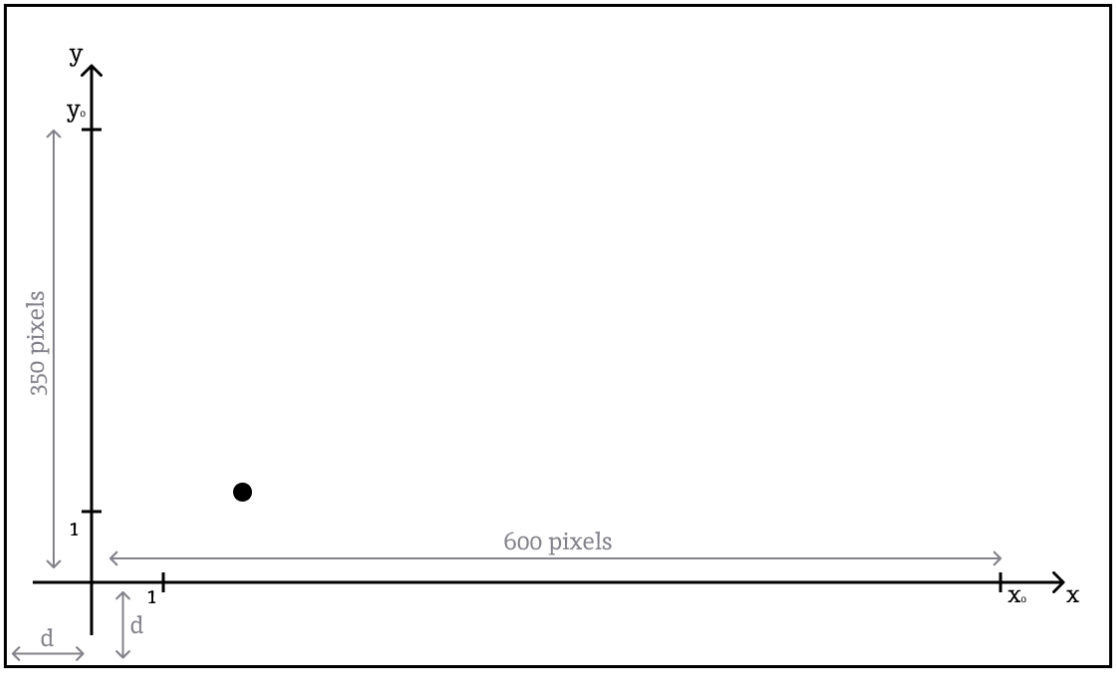
3.11.6Coordinate axes

Coordinate axes’ main purpose is to give the user some sense of the simulation scale. If a particle appears on the blank screen, they will have no idea whether it is 2 or 200 metres high. A couple of marks on coordinate axes will give the user a rough idea of where the particle is by just looking at its visual representation. They will then be able to get an exact value from the IO area.



CoordinateAxes is aggregated by the ViewSim

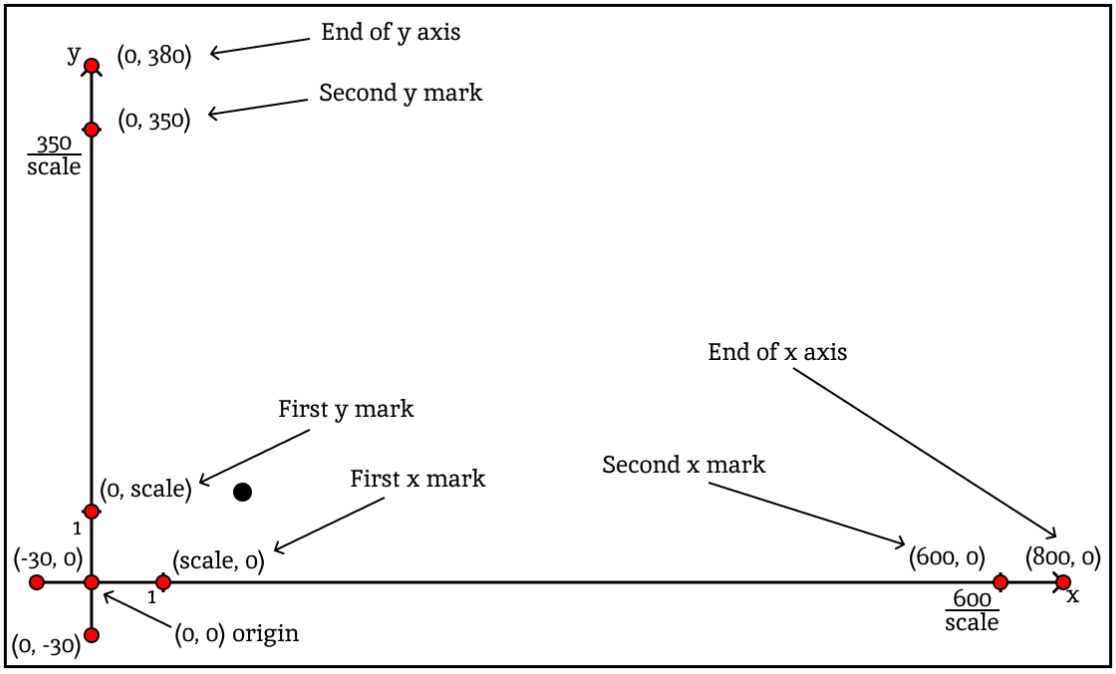
I will make the following layout for the coordinate axes object:



The main attributes of the coordinate axes will be marks on the x and y axes with little captions around them representing how many units of length in simulation space they are apart from the origin. The origin of the axes object will match the simulation's origin. The object will adjust to the current simulation scale and work out position for each mark on the screen and what its caption should look like.

Define the marks in a following way:
The first two marks will be located the same number of pixels to the right/top as the value of the scale as it represents 1 unit of length.
Two marks on the ends of two axes are fixed on coordinate axes to 600 pixels to the right and 350 pixels to the top (the exact measurements are approximate and might change when prototypes are developed). On the other hand, their captions will depend on the scale of the simulation. To work it out, you must simply divide the number of pixels by the scale. The larger the scale, the smaller value these captions will obtain.

To draw this structure on the screen, the main points must be defined:



The position of the origin (and consequently all other points) will be shifted by the global translation vector.

To save some time for recalculation, ViewSim can recalculate the coordinates only when the scale is changed (essentially making use of caching). The method for drawing coordinate axes will be contained within ViewSim as “drawCoordinateAxes()”. The method will draw all lines (arrows), put marks as defined above and put captions where necessary.

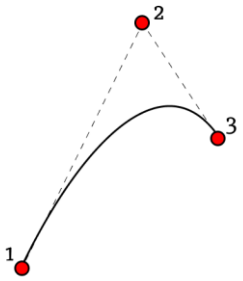
3.11.7Trajectory

The trajectory of a particle is a useful graphical representation of a path that the particle is going to undertake and can give a useful insight for the user. The Trajectory object will be another implementation of a ViewBody.

The trajectory is a curve. Two main ways to draw curves when their equations are known (the equation of the particle trajectory can be calculated from initial state):
1) moving along the curve by substituting values of x in small increments and drawing a line between them (or simply putting a dot). Even though all curves and lines are actually drawn in this way, implementing this in the project with no access to lower-level feature is highly inefficient;
2) Bézier curves: Bézier curves are already efficiently implemented into most graphical engines (e.g., “bezierCurveTo()” function of HTML canvas context). They allow to draw parametric curves based on the number of control points given.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

Consider a quadratic Bézier curve:
It is built by 3 control points. The first and the third one specify the start and end points of the curve. Lines going from first to second and from third to second points are tangents to the curve. Given that the curve has quadratic nature, two points and two tangents uniquely identify the parabola on a plane. Therefore, to build a quadratic curve, it is enough to calculate positions of these three points.



An equation of the trajectory of a particle is canonically given as:

$$y = x\tan\theta + \frac{ax^2}{2v^2cos^2\theta}$$

Given that the particle starts from (0, 0) with velocity of magnitude v with an angle θ above horizontal. Acceleration is assumed to have only vertical component represented by a. To adjust the formula to a more suitable for simulation form, we need to account for the fact that the particle starts at arbitrary point (x_0, y_0) and the velocity is given as a vector (v_x, v_y) which is equivalent to $(v\cos\theta\,, v\sin\theta)$. Translating equation by (x_0, y_0) requires substituting $(y - y_0)$ instead of y and $(x - x_o)$ instead of x. Rewriting equation in a following way:

$$y - y_0 = \frac{(x - x_o)v\sin\theta}{v\cos\theta} + \frac{a(x - x_o)^2}{2(v\cos\theta)^2}$$

Now every $v\sin\theta$ can be substituted by v_y and every $v\cos\theta$ by v_x:

$$y - y_0 = \frac{(x - x_o)v_y}{v_x} + \frac{a(x - x_o)^2}{2{v_x}^2}$$

Call $f(x)$ the function that gives y coordinate of the point at some value of x:

$$f(x) = y_0 + \frac{(x - x_o)v_y}{v_x} + \frac{a(x - x_o)^2}{2{v_x}^2}$$

The derivative of this function $f'(x) = \frac{v_y}{v_x} + \frac{a(x-x_o)}{{v_x}^2}$. It gives the value of slope of the tangent to the curve at any given x. It will be used later.

The first point to define a Bézier curve is given by the starting position of the particle. The third one is where the curve stops, i.e., drawing a trajectory is no longer required. It can be an arbitrary point on the curve in the direction the particle moves.
(x_0, y_0) – point 1 where $y_0 = f(x_0)$; (x_2, y_2) – point 3 where $y_2 = f(x_2)$.

To calculate position of the second point, also referred to as control point, we need to determine the point where tangents to point 1 and point 3 crosses. Call $m_0 = f'(x_0)$ slope of the first tangent and $m_2 = f'(x_2)$ slope of the second tangent. Their equations are therefore given as:

$y = y_0 + m_0(x - x_0)$ and $y = y_2 + m_2(x - x_2)$. Equating y values and rearranging the equation in terms of x gives following formula:

$$x = \frac{y_2 - y_0 + m_0x_0 - m_2x_2}{m_0 - m_2}$$

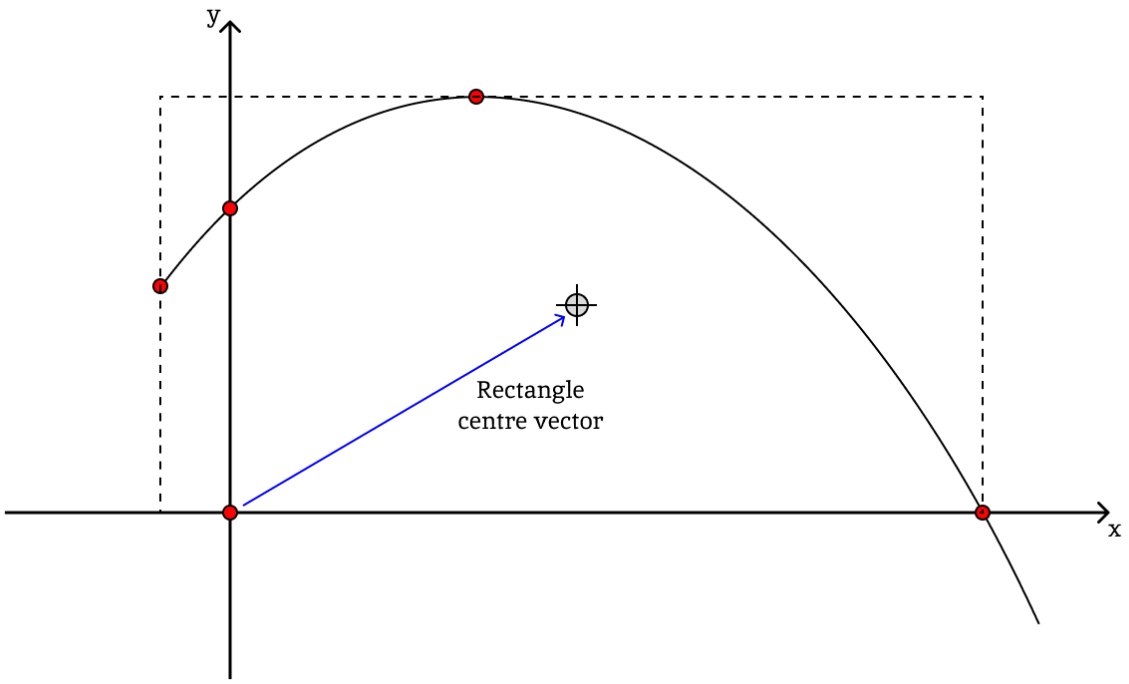
This is an equation for the x coordinate of the control point (x_1). Its y coordinate is given as $y_1 = y_0 + m_0(x_1 - x_0)$ (substituting x_1 into one of the tangent equations).

These calculations will occur when the Trajectory object is constructed. They compute values for three attributes: “starting_point”, “control_point” and “end_point” which are then passed to the ViewSim object so it can convert them into Canvas space and draw a Bézier curve representing trajectory of a particle.

3.11.8Scale adjustment

Particles that will be added to the simulation will vary in how far from the origin they will start and how far they travel. Therefore, different particles will require different scales for the best visual output.

It is impossible to estimate what will be the best scale and translation for every particle given its starting condition, so I will use a heuristic that will give an estimate of what part of particle’s trajectory is the most vital to see on the screen and how to fit it on the canvas.
To accomplish that, I will first calculate the most important points on the particle’s trajectory that need to be on the screen:



The starting point, the points of intersection with coordinate axes, the highest point on the trajectory and the origin are some of the examples of the important points. Calculating them is a similar process to how the time is worked out for events. Set up an equation projecting vectors on the required axis, solve it with respect to time and substitute back to the equation of motion to get the coordinates of the required point.

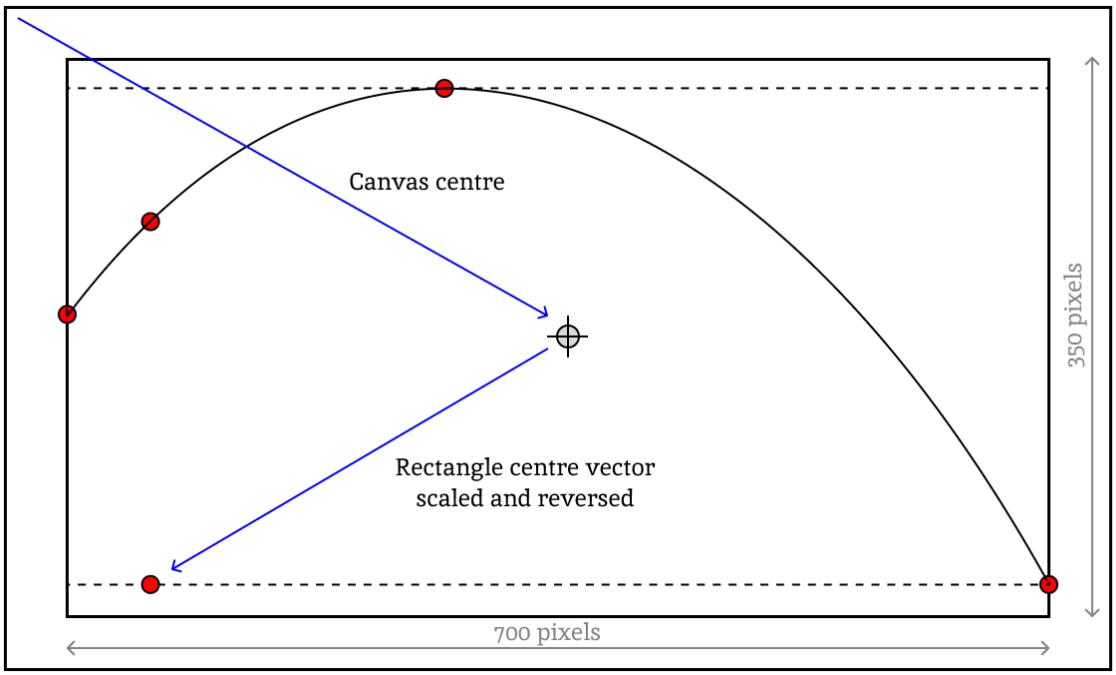
Once the points are found, they form a rectangle. It can be found by iterating over the list of points and keeping track of the maximum and minimum values by x and y axes they lie. Once the process is finished, you end up with 4 values that represent the rectangle. Its width can be found as “maximum_x – minimum_x” and height as “maximum_y - minimum_y”. The coordinates of the centre of rectangle are given as “((maximum_x + minimum_x) / 2, (maximum_y + minimum_y) / 2)” because it lies halfway between the lines that form the rectangle.

This rectangle must be located within the central area of the canvas after translation vector and scale are adjusted. To assure that, I will define the other rectangle of width 700 and height 350 located at the centre of the canvas (with each edge 50 pixels off from the edge of the canvas). The rectangle of important points must fit inside of this rectangle. To assure that, we must find such scale value that:

$$width * scale \leq 700$$

$$height * scale \leq 350$$

setting up the scale to be $\min\left(\frac{700}{width}, \frac{350}{height}\right)$ will satisfy both conditions in such a way that the rectangle is placed at the centre of the canvas and occupies as much area as possible.



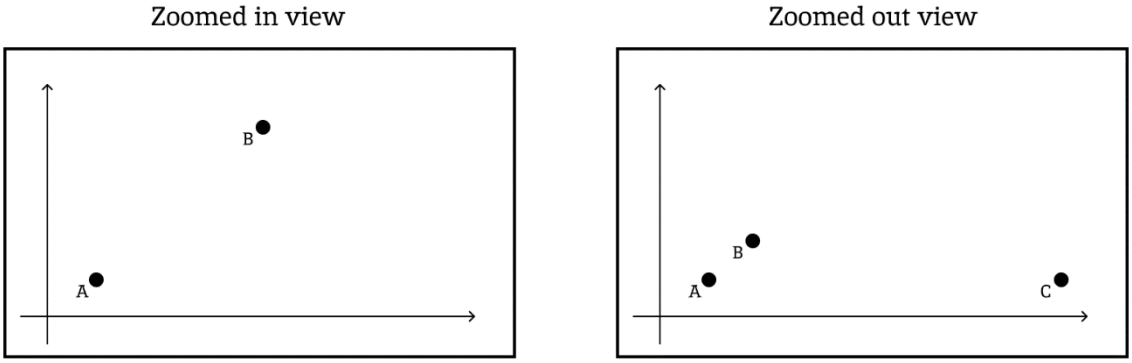
Now that the scale is found, a translation vector needs to be set up such that centres of two rectangles coincide. From the diagram, we can see that the translation vector (vector that points from the left top corner to the origin of the simulation) is given as a sum of two vectors. The first one is the vector to canvas centre (400, -225). The second point from the rectangle centre to the origin. The coordinates of the rectangle centre relative to the origin have been found previously, going back from the centre requires the reversed vector of the rectangle centre multiplied by the scale. The sum of these two vectors will give the translation vector required to achieve two centres coincide.

The scale and translation together define how simulation space is mapped on the canvas. The method that will perform these calculations will be called “adjustMapping()”. It will receive a new body as an argument and will set scale and translation to required values.

3.11.9 Drawing a point

The subroutines that draw primitives will work directly with the Canvas API. The algorithms for them will be presented in a conceptual way (not referencing exact functions from the Canvas API), the code will be created in the development section.

The first primitive to be explored is a point. The idea of a point is that its size does not depend on the scale of the simulation, no matter how the user will zoom in or out, the point will remain the same size. Creating a point can be helpful to get an idea of a scale of the simulation, for example, in instances when points are in remote locations.



In this way, the user will not lose the sight of some location if they zoom out too much, because they will be able to fix a point in that specific location. This method will be mainly used with for the point mass as this type of objects does not have specific dimensions and does not need to be scaled.

```
public procedure drawPoint(center: Vector)
      translated_center = center.reflectedInX().added(this.translation);
      //   reflect in x axis and shift
      draw a circle of radius 10 pixels at coordinates
      "translated_center" fill the circle black
endprocedure
```

Testing strategy:

Once the ViewSim is implemented, I will write a test method that will be invoked at the end of the ViewSim constructor. It will launch all temporary code that is used for visual tests.

Starting with PointMass tests. Once all required code is written, start test method by instantiating the body with coordinates (10, 10) pixels and creating ViewPointMass object with reference to the body. Add the ViewPointMass object to the ViewSim. Run the code: create new simulation on the page. This must result in a Circle of radius 10 be drawn on the left bottom corner of the canvas. The circle must touch the borders.
Try different coordinates and radii for the circle:
position = (200, 30); r = 30 -> a circle touching the bottom of the canvas.
position = (0, 400); r = 50 -> a circle touching the top of the canvas.
position = (400, 50) r = 400 -> a circle touching left, right, and top borders going beyond the canvas on the bottom.

3.11.10 Drawing a polygon

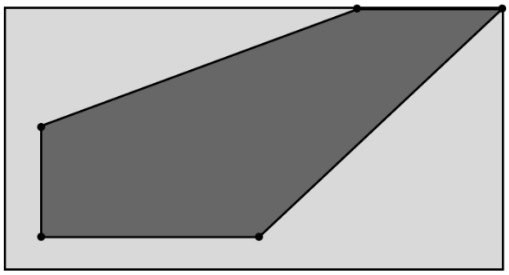
Convex polygons are the main shapes of the general-purpose simulation. The polygon drawing method will take the list of verticies and colour as parameters and will draw a polygon by given coordinates (coordinates are given in simulations space and then translated into coordinate system of the canvas by the ViewSim).

The algorithm is simple:
- start the path;
- iterate through the list of verticies;

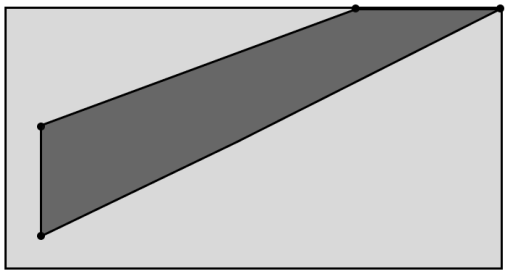
- translate coordinate of each point to canvas space;
- draw a line from the previous point to the next one on the list of vertices;
- once the list is finished, close path and fill it with required colour.

Testing strategy:
Use following lists of vertices to draw polygons and match them with expected results:

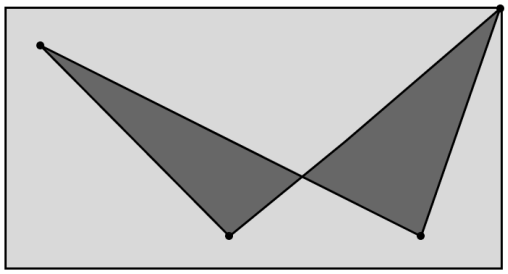
1) [(50, 50), (350, 50), (800, 450), (500, 450), (50, 220)]



2) [(50, 50), (800, 450), (500, 450), (50, 220)]

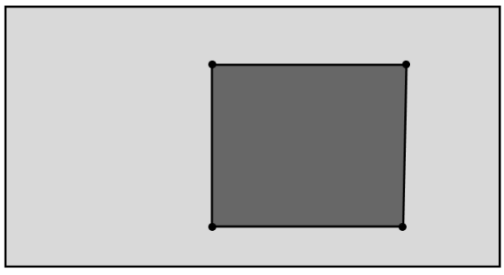


3) [(350, 50), (50, 400), (700, 50), (800, 450)]



unrealistic scenario just to check how the method handles it.

4) [(350, 50), (350, 350), (650, 350), (650, 50)]



3.11.11 Drawing a circle

This method is the same as drawing a point. The only difference is that the ViewDisc class will call it with specific radius (which will also need to be scaled). Describing the method in a structured way:

1) Calculate translated and scaled coordinates of a centre;
2) Calculate scaled radius;
3) Set the colour;
4) Move to the point with required coordinates;
5) Draw a circle;
6) Fill the circle with the colour;
7) Finish drawing.

3.12 Controller

The controller exists in a single instant per opened html page. It stores all the currently running simulations, their views and input output handlers. The controller responsible for invoking appropriate methods to update simulations and redraw them on the canvas. The user has a choice of different simulations to create on the screen. The controller will place all appropriate HTML elements to retrieve user input.

Constructor method: initialises lists for storing simulations. Places a drop-down selection, linking a function that creates simulation to the button that places a simulation. Starts the update loop. Each new simulation will be assigned a new unique index to distinguish simulations between each other.

3.12.1 Constructor

```
public procedure constructor()
      this.sim_list = new List()
      this.view_list = new List()
      this.io_list = new List()
      //   lists of simulations, views and input/output handlers, the
      //   simulation, its view and io handler are stored by the same
      //   index in the list

      this.sim_names = ["Particle projection simulation", "General-
purpose simulation"]
      this.sim_classes = ["ParticleProjection", "GeneralPurpose"]
      //   lists of simulation names and their class identifiers
      //   can be easily changed if a new simulation is added

      this.select = document.createElement("select")
      //   a javascript method that creates new HTML element:
      //   in this case, select, i.e., a dropdown menu of simulation
      //   names for user to choose one
      //   reference to it is also stored in the controller

      option = document.createElement("option")
      option.value = -1
      option.innerHTML = "Choose a simulation"
      this.select.appendChild(option)
      //   add a default option to the drop-down menu that prompts the
      //   user to choose a simulation

      for i = 0 to this.sim_names.length - 1
            option = document.createElement("option")
            //   creating an option
            option.value = i
            //   value to associate an option with a simulation
            option.innerHTML = this.sim_names[i]
            //   fills option with the name of the simulation
            this.select.appendChild(option)
            //   add the option to the dropdown menu
      next i

      add_sim_button = document.createElement("button")
      //   instantiating a button that creates a new simulation
      add_sim_button.onclick = this.add_sim
      //   assigning the function of adding a simulation to the button
      //   click on the button will invoke an add_sim function
      //   of the controller with associated index that is selected

      this.next_id = 0
      //   id of the next simulation

      add dropdown menu and the button to the window
      //   will be coded in development: puts the dropdown on the page

      this.update(0)
      //   start the update loop of the controller with the initial
      //   timestamp of 0
endprocedure
```

Testing strategy:

Check if the drop-down menu and button have been added. Write temporary placeholders for

update and add_sim methods. Test if they get invoked and have access to the lexical environment[8] of the Controller.

3.12.2 Adding a simulation

Invoked when the user clicks on the add button. Appends simulation to the bottom of all the simulations. Initialises inputs and outputs for the simulation. Assigns a unique id for the simulation. It will be stored by an IO handler and View as they create HTML elements that are required to input and output any information to and from the user. This unique identifier will be contained within all HTML tag IDs.

```
public procedure addSim()
      index = this.select.value
      //   temporary variable for the option on the dropdown menu
      if index == -1 then
            return
      endif
      //   if no option is chosen (value of -1), no simulation is added

      name = this.sim_classes[index]
      //   name of the group of classes

      create a simulation area
      //   create an HTML container with the unique id (this.next_id)
      //   view and io handler will put other elements within it

      sim = new Simulation object of the class name: (name + "Sim")
      this.sim_list.append(sim)
      view = new View object of the class name: (name + "View")
      this.view_list.append(view)
      io = new IOHandler of the class name: (name + "IO")
      this.io_list.append(io)
      //   initialise sim, view and io for a new simulation, pass
      //   this.next_id as a parameter to their constructors (IO Handler
      //   will require more parameters: discussed later)

      this.next_id += 1
      //   increment id for the next simulation

      add buttons that terminate, pause and continue simulations
      //   html input elements of type button with onclick functions
      //   this.terminate_sim(sim), sim.pause() and sim.continue()
      //   resepctively
endprocedure
```

3.12.3 Terminating a simulation

Iterate through the list of sims, delete required one and delete it from the html page.

[8] Set of variables and functions that can be accessed from within the scope of the block of code.

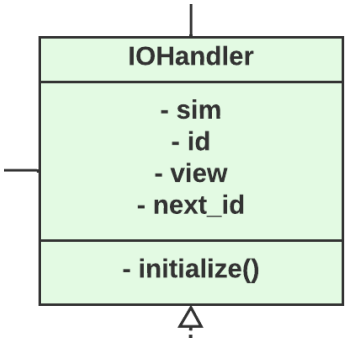
```
public procedure terminateSim(sim: Simulation)
      for i=0 to this.sim_list.length - 1
            if this.sim_list[i] == sim then
                  id = this.view_list[i].getId()
                  delete sim area by the id
                  //   deletes the simulation area dedicated for canvas
                  //   and input / output elements

                  this.sim_list.delete(i)
                  this.view_list.delete(i)
                  this.io_list.delete(i)
                  //   once the simulation is found in the list,
                  //   delete the sim, view and io handler by the index
                  return
            endif
      next i
endprocedure
```

3.12.4 Update loop

Iterate through the list of active simulations and invoke their update methods (may be several times dependent on how fast can simulation be updated). Redraw the simulation and update IO handlers (happens automatically as the callback functions for button click / text box update are handled separately). Takes dt as a parameter. It represents the time elapsed from the last update method call. It is normally handled by appropriate JavaScript methods. E.g., "`requestAnimationFrame`" method automatically passes current timestamp every time it invokes its callback functions.

```
public procedure update(timestamp: real)
      dt = timestamp - prev_timestamp
      //   calculating a time from the last frame
      for i=0 to this.sim_list.length - 1
            if this.sim_list[i].isActive()
                  this.sim_list[i].update(dt)
                  this.view_list[i].redraw()
            endif
      next i
endfunction
```

3.13 IO Handler



Input Output handler class will initialise and manage all input and output information that is received from the textboxes, sliders, checkboxes, and other types of inputs that change any parameters of the simulation. It tightly links to the model as all the new objects, their properties and nearly all parameters are controlled by the input output handler. It implements all inputs and outputs that reside in the IO area of the simulation. That is all inputs apart from stop, continue and close buttons as they are managed by the main Controller. The constructor method will receive already created view and simulation objects from the Controller. It will invoke the "`initialise()`" method that starts the process of creating input output area. This process is broken down into multiple methods that deal with their specific part of the I/O area.

Specific input output handler implementations will extend functionality of the superclass and implement methods dictated by the superclass.

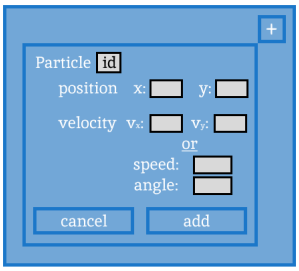
3.13.1 Particle Projection IO

Following the GUI designed defined earlier, the IO area consists of 3 main parts: particle list, event area, and the time slider (together with the reset button).

```
public procedure initialise()
      this.createParticleArea()
      this.createEventArea()
      this.createTimeSlider()
      //   invoking methods that create every part in order
endprocedure
```

```
public procedure createParticleArea()
      create dedicated section on the I/O area
      put the "List of particles" title on the section
      create particle list section on the particle area
      //   this section is dedicated for the list of particle which is
      //   initially empty
      create the "add_particle_button"
      "add_particle_button" listen for the click
            invoke this.addParticle() function on click
      //   this function starts the process of adding a new particle
endprocedure
```

"add_particle_button" will listen for any clicks invoking the function that puts up the input form to receive parameters of the new particle. An alternative GUI input might be created as an extension after meeting all the essential user requirements.



The "`addParticle()`" method puts up all the required text inputs. And once the user clicks on the add button, the I/O handler validates all the input received and if it satisfies all the checks, the particle gets created and added to the body list of the simulation. The view version of the particle is created right afterwards and added to the view list of bodies.

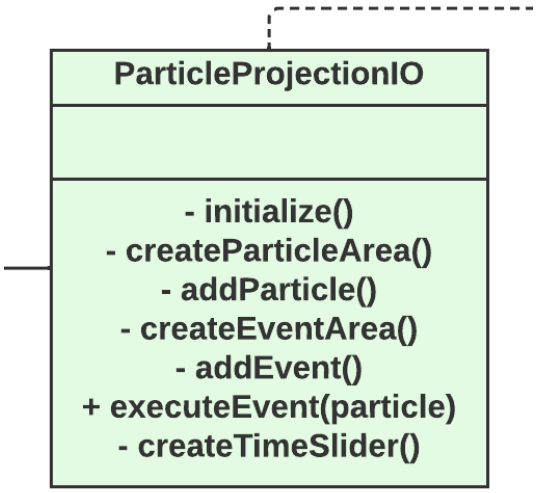
```
public procedure createEventArea()
      create dedicated section on the I/O area
      put the drop-down list on the section
      //   options of the list are PositionEvent/VelocityEvent/TimeEvent
      //   it defines the type of event to be added
      create occurred events section on the event area
      //   initially empty
      create the "add_event_button"
      "add_event_button" listen for the click
            invoke this.addEvent() function on click
      //   this function starts the process of adding a new event
endprocedure
```

The button listens for any input and responds by invoking the "`addEvent()`" that like an "`addParticle()`" method creates the input menu for a new event, validates it and adds it to the queue of events. The event input menu was discussed in the Event section earlier.

The "`executeEvent(particle)`" method is invoked by the event once it has occurred. It takes the event out of the queue and puts is to the area of occurred events mocking the parameters of the considered particle and the current time of the simulation.

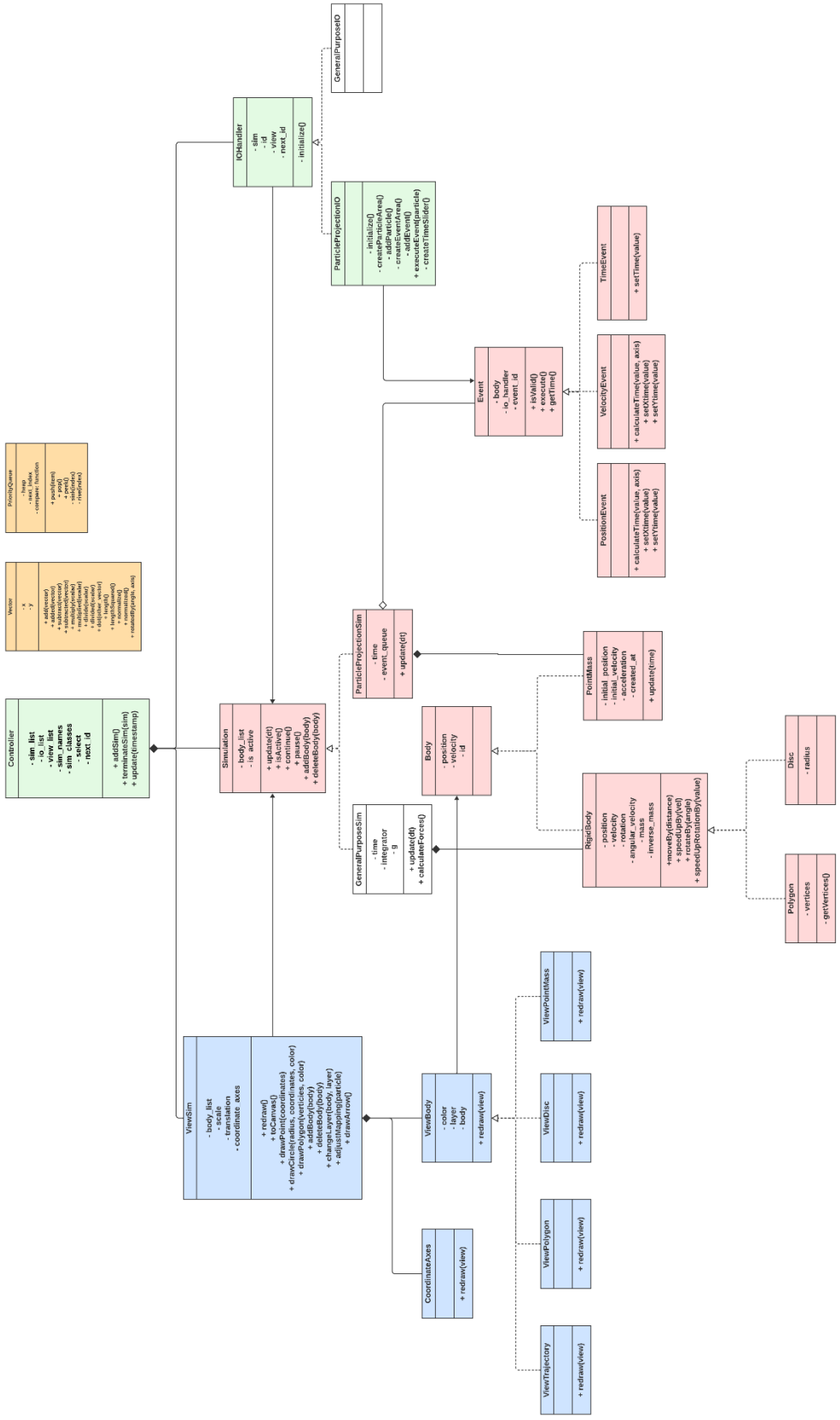
The time slider will allow the user to control the simulation time, quickly returning to earlier or later moments if necessary. The simulation gets stopped, the I/O handler accesses the time variable through a setter and changes it to the value received by the user.

The following step by step process diagram will show in which order interface must be built (Read top to bottom, left to right).




Create io_area html element
Create Particle Area
Add a title to it
Create list of particles
Create input menu
Create button for an input menu
Put the menu on the click
Remove menu on the next click
Alternate
Increment next particle id
Increment next event id
Creating input menu
Create title
Create position input
Create velocity input
Create alernative input form
Create button to add particle
Validate all input
If an errror, output error message
Add to simulation otherwise
Add a particle to the list of particles
Add particle to Sim and ViewSim
Creating event area
Create title
Create button to toggle event menu
Create an event type drop-down
Create a particle drop-down
Create condition input
Create button to add an event
Validate an input
Add an erorr message if invalid
Add to simulation otherwise
Add an event to the queue
Creating a time slider
Create area for the time slider
Create input of type range
Add an update method for the slider
Monitor new events being added
Adjust the time slider
Mark an occured event on the slider
Add reset button
Reset simulation time to 0 on click
Stop a sim on click on the slider
Set Simulation time to desired

3.14 Final UML diagram



3.15 Post development tests

Once all development is finished, the program must be capable of modelling particle projection accurately, presenting to the user state of the system at any given point in time in both visual and textual formats. Some examples of the problems that can be modelled and solved with my program are given in the section of event designs.

Main tests should involve stakeholders being presented with the program with very little / no prior knowledge of how the program works. They should be able to navigate the page with the instructions given to them when they first open it. When interviewing stakeholders, record the time they take to read instructions, it will assess how easy they are to read and understand. Right after this stage, the stakeholders should start interacting with the program. Quality of the interface is assessed by how easy it is for them to find one or the other button when they are given a task like “create a new particle” or “start simulation”. The main test data gathered from these interviews are any inconveniences stakeholders had when they used the program. These will be focus on the first stages of further development.

To test usability of the program, I will present stakeholders with several problems (the same ones mentioned in the first paragraph) and ask them to solve these problems with no aid from the interviewer. It will make interviewees think through instructions given at the start on their own and apply experience gained from the first stage of the interview. It gives very valuable data on how easy it is to apply my program to actual problems of A-level mechanics and shows weak places or lacking features of the software.

Apart from interviews with stakeholders, I will go over each success criteria and evaluate how actual program meets expectations from analysis.

4 Development and testing

The solution is developed using JavaScript for all the logic with HTML providing the graphical interface and elements to receive the user input and output information. CSS will be used to arrange HTML elements and stylize them, whenever necessary.
Often during the development some explanations will be followed by snippets of commented code or relevant screenshots.

At the start of design, I planned on making two simulations: Particle Projection Simulation and General-Purpose Simulation. Eventually, I ended up spending all the design on the first one. Nevertheless, Particle Projection Simulation is the one that can potentially be helpful for target users (students and teachers), whereas the General-Purpose Simulation does not apply to their needs, apart from some satisfaction from creating objects and moving them around simulation. This is the main reason I prioritised Particle Projection Simulation over General Purpose Simulation.

4.1 Development order

The development process will be based around satisfying all the essential features of the solution following implementation of the earlier designed algorithms and data structures. Stakeholder reviews will be gathered in between the development sections whenever a prototype attains a new level of usability, such as a new set of features implemented, or GUI redesigned.
I will divide development in the following iterations:

- Initial setup: creating a start-up page, hosting it on GitHub and setting up the structure of classes implementing the controller.
- Implementing Utility Package, i.e., classes for Vectors and Priority Queue.
- Visual representation: This step is about developing an entire View part of the solution. It includes all the methods for drawing primitive shapes. PointMass, RigidBody abstract class with its implementations, Disc and Polygon classes.
- Implementation of the Particle Projection Simulation with the system of Events in isolation from the input interface which will be implemented in the next stage.
- Implementation of the Particle Projection IO handler. It includes major part of interface development.
- Finishing off with additional functionality left out from precious stages.

As all the parts of my program are interconnected and one method will call a lot of other methods continuously, I will write “placeholder” methods for those that are not yet implemented but are already requested by other methods. For example, the first class to be implemented is a Controller. But the Controller instantiates other classes like ViewSim. So, I will make a constructor method for ViewSim to be printing “ViewSim instantiated” before I start fully implementing the class. The same principle applies to functions.
In this way, I will be able to see if the method I just implemented correctly refers to other parts of the program before I start coding them.

4.2 Initial setup

Success criteria: starting interface; web application as a static website; multiple simulations at a time; starting time control system (the controller update method).

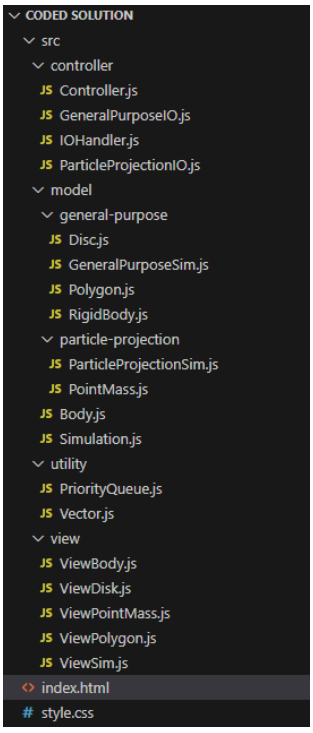
4.2.1 Structure

All the JavaScript code for the solution will be in an “src” folder sorted in other folders by its purpose to make the solution structure neater. Every class will be in a separate .js file. The code for every file will start with all the required inputs and end with an export statement. So, for example:

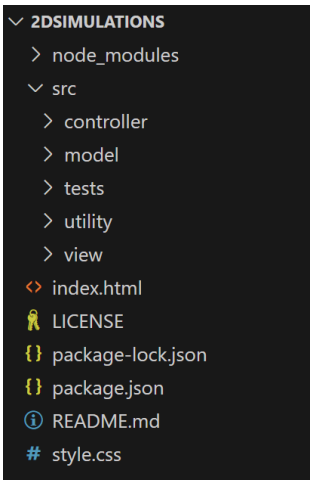
```
import Apple from "src/Apple.js"

class Tree {
    // code
}

export {Tree as default}
```

Initial structure of the project will look in a following way. All the code will reside in an “src” folder where JavaScript files are grouped by their purpose, mostly by their role in a model-view-controller pattern. An “index.html” file will represent the main and only (for now) html page where all the simulations will run. Style of some elements will be defined in a “style.css”, other elements’ style will be directly dictated by the JavaScript code or assigned a class with already defined style. This structure is initially derived from the complete UML diagram, designed earlier.

Initial start-up page will be quite empty as nearly all elements are created and added on the page by the Controller and I/O Handlers. The page will contain some meta data which is just generally a good practice to include. It will link the stylesheet and run a couple of lines of JavaScript code that instantiates the Controller and stores it in a variable. The script tag is assigned an attribute “type” as “module” to enable import statements.



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>2d Simulations</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <p>Brief instructions</p>

    <script type="module">
        import Controller from "/src/controller/Controller.js"

        let controller = new Controller();
    </script>
</body>
</html>
```

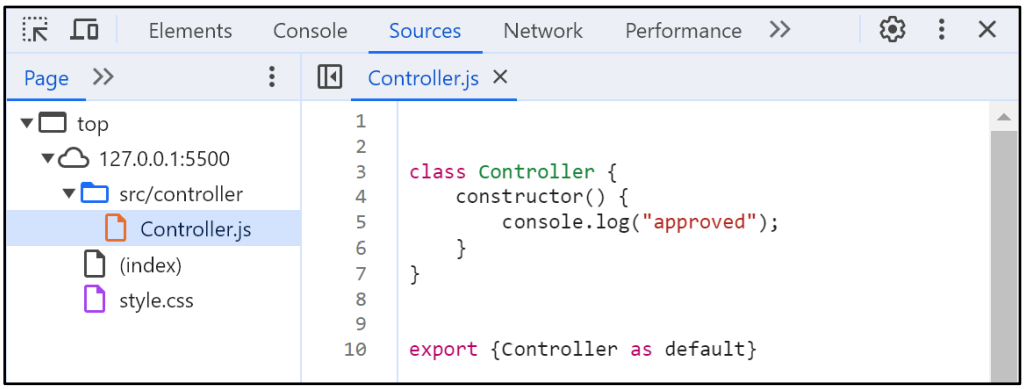


For testing purposes, I also installed “Jest” framework that I will use to run unit tests and evidence all successful and unsuccessful tests. I created GitHub repository where I will regularly commit updates on the project. The main reason for using GitHub is that it provides free to use static site hosting service that takes HTML, CSS, and JavaScript files straight from the repository. It automatically deploys all the updates.

“package.json” and “package-lock.json” files with “node_modules” folder are present for dependency and version management (Jest is the only external module so far).

4.2.2 Controller constructor

Testing that the script works as intended, I checked the Chrome’s “Sources” section which proved that the controller script was correctly imported.



4.2.2.1 Implementation

Developing the code, I will not worry too much about the appealing look of the interface leaving it for the later stages and discussions with the stakeholders. For now, I will apply the same style for all the descendent elements of the body tag. It will highlight them and show the space they take up.

```css
body * {
    background-color: rgb(46, 90, 231);
    color: aliceblue;
    padding: 10px;
    max-width: 720px;
    outline: 2px solid turquoise;
    font-size: 14pt;
}
```

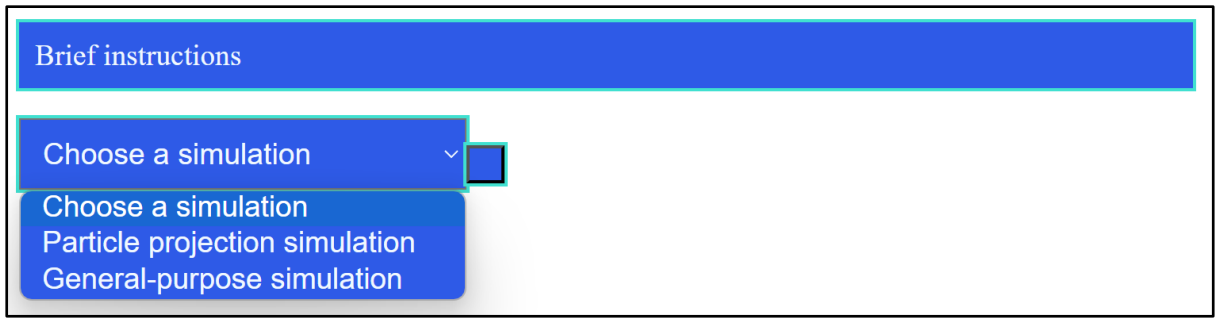
Setting the background to be blue with the turquoise outline and aliceblue (contrasting colour to blue) font colour. The page initially looks in a following way:

Coding the controller, I will firstly rewrite the pseudocode in JavaScript code with little to no changes apart from some JavaScript syntax specific amendments. The newly coded part of code retrieves the body html elements to append the drop-down menu and the button.

```
        let body_element = document.getElementsByTagName("body")[0];
        // .appendChild(this.select);
        //   retreive the body tag from the page
        body_element.appendChild(this.select);
        body_element.appendChild(add_sim_button);
        //   put the dropdown and the button on the screen
```

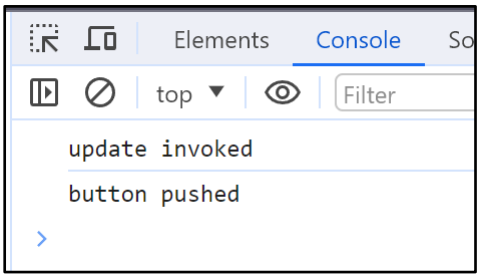
4.2.2.2 Testing

Quickly checking if the method is functional:



The drop-down works as well as the button.

```
update(timestamp) {
    console.log("update invoked");
}

addSim() {
    console.log("button pushed");
}
```

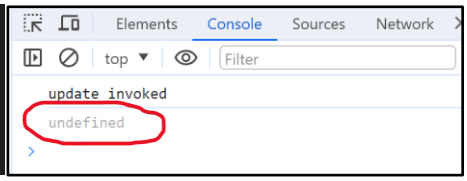


4.2.3 Adding a simulation

4.2.3.1 Implementation

Firstly, I decided to check if the method invoked from the button click has access to the Controller's attributes.

```
addSim() {
    console.log(this.select)
}
```

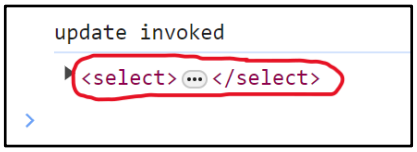


Click on the button showed that the value for the “select” attribute of a controller is unknown within the function scope. Researching the problem, I found out that this issue occurs because I assigned the “addSim()” function to the button in this way:

```
add_sim_button.onclick = this.addSim;
```

Because of that, next time the button is clicked, it calls the code for this function from the button’s namespace rather than as a controller. One way to fix this is by using closures[9]. Creating an anonymous function that has access to the controller’s scope and invoking “addSim()” within it. Passing this function instead will fix the issue:

```
add_sim_button.onclick = () => {
    this.addSim();
}
```



Now the addSim function can identify the select attribute of the controller.

Developing this function, it turned out that the “name” identifier cannot be used in JavaSctipt, so I changed it to “class_group”.
Adding the code that creates a simulation area and retrieves body tag to add the simulation on the screen further down the code.

```
        let body_element = document.getElementsByTagName("body")[0];
        //   retrieve the body element
        let sim_area = document.createElement("div");
        sim_area.id = "sim_area" + this.next_id.toString();
        //   assigning an identifier to the new simulation area
```

The next step of creating instances of the view, model and I/O handler for the newly added simulation is carried out using the “eval()” JavaScript function that runs the code passed to it as a string parameter. It allows me to use the string name of the class group to create new instances for these classes. For example, `eval("new ParticleProjectionIO()")` will return a new instance of the Particle Projection I/O handler. This step is taken to abstract the “addSim” method from specific implementations of Simulation using the same code for each of them.

```
        let view = new ViewSim(this.next_id);
        this.view_list.push(view);
        let sim = eval("new " + class_group + "Sim()");
        this.sim_list.push(sim);
        let io = eval("new " + class_group + "IO(this.next_id, sim, view)");
        this.io_list.push(io);
        //   instantiating view, sim and io for the new simulation and
        //   adding them to the list
```

[9] The term closure refers to defining a function within the scope of the other function, so created function has access to the outer’s function scope when it is called.

Creating the required buttons and assigning their corresponding functions again using closures. The code is self-explanatory.

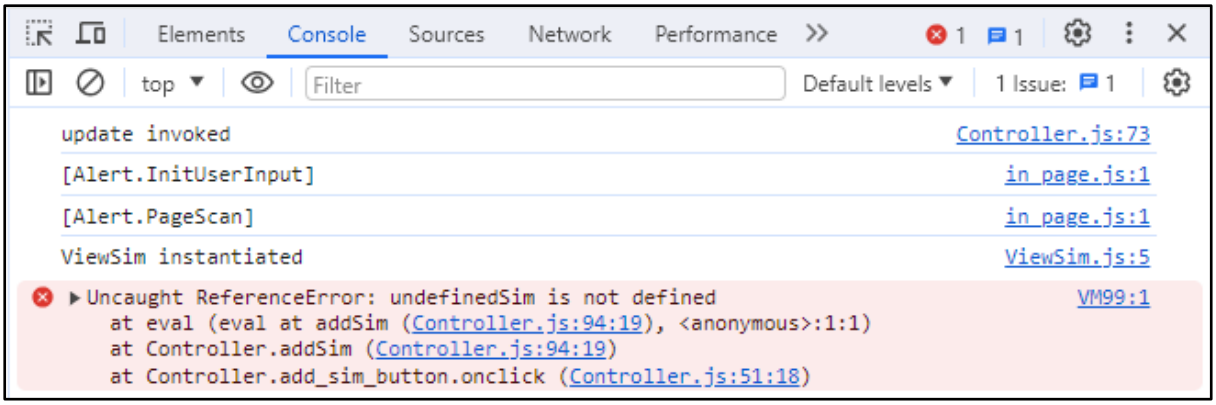
```
        let terminate_sim_button = document.createElement("button");
        let pause_sim_button = document.createElement("button");
        let continue_sim_button = document.createElement("button");

        terminate_sim_button.onclick = () => {
            this.terminateSim(sim);
        }

        pause_sim_button.onclick = () => {
            sim.pause();
        }

        continue_sim_button.onclick = () => {
            sim.continue();
        }
        //   adding buttons that terminate, pause and continue the simulation
        //   and assigning to them corresponding function using clousers
```

At last, creating a title and putting all the elements together. First put the title and all the buttons on the sim area and then entire sim area into body element.

```
        let title = document.createElement("p");
        title.innerHTML = this.sim_names[index];
        //   creating a title paragraph html tag and setting changing its
        //   content to the name of the new simulation being added

        sim_area.appendChild(title);
        sim_area.appendChild(pause_sim_button);
        sim_area.appendChild(continue_sim_button);
        sim_area.appendChild(terminate_sim_button);
        //   adding buttons and the title on the sim_area

        body_element.appendChild(sim_area);

        this.next_id++;
        //   increment id for the next simulation
```

4.2.3.2 Testing

Refreshing the page and clicking on the button resulted in an error:



First line: the update method got invoked once (temporary placeholder because the constructor calls an update function). Second and third lines are irrelevant. Fourth line: code for instantiating a ViewSim got executed and then an error occurred as using an “eval()” function as it tried to instantiate an “undefinedSim”. Having a look at this line:

```
let sim = eval("new " + class_group + "Sim()");
```

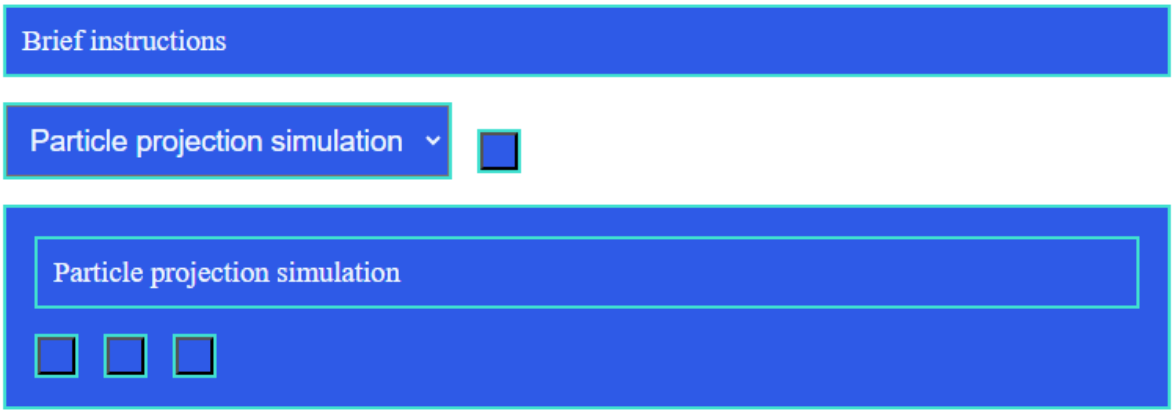
Apparently, the “class_group” variable had value “undefined” leading to this error. This variable is created with the following line of code:

```
let class_group = this.sim_classes[index];
```

where index is the currently chosen value of the dropdown menu.
The problem occurs because the value of the dropdown menu is always received as a string, even though an integer was assigned to it when the dropdown (i.e., select tag) was created. To fix the error, the index just needs to be changed to an integer when retrieved:

```
let index = parseInt(this.select.value);
```

Now the page loads with no errors and buttons respond in a following way:



* I added 10 pixels of margin to the style for now.

As expected, all elements are present on the sim area. If no option is chosen, nothing happens and simulations are added one after the other with correct response from other, not yet implemented methods (diagram on the left).

```
update invoked
ViewSim instantiated
ParticleProjectionSim instantiated
ParticleProjectionIO instantiated
>
```

Multiple simulation areas are created dynamically and have correct identifiers assigned to them as can be seen from the HTML page.

I also changed an id system such that the id is no longer separated by an underscore form the name of the element. It does not make a huge difference, except now to create a new element, its HTML id will look like “name + id” rather than “name + ‘_’ + id”.

```
<!DOCTYPE html>
<html lang="en"> == $0
  <head> ... </head>
  <body>
    <p>Brief instructions</p>
    <script type="module"> ... </script>
    <!-- Code injected by live-server -->
    <script> ... </script>
    <select> ... </select>
    <button></button>
    <div id="sim_area0"> ... </div>
    <div id="sim_area1"> ... </div>
    <div id="sim_area2"> ... </div>
    <div id="sim_area3"> ... </div>
    <div id="sim_area4"> ... </div>
  </body>
</html>
```

Pause, continue, and close (terminate) simulation buttons are also invoked correctly:

```
pause sim function invoked                    Simulation.js:5
continue simulation function invoked          Simulation.js:9
terminate sim function invoked                Controller.js:137
>
```

Again, these outputs are what is temporarily written as the methods that pause, continue, and terminate buttons invoke.

4.2.4 Terminating a simulation

4.2.4.1 Implementation

Terminating procedure is very simple as the method receives a reference to the simulation as a parameter when the button to remove the simulation from the page is created. Next, a linear search can be performed to find index of the simulation in the sim_list, view_list and io_list (the index is the same for every list). The index in the list might not match the overall simulation id as new simulation can be created and deleted. However, it can be guaranteed that the lists hold simulations in a sorted order of their indices, so linear search can be substituted with more efficient binary search. This optimisation is entirely unnecessary as the user is unlikely to have more than 3 simulations opened at a time. On such scales, the difference will not be noticeable at all.

```
terminateSim(sim) {
```

```
    for (let i = 0; i < this.sim_list.length; i++) {
        if (this.sim_list[i] == sim) {
            //   i will represent an index of the simulation in the lists
            let id = this.view_list[i].getId();
            //   getting an id of the simulation
            let sim_area = document.getElementById("sim_area" + id);
            sim_area.remove();
            //   remove entire simulation area

            this.sim_list.splice(i, 1);
            this.view_list.splice(i, 1);
            this.io_list.splice(i, 1);
            //   remove all references to the simulation from the controller
lists

            return
            //   stop the function as the simulation was found
        }
    }
}
```

The comparison is carried out with "==" operator which compares the addresses in memory of the two objects. It returns true whenever two variables contain references to the same object which is what I need for this method.

Also, for the method to correctly retrieving an id of the simulation from the View object, I added an id getter into the View class.

4.2.4.2 Testing

First, there needs to be way to differentiate buttons, so I went through all the ones created so far and added little signs indicating the button's function. I also slightly change the style of the buttons:


```
body button {
    width: 45px;
    height: 45px;
    text-align: center;
}
```

```
add_sim_button.innerHTML = "+";
terminate_sim_button.innerHTML = "X";
pause_sim_button.innerHTML = "||";
continue_sim_button.innerHTML = ">";
```

Click on the terminate sim button successfully finds correct simulation and deletes it from the lists.

4.2.5 Update loop

4.2.5.1 Implementation

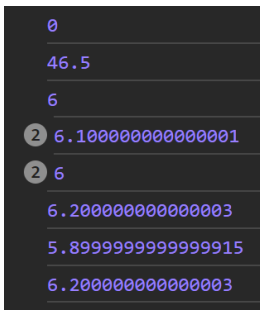
For the method to work correctly, I first added a “prev_timestamp” attribute to the Controller class that is assigned an initial value of 0 in the constructor method right before the update loop is started.

```
update(timestamp) {
    let dt = timestamp - this.prev_timestamp;
    //   calculating time from the last update
    this.prev_timestamp = timestamp;

    for (let i = 0; i < this.sim_list.length; i++) {
        if (this.sim_list[i].isActive()) {
            //   update and redraw every active simulation
            this.sim_list[i].update(dt);
            this.view_list[i].redraw();
        }
    }

    requestAnimationFrame((timestamp) => {
        this.update(timestamp);
    });
}
```

The update method is called recursive using the “requestAnimationFrame” JavaScript function that requests the browser to execute the callback function before the next repaint. The frequency of calls typically matches the display refresh rate. It passes the time elapsed from the moment the webpage was opened as a parameter to the callback function. It allows the program to frequently request an update with precise timestamps. The update method is again passed as a callback using closures.

4.2.5.2 Testing

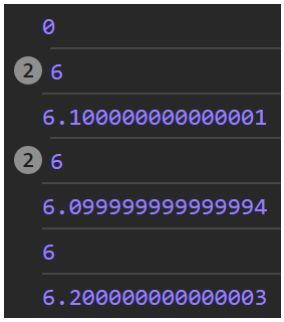
Firstly, testing if the delta time is calculated correctly. Outputting the results in the console gives an average of 6 to 6.1 milliseconds between frames. Except for the first two function calls. This occurs due to the fact that the first call of the update method is carried out without “requestAnimationFrame” method with manually passed timestamp of 0 which does not reflect reality as it takes some time to instantiate a controller and when the program reaches the first call of “requestAnimationFrame” function, the value of the timestamp exceeds previous timestamp by a substantial amount which might cause simulation to jump forward too much, if it receives this value as delta time. On the other hand, it is unlikely that the user will create a simulation within the first milliseconds they open the webpage. An easy way to fix is to first call the update method with “requestAnimationFrame” function and within that call, assign the value of the first timestamp to the ”prev_timestamp” attribute:




```
requestAnimationFrame((timestamp) => {
    this.prev_timestamp = timestamp;
    //   the first instance when the update method is called
```

```
    this.update(timestamp);
});
```

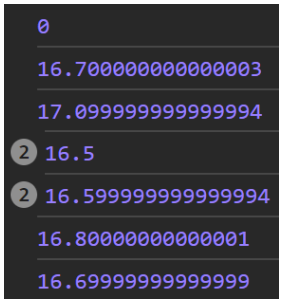
It will allow the update method to start from a delta time of 0 and work correctly afterwards. Further values do not show bigger derivations from the value of 6. It is mostly computer dependent, however the program should still run smoothly even if at some point, it takes longer to get from one frame to the other. As the overall time is kept consistent, all simulations will simply adjust to it by making a slightly bigger leap in time.



165 Hz

The values of 6 to 6.1 correspond to 1000 / 6.1 to 1000 / 6 frames per second, which is 164 to 167 fps. And it correctly matches the refresh rate of my computer (165 Hz).

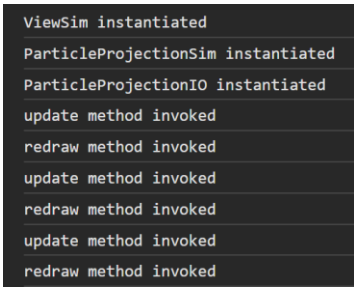
Checking results for a refresh rate of 60 Hz:
16.5 to 17 - 59 to 61 fps. An acceptable range.



60 Hz

Testing the loop through the list of functions, I will have to write the "isActive()", "pause()" and "continue()" methods for the Simulation class which consist of only one for returning "is_active" attribute, reassigning a value of false and true respectively.

Now when the simulation is created, the update and redraw methods are called repeatedly stopping when the stop or close buttons are clicked and continuing when the continue button is clicked.



4.3 Utility Classes

This section itself does not address any specific success criteria. It will assist all other parts of the program like simulation object updates / event queue, etc.

4.3.1 Vector

4.3.1.1 Constructor

```
constructor(x = 0, y = 0) {
    this.x = x;
    this.y = y;
}
```

* Vector getters and setters simply assign or return value so they will be coded but no discussed
The default values of 0 are assigned to the x and y parameters, so a zero vector can be created as simply as "new Vector()".

4.3.1.2 Mathematical operations

```
add(vector) {
    this.x += vector.x;
    this.y += vector.y;
```

```
}

added(vector) {
    let x = this.x + vector.x;
    let y = this.y + vector.y;
    //   creating temporary variables for new x and y variables,
    //   original vector is unchanged
    return new Vector(x, y);
}

multiply(scalar) {
    this.x *= scalar;
    this.y *= scalar;
}

multiplied(scalar) {
    let x = this.x * scalar;
    let y = this.y * scalar;
    return new Vector(x, y);
}
```

Here I changed the pseudocode slightly. Now I access the x and y attributes for the vector parameter directly without getters as the method is within the Vector class, therefore encapsulation will not be violated. The “subtract()”, “subtracted()”, “divide()”, “divided()” are coded similarly.

4.3.1.3 The dot product of function

```
dot(other_vector) {
    //   returns the dot product between “this” vector
    //   and the vector passed as a parameter
    //   has a commutative property: a.dot(b) = b.dot(a)
    return this.x * other_vector.x + this.y * other_vector.y;
}
```

4.3.1.4 Length and length squared functions

```
lengthSquared() {
    return this.x * this.x + this.y * this.y;
}

length() {
    return Math.sqrt(this.lengthSquared());
}
```

4.3.1.5 Normalisation

```
normalize() {
    if (this.x === 0 && this.y === 0)
        throw new Error("Zero vector cannot be normalised");
    this.divide(this.length());
}
```

```
normalized() {
    if (this.x === 0 && this.y === 0)
        throw new Error("Zero vector cannot be normalised");
    return this.divided(this.length());
}
```

The code is self-commenting. The first method divides the vector by its length. The second one returns new normalised vector. Before executing the code, methods check if the vector has no zero length, i.e., both components must not be zero at the same time.

4.3.1.6 Rotation

The method is explained in design and does not differ in development. The sine and cosine functions are attained as “Math.sin()” and “Math.cos()”.

```
rotatedBy(angle, axis = new Vector()) {
    let rotated_vector = this.subtracted(axis);
    //   first tranlate the axis to the origin

    rotated_vector = new Vector(
         rotated_vector.x*Math.cos(angle) - rotated_vector.y*Math.sin(angle),
         rotated_vector.x*Math.sin(angle) + rotated_vector.y*Math.cos(angle)
    );
    //   rotate vector like that

    rotated_vector.add(axis);
    //   translate the axis back
    return rotated_vector;
}
```

The default value for axis is zero Vector, i.e., the origin.

4.3.1.7 Testing

Testing vectors, I used the Jest framework mentioned earlier. I divided the tests into blocks by functions, with little caption for each individual test:

```
describe("addition / subtraction", () => {
    test("(2,9) + (4,8)", () => {
        expect(new Vector(2,9).added(new Vector(4,8))).toEqual(new Vector(6,17));
    })

    test("(2,9) - (4,8)", () => {
        expect(new Vector(2,9).subtracted(new Vector(4,8))).toEqual(new Vector(-2,1));
    })
```

The block continues for all addition and subtraction methods tests. The “expect” method receives a method call as a parameter and the “toEqual()” method gets expected results for the operation. Jest generates an output indicating successful and failed test describing differences between received and expected output.
Almost all tests follow the same pattern of calculating the output and matching it to precalculated expected results.

To test the “rotateBy()” method, I wrote an auxiliary function that will calculate the rotated vector and truncate the results leaving first 3 decimal places.

```
function rotatedRounded(vector, angle, axis = new Vector()) {
    let rotated_vector = vector.rotatedBy(angle, axis);
    rotated_vector.x = Math.trunc(rotated_vector.x * 1000) / 1000;
    rotated_vector.y = Math.trunc(rotated_vector.y * 1000) / 1000;
    return rotated_vector;
}
```

The default value for axis is also set to a zero vector. This method is then used for tests:

```
test("(5,8) rotated by 1 radian around (0,0)", () => {
    expect(rotatedRounded(new Vector(5, 8), 1)).toEqual(new Vector(-4.030, 8.529));
})

test("(-20, 8.1) rotated by 13.8 radian around (0,0)", () => {
    expect(rotatedRounded(new Vector(-20, 8.1), 13.8)).toEqual(new Vector(-14.260, -16.194));
})
```

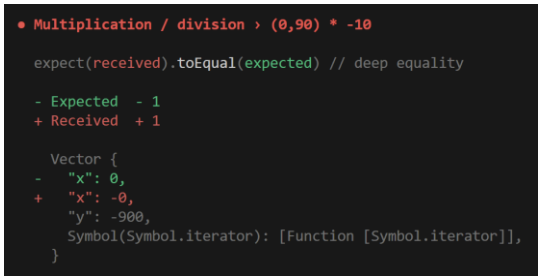
Test results:

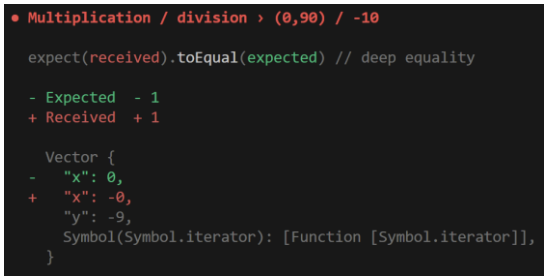
Function being tested	Expected Output	Successful
added()	(6, 17)	addition / subtraction √ (2,9) + (4,8) (2 ms) √ (2,9) - (4,8) √ (-10,-11) + (3,3) (1 ms) √ (-10,-11) - (3,3) √ (3,3) - (-10,-11) √ (3,3) - (-10,-11) (1 ms)
subtracted()	(-2, 1)	
added()	(-7, -8)	
subtracted()	(-13, -14)	
added()	(-7, -8)	
subtracted()	(13, 14)	
multiplied()	(24, 36)	Multiplication / division √ (6,12) * 3 √ (6,12) / 3 (1 ms) √ (-7,200) * 4 (1 ms) √ (-7,200) / 4 (1 ms) × (0,90) * -10 (3 ms) × (0,90) / -10
divided()	(2, 4)	
multiplied()	(-28, 800)	
divided()	(-1.75, 50)	
multiplied()	(0, -900)	
divided()	(0, -9)	
dot()	90	dot product √ (0,10) . (-7,9) √ (2,2) . (-1,2) √ (0,10) . (-7,9)
	2	
	0	
length()	5	length √ length of (3,4) √ length of (12,-5)
	13	
lengthSquared()	13	lengthSquared √ squared length of (2,3) √ squared length of (6,-10)
	136	
normalized()	(0.6, -0.8)	normalisation √ normalise (2,3) (1 ms) √ normalise (80,-60)
	(0.8, -0.6)	
rotatedBy()	(-4.030, 8.529) to 3dp rounded down	
rotatedBy()	(-14.260, -16.194) to 3dp rounded down	rotation √ (5,8) rotated by 1 radian around (0,0) √ (-20, 8.1) rotated by 13.8 radian around (0,0) × (4, 7) rotated by -Pi/2 radian around (0,0) (1 ms) √ (3, 3) rotated by -Pi/2 radian around (2, 1) √ (5,2) rotated by Pi/4 radian around (3,4)
rotatedBy()	(7, -4)	
rotatedBy()	(4, 0)	

rotatedBy()	(5.828, 4.000) to 3dp rounded down	

```
Test Suites: 1 failed, 1 total
Tests:       3 failed, 23 passed, 26 total
```

2 failed tests are for multiplication and division and 1 for rotation. Starting with division and multiplication:

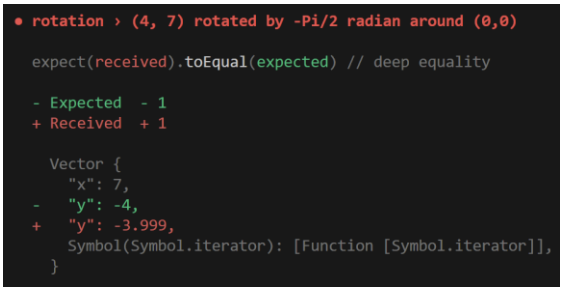




Both tests were failed due to the fact JavaScript distinguishes negative zero from the positive zero. The "-0" occurs because of 0 divided by a negative number. This difference does not affect calculations, so these results are perfectly acceptable, and tests can account for that changing expected output.

Rotation:

In this case, the test received an output of truncated -3.999 instead of -4. The problem in this case is that the "Math.cos(-Math.PI/2)" evaluated to a very small number instead of exactly 0:




```
> Math.cos(-Math.PI/2)
< 6.123233995736766e-17
```

Therefore, calculations give very close results:

```
> 4 * Math.sin(-Math.PI/2) + 7 * Math.cos(-Math.PI/2)
< -3.9999999999999996
```

Also acceptable because result is acceptable when rounded.

```
Test Suites: 1 passed, 1 total
Tests:       26 passed, 26 total
```

Reflection:

Simply rewriting pseudocode methods one to one

```
reflectedInX() {
    let new_y = this.y * (-1);
    return new Vector(this.x, new_y);
}
```

```
reflectedInY() {
    let new_x = this.x * (-1);
    return new Vector(new_x, this.y);
}
```

4.3.2 Priority Queue

Priority Queue will be developed in two stages, first implementing the "push()" and "rise()" methods that allow to add new items into the queue and next implementing "pop()" and "sink()" methods for retrieving items from the queue. Each stage will be followed with corresponding tests.

4.3.2.1 Implementation Stage 1

First implementing pseudocode with no changes apart from integer division to get the address of the parent node is calculated with following code in JavaScript:

```
let parent_index = Math.floor(index / 2);
```

```
class PriorityQueue {
    constructor(compare) {
        this.heap = [null];
        //   a list that is used to store the heap, the first element is
        //   null because the heap starts at index 1
        this.next_index = 1;
        //   index of the next free space in the list
        this.compare = compare;
    }
```

```
push(item) {
    this.heap[this.next_index] = item
    //   add item to the heap
    this.rise(this.next_index)
    //   fix the property of the heap
    this.next_index += 1
    //   increment the index for the next element
}
```

```
rise(index) {
    while (index > 1) {
        //   while the current item is not in the first node
        //   (because the root node cannot rise anymore)
        let parent_index = Math.floor(index / 2);
        //   getting index of the parent element
        let item = this.heap[index]
        let parent_item = this.heap[parent_index]
        if (this.compare(item, parent_item) > 0) {
            //   if the item is larger than its parent, perform a
            //   swap
            this.heap[index] = parent_item
            this.heap[parent_index] = item

            index = parent_index
            //   the next node to consider is now a parent of
            //   the previous node
        }
        else {
            //   if the item is less than its parent, then the
            //   property is satisfied
            break
            //   exit the loop
        }
    }
}
```

4.3.2.2 Testing Stage 1

The test client I coded will contain a function that initialises the basic queue which will be reused for both stage 1 and stage 2 tests.

```
//   function for initialising the basic priority queue:
//   [null, 30, 25, 20, 10, 3, 12, 18, 2, 4]
//   used for both stage 1 and 2 tests
function initializeQueue() {
    let pq = new PriorityQueue((a, b) => a - b);
    //   priority queue with the comparator that
    //   prioritises the larger element
    for (let element of [30, 25, 20, 10, 3, 12, 18, 2, 4]) {
        pq.push(element);
    }
    return pq;
}
```

It goes through the list of elements of the heap in the order they are desired to be in the queue.

The subject of tests will be the heap of the queue. Each test will attempt some to invoke “push” method with some argument. The diagrammatic explanations of what must happen as the result of each test was described in design. From the diagram, desired structure of the heap can be clearly seen, so tests can be formed.

```
describe("Stage 1 testing", () => {
    let pq = initialiseQueue();

    test("test (0)", () => {
        expect(pq.heap).toEqual([null, 30, 25, 20, 10, 3, 12, 18, 2, 4]);
    });

    test("test (1)", () => {
        pq.push(12)
        expect(pq.heap).toEqual([null, 30, 25, 20, 10, 12, 12, 18, 2, 4, 3]);
    });

    test("test (2)", () => {
        pq.push(29)
        expect(pq.heap).toEqual([null, 30, 29, 20, 10, 25, 12, 18, 2, 4, 3, 12]);
    });

    test("test (3)", () => {
        pq.push(-10)
        expect(pq.heap).toEqual([null, 30, 29, 20, 10, 25, 12, 18, 2, 4, 3, 12, -10]);
    });

    test("test (4)", () => {
        pq.push(42)
        expect(pq.heap).toEqual([null, 42, 29, 30, 10, 25, 20, 18, 2, 4, 3, 12, -10, 12]);
    });
});
```

Test (0) will check if the “initializeQueue()” function correctly built the heap. Results of the test are as follows:

All tests have been passed.

```
PASS src/tests/PriorityQueue.test.js
  Stage 1 testing
    √ test (0) (2 ms)
    √ test (1)
    √ test (2) (1 ms)
    √ test (3) (1 ms)
    √ test (4) (3 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        0.389 s, estimated 1 s
```

4.3.2.3 Implementation Stage 2

Again, simply rewriting the pseudocode to JavaScript.

```
sink(index) {
    let largest_child_index;
    //   declaring variable
    while (2 * index < this.next_index) {
        //   index of the left child of the node is 2 * index
        //   therefore this condition is satisfied as long as there
        //   at least one child of the node with this index
        //   if there are no children, then the node is already
        //   on the correct place
```

```
        if (2 * index + 1 < this.next_index) {
            //    if there is right child
            let left_item = this.heap[2 * index];
            let right_item = this.heap[2 * index + 1];

            if (this.compare(right_item, left_item) > 0) {
                //   if the right child is larger
                largest_child_index = 2 * index + 1;
            }
            else {
                largest_child_index = 2 * index;
            }
        }
        else {
            largest_child_index = 2 * index;
        }
        //   previous block of code finds the largest child of
        //   the current node

        let parent_item = this.heap[index];
        let largest_child_item = this.heap[largest_child_index];

        if (this.compare(largest_child_item, parent_item) > 0) {
            //   if the largest child is larger than its parent
            //   perform a swap
            this.heap[largest_child_index] = parent_item;
            this.heap[index] = largest_child_item;

            index = largest_child_index;
            //   the next node to consider is now at this index
        }
        else {
            break;
        }
        //   exit the loop otherwise
    }
}

pop() {
    //   if the queue is empty, return null
    if (this.next_index == 1)
        return null;

    let item = this.heap[1];
    //   required item temporarily stored

    this.heap[1] = this.heap[this.next_index - 1];
    //   place the last elememt at the top
```

```
    this.next_index -= 1;
    //   decrement the next_index attribute

    this.sink(1);
    //   make the new first top element of the queue to go down
    //   so the property is maintained
    return item;
}
```

The only change to the code is a “let” keyword for variable declaration.

Before implementing the “peek()” method, I decided to add “isEmpty()” method that will check if the index of the next element to add is 1, which corresponds to the state of the queue being empty. I will replace “`this.next_index == 1`” with this method and use it for the “peek()” method. It does not simplify code much, however it assists readability as the person reading the code will straight away understand that this block of the code handles the case of an empty queue. Other classes, like ParticleProjectionSimulation will also use it to avoid popping from an empty queue.

```
peek() {
    if (this.isEmpty())
        return null;

    return this.heap[1];
}
```

4.3.2.4 Testing Stage 2

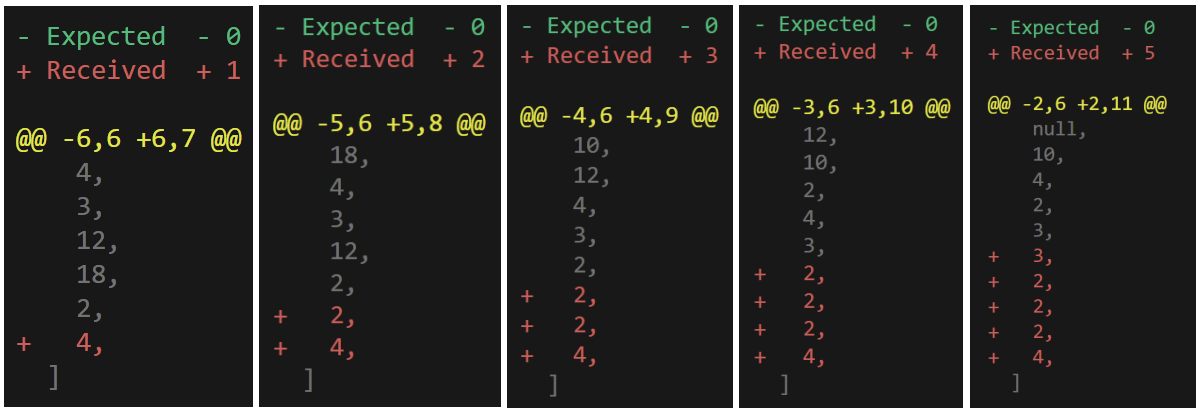
As with stage 1, first the priority queue is initialised using the “initializeQueue()” function. At every test, an element is popped from the queue and compared against expected output, following by the peek method call and heap check:

```
describe("Stage 2 testing", () => {
    let pq = initializeQueue();

    test("test (1)", () => {
        expect(pq.pop()).toEqual(30);
        expect(pq.heap).toEqual([null, 25, 10, 20, 4, 3, 12, 18, 2]);
        expect(pq.peek()).toEqual(25);
    });
```

Run the test:

```
Stage 2 testing
  × test (1) (2 ms)
  × test (2) (1 ms)
  × test (3)
  × test (4)
  × test (5) (1 ms)
  × test (6) (1 ms)
  × test (7) (1 ms)
  × test (8)
  × test (9) (1 ms)
  × test (10) (1 ms)
```

All of the tests have failed. Looking at feedback:



According to the feedback, the heap consistently has too many elements. It gave me a hint on the fact that I did not implement deletion of the element once it is popped.
To fix that, I added one line of code that deletes the last element from the heap after it is swapped with the first one.

```
this.heap[1] = this.heap[this.next_index - 1];
//   place the last elememt at the top
this.next_index -= 1;
//   decrement the next_index attribute

this.heap.splice(this.next_index, 1)
//   delete previously last element
```

Now all the tests are passed:

```
Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
```

4.4 Visual representation

Success criteria: object storage, update and **drawing** system.

4.4.1 Basic View set-up

Starting by implementing a constructor of the ViewSim class:

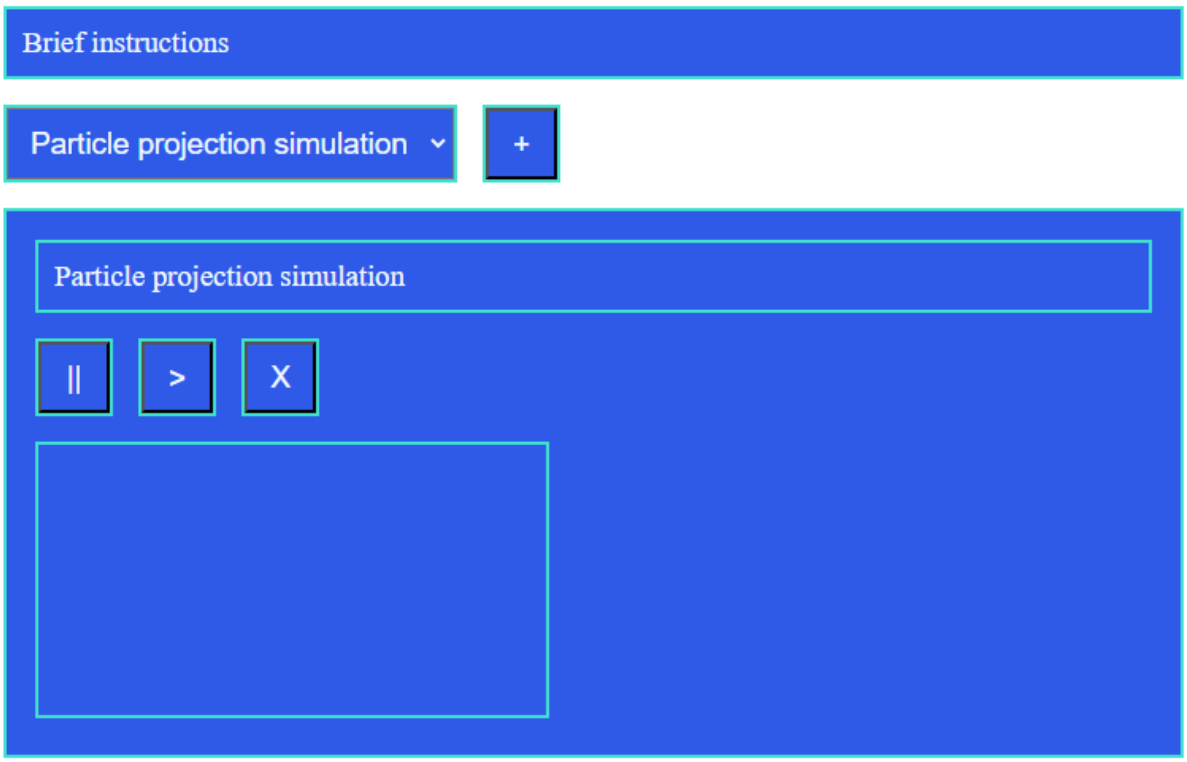
```
constructor(id) {
    this.id = id;
    this.body_list = [];
    this.scale = 1;
    // initialising set of attributes to their initial values

    this.canvas = document.createElement("canvas");
    //   creating the canvas html element
    this.translation = new Vector(0, this.canvas.height);
    //   initial translation to the left bottom corner

    let sim_area = document.getElementById("sim_area" + id);

    //   line break before adding the simulation
    sim_area.appendChild(this.canvas);
```

```
    this.ctx = this.canvas.getContext("2d");
    //   getting access to "context" of the canvas for drawing purposes
}
```

Trying to add simulation with this code results in a following error:

```
▶ Uncaught TypeError: Cannot read properties of null (reading 'appendChild')
    at new ViewSim (ViewSim.js:14:49)
    at Controller.addSim (Controller.js:112:20)
    at Controller.add_sim_button.onclick (Controller.js:54:18)
```

Investigating the problem, I found out that the error occurs because the sim_area element is not present on the html page at the time ViewSim is instantiated. To solve this, I moved the code inside the “addSim” method of the constructor such that sim_area and buttons with the title are added on the page before initialisation of the ViewSim.



Before I moved on with implementation of the ViewSim, I decided to change style of the button, so they lie on the same level as the title and align to the right. First, I put them together in a html tag and assigned a title to the “title” CSS class and buttons to the “sim_area_button” class.

```
terminate_sim_button.classList.add("sim_area_button");
pause_sim_button.classList.add("sim_area_button");
continue_sim_button.classList.add("sim_area_button");
title.classList.add("title");

let container = document.createElement("div");
container.style.display = "flex";
```

```
container.appendChild(title);
container.appendChild(pause_sim_button);
container.appendChild(continue_sim_button);
container.appendChild(terminate_sim_button);

sim_area.appendChild(container);
```

The classList.add() method assigns the class to the element and appendChild() method appends one element inside the other one. The display property of the style attribute of the container element is set to “flex” so it fits all elements in one line. The title is assigned a margin-right value “auto”, so it automatically extends its margin to the required width to push the buttons to the right.

```
.title {
    margin-right: auto;
}
body canvas {
    width: 800px;
    height: 450px;
    background-color: whitesmoke;
}
```

Applying some additional styling for the canvas, making background colour slightly greyer.

I decided to temporarily leave functionality of layers out of the project as it has little to no usage. Adding and deleting a body will be performed as simple appending to the end of the list and linear search for deletion.

```
addBody(body) {
    this.body_list.push(body);
}

deleteBody(body) {
    for (let i = 0; i < this.body_list.length; i++) {
        if (this.body_list[i] == body)
            this.body_list.splice(i, 1);
    }
}
```

The redraw method by pseudocode:

```
redraw() {
    this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
    //   clearing canvas area from the previous frame
    for (let i = 0; i < this.body_list.length; i++) {
        this.body_list[i].redraw(this);
        //   redrawing each body separately, if bodies are stored by layers,
        //   then first bodies will be below later bodies
    }
}
```

The canvas is cleared straight after the redraw method is called, then iterate through the list of bodies, and redraw each body separately.

4.4.2 Body and ViewBody superclasses

These classes are in place to save time for rewriting constructors and common getters and setters for each Body and ViewBody (child classes of the Body superclass will extend its constructor more).

```
class ViewBody {
    constructor(body, color) {
        this.body = body;
        this.color = color;
    }
}
```

```
class Body {
    constructor(id) {
        this.position = new Vector();
        this.velocity = new Vector();

        this.id = id;
    }
}
```

They simply assign values to attributes.

4.4.3 Point Mass

4.4.3.1 Implementation

The goal at this stage is to properly implement and test methods for drawing bodies.

```
class ViewPointMass extends ViewBody {
    constructor(body, color) {
        super(body, color);
        //   body is an instance of PointMass in this case
    }

    redraw(view) {
        view.drawPoint(this.body.getPosition(), this.color);
        //   the position of the point mass is its centre
    }
}
```

The ViewPointMass class will call the “drawPoint()” method of the view object. This weird way of drawing objects abstracts implementation of bodies from what shapes they represent when redrawn. For example, the drawPoint() method of the View object is only aware that the point is to be drawn at coordinates “center” whereas the ViewPointMass retrieves these coordinates as the position of the PointMass body.

```
drawPoint(center /* American spelling */, color) {
    let translated_center = center.reflectedInX().added(this.translation);

    this.ctx.beginPath();
    this.ctx.arc(
        translated_center.getX(),
        translated_center.getY(),
        10 /* radius */,
        0, 2 * Math.PI /* whole circle */
    )
    this.ctx.fillStyle = color;
    this.ctx.fill();
    this.ctx.closePath();
}
```

The point shape is achieved with the “arc()” method of the canvas context. It draws an arc of a circle with a given radius and centre coordinates. In the case of a full circle, the arc starts at 0 radians and goes up to 2π radians. Then the area is filled with required colour and the path is closed.

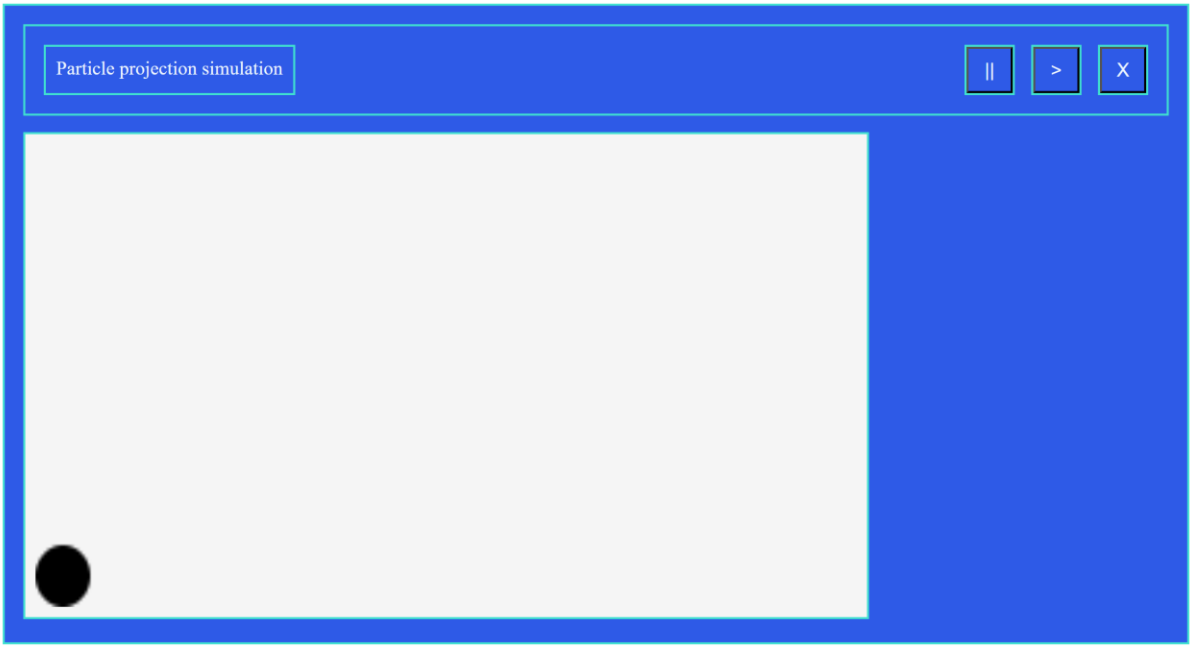
4.4.3.2 Testing

First write the testing method that must result in a circle at the bottom left corner.

```
test() {
    let body = new PointMass(
        new Vector(10,10),
        new Vector(),
        new Vector(),
        0,
        0
    );
```

```
    let view_body = new ViewPointMass(body, "black")
    this.addBody(view_body);
}
```

The result achieved:



Two issues encountered: 1) the circle is too big and low resolution for a 10-pixel radius circle and 2) the circle is slightly offset from the position it must be in (touching left and bottom borders).

As it turns out, the first one is a result of the fact that width and height of the canvas is specified with CSS code. It forces the image to be scaled during rendering to fit the styled size making the image distorted. It is fixed by setting dimensions of the canvas directly using JavaScript:

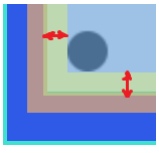
Before:

```
body canvas {
    width: 800px;
    height: 450px;
    background-color: whitesmoke;
}
```

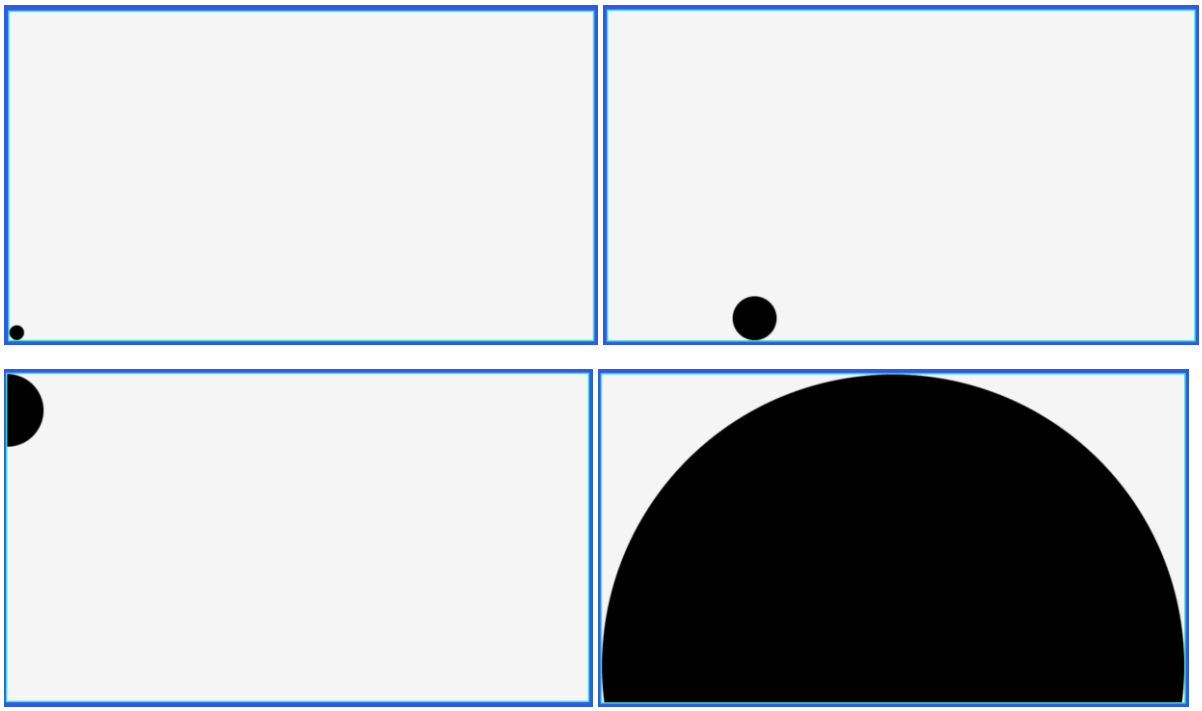
After:

```
this.canvas = document.createE
this.canvas.width = 800;
this.canvas.height = 450;
//   creating the canvas html
```

The second issue occurs because of the padding of 10 pixels that is applied to all descendent elements of the body tag. To fix that, I simply overriden the padding property of the canvas to set it to 0.



Results of all the tests described in design (testing strategy subsection at the end of the section) are presented below:



Works perfectly fine. The radius of a point I will leave for now will be 5 pixels.

4.4.4 Polygon

4.4.4.1 Implementation

The “drawPolygon()” method of the ViewSim class will be invoked by the Polygon objects that will calculate the list of verticies in simulation space converting them from their local space.

```
drawPolygon(vertices, color) {
    this.ctx.beginPath();
    this.ctx.moveTo(vertices[0].getX(), vertices[1].getY());
    //   start path at the first vertex
    for (let vertex of vertices) {
        let translated_vertex = vertex.reflectedInX().added(this.translation);
        //   translate to canvas coordinates
        this.ctx.lineTo(translated_vertex.getX(), translated_vertex.getY())
        //   line to the next vertex
    }
    //   move to each vertex in order
    this.ctx.closePath();

    this.ctx.fillStyle = color;
    this.ctx.fill();
}
```

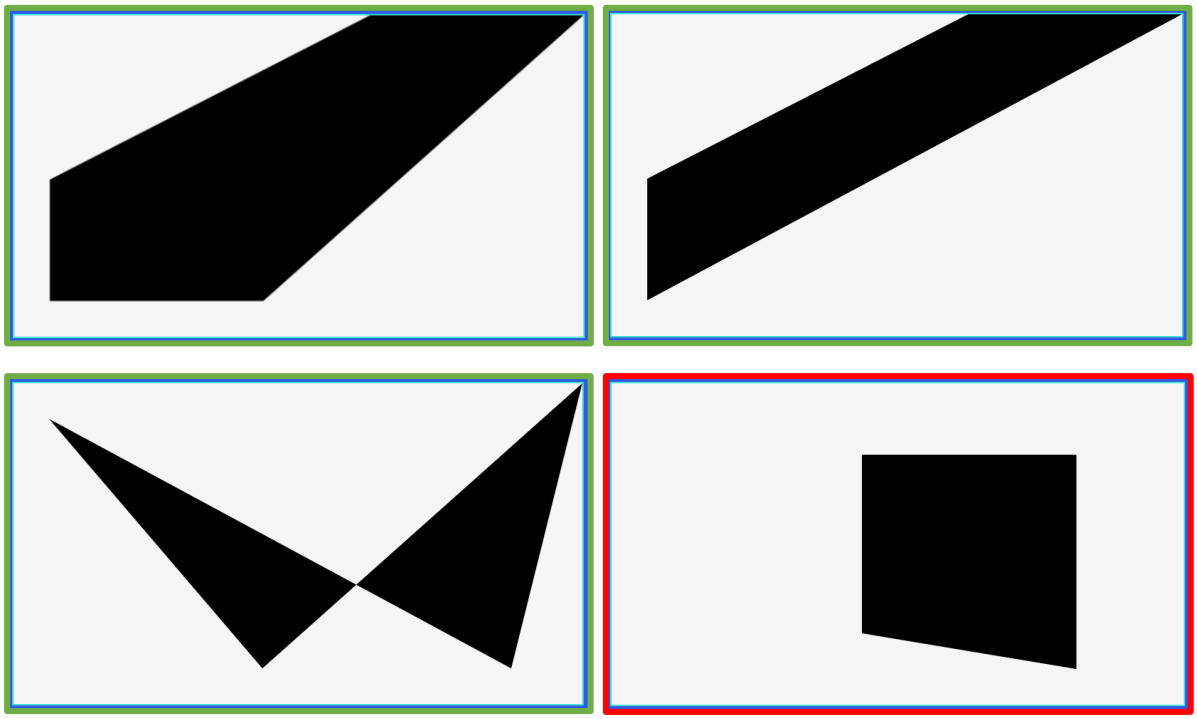
The method must accomplish what has been described.

4.4.4.2 Testing

The testing method will be temporarily put at the end of the redraw function. It will define the list of vertices and call “drawPolygon()” method on it with the black colour. For the first test the list of vertices is [(50, 50), (350, 50), (800, 450), (500, 450), (50, 220)] and respective test method:

```
test() {
    let vertices = [
        new Vector(50, 50),
        new Vector(350, 50),
        new Vector(800, 450),
        new Vector(500, 450),
        new Vector(50, 220)
    ];
    this.drawPolygon(vertices, "black");
}
```

Other “drawPolygon()” will differ only by the set of verticies used.
The results for the 4 tests described in design are:



All tests, except the last one, were successful. It seems that the path starts at the wrong point. I quickly figured out that the problem is that the “cursor” is moved to the first vertex. Swapping two lines must solve the issue:

```
drawPolygon(vertices, color) {
    this.ctx.beginPath();
    this.ctx.moveTo(vertices[0].getX(), vertices[1].getY());
    //   start path at the first vertex
```

->

```
drawPolygon(vertices, color) {
    this.ctx.moveTo(vertices[0].getX(), vertices[1].getY());
    this.ctx.beginPath();
    //   start path at the first vertex
```

So now the test is passed successfully.

4.4.5 Scaling and Optimisations

At this stage I realised that I have not been applying the scaling to any of the drawing methods, so this section will be dedicated to this and making the code neater by adding some auxiliary methods to ViewSim. For now, scaling will represent multiplication of the scale factor (scale variable of the ViewSim) by the position vector. When it comes to zooming in or out, not only scale factor will need to be adjusted, but the translation vector as well.
Because of the way scale factor will be applied, it would essentially shrink or extend the image of the simulation relative to its origin.

Starting with the conversion of vectors and constants to canvas coordinate system:

```
//   converting coordinates from simulation space
toCanvas(vector) {
    let new_vector = vector.reflectedInX();
    //   reflecting the vector
    new_vector.multiply(this.scale);
    //   scaling relative to the origin
    new_vector.add(this.translation);
    //   translating the vector to a new position
    return new_vector;
}

scaled(value) {
    return value * this.scale;
}
```

The first method will be applied to all coordinates / vectors by reflecting, scaling and translating them, so the image of the simulation correctly maps on the screen. The second method will scale all non-vector values, such as radius of a circle (disc). The radius does not need to be translated or reflected, however, scaling still applies to it. So now “drawPoint()” and “drawPolygon()” methods can use these methods instead of converting coordinates in multiple lines.

```
Before
let translated_center = center.reflectedInX().added(this.translation);
After
let translated_center = this.toCanvas(center);
```

```
Before
let translated_vertex = vertex.reflectedInX().added(this.translation);
After
let translated_vertex = this.toCanvas(vertex);
```

The other helpful optimisation to make involves more concise use of vectors. JavaScript provides a spread operator that can be used to expand iterable object inside a specific receiver. When some function (call it “f(a, b, c)”) needs to receive 3 parameters. Say they are stored in an array called “arr”. One of the ways to pass the parameters is f(arr[0], arr[1], arr[2]). With JavaScript, it can be accomplished with f(…arr), if the length of array matches the number of parameters.

Providing Vector class with an iterable interface can simplify the way vector x and y components are passed to drawing functions. Without the spread operator, the x and y components can only be retrieved using appropriate getters, whereas using "…" operator it can be accomplished in much less code and much more concise.

A way it is accomplished is by specifying a symbol iterator of the Vector class as a generator function.

```
//   Allows to easily spread vector coordinates using
//   spread operator
[Symbol.iterator] = function* () {
    yield this.x;
    yield this.y;
}
```

Once the program tries to access the iterator of some object, the iterator will first yield x and then y.

So now, following code can be changed:

```
this.ctx.arc(
    translated_center.getX(),
    translated_center.getY(),
    5 /* radius */,
    0, 2 * Math.PI /* whole circle */
)
```

->

```
this.ctx.arc(
    ...translated_center,
    5 /* radius */,
    0, 2 * Math.PI /* whole circle */
)
this.ctx.closePath();
```

```
for (let vertex of vertices) {
    let translated_vertex = this.toCanvas(vertex);
    //   translate to canvas coordinates
    this.ctx.lineTo(translated_vertex.getX(), translated_vertex.getY())
    //   line to the next vertex
}
```

to ->

```
for (let vertex of vertices) {
    let translated_vertex = this.toCanvas(vertex);
    //   translate to canvas coordinates
    this.ctx.lineTo(...translated_vertex);
    //   line to the next vertex
}
```

4.4.6 Disc

For the disc drawing operation, I decided to make a deviation from designed plans. It is possible to make use of reusable components by slightly modifying the "drawPoint()" method, so the point and disc can be drawn using the same drawing routines with very little code.

I will define methods in a following way:

"drawCircle(center, radius, color)" – draws a circle of radius "radius" (scaled before using) at coordinates "center" (translated and scaled before using) filled with colour "color".

"drawPoint(center, color)" – this function will calculate the radius required to draw the point such that no matter what scale will be applied, the radius will remain constant.

```
drawCircle(center, radius, color) {
    let translated_center = this.toCanvas(center);

    this.ctx.beginPath();
    this.ctx.arc(
        ...translated_center,
        this.scaled(radius) /* radius */,
        0, 2 * Math.PI /* whole circle */
    )
    this.ctx.closePath();

    this.ctx.fillStyle = color;
    this.ctx.fill();
    //   fill the shape
}
```

Entirely the same as previous “drawPoint()” method, except now radius is scaled.

```
drawPoint(center, color) {
    let unscaled_radius = 5 / this.scale;
    this.drawCircle(center, unscaled_radius, color);
}
```

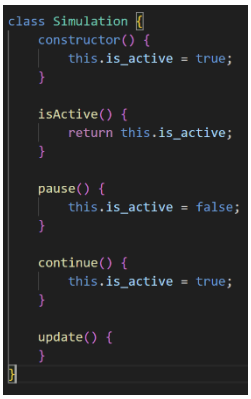
By dividing the constant 5 by the scale, the “drawCircle()” method will receive such value for radius that when it is scaled, the constant 5 is received back.

The “drawCircle()” method (previously called “drawPoint()”) was already tested here.

4.5 Particle Projection Simulation

Success criteria: object **storage, update** and drawing system; event system.

I will start this iteration with development of the Particle Projection Simulation class (and a bit of Simulation class), which will consist of rewriting JavaScript code from the designed pseudocode. Straight after, I will tackle the Event class and TimeEvent which are useful to test both events and part of the update method that deals with events.



Starting with simulation class, the basic set of methods that are shared across all simulations is simple. Other implementations will inherit (using “extend” JavaScript keyword) these method and attributes initialised in the constructor.

```
class ParticleProjectionSim extends Simulation {
    constructor() {
        super();
        this.event_queue = new PriorityQueue(Event.compare);
        this.body_list = [];
    }

    update(dt) {
        //   parameter dt denotes the change in time from the previous
        //   frame
```

```
        this.time += dt

        let event = this.event_queue.peek()
        //   peek the first event from the queue

        if (event.getTime() < this.time) {
            //   if the time the event occurs is greater than the
            //   current time the simulation is at
            event.execute();
            this.pause();
            //   execute the procedure defined when event was created
            this.time = event.getTime();
            //   set the time of the simulation to the moment when
            //   event has occured
            this.event_queue.pop();
            //   remove event from the queue
        }

        for (let i = 0; i < this.body_list.length; i++) {
            body_list[i].update(this.time);
        }
    }
}
```

The ParticleProjectionSim follows the pseudocode. The “super()” function calls the constructor of the parent class, event_queue and body_list are then initialised. The event priority queue must receive Event.compare function when instantiated. The update method, as described in design, must check if any events have occurred by the current time and execute event if so. I also decided that the simulation will be stopped if there is such event. It then proceeds to update each body of the simulation.

Note: body_list instantiation can be moved to the parent Simulation class.

4.5.1 Events (TimeEvent)

4.5.1.1 Implementation

First the parent Event class will have most methods, the only difference between the three types of events is the way the time is set. For the TimeEvent class, the time is directly set to occurs_at attribute, whereas other methods calculate it using given parameters.

The constructor of the Event class gets the body that the event is considering, reference to Input / Output handler for an “executeEvent()” method and declares “occurst_at” attribute.
The “getTime()” method is a getter for an occurs_at attribute.
“isValid()” method checks if the “occurs_at” attribute is greater than zero.

```
execute() {
    this.io_handler.executeEvent(this.body, this.occurs_at, this.event_id);
    //   provides enough details for an io handler to produce
    //   relevant output on the screen
}
```

An execute method will pass some parameters for an io_handler to generate output.

Compare static method that is passed to the Priority Queue constructor:

```
static compare(event1, event2) {
    return event2.occurs_at - event1.occurs_at;
}
```

It will return a positive number whenever the second event occurs later, in this way, prioritising the earlier event.

The first implementation of the Event is TimeEvent which adds only one new method “setTime()”.

```
import Event from "./Event.js";


class TimeEvent extends Event {
    constructor(body, io_handler, event_id) {
        super(body, io_handler, event_id);
    }

    setTime(time) {
        this.occurs_at = time;
    }
}

export {TimeEvent as default};
```

4.5.1.2 Testing

This part of tests will examine if the system of events was integrated correctly. At the end of the constructor for ParticleProjectionSimulation call “test()” method. This method must create 4 instances of TimeEvent class and set there time to 5, 10, 12 and 20. Right afterwards, output current time using “new Date().getTime()”.

```
test() {
    for (let time of [5, 10, 12, 20]) {
        let event = new TimeEvent(null, null, 0);
        event.setTime(time*1000);
        //   time multiplied by 1000 to convert it to miliseconds
        this.event_queue.push(event);
    }
    console.log(new Date().getTime());
}
```

Also, “event.execute()” method must temporarily be deactivated as there are no Input Output handlers implemented yet.

Running the program now:

```
1707067465068
ParticleProjectionIO instantiated
Simulation time now: 5002.7
Absolute time now: 1707067470066
```

Calculating the time difference: 4998 milliseconds, indeed nearly 5 seconds. Continue simulation:

```
Time start: 1707067534299
Simulation time now: 10008.600000000006
Absolute time now: 1707067539305
```

5006 milliseconds: exactly the time between first (5s) and second (10s) events.

```
Time start: 1707067683245
Simulation time now: 12002.5
Absolute time now: 1707067685242
```

1997 milliseconds: 12s – 10s

```
Time start: 1707067748923
Simulation time now: 20002.199999999997
Absolute time now: 1707067756911
```

7988 milliseconds: 20s – 12s

Events are retrieved correctly. However, if simulation is continued further, an error occurs:

```
▶ Uncaught TypeError: Cannot read properties of null (reading 'getTime')    ParticleProjectionSim.js:23
```

An error occurs because at this point event_queue is empty and simulation retrieves null for an event, attempting to use event's "getTime()" method. An update method must check if the event queue is empty before attempting to use "getTime()" method:

```
if (!this.event_queue.isEmpty() && event.getTime() < this.time) {
    //   if the time the event occurs is greater than the
    //   current time the simulation is at
    event.execute();
```

the "event.getTime() < this.time" will be reached only if event queue is not empty.

4.5.2 Point Mass

4.5.2.1 Implementation

The point mass constructor will simply assign values to initial_position, initial_velocity, accelearation and created_at attributes.

The update method was extensively described in design, rewriting the pseudocode:

```
update(time) {
    let t = (time - this.created_at) / 1000;
    //   calculate time from the creation of the object

    let ds = this.initial_velocity.multiplied(t);
    //   initialising ds variable that represents change in position
    //   and calculating the first part of the equation

    ds.add(this.acceleration.multiplied(t * t / 2));
    //   adding the change caused by acceleration

    this.position = this.initial_position.added(ds);
    //   updating position
```

```
    let dv = this.acceleration.multiplied(t);
    //   initialising change in velocity variable
    this.velocity = this.initial_velocity.added(dv);
    //   updating velocity
}
```

The variable t is divided by 1000 to convert it to seconds, because both “time” parameter and “created_at” attribute are in milliseconds, whereas all calculations use seconds.

4.5.2.2 Testing

To carry out testing described in design, new PointMass objects will be added from the controller class at the end of the “addSim()” method. In this way, objects for both PointMass and ViewPointMass can be integrated, so the visual output can also be received.
Start with the first object that is added straight when new Simulation is created with the click of a button.

```
test(sim, view) {
    let particle1 = new PointMass(
        new Vector(100, 100),
        new Vector(20, 100),
        new Vector(0, -10),
        sim.getTime()
    );
    let view_particle1 = new ViewPointMass(particle1, "black");
    sim.addBody(particle1);
    view.addBody(view_particle1);

    for (let time of [5, 10, 60]) {
        let event = new TimeEvent(particle1, null, null);
        event.setTime(time * 1000);
        sim.addEvent(event);
    }
}
```

First, new particle is created with the position of (100, 100) and velocity of (20, 100), then the view representation of it. They are added to sim and view of the respective simulation. To mock the state of this object, TimeEvents that occur in 5, 10 and 60 seconds are added. The PointMass object will output its state at the end of every update, so it can be compared to precalculated results once simulation stops at one of the events. Put this function to “addSim()” and pass corresponding arguments. Run simulation:

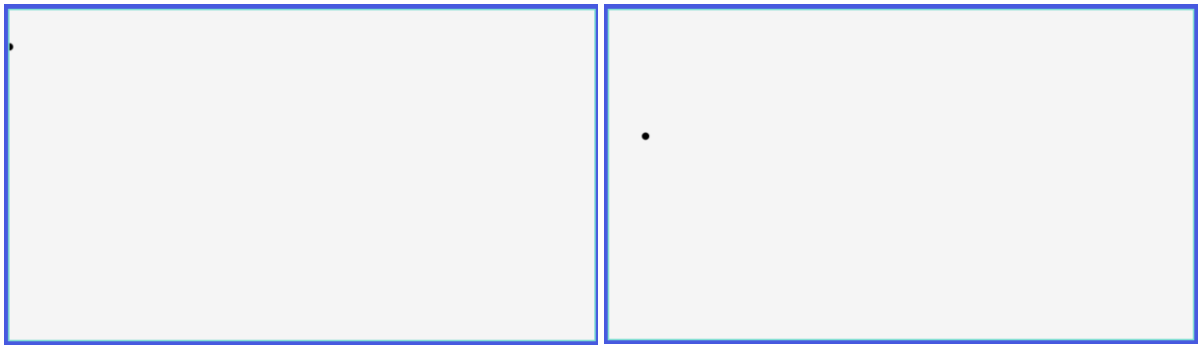
The particle moves by seemingly correct trajectory. Proceed to state checks:

Position/Velocity	Expected	Received	Successful
Position	(200, 475)	▶ Vector {x: 200, y: 475,	
Velocity	(20, 50)	▶ Vector {x: 20, y: 50,	
Position	(300, 600)	▶ Vector {x: 300, y: 600,	
Velocity	(20, 0)	▶ Vector {x: 20, y: 0,	
Position	(1300, -11900)	▶ Vector {x: 1300, y: -11900,	
Velocity	(20, -500)	▶ Vector {x: 20, y: -500,	

To test for the second particle, the “test()” method must be moved to the update method of the controller to check when the simulation first hits 5 seconds.

```
for (let i = 0; i < this.sim_list.length; i++) {
    if (this.sim_list[i].isActive()) {
        //   update and redraw every active simulation
        this.sim_list[i].update(dt);
        if (this.sim_list[i].getTime() >= 5000 && !this.stopped) {
            console.log("hit 5s");
            this.test(this.sim_list[i], this.view_list[i]);
            this.stopped = true;
            this.sim_list[i].pause();
        }
    }
    this.view_list[i].redraw();
}
```

The “stopped” variable is in place to execute the test method only once. The only changes to the test method are the initial state of the particle.



Position/Velocity	Expected	Received	Successful
Position	(50, 277.5)	▶ Vector {x: 50, y: 277.5,	
Velocity	(10, -49)	▶ Vector {x: 10, y: -49,	
Position	(100, -90)	▶ Vector {x: 100, y: -90.00000000000006,	
Velocity	(10, -98)	▶ Vector {x: 10, y: -98,	
Position	(600, -17240)	▶ Vector {x: 600, y: -17240,	

Velocity	(10, -588)	▶ Vector {x: 10, y: -588,	

To obtain screenshots, I made the PointMass output its position and velocity every time an update method is called.

All tests are successful.

4.5.3 Position and Velocity Events

Starting with basic set-up for both classes:

```
import Vector from "../../utility/Vector.js";
import Event from "./Event.js";


class VelocityEvent extends Event {
    constructor(body, io_handler, event_id) {
        super(body, io_handler, event_id);
    }

    calculateTime(value, axis) {

    }

    setXtime(value) {
        this.calculateTime(value, new Vector(1, 0));
    }

    setYtime(value) {
        this.calculateTime(value, new Vector(0, 1));
    }
}


export {VelocityEvent as default};
```

Both of them will extend (inherit) from the Event superclass. The “setXtime()” method will call “calculateTime()” method giving (1, 0) vector as the axis, so all vectors are projected on the x axis before performing any calculations. The “setYtime()” method on the other hand, passes (0, 1) vector, which is parallel to y-axis. PositionEvent and VelocityEvent classes will differ only be their implementation of the “calculateTime()” method.

4.5.3.1 PositionEvent Implementation

```
calculateTime(value, axis) {
    const a = this.body.getAcceleration().dot(axis) / 2;
    const b = this.body.getInitialVelocity().dot(axis);
    const c = this.body.getInitialPosition().dot(axis) - value;
    //   calculating coefficients for the quadratic equation
    //   from the equation of motion in a form of ax^2+bx+c=0

    if (a == 0 && b == 0) {
        this.occurs_at = -1;
```

```
        return;
        //   invalid if both a and b are zero
    }

    if (a == 0) {
        //   if only a is zero, then there might be a solution
        let x = (-1) * c / b;
        if (x <= 0)
            this.occurs_at = -1;
        else
            this.occurs_at = x;
        return
    }

    let D = b * b - 4 * a * c;
    //   calculating discriminant

    if (D < 0) {
        this.occurs_at = -1;
        return
        //   end function
    }

    let x1 = ((-1) * b - Math.sqrt(D)) / (2 * a);
    let x2 = ((-1) * b + Math.sqrt(D)) / (2 * a);
    //   two solutions where x1 <= x2

    if (x2 <= 0) {
        this.occurs_at = -1
        return
    }
    //   if the largest solution is negative, then the particle
    //   never reaches the value

    if (x1 <= 0)
        this.occurs_at = x2
    else
        this.occurs_at = x1
    //   if the first value is positive, then the particle first
    //   reaches the value after x1 seconds and the second time
    //   after x2 seconds
}
```

The method is derived from the formula for a particle motion under constant acceleration. To perform calculation, the method first calculates projections on the required axis, getting an equation of the form $ax^2 + bx + c = 0$ which can be mathematically solved with 2 solutions. The method basically implements this with additional checks for zero values of a, b, and c.

4.5.3.2 VelocityEvent Implementation

```
calculateTime(value, axis) {
    b = this.body.getAcceleration().dot(axis);
    c = this.body.getInitialVelocity().dot(axis) - value;
    //   calculating coefficients for the equation
    //   bx + c = 0

    if (b == 0) {
        this.occurs_at = -1;
        return
        //   invalid if b is zero
    }

    let x = (-1) * c / b;
    if (x <= 0)
        this.occurs_at = -1;
    else
        this.occurs_at = x;
}
```

Slightly simpler code, for velocity the equation is linear and therefore has only one possible root. The method basically calculates it with validations along the way.

4.5.3.3 Testing

Use a common strategy of injecting a “test()” method at the end of the “addSim()” method of a controller to add a particle to the simulation and its view. Instantiate and set the time of the event.

```
test(sim, view) {
    let particle = new PointMass(
        new Vector(),
        new Vector(13*Math.cos(Math.PI/6), 13*Math.sin(Math.PI/6)),
        new Vector(0, -9.8),
        sim.getTime()
    );
    let view_particle = new ViewPointMass(particle, "black");
    sim.addBody(particle);
    view.addBody(view_particle);

    let event = new PositionEvent(particle, null, null);
    event.setYtime(2);
    console.log(event.getTime());
}
```

The particle used is described in testing designs.

1) The first test involves calculating the time before the particle reaches 2m height which corresponds to the “setYtime(2)” call. The value outputted must be equal to 0.485059 to 6sf. The particle is at this height two times on its trajectory. First at 0.485059s and then at 0.841472s. Because the values are very small, the particle will be barely seen. To adjust for that, I have set a “scale” attribute of a ViewSim class to “70”.

The particle moves by expected trajectory with expected speed. Another time tests PointMass and ViewSim classes. The value received as an output for the event is:

```
0.8414719305772747                                    Controller.js:215
```

It corresponds to the second expected time value which suggests that x1 and x2 variables are not treated correctly. Make the event output both x1 and x2 once they are calculated:

```
0.8414719305772747 0.48505868166762306                PositionEvent.js:46
```

We see that they are swapped places, giving a hint on the fact they are not properly arranged, and I realised why. The way I thought about it is that the solution which is calculated with negative square root of the discriminant is smaller, however, in cases when "a" is negative, it is directly opposite. To arrange roots such that $x1 \leq x2$ I will add a check:

```
if (x2 < x1) {
    let temp = x2;
    x2 = x1;
    x1 = temp;
    //   swap x1 and x2 so x2 >= x1
}
```

The output now is:

```
0.48505868166762306 0.8414719305772747                PositionEvent.js:53
0.48505868166762306                                   Controller.js:215
```

which is what expected.

2) To test for event being created not at the start of the simulation, use the same method as for PointMass "update()" method, inject a test method to the controller's update loop, checking for when a simulation first reaches, for example, 5 seconds:

```
if (!this.stopped && this.sim_list[i].getTime() > 5000) {
    this.test(this.sim_list[i], this.view_list[i]);
    console.log("5 seconds reached");
    this.stopped = true;
}
```

inside the update loop.

```
0.48505868166762306 0.8414719305772747
0.48505868166762306
5 seconds reached
```

The event still outputs the same time as the event's logic does not account for the fact that particle might not have been created at the start of the simulation. It calculates time **since** the particle was created. To fix that, the event must add the time the particle was created to its "occurs_at" attribute:

```
if (x1 <= 0)
    this.occurs_at = x2 + this.body.createdAt();
else
    this.occurs_at = x1 + this.body.createdAt();
```

```
0.48505868166762306 0.8414719305772747
5002.985058681667
5 seconds reached
```

"x1" and "x2" are still computed correctly, however they are initially in seconds whereas other simulation objects deal in milliseconds. To account for that, the x1 and x2 values must be multiplied by 1000.

As the particle is not created exactly at 5 seconds, the value may vary by a couple of milliseconds. This is not an error, however, to test the event methods, I will be creating them exactly at the start of each simulation.

3) Other tests for PositionEvent with this setup:

setXtime(7) -> 0.621762s

```
621.7618283580584                                    Controller.js:216
```

Successful (the output is in milliseconds)

setXtime(12) -> 1.06588s

```
1065.877420042386                                    Controller.js:216
```

Successful

setYtime(5) -> -1

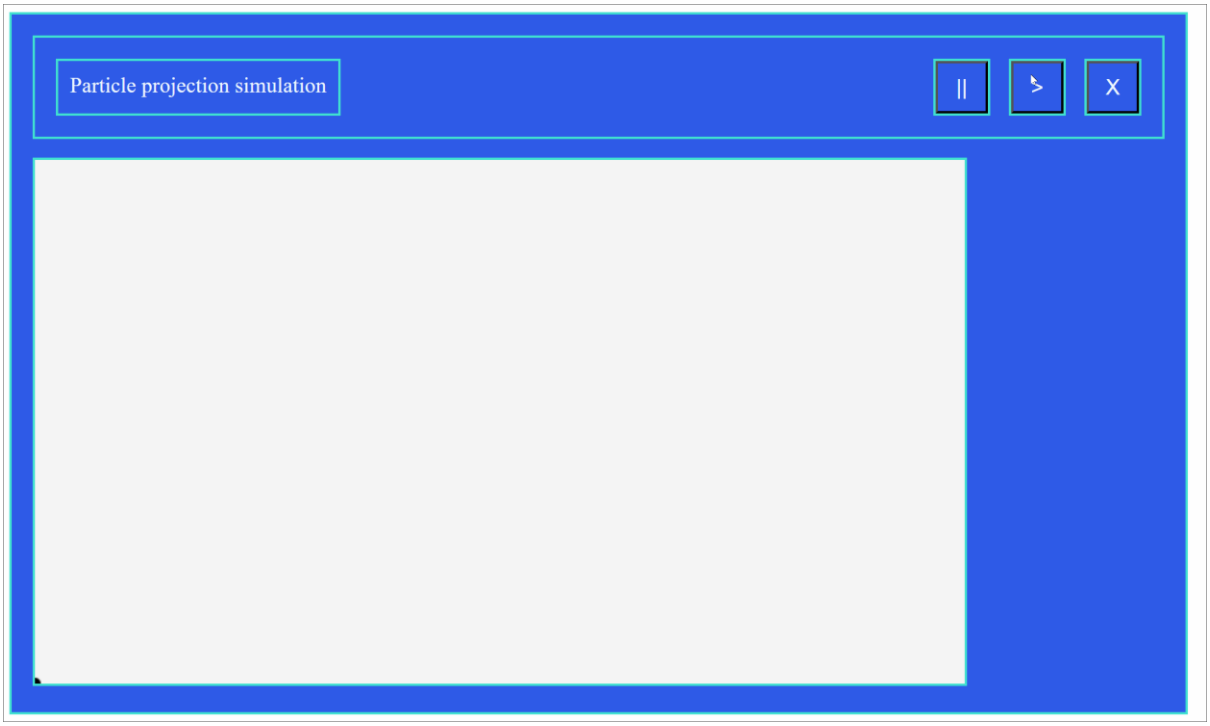
```
-1                                                   Controller.js:216
```

Successful

setYtime(0) -> -1

```
1326.5306122448976                                    Controller.js:216
```

In this case, the output is the time when the particle reaches value of 0 by y axis for the second time (the particle starts at 0 by y). Looking at this now, I consider the output of the second value to be preferable, so I will leave it like that. Looking at what happens to simulation if I add this event:



Simulation stops exactly when the particle has reached the ground. *I changed the scale to 40.

4) **<u>VelocityEvent tests for the first setup</u>**:

The same changes as to PositionEvent, must be applied to VelocityEvent:

```
else
    this.occurs_at = x * 1000 + this.body.createdAt();
```

Convert seconds to milliseconds and adjust to the time particle was created.

setXtime(13*cos(30^0)) -> -1

```
-1                                                    Controller.js:257
```

Successful

setXtime(2) -> -1

```
-1                                                    Controller.js:255
```

Successful

setXtime(-200) -> -1

```
-1                                                    Controller.js:255
```

Successful

setYtime(4) -> 0.255102s

```
255.10204081632642                                    Controller.js:255
```

Successful

setYtime(-10) -> 1.68367s

```
1683.6734693877552                                    Controller.js:255
```

Successful

setYtime(6.5) -> -1

```
-1                                                    Controller.js:255
```

Successful

setYtime(10) -> -1

```
-1                                                    Controller.js:255
```

Successful

5) Now a different setup is used, so the test method will be changed to create a particle with velocity of (5*cos(35^0), 5*sin(35^0)) and initial position (0, 3):

```
test(sim, view) {
    let particle = new PointMass(
        new Vector(0, 3),
        new Vector(5*Math.cos(35*Math.PI/180), 5*Math.sin(35*Math.PI/180)),
        new Vector(0, -9.8),
        sim.getTime()
    );
    let view_particle = new ViewPointMass(particle, "black");
```

Carrying out following test for PositionEvent:

setXtime(5) -> 1.22077s

```
1220.774588761456                                     Controller.js:255
```

Successful

setXtime(-1) -> -1

```
-1                                                    Controller.js:257
```

Successful

setYtime(3.1) -> first: ***0.0372382s***, second: 0.548044s

```
37.238191642510095                                    Controller.js:260
```

Successful

setYtime(3) -> 0.585282s (boundary data as the particle is already at 3 by y)

```
585.2820779092306                                     Controller.js:260
```

Successful

setYtime(-10) -> 1.94754s

```
1947.54284938635                                    Controller.js:260
```

Successful

setYtime(5) -> -1

```
-1                                                  Controller.js:260
```

Successful

VelocityEvent:
setYtime(2.5) -> 0.037539s

```
37.5389981382888                                    Controller.js:256
```

Successful

setYtime(-12) -> 1.51713s

```
1517.1308348729826                                  Controller.js:256
```

Successful

setYtime(0) -> 0.292641s

```
292.6410389546153                                   Controller.js:256
```

Successful

4.5.4 Test reflections

It is important that all units of measurements are clearly defined and consistent across classes. So, the Event class carries out all calculations using SI units, whereas Simulation and Controller classes use milliseconds. To maintain consistency, the Event class converts seconds to milliseconds before assigning the value to “occurs_at” attribute. PositionEvent and VelocityEvent have been amended during tests.

The TimeEvent must be defined in a following way: a TimeEvent is thought of as an occurred event if “***time***” seconds have elapsed from the moment it was created. (where “***time***” is a parameter of the “setTime()” method). Therefore, the “setTime()” method needs another parameter of current time of the simulation. The “***time***” will represent user input time in seconds. The “***current_time***” parameter will represent the “Simulation.getTime()” function call at the exact moment event is created.

To adapt to this definition, the “setTime()” method will be changed:

```
setTime(time, current_time) {
    this.occurs_at = time * 1000 + current_time;
}
```

4.6 Input Output Handler

Success criteria: nearly entire interface; time control system; all user inputs and validations (for particle / event system, etc.).

4.6.1 Creating I/O area

Starting with the constructor, the IO handler must create an io_area html element that will be put within the sim_area. This functionality is common across all IO handlers, so it will be implemented for an IOHandler superclass. The “initialize()” method will be left for IOHandler child classes to implement.

```
class IOHandler {
    constructor(id, sim, view) {
        this.id = id;
        this.sim = sim;
        this.view = view;

        this.createIOarea();
        this.initialize();
    }

    createIOarea() {

    }

    initialize() {

    }
}

export {IOHandler as default};
```

Starting implementation of the “createIOarea()” method, I realised that for the io_area to be located rightwards to the canvas, as it was designed, the sim_area must have “flex” value for its display style property, allowing multiple elements to be in one row. However, setting its property directly will force all elements inside the sim_area to be in one row:

To fix that, the sim_area must wrap elements once they do not have enough space in one row. To force a line break after the container for a title and buttons, the container’s width must be set to “100%”. For an io_area to take up rest of the space, its flex-grow property must be set to “1”:

```
sim_area.style.flexFlow = "row wrap"
container.style.width = "100%";
this.io_area.style.flexGrow = "1";
```

So now the io_area fits to the right of the canvas and will even wrap around if there is insufficient space.
The “createIOarea()” method is implemented in a following way:

```
createIOarea() {
    this.io_area = document.createElement("div");
    this.io_area.style.display = "flex";
    this.io_area.style.flexGrow = "1";
    //   create an io_area and specify its style

    let sim_area = document.getElementById("sim_area" + this.id);

    let p = document.createElement("p");
    p.innerHTML = "text";
    this.io_area.appendChild(p);
    //   test paragraph

    sim_area.appendChild(this.io_area);
    //   put io_area on the screen
}
```

It simply creates an io_area div element and puts it on the sim_area.

4.6.2 Reorganisation work

Before moving on with I/O Handlers, I would like to come back to the controller method to make reorganise code in more modular way. Now it comprises of four methods, “constructor()”, “update()”, “addSim()” and “terminateSim()”.

```
constructor() {
    this.sim_list = [];
    this.view_list = [];
    this.io_list = [];
    //   lists of simulations, views and input/output handlers, the
    //   simulation, its view and io handler are stored by the same
    //   index in the list

    this.sim_names = ["Particle projection simulation",
                      "General-purpose simulation"];
    this.sim_classes = ["ParticleProjection", "GeneralPurpose"];
    //   lists of simulation names and their class identifiers
    //   can be easily changed if a new simulation is added

    this.createSimulationsDropDown();

    this.next_id = 0;
    //   id of the next simulation

    this.createAddSimButton();

    this.startUpdateLoop();
}
```

The “constructor()” method can be split up into: createSimulationsDropDown(), createAddSimButton() and startUpdateLoop(). These methods will make the code more manageable, modular, and self-documenting. They do not change any functionality, but simply rearrange the code from the constructor method.

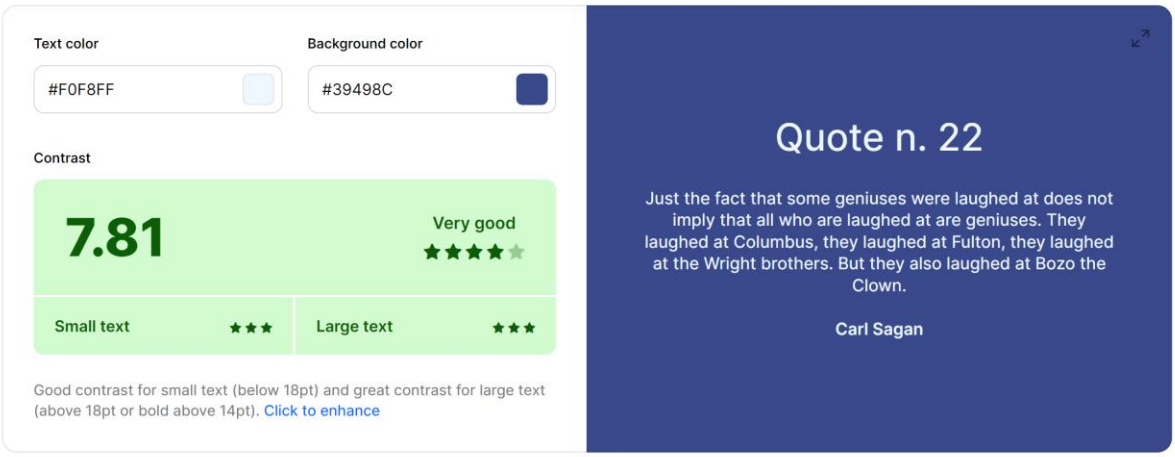
The update method can be left as it was.
The “addSim()” method can be rearranged in a following way: createSimArea(index) which will return the HTMLDivElement object of the sim_area of the same type as the class group selected by the user (represented by the index parameter); createSimHeader(sim, sim_area) which will be responsible for creating a container for title and sim buttons. It will then create the title and buttons with the following function: createSimButtons(sim, container, index) will accept a sim, container, and index as parameters to put them on the appropriate place on the screen.

4.6.3 Interface design changes

As a result of discussions of the interface designs and mock-ups created earlier, we came up with the following colour scheme for the interface:

Text with this colour scheme

This colour scheme was accepted by all stakeholders. Furthermore, it gets an excellent score for the background to text colour contrast ratio according to Web Content Accessibility Guidelines (WCAG). The main colours used are #B9BFE0 for the page background; #39498C for the background of all elements inside the body tag; #F0F8FF (also called aliceblue, nearly the same as white) for the text colour; #383D6D for the outline, this colour is a bit darker than the background colour, it makes a boundary between logically separate elements. For example, a sim_area will be surrounded with this outline.



Screenshot from the "Coolors" website (https://coolors.co/contrast-checker/f0f8ff-39498c)

The main font will be a "Lexend" with "sans-serif" as a backup. To link the font from google fonts, put following html code in the page head tag:

```
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link
href="https://fonts.googleapis.com/css2?family=Lexend:wght@100..900&display=sw
ap" rel="stylesheet">
```

Applying these changes with a CSS stylesheet:

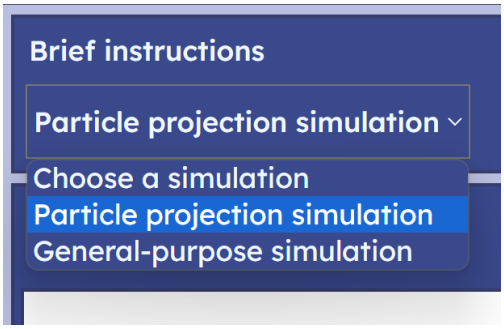
```
html {

    background-color: #b9bfe0;
    font-size: 13pt;
    font-family: "Lexend", sans-serif;
    font-optical-sizing: auto;
    font-weight: 500;
    font-style: normal;
}

body * {
    background-color: #39498C;
    color:aliceblue;
}
```

The main attributes of the interface pointed out by the stakeholders are:
- being easy to navigate;
- not overloaded with information (keeping it relevant to the main problem);
- intuitive to use.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

Moving on to **button design**.
Button's design itself is not as important as the action it represents. For the button styling, I used a preset from the getcssscan.com website (https://getcssscan.com/css-buttons-examples button 30):
Its main advantage is that it makes clear to the user when the button has been pressed and released. It also smoothly blends into website design.

The regular text elements **(<p> tag**) will be assigned margin and padding of 3 pixels and align-self property as "center" to add some symmetry.



For **drop-down menus** to look nice, they will need to inherit properties defined by the html tag (other elements do it by default). Other than that, the padding does not apply to them, so they will need to be adjusted accordingly to align with other text. The options will be left as they are to be distinguishable from other text. CSS code for that:

```
p {
    margin: 3px;
    padding: 3px;
    align-self: center;
}

select {
    font-family: inherit;
    font-size: inherit;
    font-weight: inherit;
    margin: 4px;
}
```

To apply styling for **containers that are outlined**, I will introduce a ".box" CSS class. By default, "div" elements will not take more space than its contents. It must be different for containers that intend to encapsulate logically different sections (for example sim_area and a header). They will be assigned a box class which will put appropriate margin, padding and outline around them:

```
.box {
    margin: 8px 3px 8px 3px;
    padding: 3px;
    outline: 2px solid #383D6D;
}
```

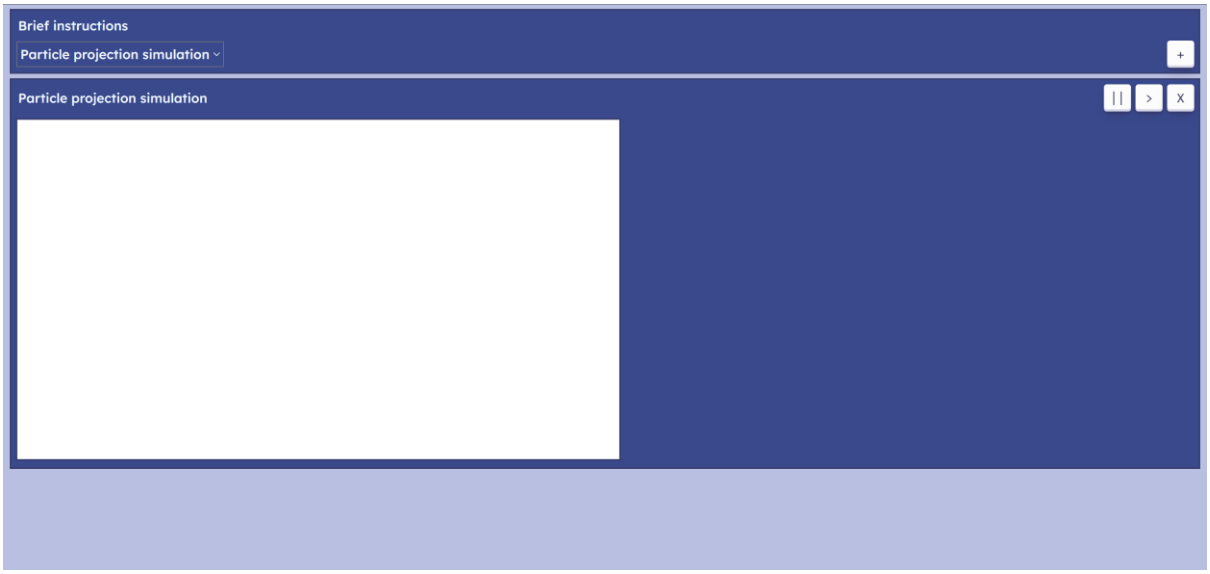
8px margin represents margin at the top and the bottom of the elements to create space in between boxes.

The **canvas** will share similar properties to the box class elements. It will not need any padding as it would create an unnecessary invisible border between the image and edges of the canvas.

```
canvas {
```

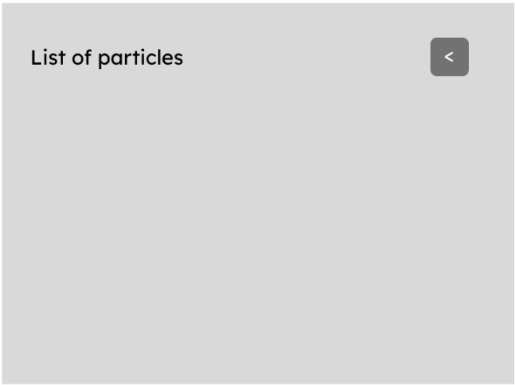
```
    background-color: white;
    outline: 2px solid #383D6D;
    margin: 8px 3px 8px 5px;
    padding: 0;
    /* space before io_area */
    align-self: "start";
    /* prevent the canvas from stretching */
}
```

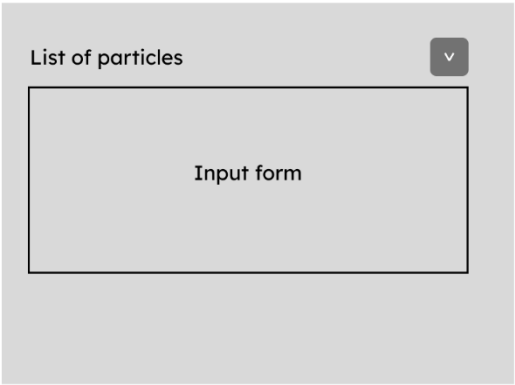
The final look for the page once a Particle Projection Simulation has been added:



*Buttons were pushed to the right by making the containers for titles flexible and making their right margin to automatically set its value to fill the space between.

The overall design was also discussed with the stakeholders and some advancements over previous designs were suggested. First, the input forms to add particles and events can be improved if they will appear/disappear on a click of the button.





It was suggested that input for position and velocity will be better in the form of a column vector.

The list of occurred events was redundant, and it was decided to omit it from the io_area.

4.6.4 Particle area

According to the process defined in design (Particle Projection IO section), I started with particle area.

```
createParticleArea() {
    let particle_area = document.createElement("div");
    particle_area.id = "particle_area" + this.id;

    this.io_area.appendChild(particle_area);

    let header_container = document.createElement("div");
    header_container.style.display = "flex";
    //   header container
    particle_area.appendChild(header_container);

    let title = document.createElement("p");
    title.style.marginRight = "auto";
    title.innerHTML = "List of particles";
    header_container.appendChild(title);
    //   first the title is created

    this.createParticleMenuButton(header_container);
    //   then the button is created

    this.createParticleInput(particle_area);
    //   create an input area

    let particle_list = document.createElement("div");
    particle_list.id = "particle_list" + this.id;
    particle_area.appendChild(particle_list);
    //   then the particle list area
}
```

The header container will not have outline. Its purpose is to allow the menu button to be to the right of the title by assigning “flex” to container’s display property and margin-right to “auto”.
The title is created next and added to the header of the particle_area.
Then the menu button.
And finally the particle list.
(The order elements are created follows their locations on the interface. Title first, the button is on the same row to the right. Input area below and list of particles below)

4.6.4.1 Menu button

```
createParticleMenuButton(header_container) {
    let button = document.createElement("button");
    button.innerHTML = "<";

    button.onclick = () => {
        let particle_input = document.getElementById("particle_input" +
this.id);
```

```
        if (particle_input.style.display == "none") {
            button.innerHTML = "v";
            particle_input.style.display = "block";
        }
        else {
            button.innerHTML = "<";
            particle_input.style.display = "none";
        }
    }

    header_container.appendChild(button);
}
```

The menu button is a simple html <button> element that is initially in “<” state, indicating that the input area is closed. When the user clicks on the button, the button changes its state between “<” and “v” depending on which action is to be performed. To close the input area, its display style property is set to “none”, meaning that it still exists on the page and does not need to be created every time again, while being invisible and not interactable for the user.

4.6.4.2 Particle input

```
createParticleInput(particle_area) {
    let particle_input = document.createElement("div");
    particle_input.classList.add("box");
    particle_input.style.display = "none";
    particle_input.id = "particle_input" + this.id;
    //   particle input area

    particle_area.appendChild(particle_input);

    let title = document.createElement("p");
    title.style.marginRight = "auto";
    title.id = "particle_input_title" + this.id;
    title.innerHTML = "New Particle " + this.next_particle_id;
    //   title for the area
    //   it must be updated every time new Particle is added

    let top_container = document.createElement("div");
    top_container.style.display = "flex";
    top_container.style.flexFlow = "wrap";
    top_container.appendChild(title);
    //   container for the title and inputs
    //   it separates them from the button

    particle_input.appendChild(top_container);

    this.createPositionInput(top_container);
    this.createVelocityInput(top_container);
    //   creating inputs for position and velocity
```

```
    this.createAddParticleButton(particle_input);
    //   button that adds new particles
}
```

This method starts by creating the input area and putting it on the particle area. The “box” class is assigned to outline the area. Initially, the menu is closed, so the display property is set to “none”. The id is assigned as the global name of the area + id of this particular simulation represented by “this.id”.
Create the title next, it shows to the user an id of the next particle to be added, therefore, it will have to be updated every time new particle is added successfully. It also has margin-right as “auto” to create space before inputs.
The top container separates the inputs from the button. Its display property is “flex”, so all the inputs are in one row.
The button is created at last.

Reviewing two methods for position and velocity inputs:

```
createPositionInput(top_container) {
    let position_container = document.createElement("div");
    position_container.style.display = "flex";
    //   a container for all position input

    let position_title = document.createElement("p");
    position_title.innerHTML = "Position =  ";

    let column = this.createColumn(
        "position_x_input" + this.id,
        "position_y_input" + this.id
    );
    //   column input

    position_container.appendChild(position_title);
    position_container.appendChild(column);

    top_container.appendChild(position_container);
}
```

All position and velocity input will be located within their dedicated containers of display “flex”. To create the column input, I decided to create an auxiliary method that takes as input two id values. First one for top row and second one for bottom row. It returns an HTML Node element that represents the column with input fields at the top and bottom with required ids. This column is then put on the position_container.
The “createVelocityInput()”differs only by the title (“Velocity = ” instead of “Position = ”) and the ids for input fields (“velocity_x_input” + this.id and “velocity_y_input”)

For the “createColumn()” method I decided to use MathML, Mathematical Markup Language for describing mathematical notations. It will help to create brackets around input fields to represent column vectors. Nowadays, it is supported by most popular browsers, even on mobile phones.

```
createColumn(top_id, bottom_id) {
    let column = document.createElement("div");
```

```
    column.innerHTML = `
    <math xmlns="http://www.w3.org/1998/Math/MathML">
        <mrow>
            <mo>(</mo>
            <mtable>
                <mtr>
                    <mtd><mi><input id="${top_id}"/></mi></mtd>
                </mtr>
                <mtr>
                    <mtd><mi><input id="${bottom_id}"/></mi></mtd>
                </mtr>
            </mtable>
            <mo>)</mo>
        </mrow>
    </math>
    `
    return column;
}
```

The method starts with creating another container for a column itself. Then the innerHTML of this container is manually changed to the <math> notation that defines the structure of the column. The ids are specified using JavaScript's template literals. The column is then returned. This same method can be used for output column vectors as well.

4.6.4.3 Adding a particle

```
createAddParticleButton(particle_input) {
    let container = document.createElement("div");
    container.style.display = "flex";
    //   flex container for a button to extend

    let add_button = document.createElement("button");
    add_button.innerHTML = "Add";
    add_button.style.flexGrow = "1";
    //   allows button to take up all space
    add_button.onclick = () => {
        this.addParticle();
    }

    container.appendChild(add_button);

    particle_input.appendChild(container);
}
```

The button will be located under the input and the title. To make it look better, I will make it take up the whole space under the inputs by putting it inside a flex container and making its flex-grow attribute "1". The main functionality of the button is inside an "addParticle()" method.

Moving on to the "addParticle()" method, its main purpose is to validate all the input provided by the user and output an error message if any of the input is not of required format or range. The error message logic will be like the input menu's one. Whenever an error occurs, the message's display property will be set to "block" and "none" when the parameters are entered correctly.

To implement an error message, I must come back to the “createParticleInput()” method and create an html element for an error message text:

```
let error_message = document.createElement("p");
error_message.id = "particle_error_message" + this.id;
error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
error_message.style.display = "block";
error_message.classList.add("error_message");
particle_input.appendChild(error_message);
```

Initially invisible.

To simplify code further, I will define all errors as constants in the “initialize()” method.

- INCORRECT_TYPE_ERROR = “Input values must be real numbers with a decimal part separated by the dot (not comma) or in the format Xe+/-Y for $X * 10^{+/-Y}$”
  (Use MathML to create the math notation)
- OUT_OF_RANGE_VALUES = “Input values must be between -10000 and 10000”
- INVALID_EVENT_ERROR = “Such event will never occur”

The MathML script developed for the incorrect type error:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
    <mrow>
        <mi>X</mi>
        <mo>*</mo>
        <msup>
            <mi>10</mi>
            <mrow>
                <mo>+/-</mo>
                <mi>Y</mi>
            </mrow>
        </msup>
    </mrow>
</math>
```

Simply recreates $X * 10^{+/-Y}$ as a maths expression.

“addParticle()” routine itself:

```
addParticle() {
    let position_x = parseFloat(document.getElementById("position_x_input" +
this.id).value);
    let position_y = parseFloat(document.getElementById("position_y_input" +
this.id).value);
    let velocity_x = parseFloat(document.getElementById("velocity_x_input" +
this.id).value);
    let velocity_y = parseFloat(document.getElementById("velocity_y_input" +
this.id).value);
    //   retrieve values from inputs

    let error_message = document.getElementById("particle_error_message" +
this.id);
```

```
    for (let value of [position_x, position_y, velocity_x, velocity_y]) {
        if (isNaN(value)) {
            error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
            error_message.style.display = "block";
            return;
        }

        if (value < -10000 || value > 10000) {
            error_message.innerHTML = this.OUT_OF_RANGE_VALUES;
            error_message.style.display = "block";
            return;
        }
    }
    //   validations

    let particle = new PointMass(
        new Vector(position_x, position_y),
        new Vector(velocity_x, velocity_y),
        new Vector(0, -9.8),
        this.sim.getTime(),
        this.next_particle_id++ //  get and increment at the same time
    );
    //   initialise new particle

    this.sim.addBody(particle);

    let view_particle = new ViewPointMass(particle, "black");
    this.view.addBody(view_particle);

    this.body_list.push(particle);
    //   add the particle to io_list's own body_list

    this.createParticleOutput(particle);
    //   create an output window for the particle

    this.resetParticleChoice();
    //   reset the select element for the events
    //   because new particle has been added

    let particle_input_title = document.getElementById("particle_input_title"
+ this.id);
    particle_input_title.innerHTML = "New Particle " + this.next_particle_id;
    //   update title for the next particle

    error_message.style.display = "none";
    //   hide error message when particle is added successfully
}
```

Comments describe actions in order. The “resetParticleChoice()” method refers to the particle drop-down menu as it would need to be updated every time a particle is updated or deleted. For now, it shall be left empty.

4.6.4.4 Particle output block

Generating an output block:

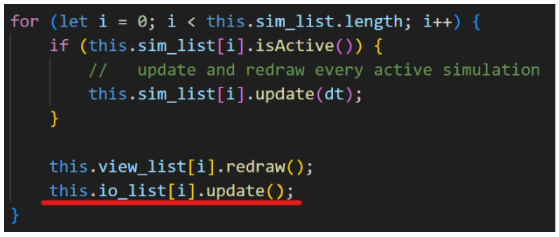
```
createParticleOutput(particle) {
    let particle_list = document.getElementById("particle_list" + this.id);

    let output_container = document.createElement("div");
    output_container.classList.add("box");
    //   container for an output block

    let top_container = document.createElement("div");
    top_container.style.display = "flex";
    //   container for a title and button

    let title = document.createElement("p");
    title.style.marginRight = "auto";
    title.innerHTML = "Particle " + particle.getId();
    top_container.appendChild(title);

    this.createPositionOutput(top_container, particle.getId());
    this.createVelocityOutput(top_container, particle.getId());
    //   creating outputs for each property

    output_container.appendChild(top_container);

    particle_list.appendChild(output_container);

    this.createRemoveButton(output_container, particle.getId());
    //   button to remove the particle after it is added
}
```

Again, elements are created one by one.

```
createPositionOutput(top_container, particle_id) {
    let position_output = document.createElement("div");
    position_output.style.display = "flex";
    //   container for an output block

    let position_title = document.createElement("p");
    position_title.innerHTML = "Position = ";

    let column = this.createColumn(
        "position_x_value" + particle_id + "_" + this.id,
        "position_y_value" + particle_id + "_" + this.id
    );
```

```
    position_output.appendChild(position_title);
    position_output.appendChild(column);

    top_container.appendChild(position_output);
}
```

The output is created in the same way as input, using "createColumn()" method. These will have to be updated when the particle gets updated. It requires an IO handler to also have an update method that will be invoked together with Simulation and ViewSim update methods in the controller class.




```
update() {
    for (let particle of this.body_list) {
        if (particle == null)
            continue;
        this.updatePositionOutput(particle);
        this.updateVelocityOutput(particle);
    }
}
```

The method will go over every element of the body list and update output blocks for every particle that is not null (deleted particles will be marked as null).

```
updatePositionOutput(particle) {
    let x_value = document.getElementById("position_x_value" +
particle.getId() + "_" + this.id);
    let y_value = document.getElementById("position_y_value" +
particle.getId() + "_" + this.id);
    //   retrieve html elements that represent output

    let position = particle.getPosition();

    x_value.value = Math.round(position.getX() * 100) / 100;
    y_value.value = Math.round(position.getY() * 100) / 100;
    //   assign values rounded to 2 dp
}
```

Assign current values, rounded to 2 decimal places, to the respective output elements.

```
createRemoveButton(output_container, particle_id) {
    let button = document.createElement("button");
    button.style.width = "100px";
    //   fixed small width to avoid accidental clicks
    button.innerHTML = "Remove";

    button.onclick = () => {
        let particle = this.body_list[particle_id];
        this.sim.deleteBody(particle);
        this.view.deleteBody(particle);
```

```
        this.body_list[particle_id] = null;
        //   remove particle from sim, view and io

        this.resetParticleChoice();
        //   reset a particle drop down

        output_container.remove();
        button.remove();
        //   remove particle from HTML
    }

    output_container.appendChild(button);
    //   add button on the page
}
```

Lastly, after all output routines are finished, the remove particle button is added, so the user can remove particle from the screen.

4.6.4.5 Tests and corrections

Now that all routines for the area are finished, observe how it all functions:

Problems encountered are
input elements are too large; error message is causing the io_area to wrap under the canvas; the MathML formula does not inherit style.
First, adjusting the styling:

```
mtd {
    padding: 2px;
}

math, mrow, mi, mo, msup {
    color: inherit;
    font-weight: 600;
```

```
}

input {
    font-size: 11pt;
    width: 60px;
    height: 16px;
    align-self: center;
    color: #36395A;
    background-color: aliceblue;
}
```

The colour and background colour are swapped, font size is made smaller than other text and width and height are fixed to specific size. The <mtd> tag is a column cell, I made its padding a little smaller so each cell occupies a bit less space. Other MathML elements are forced to inherit the font colour (red for formulas) and font-weight is set to 600.

To fix wrapping, the optimal solution I came up with is to put the canvas and io_area into one flex container that is unable to wrap while keeping the io_area able to wrap by rows. The sim_area itself will be kept as an element with block display to prevent io_area wrapping below the canvas.

To accommodate these changes:

```
createSimArea() {
    let sim_area = document.createElement("div");
    sim_area.id = "sim_area" + this.next_id.toString();
    //   assigning an identifier to the new simulation area

    sim_area.classList.add("box");
    sim_area.style.display = "flex";
    sim_area.style.flexFlow = "row wrap";
    sim_area.style.marginTop = "10px";
    sim_area.style.marginBottom = "10px";

    //   sim area styling
```

First remove these two lines that make sim_area's display flex.

```
this.createSimHeader(sim, sim_area, index);

let io_canvas_container = document.createElement("div");
io_canvas_container.style.display = "flex";
sim_area.appendChild(io_canvas_container);
//   container for an io area and canvas

let view = new ViewSim(this.next_id, io_canvas_container);
this.view_list.push(view);
let io = eval("new " + class_group + "IO(this.next_id, sim, view, io_canvas_container)");
this.io_list.push(io);
//   creating ViewSim and IOHandler objects

this.next_id++;
//   increment id for the next simulation;
```

Create io_canvas_container and pass it to ViewSim's and IOHandler's constructors. They will now put the canvas and io_area on this container instead of the sim_area directly, as it was before.

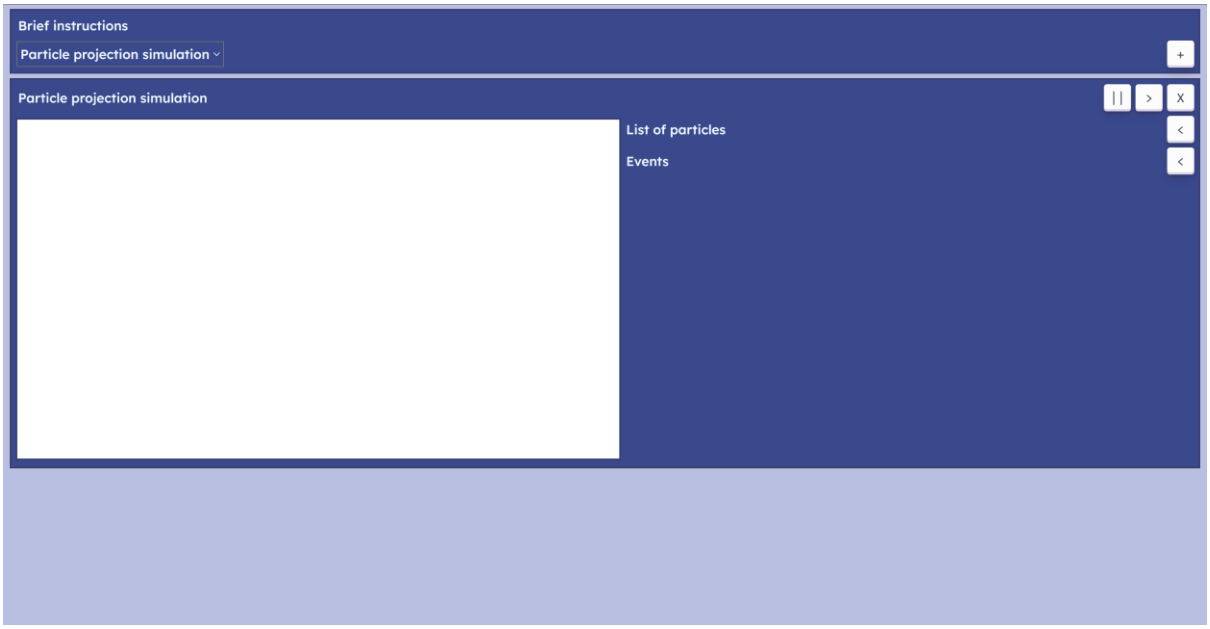
Having these issues fixed, next test:

All features work as expected.

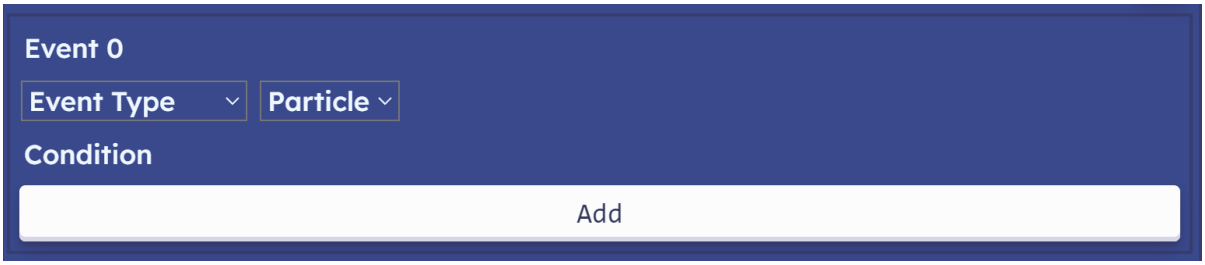
Notes: the mathematical expression evaluation inside input boxes as a feature was decided to be put for further development features. It does not contribute to the project enough for the time it would take to develop it.

4.6.5 Event area

The development process for event area uses mostly the same ideas of creating html elements and arranging them on io_area, the input menu is toggled with the button in the same way as for the particle area. Starting with "createEventArea()" method, the title and the button are created.



The event input menu gets following layout:



The drop-down menus for event type and particle choice are managed with “createEventTypeChoice()” and “resetParticleChoice()” methods respectively. Conditions for the events are created before the type is chosen and made invisible.

Methods that create event condition input fields are “createPositionEventInput()”, “createVelocityEventInput()” and “createTimeEventInput()”. If they are made visible, following output is produced:

4.6.5.1 Event type drop-down

The event type drop-down:

```
createEventTypeChoice(event_input) {
    let drop_down = document.createElement("select");
```

```
    drop_down.id = "event_type_choice" + this.id;

    let option = document.createElement("option");
    option.value = -1;
    option.innerHTML = "Event Type";
    drop_down.appendChild(option);
    //   default option

    let event_types = ["Position Event", "Velocity Event", "Time Event"];
    //   defines the order where
    //   0 - Position; 1 - Velocity; 2 - Time;

    for (let i = 0; i < event_types.length; i++) {
        let option = document.createElement("option");
        //   creating an option
        option.value = i;
        option.innerHTML = event_types[i];
        drop_down.appendChild(option);
        //   add the option to the dropdown menu
    }

    event_input.appendChild(drop_down);

    drop_down.onchange = () => {
        this.toggleCondition(drop_down);
    }
}
```

Once the user chooses some event type, the “onchange” event of the HTML select element is triggered. The ”toggleCondition()” method is invoked for onchange to set one of the conditions visible depending on the selected value of the drop-down.

```
resetParticleChoice() {
    let drop_down = document.getElementById("particle_choice" + this.id);
    drop_down.innerHTML = "";
    //   empty previous contents

    let option = document.createElement("option");
    option.value = -1;
    option.innerHTML = "Particle";
    drop_down.appendChild(option);
    //   default option

    for (let i = 0; i < this.body_list.length; i++) {
        if (this.body_list[i] == null)
            continue;
        let option = document.createElement("option");
        //   creating an option
        option.value = i;
        option.innerHTML = "Particle " + i;
```

```
        drop_down.appendChild(option);
        //   add the option to the dropdown menu
    }
}
```

The “resetParticleChoice()” method was mentioned earlier. It is triggered when new particle is added or removed. It resets previous contents of the drop-down menu and iterates over the list of bodies to add all particles again. It is also used to create an empty drop-down menu when the event input menu is created.

4.6.5.2 Adding a new event

Each event has corresponding method that initialises and adds the event output frame to the queue of events. When the event type is chosen, the button to add new particle is assigned one of the methods “addPositionEvent()”, “addVelocityEvent()” and “addTimeEvent()” is assigned as the button’s onclick callback function.
PositionEvent and VelocityEvent have exact same methods that differ only by the ids and Event types they use:

```
addPositionEvent() {
    let x_value = document.getElementById("x_event_condition" +
this.id).value;
    let y_value = document.getElementById("y_event_condition" +
this.id).value;
    //   retrieving user input for conditions

    let error_message = document.getElementById("event_error_message" +
this.id);
    error_message.style.display = "none";

    let particle_id = parseInt(document.getElementById("particle_choice" +
this.id).value);
    if (particle_id == -1) {
        error_message.innerHTML = this.PARTICLE_NOT_CHOSEN;
        error_message.style.display = "block";
        return;
    }
    //   validating particle choice
    let particle = this.body_list[particle_id];
    //   retrieving the required particle

    let event = new PositionEvent(particle, this, this.next_event_id);

    if (!isNaN(x_value) && !(x_value === ""))
        event.setXtime(parseInt(x_value));
        //   validating x_value
    else if (!isNaN(y_value) && !(y_value === ""))
        event.setYtime(parseInt(y_value));
        //   validating y_value
    else {
        error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
```

```
        error_message.style.display = "block";
        return;
    }

    if (!event.isValid()) {
        error_message.innerHTML = this.INVALID_EVENT_ERROR;
        error_message.style.display = "block";
        return;
    }

    this.sim.addEvent(event);

    this.enqueueEvent(event, this.sim);

    this.next_event_id++;
    this.resetNextEventTitle();
}
```

Each routine first validates the data from the input fields. Starting with checking whether a particle is chosen, outputting “particle_not_chosen” error if the particle was not chosen. Then the method checks if the values entered to input fields are of valid form (the same format as for particle input). If no valid input is found for x, then the y value is checked. In this way, the user can choose one of the inputs to define condition of the event, but if they do both, then the x value will be prioritised.
If the event itself has calculated that it will never occur (indicated by negative value for “occurs_at” attribute), then the “ivalid_event_error” is outputted. Finally, if all the checks are passed, the event is added to the simulation, “next_event_id” is incremented, the event title is reset, and event is enqueued.

The “enqueueEvent()” method indicates to the user the time till the condition is met. It puts an output block, like the particle output. The time is calculated like that:

```
let time = (event.getTime() - sim.getTime()) / 1000;
```

Difference between the moment event occurs and current time.

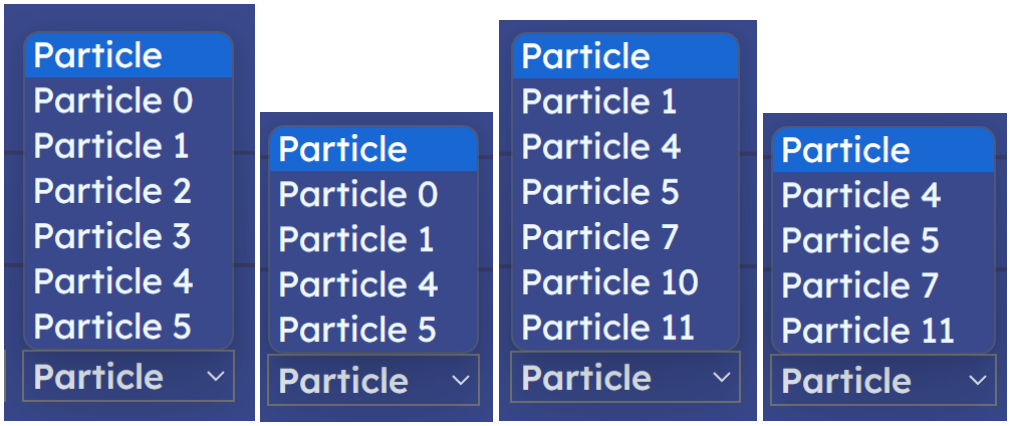
And finally, the callback function for the moment event is executed by the ParticleProjectionSim class:

```
executeEvent(event) {
    let enqueued_event = document.getElementById("enqueued_event" +
event.getId() + "_" + this.id);
    enqueued_event.remove();
}
```

For now, it will simply retrieve an HTML element representing the event’s output block and remove it.

4.6.5.3 Tests and corrections

Switching between different event types works perfectly.

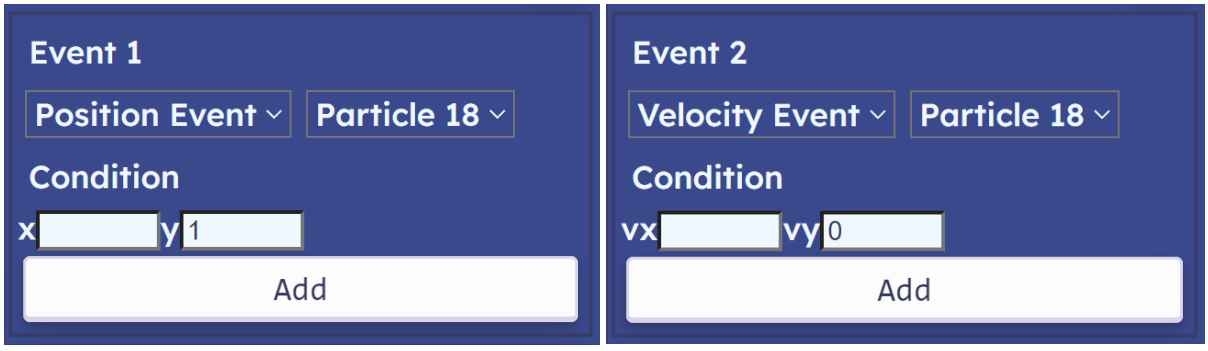


The particle drop-down is updated as particles are added and removed.

Error messages outputed correctly.

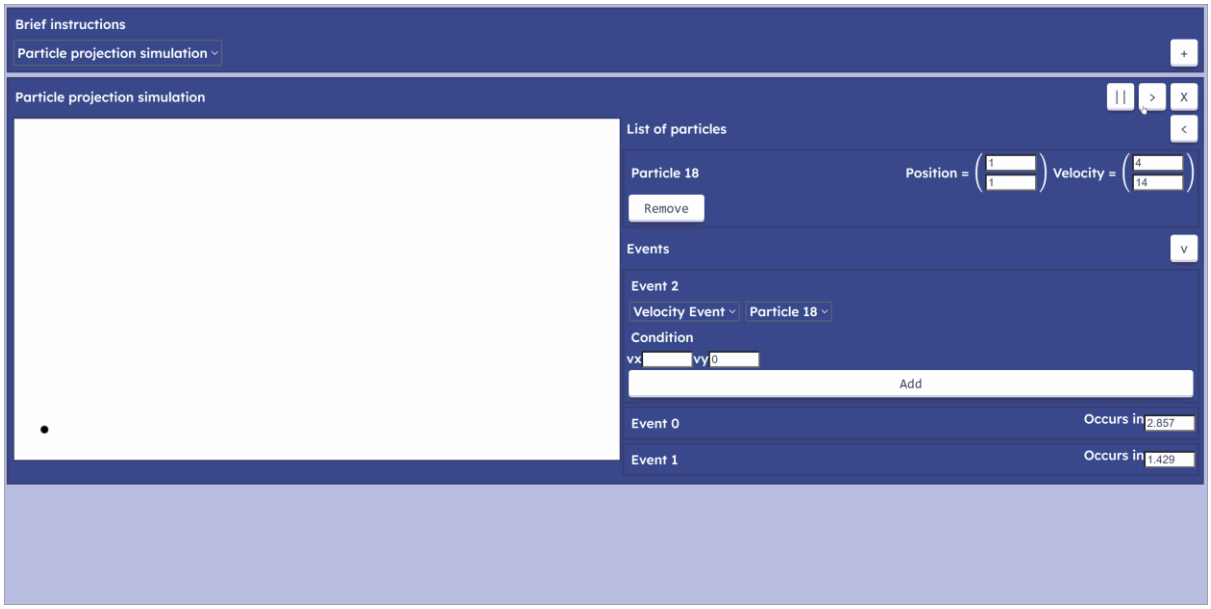
Now check if the event system work as intended. First, add the following particle to simulation:

Define events in a following way:



The first one corresponds to the particle returning back to its starting height. And the second one corresponds to the particle reaching its greatest height.

The output blocks are correctly computed and put on the screen.



The simulation is stopped at each event and the output blocks are removed as planned.
Now only the labels look strange and not symmetric. To fix that, I will set their styling to the same one as for the <p> tag by listing it next to <p> in the CSS stylesheet.

```
p, label {
    margin: 3px;
    padding: 3px;
    align-self: center;
}
```

4.6.6 Time slider

Now that I went through the development of the major part of the interface, I realise that the io_area is likely to extend down below the edge of the canvas, leaving some unused space for under the canvas. It can be utilised to place the time slider there.
It was also pointed out by stakeholders that inputting time by hand can be a useful feature. So, I suggested the following layout, which was widely accepted.

The button on the left would set simulation time to 0. The slider will tell the user current time and allow them to change it by sliding the point along itself. The input field on the right will allow the user to input any time in seconds, changing current simulation time.
Also, it is more appropriate to name the method that starts the process of creating this structure a “createTimeControl()” while leaving “createTimeSlider()” for the slider specifically.

4.6.6.1 Reset button

To place the slider directly below the canvas, I will need to create another container that will hold canvas and the time control container together, while allowing io_area to extend below indefinitely.

The routine of creating a time control panel consists of three main parts: set to zero button; time slider and time input.
The button simply sets time of the simulation to zero and calls an update method of the simulation, so if it is stopped, positions still get updated.

```
createSetToZeroButton(time_control_container) {
    let button = document.createElement("button");
    button.innerHTML = '<i class="material-icons" style="background-color:
rgba(0, 0, 0, 0); color: #39498C">&#xe042;</i>';

    button.onclick = () => {
        this.sim.time = 0;
        this.sim.update(0);
        //   update positions
    }

    time_control_container.appendChild(button);
}
```

At this stage, I realised that “set to 0” is a too big label for a button. It is also not the most intuitive. Consulting with the stakeholders, we came up with an idea of using a special symbol representing the reset. So, I imported the “material class” from google fonts and used the code above to put the special character “&#xe042”, producing following result: (the background is set to transparent as rgb(0, 0, 0, 0))

4.6.6.2 Text time input

The time input is a simple html input tag of type text that accepts a value, converting it into float and milliseconds before assigning its value to the time attribute of the simulation:

```
createTimeInput(time_control_container) {
    let input = document.createElement("input");
    input.type = "text";
    input.style.marginRight = "4px";
    input.style.marginLeft = "4px";

    input.onchange = () => {
        let value = input.value;
        if (!isNaN(value)) {
            this.sim.pause();
            this.sim.time = parseFloat(value) * 1000;
            this.sim.update(0);
        }
    }

    time_control_container.appendChild(input);
}
```

Margin left and right create some space between the input window, input field and io_area.

4.6.6.3 Time slider

The largest part of this section: the time slider. Developing the “createTimeSlider()” method:

```
let slider = document.createElement("input");
slider.id = "time_slider" + this.id;
slider.type = "range";
slider.style.flexGrow = 1;
```

The time slider is created as the input of type range.

```
slider.min = -5;
slider.max = 70;
slider.value = 0;
slider.step = "any";
```

The range is set up to be from -5 to 70 with initial value of 0. The step value defines how many units by the slider can be moved at a time. The value of any gives the user ability to adjust the time however they want. The value represents seconds since the start of simulation (need to be converted to milliseconds before putting into simulation).

```
let datalist = document.createElement("datalist");
for (let i of [0, 60]) {
    let option = document.createElement("option");
    option.innerHTML = i;
    datalist.appendChild(option);
}
slider_container.appendChild(datalist);
slider.setAttribute("list", "marks");
```

The datalist allows to assign specific points that the thumb of the slider will be attracted to when it is very close to them. For now, I will define two such points: at 0 seconds and 60 seconds.

```
slider.oninput = () => {
    this.sim.pause();
    this.sim.time = parseFloat(slider.value) * 1000;
    this.sim.update(0);

    this.updateThumb();
}
```

The function is triggered every time the user interacts with it (every page update when the user touches the thumb, not only on release). It stops the simulation, changes its time, and updates it. The update thumb method is called to change position of the subscript below the thumb (it will be described later). The tick marks are also indicated by a separate structure of elements as ticks and elements as subscripts. They are created with the function “createTickMarks()”, described later.

To make the tick marks, I will need to create another container that will enclose the slider, tick marks and their subscripts. The container will need to have position set to “relative”, so other elements can be positioned relatively to it using “absolute” property for their positions:

```
let slider_container = document.createElement("div");
slider_container.style.display = "flex";
```

```
slider_container.style.flexGrow = 1;
slider_container.style.position = "relative";
slider_container.style.marginLeft = "10px";
slider_container.style.marginRight = "10px";
```

```
let thumb_text = document.createElement("span");
thumb_text.id = "thumb_text" + this.id;
thumb_text.classList.add("ticktext");
thumb_text.style.top = "22px";
slider_container.appendChild(thumb_text);
this.updateThumb();
```

Lastly, the thumb text is created as text subscript that will be changing its position depending on the position of the thumb. To style it appropriately, the “ticktext” CSS class is used, for which I wrote following code:

```
.ticktext {
    position: absolute;
}
```

It could have been substituted with a line “thumb_text.style.position = ‘absolute’ ”, but my approach allows to further easily tweak code, so it applies to all tick marks. The position absolute allows to define its position relative to the slider container.

Now the time control panel looks like this:

To start with, I reset the default properties of the slider, applying following style:

```
input[type="range"] {
    -webkit-appearance: none;
    appearance: none;
    align-self: center;
    height: 3px;
    background-color: aliceblue;
    border-radius: 3px;
}
```

Result is:

The thumb will be styled using the following CSS code:

```
input[type="range"]::-webkit-slider-thumb {
    -webkit-appearance: none;
    appearance: none;
    height: 12px;
    width: 12px;
    border-radius: 50%;
    outline: 3px solid aliceblue;
```

```
    background: #373d56;
    cursor: pointer;
    margin-top: -16px;

    transition: all .2s ease-in-out;
}

input[type="range"]::-webkit-slider-thumb:hover {
    transform: scale(1.1);
}
```

Its default appearance is reset with the first two lines. The height and width are set to 12px and border radius to 50% to make it a circle of radius 6px. The outline is of the same colour as the slider. Margin-top of -16px pulls the thumb up so it slides along the slider. When the user hovers the mouse over thumb, it gets scaled by 1.2, so the users know that they now interact with the thumb:

"createTickMarks()" method:
Its main purpose is to create the tick marks and position them according to the value on the slider the represent. The tick mark style:

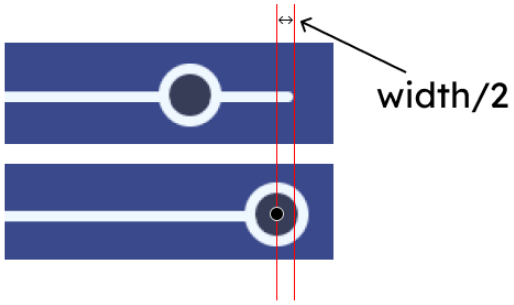
```
.tickmark {
    height: 28px;
    width: 2px;
    background-color: aliceblue;
    position: absolute;

    align-self: center;
    border-radius: 3px;
}
```

The horizontal bar with rounded edges of the same colour as the slider.

To figure out the position of the first tick mark (representing 0), I calculate the percentage it must be located from the left edge of the slider. The slider starts at -5 and goes up to 70, therefore the percentage of 0 on this scale is $\frac{5}{75} * 100$:

```
tick0.style.left = `calc(${5 * 100 / 75}%)`;
tick60.style.left = `calc(${65 * 100 / 75}%)`;
```

Both marks are off by a couple of pixels, however, the first mark is leftwards to its intended position and the second one is rightwards. Investigating this discrepancy, I noticed that the thumb does not go all the way to the edge of the slider. It stops when its border hits the edge of the slider. Therefore, the effective length of the slider is not 100%, but "100% - 12px" (6px form both edges). Therefore, the position of tick marks must be calculated with respect to this fact:



$$left\ edge\ position = 6px + \frac{value + 5}{75} * (100\% - 12px) - 1px$$

The effective width of slider starts at 6px to the right of the left edge. The proportion of the total width is calculated as $\frac{value+5}{75}$ and then multiplied by the effective width $(100\% - 12px)$. One pixel is subtracted to account for the width of the tick mark.

```
tick60.style.left = `calc(6px + 65 * (100% - 12px) / 75 - 1px)`;
tick0.style.left = `calc(6px + 5 * (100% - 12px) / 75 - 1px)`;
```

Gives following results:

Tick marks are positioned correctly. To push the thumb to the front layers, set its "z-index" CSS property to "1". Text is positioned using similar formula for the right edge:

$$right\ edge\ position = 6px + \frac{70 - value}{75} * (100\% - 12px) + 8px$$

The largest value to the right is 70, subtracting required value from 70 gives the distance from the right edge of the slider. Then shift 8px to the right to avoid collision with the tick mark:

```
ticktext60.style.right = `calc(6px + 10 * (100% - 12px) / 75 + 8px)`;
ticktext0.style.right = `calc(6px + 70 * (100% - 12px) / 75 + 8px)`;
```

Gives following result:

Now configure the subscript for the thumb text. As the simulation will be advancing its time separately from the IO handler, the thumb text will need to be regularly updated, but only when simulation is active. Put the following two lines in the update method of the IO handler:

```
if (this.sim.isActive())
    this.updateThumb();
```

The "updateThumb()" method itself will receive time from the simulation, convert it to seconds and change the value of the slider accordingly:

```
let value = this.sim.getTime() / 1000;
let clamped_value = Math.max(-5, Math.min(value, 70));
```

```
slider.value = clamped_value;
```

However, simulation time can go beyond the range of [-5 to 70]. To avoid setting the value of the slider outside this range, I clamp its value, so if it lower then -5, it is clamped to -5 and if it is larger than 70, it is clamped to 70.

4.6.6.4 Final tests

The thumb moves correctly, and simulation is stopped when you interact with it.

Now, configuring the text under the thumb:

```
thumb_text.innerHTML = Math.round(value);
thumb_text.style.right = `calc(6px + ${70 - clamped_value} * (100% - 12px) / 75 + 8px)`
```

The value is rounded before putting it into text. The position is calculated using the same formula as for the text under tick marks.

4.7 Further functionality

Success criteria: moving camera around simulation; simulation scale adjustment; coordinate axes; trajectory; drawing objects.

4.7.1 ViewSim additions

Before implementing other classes, I will write the methods required from the ViewSim to assist the axes and trajectory objects. First, the coordinate axes' dimensions are not affected by the scale, so the earlier created method "toCanvas()" that convers the point in simulation space to the canvas space will not be used. However, coordinates still need to be inversed and translated. It will be accomplished with the "inversedTranslated(vector)" method.

```
inversedTranslated(vector) {
    let new_vector = vector.reflectedInX();
    //   reflecting the vector
    new_vector.add(this.translation);
    //   translating the vector to a new position
    return new_vector;
}
```

Simply returns desired vector.

```
drawLine(start, finish, color="black") {
    //   unscaled
    this.ctx.beginPath();
    this.ctx.moveTo(...this.inversedTranslated(start));
    //   starting point
    this.ctx.lineTo(...this.inversedTranslated(finish));
    //   end point
    this.ctx.closePath();
    this.ctx.strokeStyle = color;
    this.ctx.stroke();
}
```

Drawing a line by coordinates of two points (unscaled line). Reminder, the "…vector" notations "unpacks" the vector using iterator interface, so instead of passing vector.getX() and vector.getY() as two separate parameters, the "…vector" notation is used, which accomplishes the same task.

```
renderTextByTopRight(point, text) {
    this.ctx.font = "18px serif";
    this.ctx.textAlign = "right";
    this.ctx.textBaseline  = "top";
    this.ctx.fillText(
        text,
        ...this.inversedTranslated(point)
    );
}
```

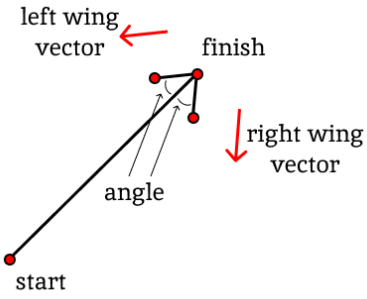
The other useful method to draw text. For the coordinate grid, all text on the screen can be drawn aligning it to the top right corner of the text. This can be accomplished with "textAlign" and "textBaseline" properties of the canvas context as shown in the code above. Then the text as a parameter is filled by the coordinates of its top right corner represented by the "point" parameter.

4.7.1.1 Arrow drawing

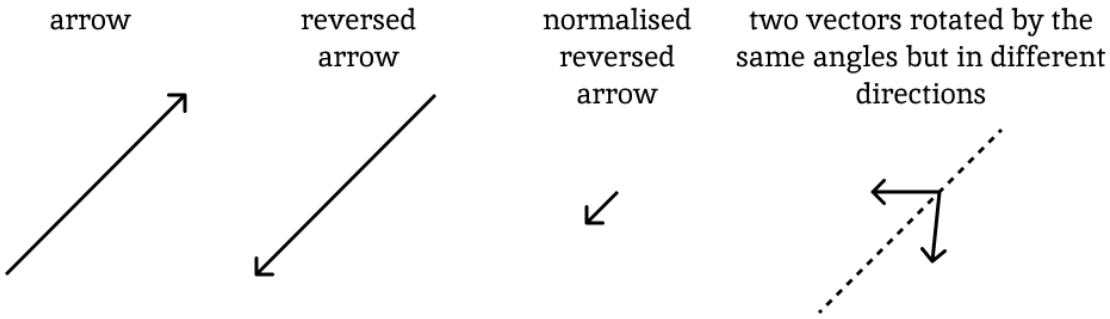
I will add another method for arrow drawing to the ViewSim. Primarily, it will be used for coordinate axes, but can be then reused in future, e.g., to show velocity on the screen.

To draw an arrow, I will need to know positions of several points. They are shown as red dots on the diagram on the right. The start and finish are received as parameters. The method in its turn calculates the positions for "wings" of the arrow.

The process of calculating the vectors pointing out of the end point of



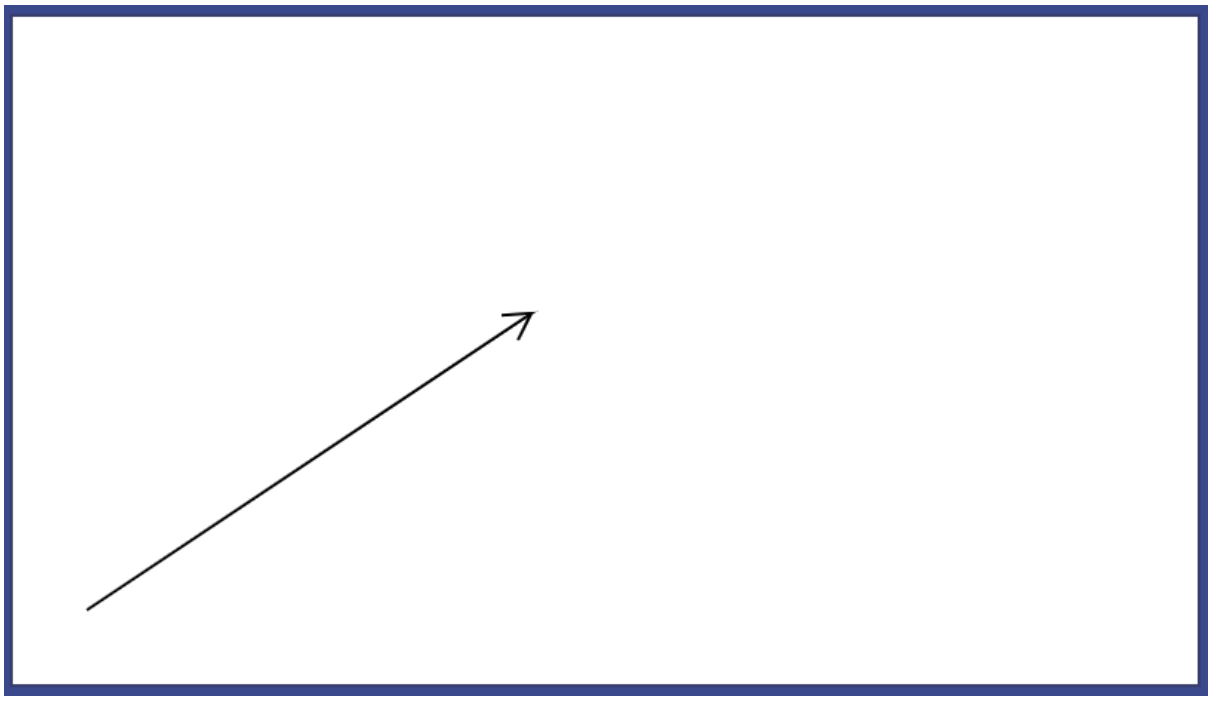
the vector can be divided into following stages:



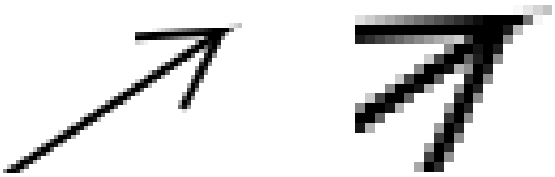
First, I need to get an arrow vector which will define the direction of the wing vectors. Then I reverse it (multiply by -1) and normalise to get the direction as a unit vector. The wing vectors can be calculated as this unit vector rotated anticlockwise by a certain angle (for the right-wing vector) and negative angle (for a left-wing angle). The positions of the points representing the ends of wings are calculated as “finish + left wing” and “finish + right wing”. Putting it into code:

```
drawArrow(start, finish, color="black") {
    this.ctx.strokeStyle = color;

    let angle = Math.PI / 6;
    let arrow = finish.subtracted(start);
    let reversed_arrow_unit = arrow.multiplied(-1).normalized();
    //   direction of the arrow
    let left_wing = reversed_arrow_unit.rotatedBy(-angle).multiplied(20);
    let right_wing = reversed_arrow_unit.rotatedBy(angle).multiplied(20);

    this.ctx.beginPath();
    this.ctx.moveTo(...this.inversedTranslated(start));
    this.ctx.lineTo(...this.inversedTranslated(finish));
    //   base of the arrow
    this.ctx.lineTo(...this.inversedTranslated(finish.added(left_wing)));
    this.ctx.moveTo(...this.inversedTranslated(finish));
    this.ctx.lineTo(...this.inversedTranslated(finish.added(right_wing)));
    //   wings
    this.ctx.stroke();
}
```

First, all the variables are defined. The angle is taken as $\frac{\pi}{6}$ $(30°)$. Earlier implemented vectors for rotations are reused to create “wings”. First the “pen” of the canvas draws the base of the arrow, moves to the left wing, comes back to the end point and moves to the right wing. The following result is achieved for a test arrow:



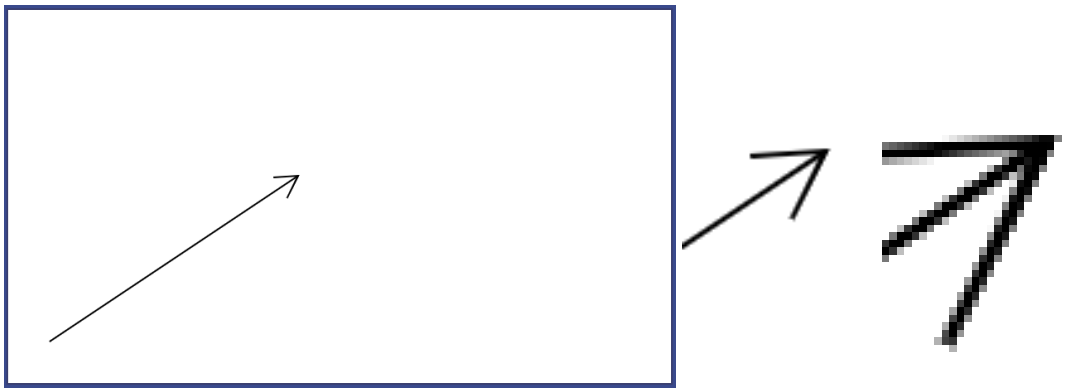
It may look like the correct result, however looking closely at the tip of the arrow:



The canvas incorrectly handles lines connecting when they make sharp turns like moving from the finish to left wing and back to the finish. Researching this issue, I discovered a solution:

```
this.ctx.beginPath();
this.ctx.moveTo(...this.inversedTranslated(start));
this.ctx.lineTo(...this.inversedTranslated(finish));
this.ctx.stroke();
//   first draw the base completely
this.ctx.beginPath();
//   start new path for the head of the arrow
this.ctx.moveTo(...this.inversedTranslated(finish.added(left_wing)));
this.ctx.lineTo(...this.inversedTranslated(finish));
this.ctx.lineTo(...this.inversedTranslated(finish.added(right_wing)));
this.ctx.stroke();
```

The arrow can be drawn in two separate stages as two different paths. The base first and then the head. The head is also drawn starting from left wing, going to the end point of the arrow, and finishing at the right wing.



Now results are produced correctly.

4.7.1.2 Layers

Abandoned previously idea which I decided to implement now. As the number of particles does not reach large numbers, I will implement a simple solution: keep the list sorted in the order of layers. Every time a new object is added, it will be appended to the end of the list and moved leftwards until it is at the right place. It results in the worst-case time complexity of $O(n)$ where n is the size of the list. The order of the body list must be such that elements on lower layers appear earlier in the body list. So, when the "redraw()" method iterates through the list of bodies, it first encounters the elements on lower layers on top of which others are drawn.

```
addBody(body) {
    this.body_list.push(body);
    let i = this.body_list.length - 1;
    while (i > 0 &&
           this.body_list[i - 1].getLayer() > this.body_list[i].getLayer()) {
        //   while the end of the list has not been reached
        //   and the body at (i) is on lower layers than body (i+1)
        let temp = this.body_list[i - 1];
        this.body_list[i - 1] = this.body_list[i]
        this.body_list[i] = temp;
        //   swap them
        i--;
    }
}
```

The while loop has two conditions. The first one ensures that the loop stops when the pointer i (points at new added element) reaches end of the list, the second one checks if a swap is required; if either of these conditions break then the element is at the right place and algorithm is finished. No further checks are required because the list is always kept sorted. Deletion of an element does not break the order because elements to the right and to the left are in sorted order.
ViewBody subclasses will need to have a layer as an attribute to accommodate these changes.

4.7.2 Keyboard Input

4.7.2.1 Implementation

First create "increaseTransaltionBy(vector)" and "increaseScaleBy(value)" ViewSim methods. They will allow external modification of scale and translation parameters of the ViewSim objects.

```
increaseTranslationBy(vector) {
```

```
    this.translation.add(vector);
}

increaseScaleBy(value) {
    this.scale *= value;
}
```

Next add “createKeyBoardInput()” method to the IO handler, that will define an “onkeypress” property for the canvas. It will detect any keyboard input from the user. The WASD keys will represent camera moving around the canvas. The callback function will identify the exact key that was pressed and perform appropriate action: key W – increase translation by 2 down (+2 as the canvas y-axis is oriented down); key A – by 2 rightwards; key S – by 2 up; key D – by 2 leftwards. If the shift key is pressed, the rate at which the translation changes will be increased. The “shiftKey” attribute of the event returns True or False depending on whether the shift key is pressed. Putting it in a mathematical expression will convert it into 0 or 1.
To change the scale: key O – increase by a factor of 1.2; I - decrease by a factor of 1.2:

```
canvas.onkeydown = (event) => {
    if (event.code == "KeyW") {
        this.view.increaseTranslationBy(new Vector(0, 2 + event.shiftKey*10));
    }
    else if (event.code == "KeyA") {
        this.view.increaseTranslationBy(new Vector(2 + event.shiftKey*10, 0));
    }
    else if (event.code == "KeyS") {
        this.view.increaseTranslationBy(new Vector(0, -2 - event.shiftKey*10))
    }
    else if (event.code == "KeyD") {
        this.view.increaseTranslationBy(new Vector(-2 - event.shiftKey*10, 0))
    }
    else if (event.code == "KeyI") {
        this.view.increaseScaleBy(1/1.2);
    }
    else if (event.code == "KeyO") {
        this.view.increaseScaleBy(1.2);
    }
}
```

The callback function receives KeyboardEvent object as a parameter. The “code” attribute gives the code of the key being pressed.

Also add the hotkey to toggle the “isActive” Simulation attribute on Space key and reset simulation time to 0 on Backspace key.

```
else if (event.code == "Space") {
    this.sim.toggle();
}
else if (event.code == "Backspace") {
    this.sim.resetTime();
    this.update();
```

```
    //    update positions
}
```

Also add helper method for accessing attributes of simulation: “sim.toggle()” and “sim.resetTime()”:

```
toggle() {
    this.is_active = !this.is_active;
}

resetTime(value=0) {
    this.time = value;
    this.update(0);
}
```

The “resetTime()” method automatically updates simulation as well. Now all instances where the time attribute was modified by the Controller directly can be replaced with this method:

```
createSetToZeroButton(time_control_container) {
    let button = document.createElement("button");
    button.innerHTML = '<i class="material-icons" st

    button.onclick = () => {
        this.sim.time = 0;
        this.sim.update(0);
        this.update();
        //    update positions
    }

    time_control_container.appendChild(button);
}
```

->

```
createSetToZeroButton(time_control_container) {
    let button = document.createElement("button");
    button.innerHTML = '<i class="material-icons" style='

    button.onclick = () => {
        this.sim.resetTime();
        this.update();
        //    update positions
    }

    time_control_container.appendChild(button);
}
```

(and others)

4.7.2.2 First tests

To test the correct input processing, I will place four static points to the simulation. One at the origin, one 5 units to the right, one 5 units above and the other one both 5 to the right and above.

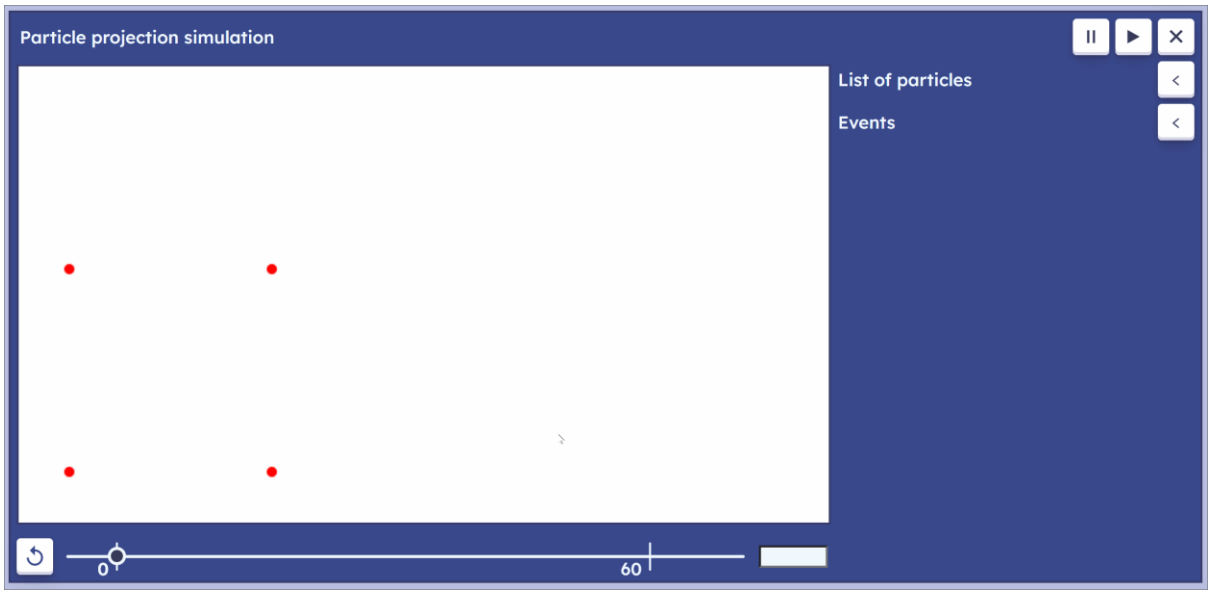
```
this.addBody(new ViewPointMass(
    new PointMass(
        new Vector(0, 0),
        new Vector(),
        0,
        0
    ),
    "red"
))
```

```
this.addBody(new ViewPointMass(
    new PointMass(
        new Vector(0, 5),
        new Vector(),
        0,
        0
    ),
    "red"
))
```

```
this.addBody(new ViewPointMass(
    new PointMass(
        new Vector(5, 0),
        new Vector(),
        0,
        0
    ),
    "red"
))
```

```
this.addBody(new ViewPointMass(
    new PointMass(
        new Vector(5, 5),
        new Vector(),
        0,
        0
    ),
    "red"
))
```

They will act as reference points representing simulation being translated. Now run simulation and test keyboard input:



The GIF clearly shows that simulation is being translated. When I / O keys are pressed, the distance between points changes because of a change in scale. The bottom left particle remains static as it is located at the origin and scale is applied relative to the origin.

However, this system of keyboard input processing does not allow multiple keys being processed at a time.

4.7.2.3 Improvements

To allow the program to handle multiple key input at a time, I had to rework system slightly. One of the ways to handle this is to keep a dictionary of pressed keys / key codes corresponding to the Boolean value of true if the key is pressed and false otherwise. When a key is pressed, first "keydown" event is fired which indicates that the key is pressed. When a key is released, the "keyup" event is fired for that key, which can be recognised by the program to set the value for this key in the dictionary to false.

```
createKeyboardInput() {
    let canvas = this.view.getCanvas();
    key_dictionary = {};

    canvas.onkeyup = canvas.onkeydown = (event) => {
        const is_pressed = event.type == "keydown";
        key_dictionary[event.code] = is_pressed;
    }
}
```

The same function is defined for both keyup and keydown events. At the start, the function identifies if the event type. If it is a keydown event, then the value of the key in the dictionary will be true and false otherwise.

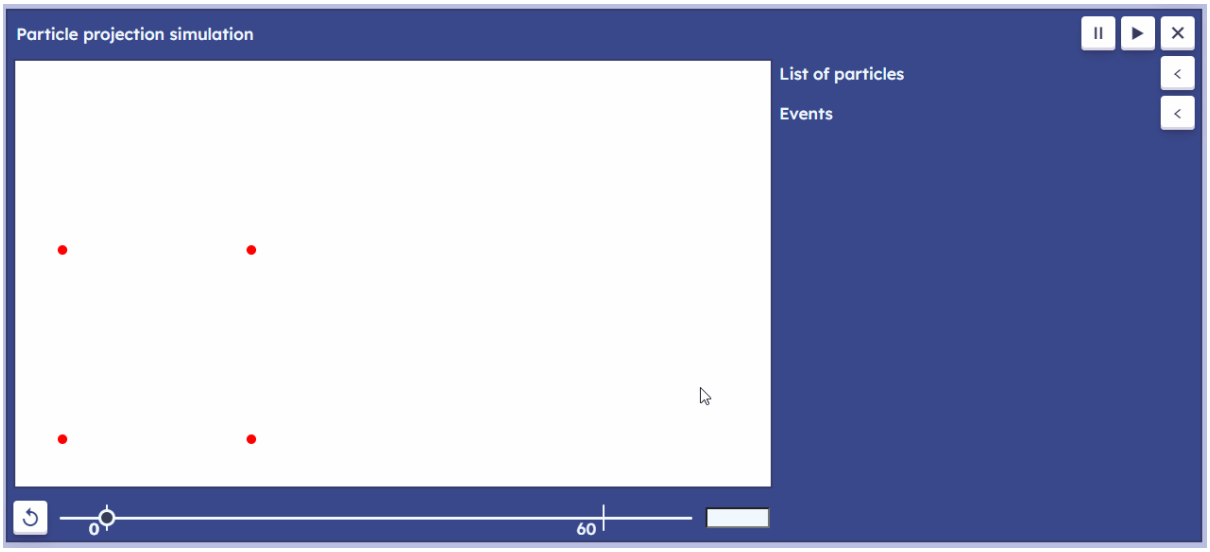
Logic for input handling is the same, but each key is checked with a separate if statement rather than a chain of if else statements, as multiple conditions can be simultaneously true at a time:

```
if (key_dictionary["KeyW"]) {
    this.view.increaseTranslationBy(new Vector(0, 2 + event.shiftKey*10));
}
```

```
if (key_dictionary["KeyA"]) {
    this.view.increaseTranslationBy(new Vector(2 + event.shiftKey*10, 0));
}
if (key_dictionary["KeyS"]) {
    this.view.increaseTranslationBy(new Vector(0, -2 - event.shiftKey*10));
}
if (key_dictionary["KeyD"]) {
    this.view.increaseTranslationBy(new Vector(-2 - event.shiftKey*10, 0));
}
if (key_dictionary["KeyI"]) {
    this.view.increaseScaleBy(1/1.2);
}
if (key_dictionary["KeyO"]) {
    this.view.increaseScaleBy(1.2);
}
```

Now if two or more keys are pressed at the same time, appropriate action will be executed for each of them and not only the one that event is being fired on.

4.7.2.4 Further tests



The GIF illustrates that two buttons pressed at the same time result in a diagonal movement. If both buttons to increase and decrease scale are pressed, nothing changes, because actions for both keys are executed.

4.7.3 Coordinate axes

4.7.3.1 Implementation

The idea of coordinate axes is described in its respective section in design. To implement it into the project, I decided to create a separate ViewBody class, called CoordinateAxes. It will be drawn as other objects of the ViewSim in the update loop, using the “redraw()” method. As the object must be update every time the scale is changed, it will have a scale attribute and its setter. The overall routine of drawing the axes will be divided into several parts, each represented by its own method.

```
import Vector from "../utility/Vector.js";
import ViewBody from "./ViewBody.js";


class CoordinateAxis extends ViewBody {
    constructor(scale) {
        super();
        this.scale = scale;
    }

    setScale(scale) {
        this.scale = scale;
    }

    redraw(view) {
        view.drawArrow(
            new Vector(-30, 0),
            new Vector(800, 0)
        );

        view.drawArrow(
            new Vector(0, -30),
            new Vector(0, 380)
        );
    }
}

export {CoordinateAxis as default};
```

The main “redraw()” method starts the routine. In the same way, as for other ViewSim objects, it accepts the “view” as a parameter to be able to access the canvas and drawing functions. Starting with the setup shown on the left, the simulation gets the following look:

The x axis goes beyond the canvas as its end point was set to (800, 0) and because the simulation is already initially translated by 50 pixels to the right, the point ends up beyond the screen. Change it to (730, 0). The head of the arrow is also too big. Change the size of the wings to 15 and the angle to $\frac{\pi}{8}$. Result:

The “drawHorizontalTickMark()” and “drawVerticalTickMark()” will be used for the four tick marks defined in design. They will draw the tickmark at specified position of the centre and put a text such that the centre position will represent the top right corner of the box that text occupies (offset by a couple of pixels to avoid intersection with ticks and axes).

```
drawHorizontalTickMark(center, text, view) {
    let offset = new Vector(10, 0);
    view.drawLine(
        center.subtracted(offset),
        center.added(offset)
    )

    view.renderTextByTopRight(center.added(new Vector(-3, -3)), text)
}
```

The tick mark is represented as the line of width 20, (10 to the left of the centre and 10 to the right). Then the text is rendered by top right corner as a centre – (3, 3).
At this point, I realised that methods for horizontal and vertical tick mark drawing methods differ only by the offset used. So, they can be written in a following way:

```
drawHorizontalTickMark(center, text, view) {
    this.drawTickMark(center, text, view, new Vector(10, 0));
}

drawVerticalTickMark(center, text, view) {
    this.drawTickMark(center, text, view, new Vector(0, 10));
}

drawTickMark(center, text, view, offset) {
    view.drawLine(
        center.subtracted(offset),
        center.added(offset)
    )

    view.renderTextByTopRight(center.added(new Vector(-3, -3)), text)
```

```
}
```

The “drawTickMark()” method is used for both cases where each specific method passes offset as an additional parameter.

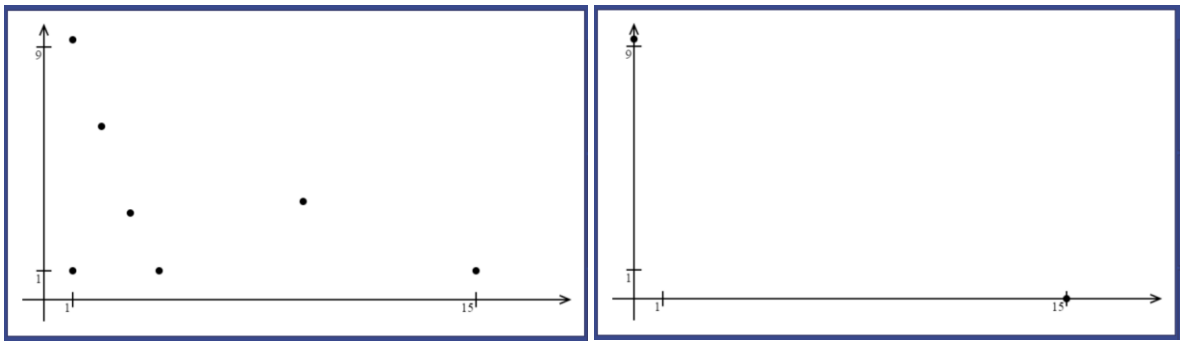
Add to the “redraw()” method:

```
this.drawHorizontalTickMark(
    new Vector(0, this.scale),
    "1", view
)

this.drawHorizontalTickMark(
    new Vector(0, 350),
    Math.round(350 / this.scale), view
)

this.drawVerticalTickMark(
    new Vector(this.scale, 0),
    "1", view
)

this.drawVerticalTickMark(
    new Vector(600, 0),
    Math.round(600 / this.scale), view
)
```

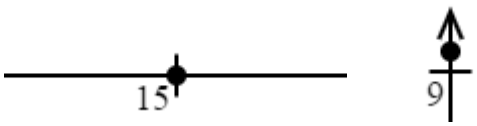
Drawing tick marks on their positions, the label for further marks is calculated as the number of pixels the mark is offset from the origin divided by the scale.

4.7.3.2 Testing

Run the simulation and put several particles on the screen:



The second screenshot represents two particles added with positions represented by the tick marks:

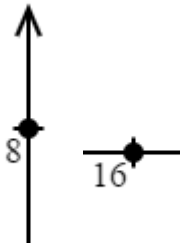


The first particle aligns perfectly. This is because the scale of the simulation is 40 with the tick mark located 600 pixels to the right. $\frac{600}{40} = 15$ with no rounding. The tick mark for 9 is at 350 pixels to the top, $\frac{350}{40} = 8.75$ which is rounded to 9. Therefore, the particle is placed visibly further from the tick mark.

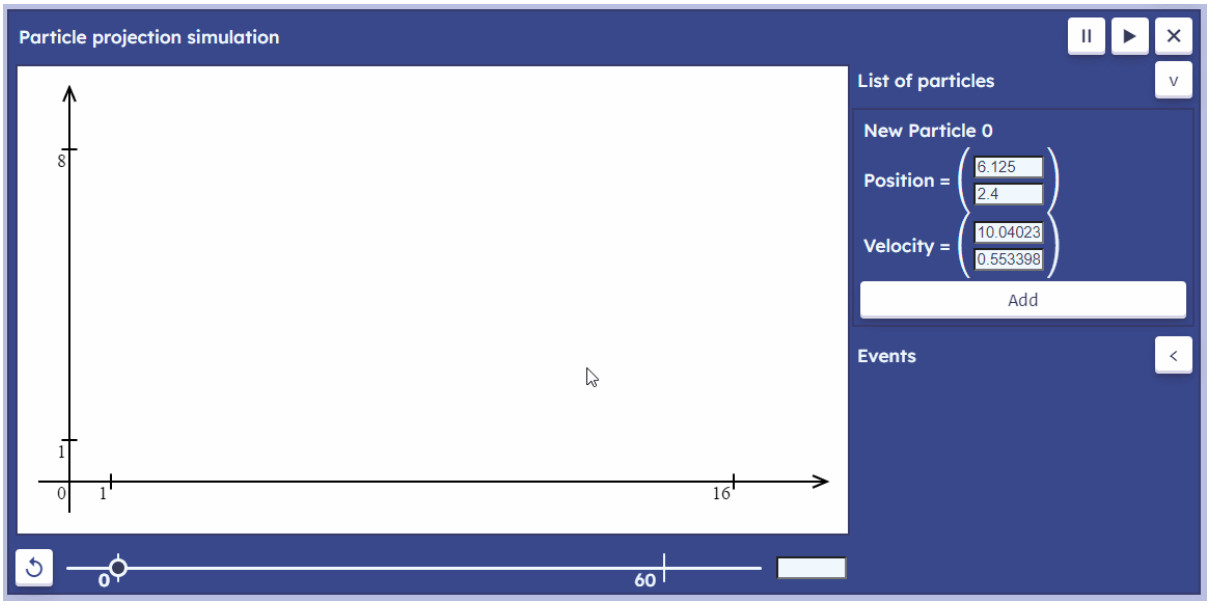
The solution I came up with is to first calculate the value that would have been represented by the tick mark with no round (x or y divided by the scale) and then round down this value. The number of pixels from the origin to the tick mark can then be calculated as this value times the scale:

In this way, x_tick_value and y_tick_value variables represent the largest whole number in simulation coordinates below 650 pixels in canvas coordinates for x and 330 pixels for y. Then it is scaled back to get the coordinates of the tick mark on the canvas.

```
let x_tick_value = Math.floor(650 / this.scale);
let y_tick_value = Math.floor(330 / this.scale);

this.drawHorizontalTickMark(
    new Vector(0, y_tick_value * this.scale),
    y_tick_value, view
)

this.drawVerticalTickMark(
    new Vector(x_tick_value * this.scale, 0),
    x_tick_value, view
)
```



Now the particles are placed correctly and for the scale of 40, the y tick mark value is rounded to 8. (I also made the tick marks slightly smaller, before: 20px wide, now: 14px wide).



Axes tick marks are adjusted whenever the scale is changed. The axes themselves are not scaled as intended.

However, the GIF shows a couple of minor issues: for large values of scale (very zoomed in view), the "1" tick marks are being shifted off the axes; for smaller values of scale (very zoomed out view), the "1" tick marks move too close to the origin and become indistinguishable. These problems occur only for extreme values of scale and do not largely affect usability, so I will leave them to be fixed in further development and focus on remaining functionality.

4.7.4 GUI input

At this stage, I realise that implementing a feature of creating a particle with a click on the canvas does not require a lot of effort since the entire program is modular. At the same time, it adds a lot of interactiveness to the program.

4.7.4.1 Implementation

Once the user clicks on the "new particle button", the "onclick" attribute of the canvas will be defined as a function that retrieves coordinates of the point the user clicked on, converting them into Simulation space and defining values for position and velocity input fields with two clicks. The second click will also add the particle into simulation and close the input menu.

First, I went to the "createParticleMenuButton()" method that creates the button to toggle menu and initialises its click callback function. Now, alongside of turning on and off the input menu, it will also initialise and remove the canvas "onclick" attribute:

```
if (particle_input.style.display == "none") {
    //   change button state to show that the menu is opened
    button.innerHTML = "v";
    //   reveal the menu
    particle_input.style.display = "block";

    //   start canvas input chain
    this.view.getCanvas().onclick = (event) => {
        this.canvasPositionInput(event);
    }
}
```

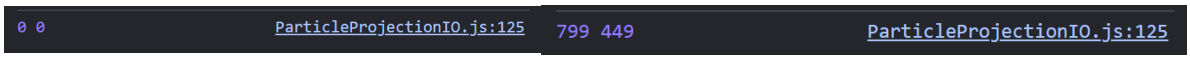
```
else {
    //   change button state to show that the menu is closed
    button.innerHTML = "<";
    //   hide the menu
    particle_input.style.display = "none";

    //   remove canvas input
    this.view.getCanvas().onclick = null;
}
```

The "canvasPositionInput()" method starts a chain of methods that receive input from the user clicking on the canvas.

```
//   input from the canvas
canvasPositionInput(event) {
    //   Get the target (canvas element itself)
    const target = event.target;
    //   Get the bounding rectangle of target
    const rect = target.getBoundingClientRect();
    const x = event.clientX - rect.left;
    const y = event.clientY - rect.top;
    //   x and y relative to the top left corner
    //   with y axis pointing downwards
    console.log(x, y);
}
```

Whenever a callback function is called, it passes the event object as a parameter. This object carries a lot of information about the click and other objects involved, such as the target object, which in this case is a canvas itself. The DOMRect object is then retrieved from the canvas, it provides information about the size of an element and its position relative to the viewport. As both, the cursor and the element positions are registered relative to the viewport, to calculate the position of a click relative to the canvas itself (which is what we need to process the click further) we need to subtract the position of the canvas, given by (rect.left, rect.top) from the position of the click. (x, y) vector represents a vector in canvas space, pointing from the top left corner of the canvas to the point, on which the user clicked.

Following two screenshots show values of x and y for clicks on the very top left and bottom right corners:



Next, the values retrieved from the click must replace the values for position input. Next click will define velocity of the particle as the vector pointing from the particle to the next point the user clicked on. It will be accomplished with the method “canvasVelocityInput(event, position)” which will be set to the canvas onclick by the “canvasPositionInput()” method. The cycle will continue as the user clicks on the canvas until they decide to add the particle. Both methods for position and velocity input will need to retrieve position from the event, so I added the method “getPositionFromClickEvent(event)” that will return the vector in canvas space for the click.

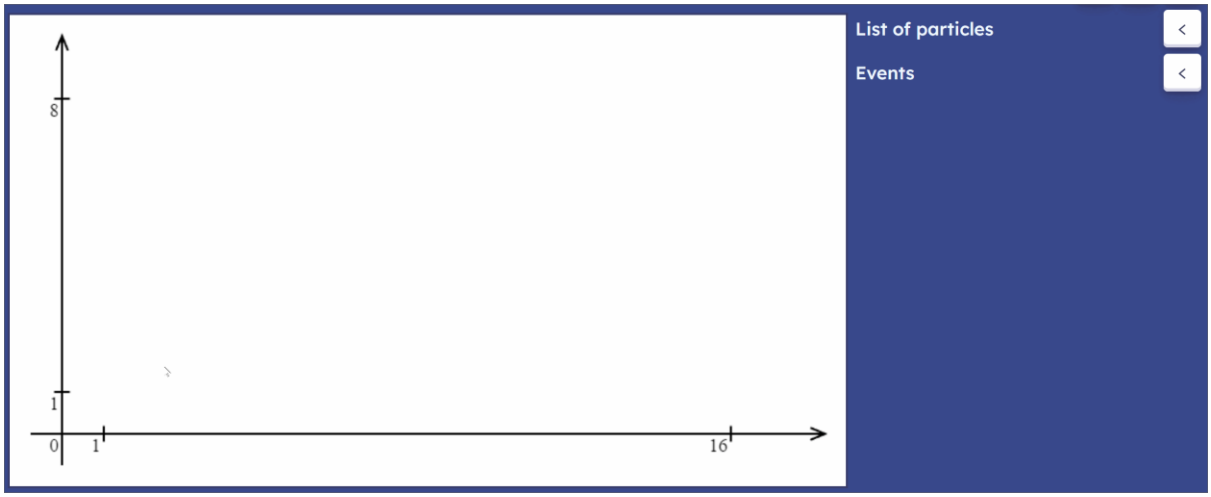
```
canvasPositionInput(event) {
    let position = this.getPositionFromClickEvent(event);

    document.getElementById("position_x_input" + this.id).value =
position.getX();
    document.getElementById("position_y_input" + this.id).value =
position.getY();

    this.view.getCanvas().onclick = (event) => {
        this.canvasVelocityInput(event, position);
    }
}

canvasVelocityInput(event, position) {
    let nextClick = this.getPositionFromClickEvent(event);
    let velocity = nextClick.subtracted(position);

    document.getElementById("velocity_x_input" + this.id).value =
velocity.getX();
    document.getElementById("velocity_y_input" + this.id).value =
velocity.getY();
```

```
    this.view.getCanvas().onclick = (event) => {
        this.canvasPositionInput(event);
    }
}
```

Methods change input fields of the input menu and pass the callback to each other, until the callback is set to null when the input menu is closed.
For convenience of development and testing, I also add a hotkey that invokes and “addParticle()” method (Shift + B):

```
else if (event.code == "KeyB" && event.shiftKey) {
    this.addParticle();
}
```

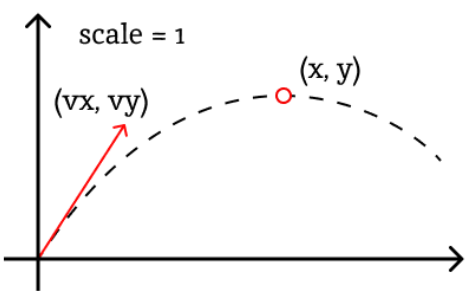
4.7.4.2 Initial test

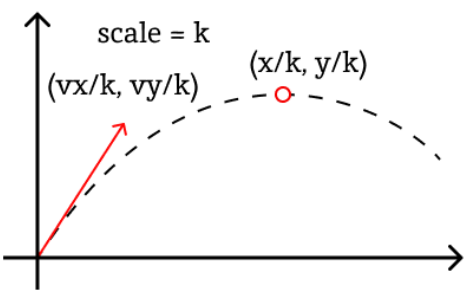


The problem discovered on the GIF above is that for larger scales, the same velocity (in canvas space) has larger effect on the trajectory of the particle. Ideally, if the user defines a particle with two clicks on the canvas with a scale of 1 and then with the scale of 20, the trajectory observed on the screen must not differ. In this way, the user will be able to intuitively expect the same behaviour regardless of the simulation scale, improving their experience.

4.7.4.3 Improvements

This problem ended up being more complicated than I expected. I will explain my solution using following diagrams:





Say the first time, the scale is 1 and the user clicks on some point to define velocity (assuming initial position is (0, 0)). Say the velocity vector is (v_x, v_y) and coordinates of the highest point on its trajectory is (x, y). The second time, simulation is scaled by a factor of k. If the user now clicks on the same point to define velocity, it will be received as $\left(\frac{v_x}{k}, \frac{v_y}{k}\right)$ because when the point on canvas space is received as a click, it gets divided by the scale. For the trajectory to look the same, the position of the highest point on it must coincide with (x, y) as it was before. As the simulation coordinates are multiplied by the scale before being drawn on the screen, the coordinates of this point must be $\left(\frac{x}{k}, \frac{y}{k}\right)$, so when it gets multiplied by k, it is mapped onto (x, y).

Now if we look at the formula for the x position of the highest point on the trajectory:

$$x = \frac{-v_x v_y}{2a}$$

Say that the velocity in the second case is (w_x, w_y) which is some multiple of the $\left(\frac{v_x}{k}, \frac{v_y}{k}\right)$ Then:

$$w_x = \frac{v_x}{k} * p \qquad w_y = \frac{v_y}{k} * p$$

$$\frac{x}{k} = \frac{-w_x w_y}{2a}$$

$$\frac{-v_x v_y}{2ak} = \frac{-v_x v_y}{2ak^2} * p^2$$

$$k = p^2$$

$$p = \sqrt{k}$$

Therefore, for a trajectory to appear the same if the user clicks on the same points, regardless of the scale, the velocity for a new particle must be additionally multiplied by a square root of the scale:

```
let nextClick = this.getPositionFromClickEvent(event);
let velocity = nextClick.subtracted(position);
velocity.multiply(Math.sqrt(this.view.getScale()) / 4);
```

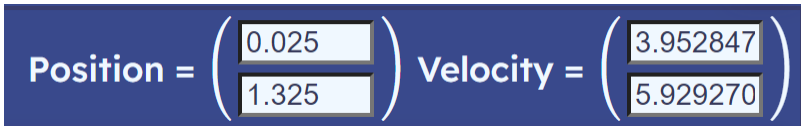
The additional division by 4 just stabilises the velocity, so it does not burst to large values too fast. This is a constant and therefore not dependent on the scale.

4.7.4.4 Final tests

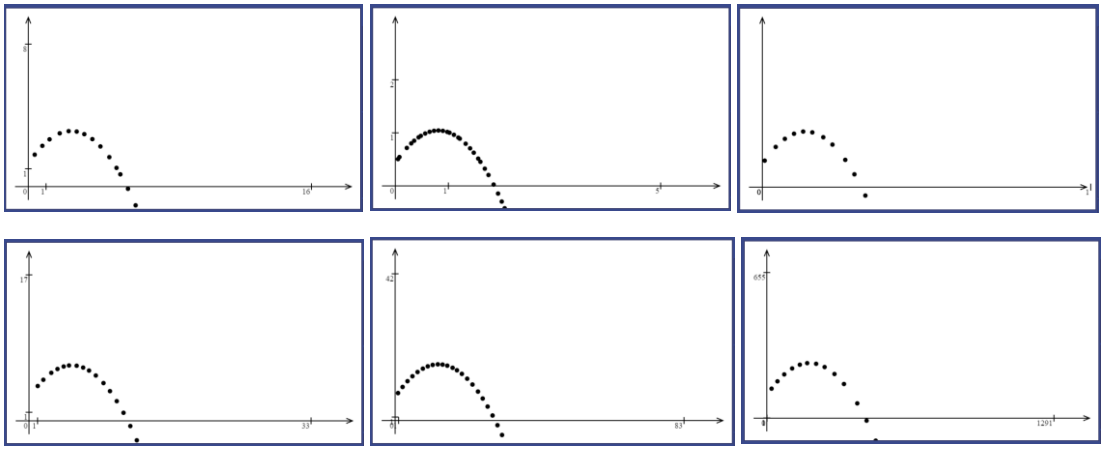
To test if the trajectory remains the same for the same clicks, I created a temporary hotkey that imitates 2 clicks. It ensures that for every test, the positions of two clicks will be the same (omits human error).

```
else if (event.code == "KeyK") {
    this.view.getCanvas().dispatchEvent(new MouseEvent("click", {clientX: 70, clientY: 500, target: this.view.getCanvas()}));
    this.view.getCanvas().dispatchEvent(new MouseEvent("click", {clientX: 170, clientY: 350, target: this.view.getCanvas()}));
}
```

When the “K” key is pressed, two events are dispatched. For initial scale of 40, it creates a particle with following parameters:

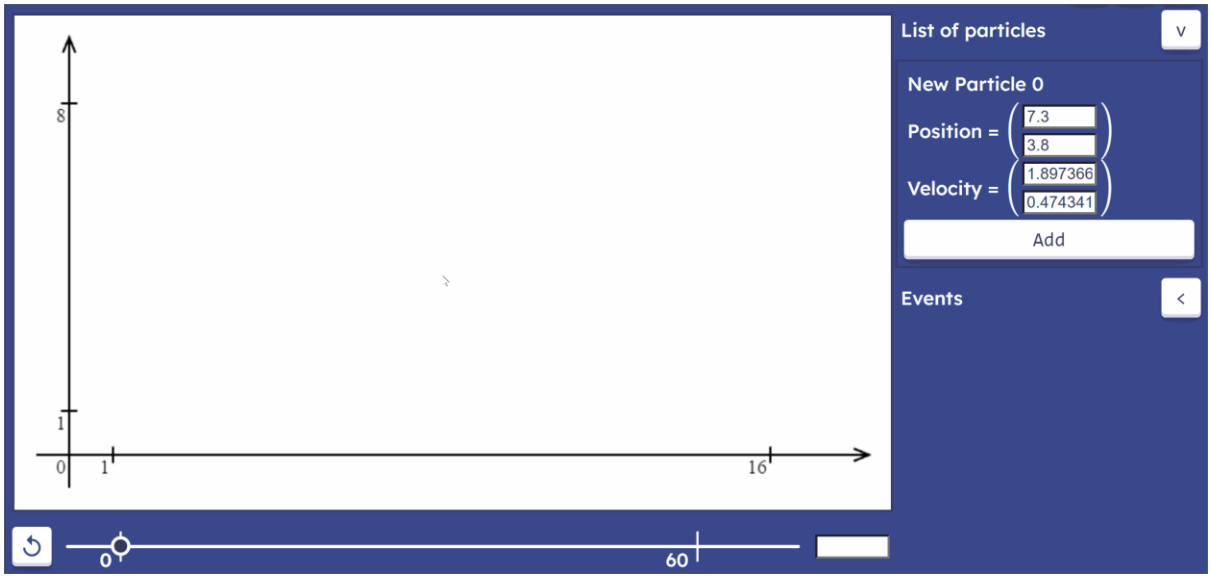


First, I will test creation of the particle using this click for different values of scale and observe their trajectories by spamming the same particle repeatedly:



The difference in scale is noticeable by the position of tick marks. The top row depicts higher scales: 40 and larger. The bottom row shows scales of less than 40. Trajectories are the same as can be inferred from the screenshots.

Now the general test showcased by a following fragment:



Observations: particles move seemingly faster for lower scales (which match to the larger values of scale); some visual indicators for where the particle will be placed would be useful.
The first observation does not appear to be a problem as it is a result of velocity being lower for the larger scales. The second one is not of the priority and can be left for further development.
Other than that, everything works perfectly.

4.7.5 Trajectory drawing

Having multiple conversations with stakeholders, all of them agreed that having a trajectory of the particle (preferably, with an ability to toggle it on demand) is essential.

4.7.5.1 Implementation

Before I implement the trajectory, I will first rework the coordinate system. It exists in only one instance per simulation and does not behave in the same way as other ViewBody objects that link to

a specific body. Therefore, it is more appropriate to make it a separate class, not a child class of ViewBody. Therefore, it will not be added to the body_list as well. An instance of coordinate axes will be stored as one of ViewSim's attributes and it will be redrawn with a separate line in the "redraw()" method of the ViewSim:

```
this.coordinate_axes.redraw(this);
```

The trajectory as an object will be linked to the particle, therefore it is a valid child class of ViewBody. The IO handler will need a way to find trajectory and all other objects linked to a specific particle. Introducing new method: "deleteById()". It will delete all the objects with a body that has a specific id passed to the method as a parameter.

```
deleteById(id) {
    for (let i = 0; i < this.body_list.length; i++) {
        if (this.body_list[i].getBody().getId() == id) {
            this.body_list.splice(i, 1);
            i--;
        }
    }
}
```

The method Iterates through the list of bodies and deletes ones that match by the id. It deletes the element and subtracts one from the current element pointer to check that position of an array again (now it is occupied by a different element because of shift caused by removed element).

The ViewSim method for drawing a quadratic Bézier curve representing the trajectory will be called "drawParabolaBy()". It accepts three points and other attributes (like "color") as parameters and calls the "quadraticCurveTo()" method of the canvas context:

```
drawParabolaBy(starting_point, control_point, end_point, color="black") {
    this.ctx.beginPath();
    this.ctx.moveTo(...this.toCanvas(starting_point));
    this.ctx.quadraticCurveTo(
        ...this.toCanvas(control_point),
        ...this.toCanvas(end_point)
    );

    this.ctx.strokeStyle = color;
    this.ctx.stroke();
}
```

For now, default colour will be black.

As discussed in design, the Trajectory object performs required calculations in its constructor method. At the end, it leaves three attributes required to draw a Bézier curve.

The constructor starts by defining several constants:

```
this.starting_point = body.getInitialPosition();

let x0 = this.starting_point.getX();
let y0 = this.starting_point.getY();
```

```
let vx = body.getInitialVelocity().getX();
let vy = body.getInitialVelocity().getY();
let a = -9.8;
```

```
let f = (x) => {
    return vy*(x - x0)/vx + a*(x - x0)*(x - x0)/(2*vx*vx);
}
// y = f(x)

let df = (x) => {
    return vy/vx + a*(x - x0)/(vx*vx);
}
// derivative of f(x)
```

The $f(x)$ function defined in the design and its derivative $f'(x)$ (named "df" in code):

```
let x2 = 100;
let y2 = f(x2);
this.end_point = new Vector(x2, y2);
```

The end point taken arbitrarily (for now) as 100 units rightwards.

```
let m0 = df(x0);
let m2 = df(x2);
//   derivatives of two tangents

let x1 = (y2 - y0 + m0*x0 - m2*x2) / (m0 - m2);
//   x coordinate of the intersection of two tangents
this.control_point = new Vector(x1, y0 + m0 * (x1 - x0));
```

Calculating the slopes of the tangents and putting it all together to calculate position of the control point.

The redraw method simply refers to the "drawParabolaBy()" method:

```
redraw(view) {
    view.drawParabolaBy(this.starting_point, this.control_point,
this.end_point, this.color);
}
```

The trajectory object will occupy lower layers of value 0 by default and have grey colour initially.

4.7.5.2 Tests and corrections

Adding lines that create trajectory object for every new particle in the IO handler:

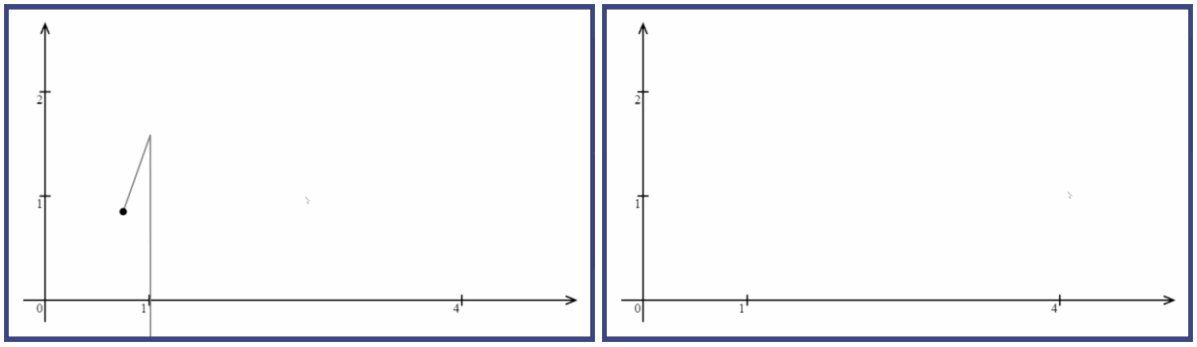
```
let trajectory = new Trajectory(particle);
this.view.addBody(trajectory);
```

Now, every particle created will have a trajectory object.

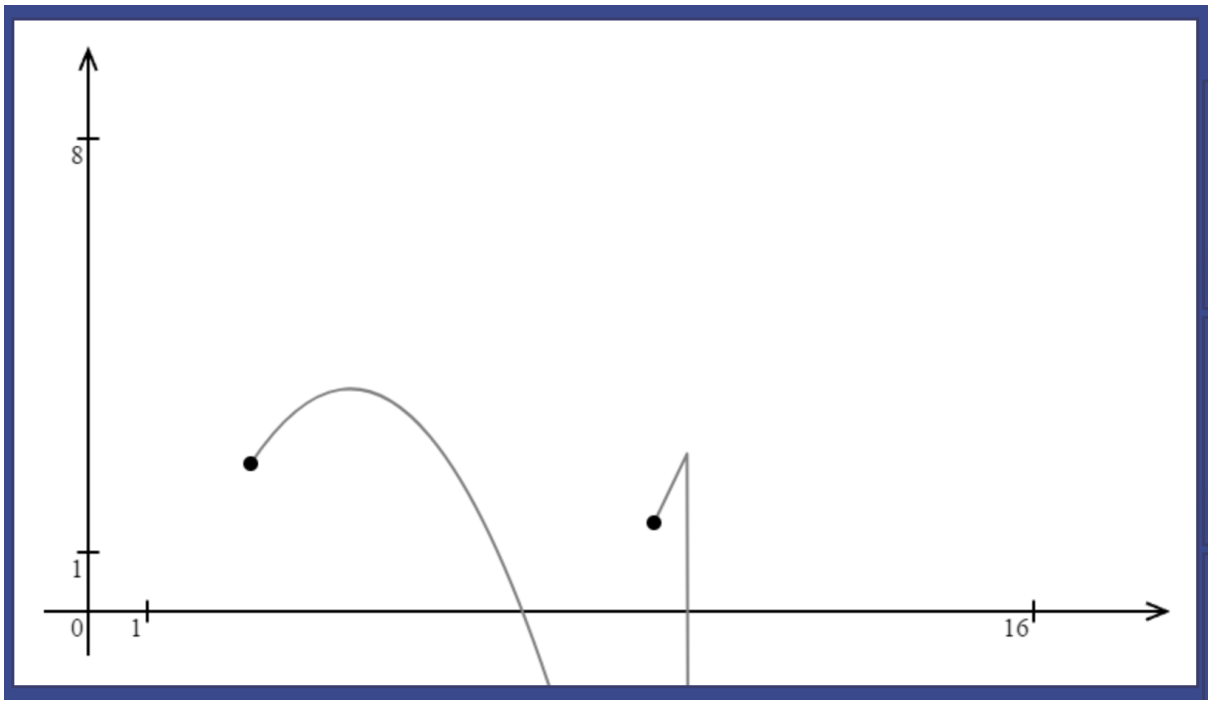
Mykyta Osovskyi
Dronfield Henry Fanshawe School

Particles indeed trace the trajectories drawn.

Making some further tests, two issues were noticed:



The first one relates to the problem of points of Bézier curve being too far away from each other. Because of that, the curve makes too sharp corners. In this case, the particle starts with small velocity by x axis and by the time it reaches 100 units by x, the downwards acceleration makes the particle too far down the y axis when 100 units by x are reached.



These two graphs correspond to the following end points:

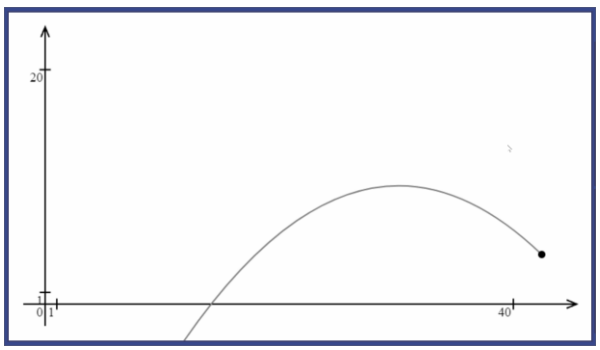
```
                                                                Trajectory.js:32
▶ Vector {x: 100, y: -4054.986111111112, Symbol(Symbol.iterator): f}
                                                                Trajectory.js:32
▶ Vector {x: 100, y: -30111.18622473234, Symbol(Symbol.iterator): f}
```

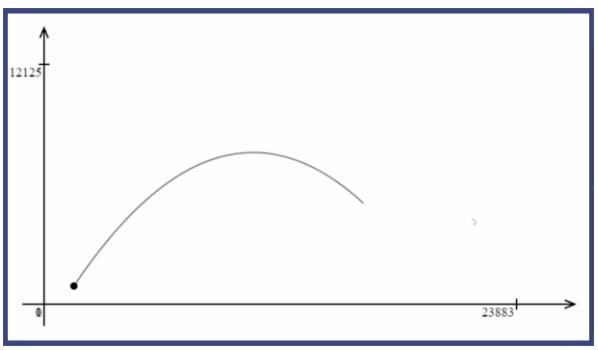
Even for the first one, by the time particle reaches 100 units by x, it is already at -4054 by y. For the trajectory that makes a sharp angle, it is over -30000. And it is even before any scales have been applied (will increase it by 40 times). The problem gets even worse when smaller scales are involved.

The second problem occurs since the end point is chosen at a fixed value of 100 by x. For particles that move leftwards, the trajectory will still be drawn to the right.

A smarter way of choosing the end point will be to choose a value that will be reached by the particle after, for example, 60 seconds of its movement as $x = x_0 + v_x t$ with $t = 60$.

```
let x2 = x0 + 60*vx
let y2 = f(x2);
this.end_point = new Vector(x2, y2);
```



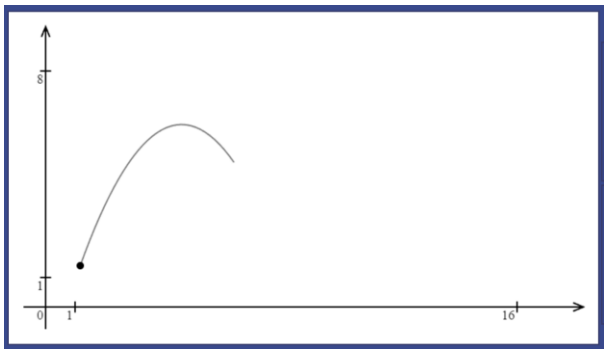


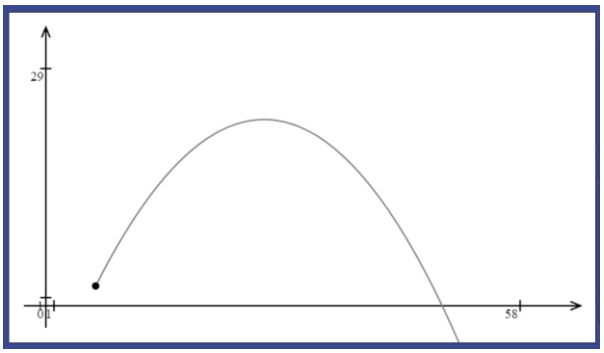
This optimisation solves issue of a particle going in different directions and minimizes the number of cases where sharp turns occur. However, for larger scales, particles move slower from the user perspective and in 60 seconds of their movement, they might not have even crossed x axis.

Therefore, it makes sense to take the scale into account when calculating how far the trajectory should extend. Sharp turns in the trajectory occur because the trajectory extends too far below. For lower scale, trajectory should extend less than for larger scales. Therefore, the value of t in $x = x_0 + v_x t$ must be inversely proportional to the scale:

```
let x2 = x0 + vx * (60/scale);
```

Looking at what results it yields:





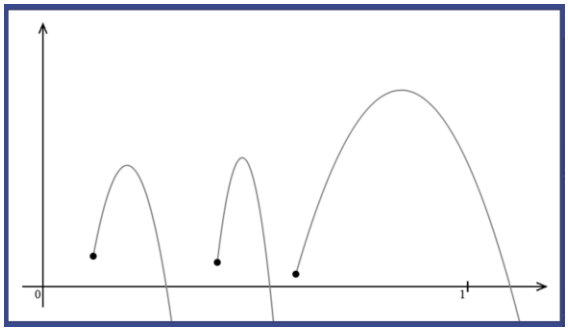

```
Unscaled: 4.901713001398495
Scaled: 196.06852005593979
```

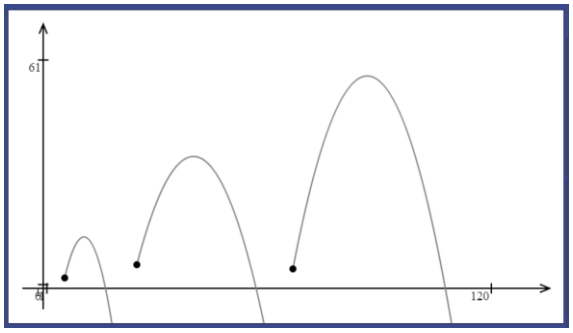
```
Unscaled: -31.755287943913316
Scaled: -354.4927227106523
```

The trajectory now indeed extends further for larger scales and smaller for lower scales, but effect is too large. To decrease it, use a square root of the scale:

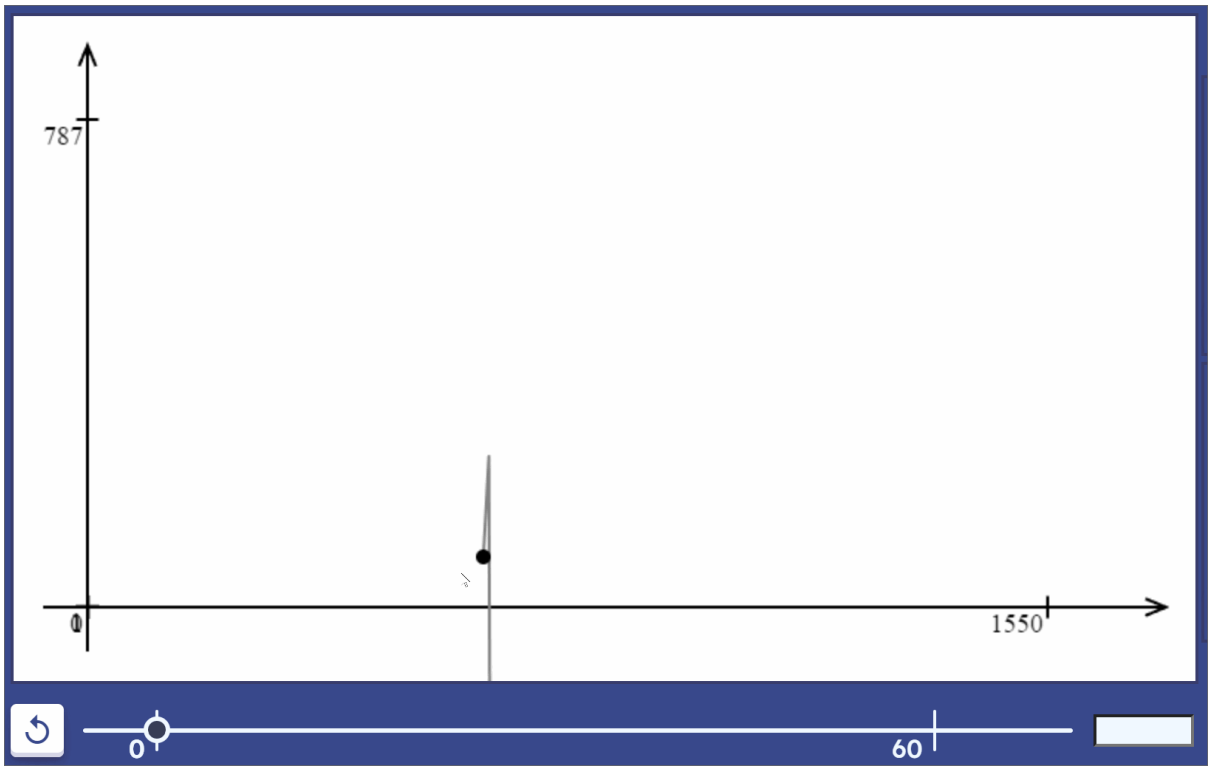
```
let x2 = x0 + vx * (60/Math.sqrt(scale));
```

Looking at what results it yields:





Now, the results are perfectly accurate for both lower and larger scales. It is still occaisionally possible to get a sharp corner, but even in these cases, the particle's movement is indistinguishable from being vertical:



4.7.6 Automatic scale adjustment

In normal use cases, it is rare that the user will have many particles at the same time on one simulation. Every time a new particle is added, the "adjustMapping()" method of the ViewSim will be invoked. Its task consists of computing the rectangle enclosing most significant points on the trajectory of a particle and changing translation and scale such that all the points are on the screen of the user.

4.7.6.1 Selecting points

The first significant point is the particle's initial position as the user needs to be able to clearly see where the particle started its path. Most problems consider particle movement in regions surrounding origin, so it will be the next point. Points where the particle crosses coordinate axes are also common subject of particle projection problems as well as the highest point on the particle's trajectory.

```
let initial_position = particle.getInitialPosition();
let list_of_points = [initial_position];
```

First point.

To determine intersections with axis, the program will need to solve a quadratic equation. Using the equation of particle trajectory derived in the Trajectory section of the Design.

$$y = y_0 + \frac{(x - x_o)v_y}{v_x} + \frac{a(x - x_o)^2}{2{v_x}^2}$$

Points of intersection with x axis are characterised by 0 value of their y coordinates. Quadratic will look in a following way:

$$\frac{a}{2{v_x}^2}(x - x_o)^2 + \frac{v_y}{v_x}(x - x_o) + y_0 = 0$$

First solve it in terms of $t = (x - x_o)$ where $a = \frac{a}{2{v_x}^2}$ $b = \frac{v_y}{v_x}$ $c = y_0$ are coefficients of the equation in standard form $at^2 + bt + c = 0$.
To get solutions for x, add x_0 to solutions in terms of t: $x_1 = t_1 + x_0$ and $x_2 = t_2 + x_0$
Required points have coordinates: $(x_1, 0)$ and $(x_2, 0)$.

Before moving on implementing this, I decided to introduce "Util" class. Several classes make use of solving a quadratic equation. This task can easily be abstracted in a function "solveQuadratic(a, b, c)" which receives coefficients a, b and c for an equation of the form ax^2+bx+c and outputs an array of two roots in ascending order. If there are no real roots, then an array of two NaN values (not a number) is outputted.

```
class Util {
    //   define as accepting a, b, c for an equation of the form
    //   ax^2 + bx + c = 0
    //   output list of two roots (in ascending order)
    //   if there is only one root, both elements are the same
    //   if there no roots, both elements are NaN
    static solveQuadratic(a, b, c) {
        let D = b * b - 4 * a * c;
        //   calculating discriminant

        if (D < 0) {
            return [NaN, NaN];
        }

        let x1 = ((-1) * b - Math.sqrt(D)) / (2 * a);
        let x2 = ((-1) * b + Math.sqrt(D)) / (2 * a);

        if (x2 < x1) {
            let temp = x2;
            x2 = x1;
            x1 = temp;
            //   swap x1 and x2 so x2 >= x1
        }
        //   two solutions where x1 <= x2

        return [x1, x2]
    }
}
```

The method is described in comments. The check for a negative discriminant can be omitted as a square root of a negative number automatically produces NaN, but I added it for consistency.

Coming back to the calculation of the axis intersection points:

```
let x0 = initial_position.getX();
let y0 = initial_position.getY();
let vx = particle.getInitialVelocity().getX();
let vy = particle.getInitialVelocity().getY();
let a = -9.8;
```

Mykyta Osovskyi
Dronfield Henry Fanshawe School

First define all constants used in equations.

```
let roots = Util.solveQuadratic(a / (2*vx*vx), vy / vx, y0);
```

Solve quadratic, using $a = \frac{a}{2{v_x}^2}$ $b = \frac{v_y}{v_x}$ $c = y_0$

```
if (!isNaN(roots[0])) {
    let x1 = roots[0] + x0;
    let x2 = roots[1] + x0;
    list_of_points.push(new Vector(x2, 0));
    list_of_points.push(new Vector(x1, 0));
}
```

If roots are not NaN (it is enough to check only the first one because if one is NaN, the second one is NaN as well), then shift them by x_0 to get coordinates of intersections.

Now define function of the trajectory, the same one was used in previous section:

```
let f = (x) => {
    return y0 + vy * (x - x0) / vx + a * (x - x0) * (x - x0) / (2 * vx * vx);
}
```

The second point is $(0, f(0))$:

```
list_of_points.push(new Vector(0, f(0)));
```

To find the highest point on the particle's trajectory, use the derivative of the trajectory function and equate it to 0. It gives a linear equation of the form bx + c:

$$f'(x_3) = \frac{v_y}{v_x} + \frac{a(x_3 - x_o)}{{v_x}^2} = 0$$

$$\frac{a}{{v_x}^2}(x_3 - x_o) + \frac{v_y}{v_x} = 0$$

To solve this equation, I will add another method to the Util class:

```
//   b, c for
//   bx + c = 0
//   output a single number
//   if no roots, output NaN
static solveLinear(b, c) {
    if (b == 0) {
        return NaN;
        //   no roots if b is zero
    }

    let x = (-1) * c / b;
    return x;
}
```

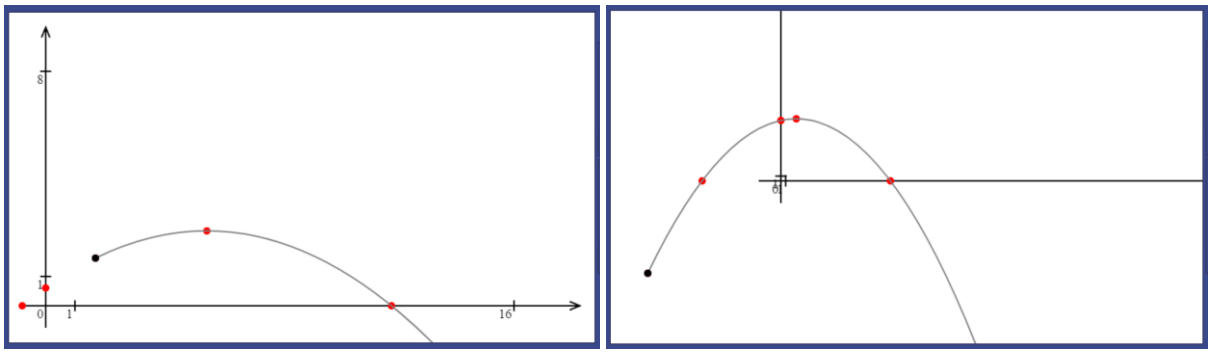
Using this method to calculate x coordinate of the highest point on the trajectory:

```
let x3 = Util.solveLinear(a / (vx * vx), vy / vx);
if (x3 != null)
    list_of_points.push(new Vector(x3 + x0, f(x3 + x0)));
```

x3 is again shifted by x0 because solutions are calculated with respect to (x3 – x0) initially.

4.7.6.2 Testing point selection

To test if the points selected are correct, I will iterate through a list of points and add them to the body list as red points, so whenever a particle is added, it is accompanied by several red dots representing the “significant” points. Running this for a couple of examples:



Point selection has proved to be correct regardless of scale and parameters of particles. It is worth noting that when the particle is below x axis, there will be no intersections with x-axis.

4.7.6.3 Identifying the rectangle

Next task is to calculate the rectangle that encloses all the significant point to then position it in the middle of the simulation space.

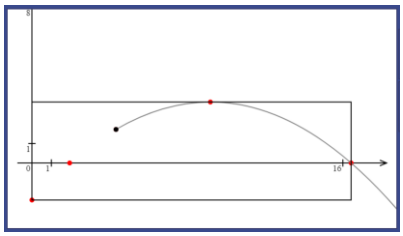
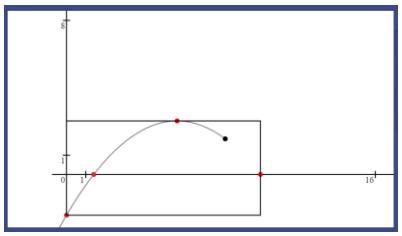
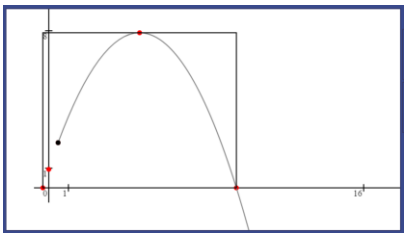
The algorithm for that was described in design: iterate through the list of points, keeping track of minimum and maximum values of x and y coordinates of these points. At the end, it will give four values that uniquely represent the rectangle:

```
let min_x = 0, max_x = 0, min_y = 0, max_y = 0;

for (let point of list_of_points) {
    min_x = Math.min(min_x, point.getX());
    max_x = Math.max(max_x, point.getX());
    min_y = Math.min(min_y, point.getY());
    max_y = Math.max(max_y, point.getY());
}
```

To quickly test if the algorithm accomplishes desired task, create a polygon with vertices at (min_x, min_y), (min_x, max_y), (max_x, max_y), (max_x, min_y)

```
let rect = new Polygon(particle.getId(), [
    new Vector(min_x, min_y),
    new Vector(min_x, max_y),
    new Vector(max_x, max_y),
    new Vector(max_x, min_y)
])
this.addBody(new VeiwPolygon(rect));
```

It will put a rectangle enclosing all points on the screen, see results:

Also worth noting that the origin, point (0, 0) is automatically included because initial values for min/max values of x/y are zeros.

Now calculating parameters of the rectangle required to centre it:

```
let width = max_x - min_x;
let height = max_y - min_y;
let rect_centre = new Vector((min_x + max_x) / 2, (min_y + max_y) / 2)

this.scale = Math.min(700 / width, 350 / height);
```

The scale is calculated using the formula described in design: $\min\left(\frac{700}{width}, \frac{350}{height}\right)$

```
let screen_centre = new Vector(this.canvas.width/2, this.canvas.height/2);
this.translation = screen_centre.added(
rect_centre.multiplied(-this.scale));
```

Finally, the screen centre is calculated as half canvas width and half canvas height vector. Now, to calculate the translation, use process defined in design: reverse rectangle centre and scale it. It can be accomplished with one multiplication by a negative value of scale.

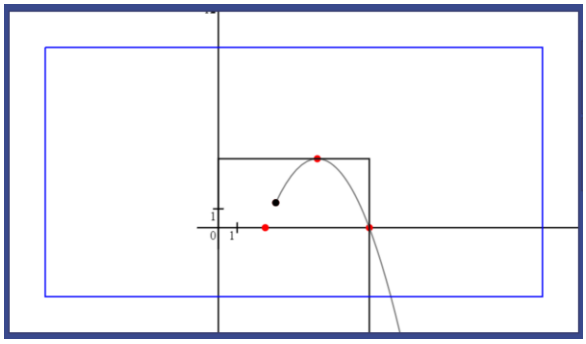
4.7.6.4 Testing entire mapping adjustment

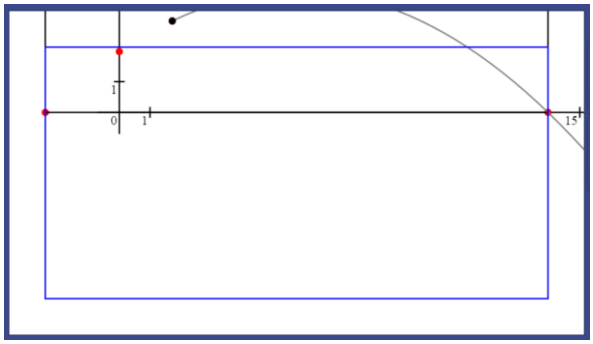
To test how simulation positions the rectangle, I will draw a blue rectangle on canvas that represents the central area of the screen within which the rectangle of “important” points on the trajectory must fit:

```
this.ctx.beginPath();
this.ctx.moveTo(50, 50);
this.ctx.lineTo(750, 50);
this.ctx.lineTo(750, 400);
this.ctx.lineTo(50, 400);
this.ctx.lineTo(50, 50);

this.ctx.strokeStyle = "blue";
this.ctx.stroke();
```

This code goes to the end of “redraw()” method. The blue rectangle will not depend on scale or translation:

Now add a particle to see how its rectangle is positioned:

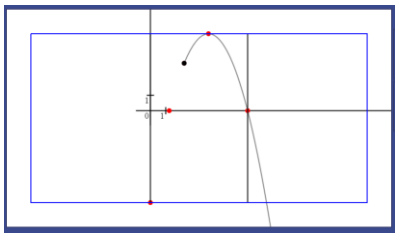


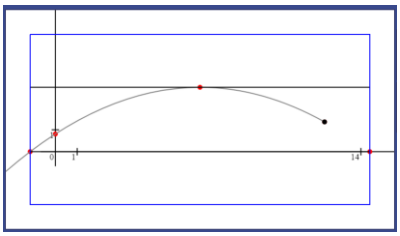


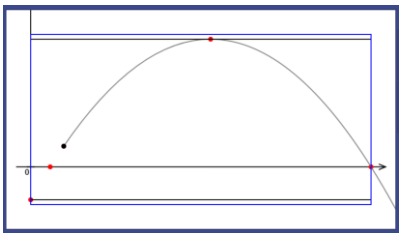
From screenshots, it is clear that translation is adjusted incorrectly. However, only in direction of y axis. It gave me a hint on the fact that canvas deals with a different direction of y-axis, so to use the rectangle centre coordinates, the vector must be reflected in x axis:

```
this.translation = screen_centre.added(
rect_centre.multiplied(-this.scale).reflectedInX());
```

Repeat the test:







The rectangle is positioned properly now. The width and the height of the rectangle of important points matches the central rectangle of the canvas.

One problem I noticed during the testing is that once the scale and translation are adjusted, the coordinate system changes drastically. Even though, trajectory of the particle is seen clearly and fully

now, the user might want to go back to the initial state of simulation. And there is no other way to do that quickly than close this simulation and open a new one. To address this problem, I will extend functionality of “set to 0 button”, which I also called “reset button”. Now, it will not only set simulation time to 0, but it will also return translation and scale to their initial values.

To make these changes, I added two new attributes to the ViewSim class: initial_scale and initial_translation.

```
this.initial_scale = 40;
this.scale = this.initial_scale;
this.initial_translation = new Vector(0 + 50, this.canvas.height - 50);
this.translation = this.initial_translation;
```

And two new methods: “resetScale()” and “resetTranslation()” which do exactly what their identifiers imply:

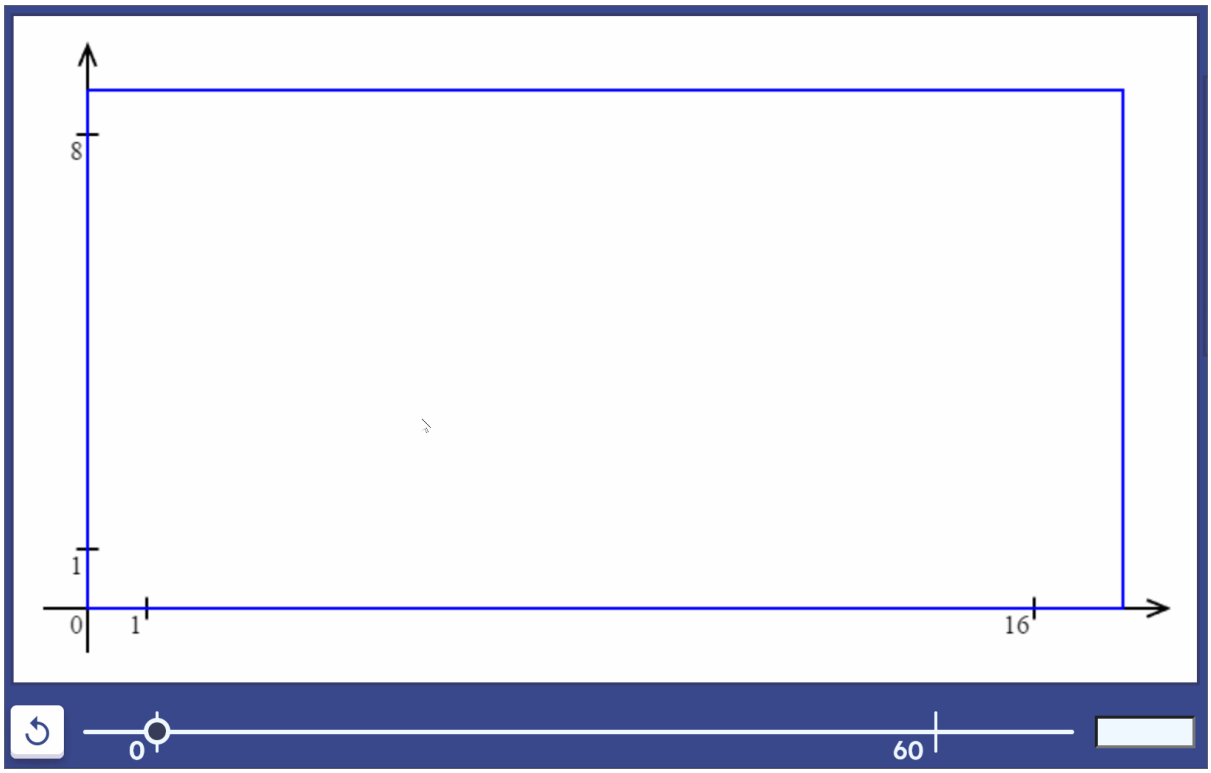
```
resetScale() {
    this.scale = this.initial_scale;
}

resetTranslation() {
    this.scale = this.initial_translation;
}
```

These two methods are to be invoked when the reset button is clicked:

```
createResetButton(time_control_container) {
    let button = document.createElement("button");
    button.innerHTML = '<i class="material-icons" style="backg

    button.onclick = () => {
        this.sim.resetTime();
        this.updateThumb();
        //   update positions
        this.view.resetScale();
        this.view.resetTranslation();
        //   reset mapping of simulation space on the screen
    }

    time_control_container.appendChild(button);
}
```

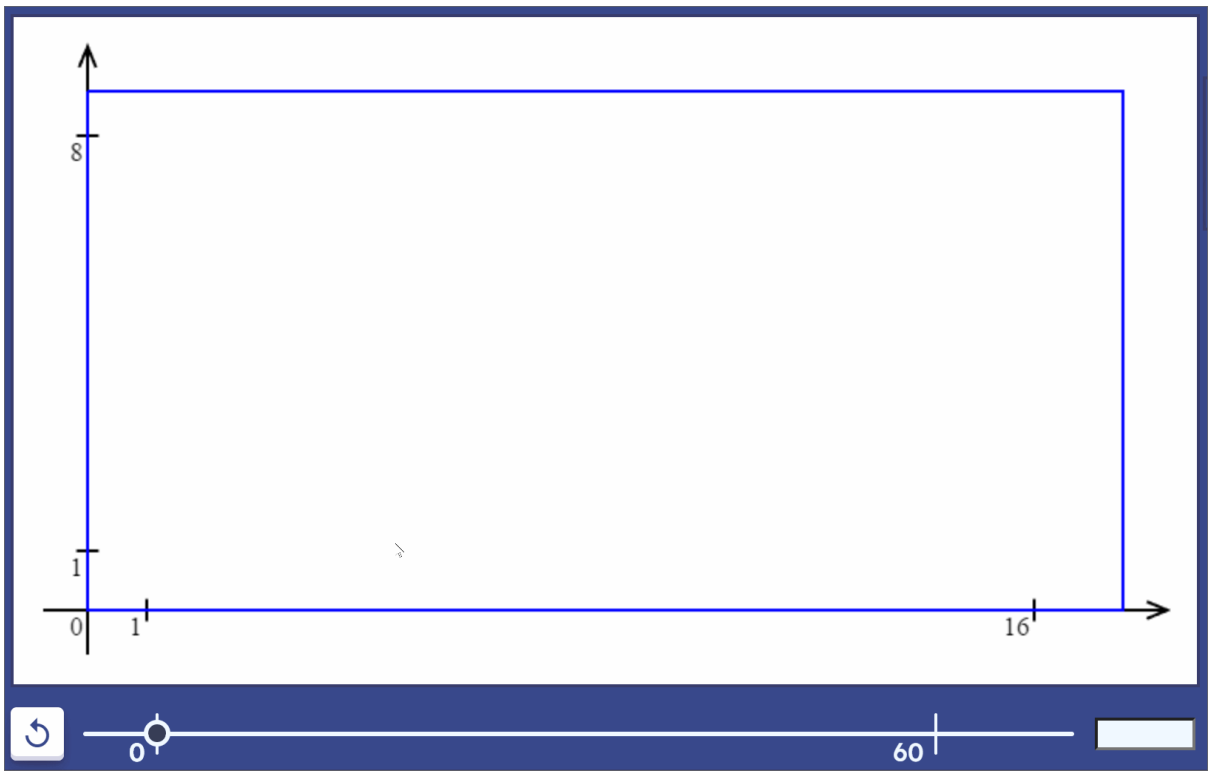
Running a quick test:

Mykyta Osovskyi
Dronfield Henry Fanshawe School



The button correctly resets time and canvas to the initial time, scale, and translation. Particles remain in their positions in simulation space.

Other problem is that when the particle crosses y axis too far below its starting point, the height of the rectangle becomes too big compared to its width, giving following mapping:



At this moment, the y intersection point becomes irrelevant and therefore can be completely discarded.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

Let the condition for when the y intersection is discarded be when distance in y axis between starting point and y intersection increases distance between starting point and highest point of the trajectory time 1.5. If y component of starting point is given as y_0, of highest point y_3 and y intersection as y_2 then the rejection condition looks like this:

$$|y_2 - y_0| > |y_3 - y_0| * 1.5$$

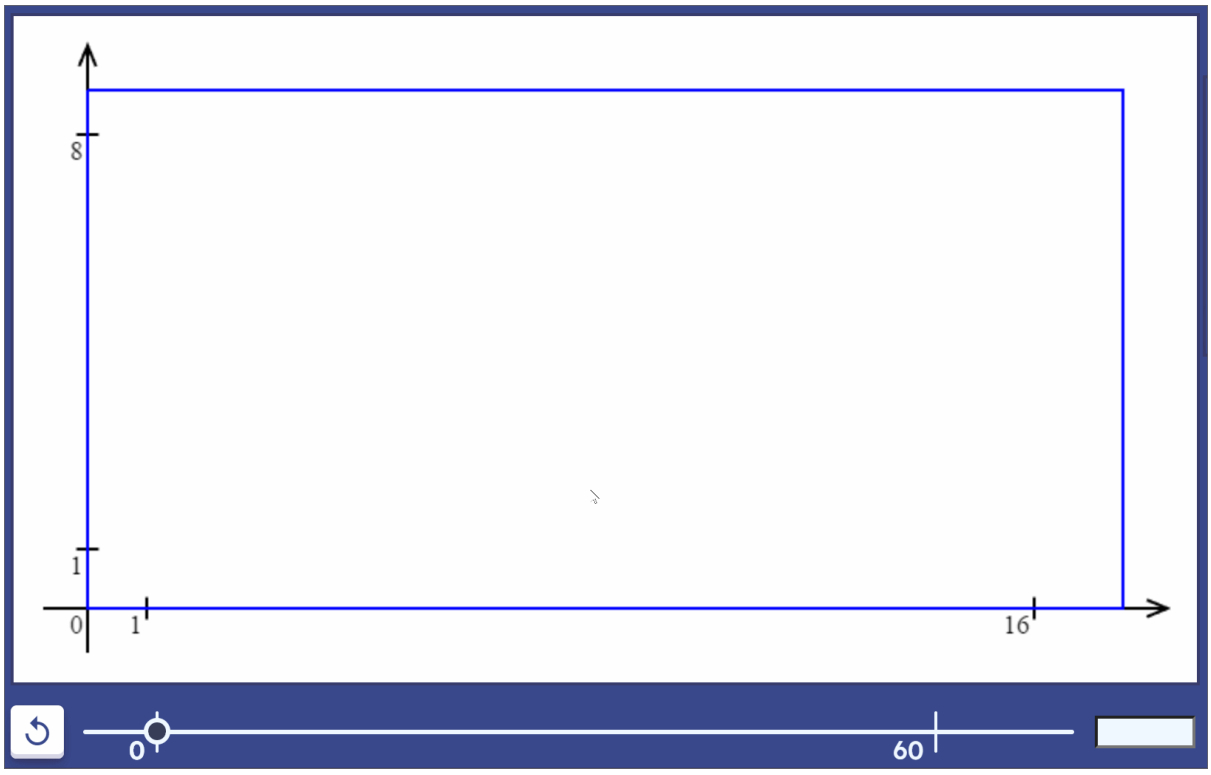
And accepting condition:

$$|y_2 - y_0| \leq |y_3 - y_0| * 1.5$$

Putting it into code:

```
let y2 = f(0);
if (Math.abs(y2 - y0) <= Math.abs(y3 - y0) * 1.5)
    //   add only if it is closer to starting point than
    //   1.5 times distance by y between starting point and highest point
    list_of_points.push(new Vector(0, y2));
//   y intersection point
```

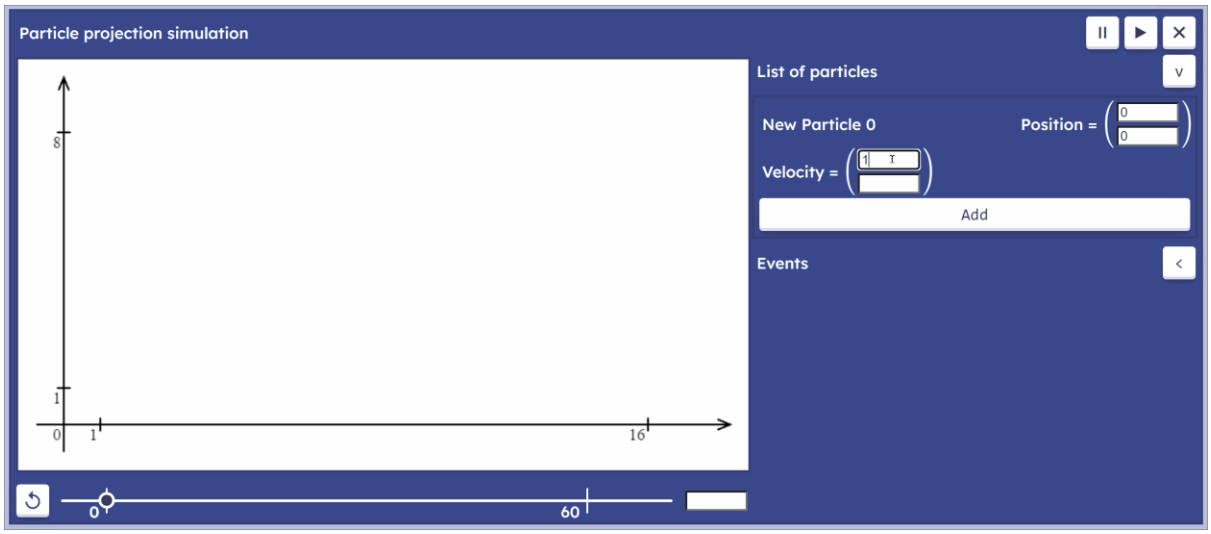
Output:



Now if the particle is placed far to the right of the origin and, as a result, has a y intersection point far below the starting point, it will not be added into rectangle. Trajectories can be clearly observed and if the user wishes, they can move around simulation, zoom in and zoom out, using their keyboard.

The last problem that must be addressed is that in some cases, all points coincide. Consider the case when a particle is launched horizontally from the origin. It crosses both axes at the origin and has its highest point there as well.

Mykyta Osovskyi
Dronfield Henry Fanshawe School



As a result, the rectangle has zero width and height. The scale is set to 0 which results in several divisions by zero.
It can be fixed by adding another point along the trajectory of a particle to the list of significant ones. Let this point be the point after the first 2 seconds of its movement. Apart from that, the position of a particle after the first 2 seconds is likely to already be in the rectangle.

The PointMass class already has sufficient functionality for that. To reuse the code, I will separate the position calculations in a separate method called “calculatePositionAt()”:

```
//   time in seconds
calculatePositionAt(time) {
    let ds = this.initial_velocity.multiplied(time);
    //   initialising ds variable that represents change in position
    //   and calculating the first part of the equation

    ds.add(this.acceleration.multiplied(time * time / 2));
    //   adding the change caused by acceleration

    return this.initial_position.added(ds);
}
```

It can be then reused for the “update()” method in the PointMass class:

```
//   time in miliseconds since simulation started
update(time) {
    let t = (time - this.created_at) / 1000;
    //   calculate time from the creation of the object
    let ds = this.initial_velocity.multiplied(t);
    //   initialising ds variable that represents change in position
    //   and calculating the first part of the equation

    ds.add(this.acceleration.multiplied(t * t / 2));
    //   adding the change caused by acceleration

    this.position = this.initial_position.added(ds);
    //   updating position

    let dv = this.acceleration.multiplied(t);
    //   initialising change in velocity variable
    this.velocity = this.initial_velocity.added(dv);
    //   updating velocity
}
```

->

```
//   time in miliseconds since simulation started
update(time) {
    let t = (time - this.created_at) / 1000;

    this.position = this.calculatePositionAt(t);
    //   updating position

    let dv = this.acceleration.multiplied(t);
    //   initialising change in velocity variable
    this.velocity = this.initial_velocity.added(dv);
    //   updating velocity
}
```

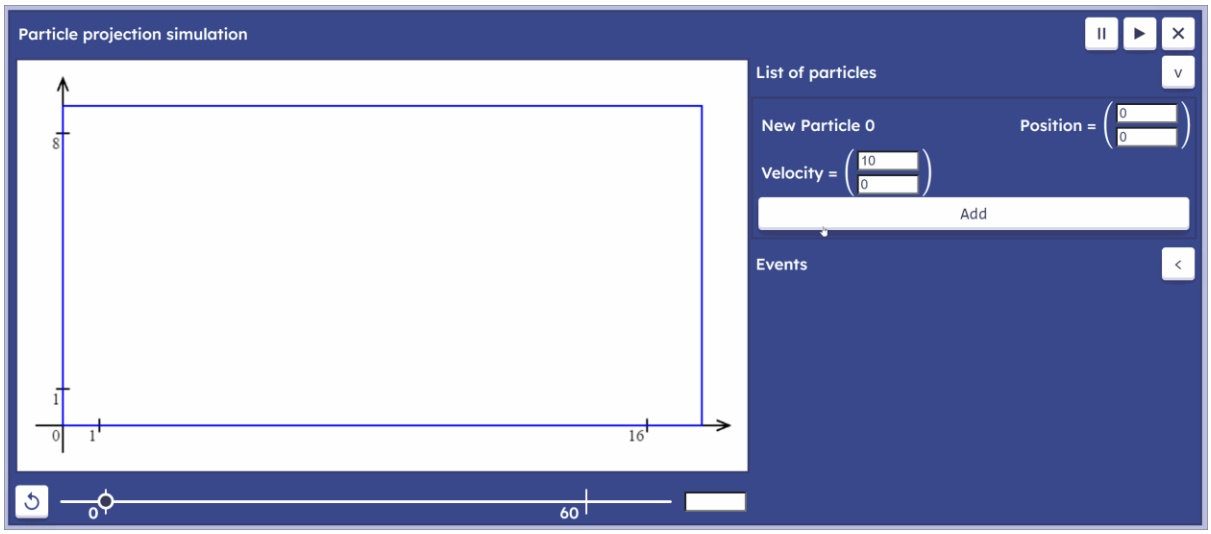
Now come back to the “adjustMapping()” function:

```
list_of_points.push(
```

```
    particle.calculatePositionAt(2)
)
// position after first 2 seconds of movement
```

Position of desired point is calculated using the same method and added to the list:

Carrying out the same test now:



It not only handles the edge case of a particle created at origin but also provides additional point for every other particle, including larger parts of trajectories for particles at small scales.

4.7.7 Smooth camera movement

Addressing the problem developed from the scale adjustment when camera (camera meaning area of simulation that eventually gets mapped on the screen) moves drastically, leaving the user confused, I decided to implement smooth camera movement.

It does not require any changes outside of the ViewSim class, because of the modular nature of the program. The main idea of algorithm that will make camera movement smoother is to keep target translation and scale values. Actual translation and scale values will be approaching their target values with each frame at some rate. Actual translation and scale will be used in drawing routines, while any changes first apply to their target values.

A simple and effective way of making actual value approach its target value is to increment actual value by a fraction of the difference between target and actual value:

$$actual_{value} = actual_{value} + \frac{target_{value} - actual_{value}}{k}$$

Where k is some constant value that defines how fast the actual value approaches its target.

First define these attributes in the constructor:

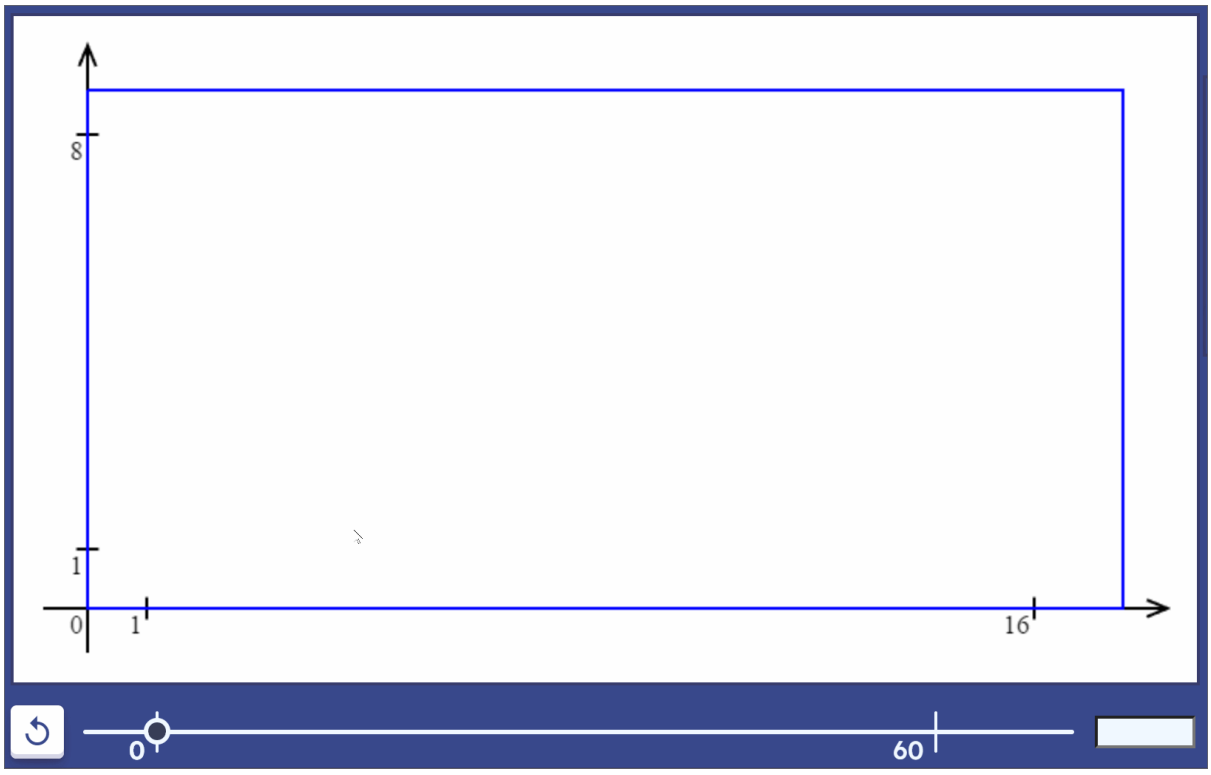
```
this.target_scale = 40;
this.INITIAL_SCALE = 40;
this.scale = 40;

this.target_translation = new Vector(0 + 50, this.canvas.height - 50);
this.INITIAL_TRANSLATION = new Vector(0 + 50, this.canvas.height - 50);
```

```
this.translation = new Vector(0 + 50, this.canvas.height - 50);
```
Initial values remain unchanged and are constants.

Putting this change into ViewSim class, every instance where translation and scale were used for drawing routines must be left with their actual values represented by “translation” and “scale” attributes respectively. It applies to methods: “getScale()”, “toCanvas()”, “toSimSpace()”, “scaled()”, “inversedTranslated()” (there are more drawing methods, but they use one of the methods listed above).
On the other side, every method where scale and translation are modified, must use their target values: “resetScale()”, “resetTranslation()”, “setScale()”, “increaseTranslationBy()”, “increaseScaleBy()” and “adjustMapping()”.

The actual values for translation and scale will be changed at the start of the “redraw()” method using the formula for actual values approaching their targets with a value of 15 for k:

```
this.scale += (this.target_scale - this.scale) / 15;
this.translation.add(
this.target_translation.subtracted(this.translation).divided(15));
```
Run the code for a test:



Rate at which actual value approaches its target becomes slower when actual value is closer to the target. The value of 15 for constant k seems to do a good job making the rate of approach pleasurable visually. However, it may be easily altered at any moment if desired.

My algorithm may not be perfect, but judging from all tests, it achieves smooth camera movement. Its advantage is that the rate at which actual values approach their targets is dependent on the gap between them. Therefore, the larger the gap, the quicker it will be eliminated and when two values are closer, they will be approaching each other slower, creating an effect of smoothness.

4.7.8 Brief instructions text

This is the last section where I will acknowledge all features added throughout development and compile a brief instructions text that is meant to guide the user on the interface.

Starting with most important information:
“To start, pick a simulation from the drop-down list below and click ‘+’ button on the right.”

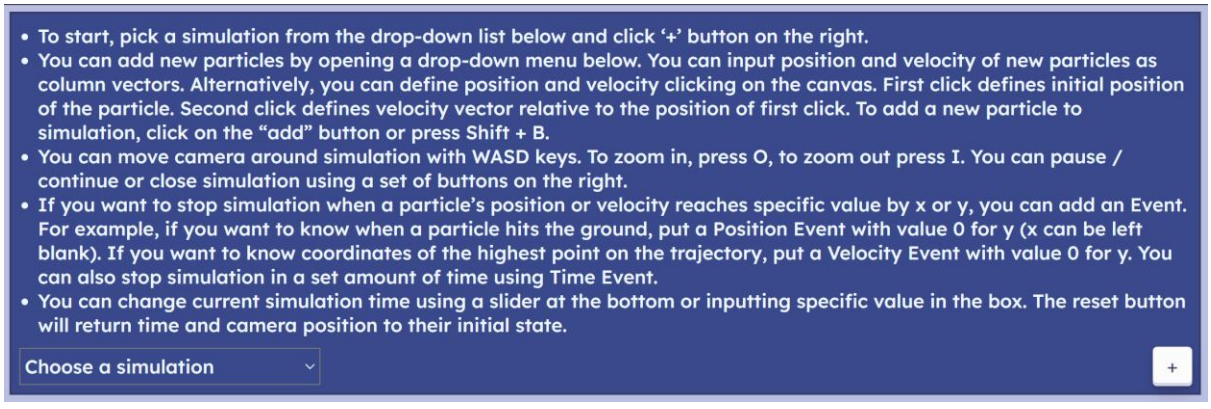
Next talk about particles:
“You can add new particles by opening a drop-down menu below.”
“You can input position and velocity of new particles as column vectors.”
“Alternatively, you can define position and velocity clicking on the canvas.”
“First click defines initial position of the particle.”
“Second click defines velocity vector relative to the position of first click.”
“To add new particle to simulation, click on the “add” button or press Shift + B.”
(from most important / unintuitive pieces of information to less important / obvious)

Camera movement and other controls:
“You can move camera around simulation with WASD keys.”
“To zoom in, press O, to zoom out press I.”
“You can pause / continue or close simulation using a set of buttons on the right.”

Event system:
“If you want to stop simulation when a particle’s position or velocity reaches specific value by x or y, you can add an Event.”
“For example, if you want to know when a particle hits the ground, put a Position Event with value 0 for y (x can be left blank).”
“If you want to know coordinates of the highest point on the trajectory, put a Velocity Event with value 0 for y.”
“You can also stop simulation in a set amount of time using Time Event.”

Time control:
“You can change current simulation time using a slider at the bottom or inputting specific value in the box.”
“The reset button will return time and camera position to their initial state.”

Organise instructions in an unordered list in HTML and apply some the same style as the one for “p” and “label” tags. Additionally, set padding-left to 18px for bullet points. It gives following result:



5 Evaluation

5.1 Structure

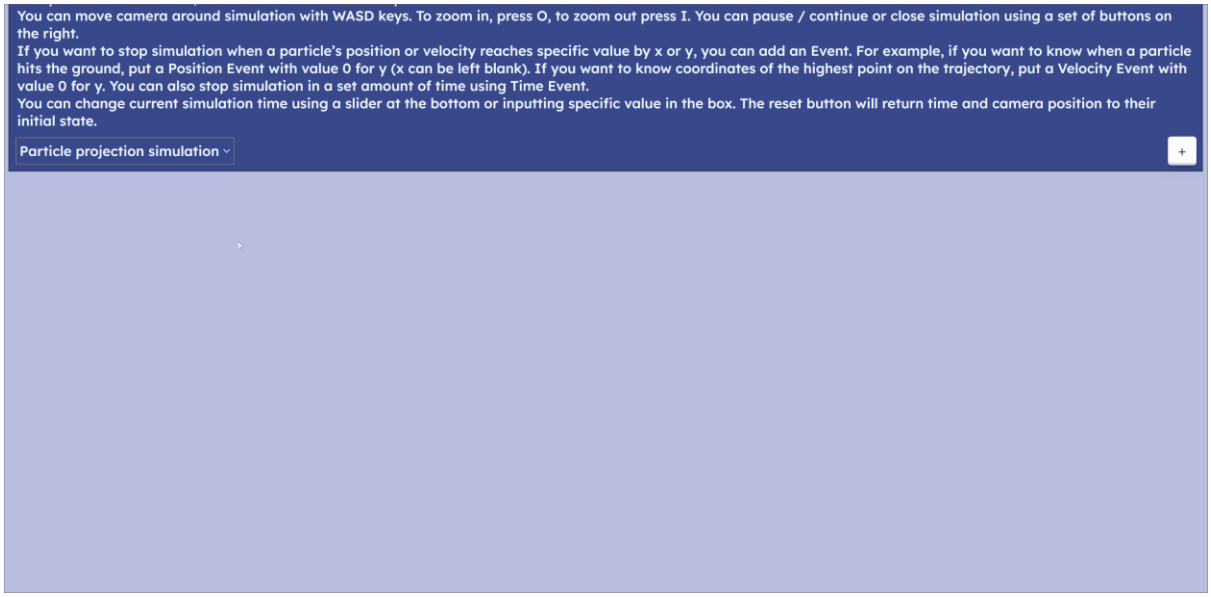
This section describes the structure of the Evaluation section.

- I will start evaluation with post development tests of the system as a whole, going over each initial objective from the success criteria. I will provide test evidence explaining how each criterion has been met, partially met, or not met commenting on how it can be addressed in future.
- Then I will carry out beta testing with my stakeholders and go through usability features providing evidence and comments on how successful each feature was and how any issues can be addressed in future. At the end, I will carry out tests for robustness, by spamming buttons, and trying to input extreme and erroneous values.
- Then I will discuss any limitations discovered throughout the project and how future improvements can be done to address them.
- Lastly, I will consider further maintenance of my project.

Features for general-purpose simulation are not considered as I dropped this part of initial solution entirely as I decided to dedicate all time to enhancing particle projections simulation and general features like camera movement. I believe these have the greatest usability potential.

5.2 Acceptance testing

5.2.1 Multiple simulations at a time



Illustrated by the gif, multiple simulations are opened and each of them functions independently as I created simulations, added particles, paused, and deleted simulations, only simulation I worked with was affected. Each of them was advancing in time separately.

Stakeholders, however, suggested that in the current structure of the webpage, it is hard to effectively use multiple simulations as you need to scroll down to see the canvas of the second simulation and use its IO area.

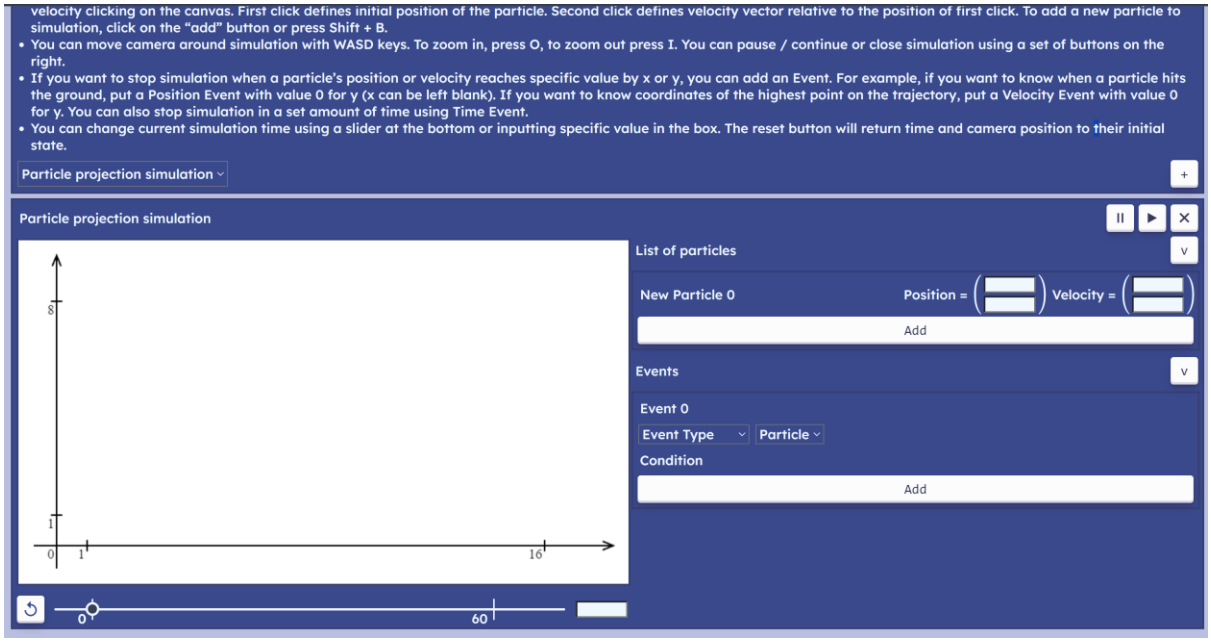
Success criteria was partially met.

Even though the technical side is solid and works as intended, there is still work to be done on the usability part. Simulations can be rearranged in a different way, e.g., have two simulations next to each other with input area underneath.

Button to stop and continue all simulations at once can be added. It would make differences between two simulations more obvious to notice.

5.2.2 Simple interface

This success criterion is assessed from usability testing which carried out later. Here I will leave general comments by myself and my stakeholders.



I tried to keep interface as simple as possible while retaining all planned functionality. Some parts of the interface are quite complicated to use and can probably be made easier to use.
Instructions are better to be substituted with video explanations and ideally, prompts when the user first uses the program. The assistant would highlight different parts of the interface and explain how to use them. As part of the users consists of children, explanations can be given by a virtual assistant in step-by-step manner, like a Microsoft Office Assistant and Clippy.

The particle output block might be taking a bit too much space for each particle. The time text input has no labels and error messages which must be changed in future. However, invalid input will be rejected and will not affect simulations. Most feedback for interface is gathered from the stakeholders during usability testing. It is summarised here.

Success criteria was partially met.

In terms of colours, blue colour is considered the most colour-blindness-friendly colour. However, no specific palette was picked to increase accessibility for colour blind people.

I would say that current interface provides a lot of functionality, however there is still a lot of work to be done on making it simpler and more intuitive. There should be more comprehensive introduction to the system when the user first opens the page. There is not yet much support for

mobile devices. Accessibility needs of colour-blind people or people with special needs are not considered enough.

5.2.3 Static web application

The program is currently hosted using the “GitHub Pages” service. The code for my project is stored at my GitHub repository: https://github.com/kij-exe/2dSimulations. The website can be accessed with the following link: https://kij-exe.github.io/2dSimulations/.

There is no server-side processing, so when the user opens the page, all the code is downloaded on their computer. It runs locally on the browser. The page can be opened and used from any browser and device, even though it may be hard to use it for some device types (this is considered when testing for robustness).

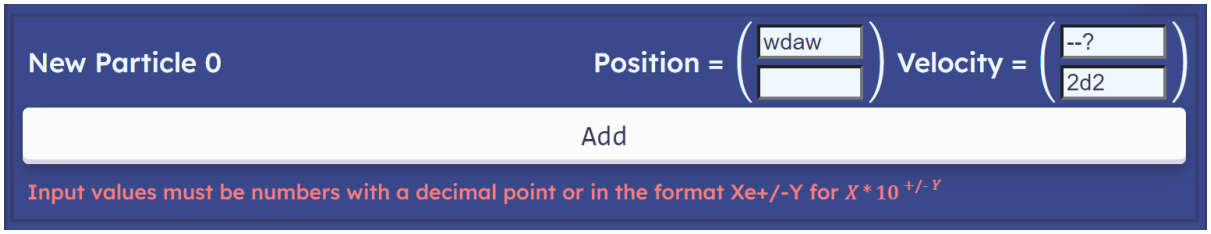
Success criteria was fully met.

All requirements for the feature are met, however using “GitHub Pages” as a hosting service is a temporary solution. Ideally, the website will be hosted on a rented server. It would provide more low-level features to its management and allow to include some server-side processing features to the program. For example, the program can potentially transform into more sophisticated learning tool with an account system. The users will be able to use simulations to solve a series of problems, saving their progress.

5.2.4 Simulation objects system

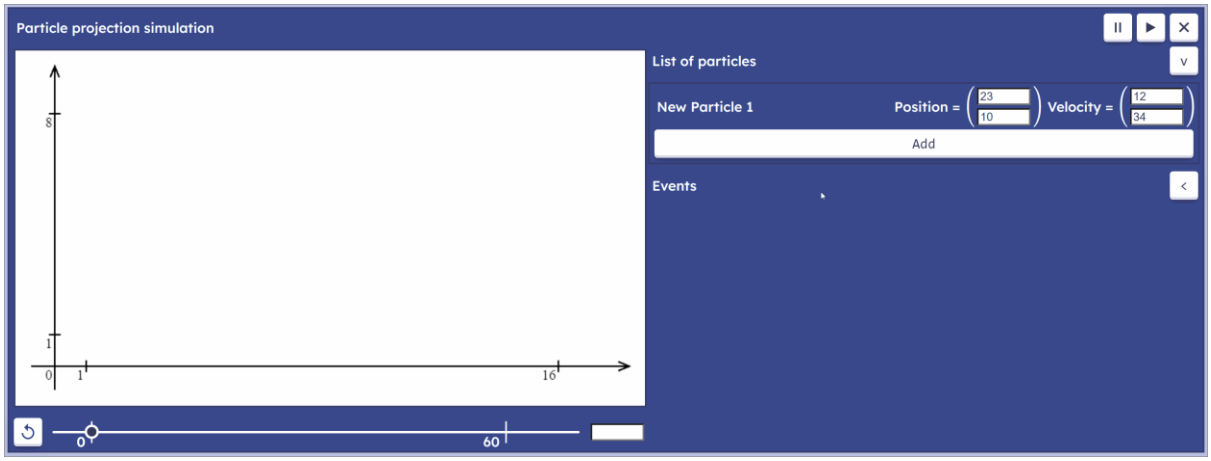
The menu for adding new objects (particles) is opened with the press of a button which changes a little sign from “<” to “v” indicating folded and opened states:

When the user attempts to pass input with one of the boxes empty or not in correct format, correct error message is displayed:



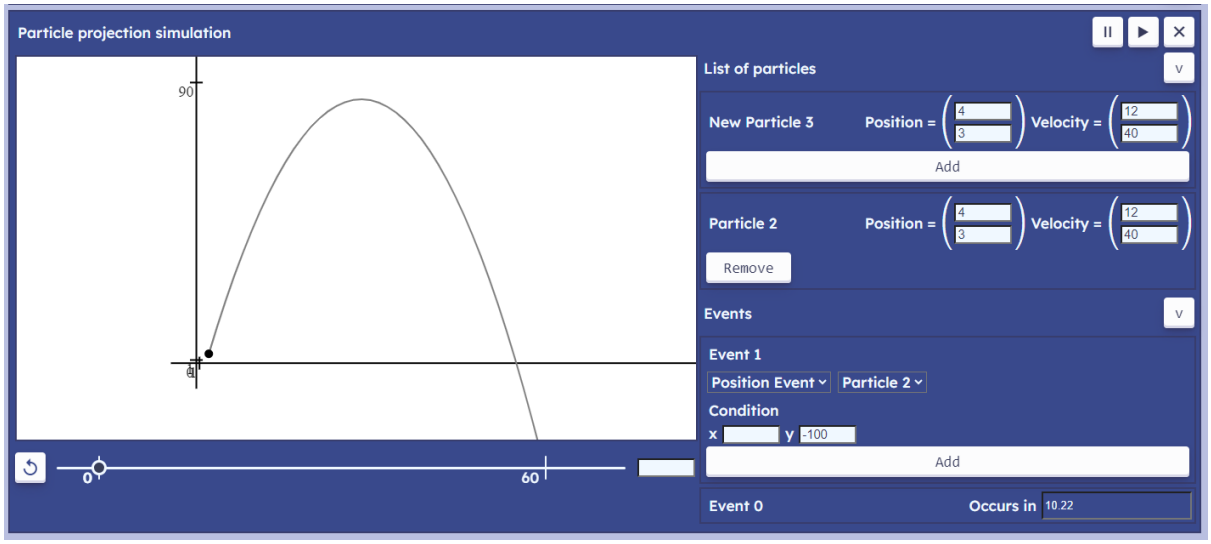
If one of the values is too large:

Finally, when correct input is processed, the particle appears on the screen and output block is added:



This GIF shows how when the new particle is added, it immediately appears on the screen and output block is created. When simulation is active, the particle is being updated from frame to frame. When the particle is deleted, it no longer appears on the screen and trajectory related to the particle is also deleted.

To test if the particle is updated consistently regardless of frame rate, I will run my program on computers with different refresh rate as it defines how frequently the website invokes callback of the “requestAnimationFrame()” method.
First, I will define the same particle for every test: position (4, 3); velocity (12, 40)
Then, I will define a position event that occurs when particle reaches -100 units by y.
It creates following set-up:



Simulation calculates Position event to occur in 10.22 seconds (rounded to 2dp). Next, I will register the moment in time I press continue button and the moment in time, event occurs using:

```
console.log(new Date().getTime());
```

To get the time elapsed between these moments, subtract first value from the second one and divide by 1000.

To verify the frame rate, I call following method in the update method of the Controller:

Mykyta Osovskyi
Dronfield Henry Fanshawe School

```
test(dt) {
    console.log("Current fps: " + Math.round((1000 / dt) * 10) / 10);
    //   dt is the number of miliseconds between two consecutive frames
    //   to get the number of frames per seconds, divide 1000 by dt
}
```

Outputs fps rounded to 1dp.

First test:

Refresh rate: 60 Hz FPS: 59.5 to 60.2

Resulting values:

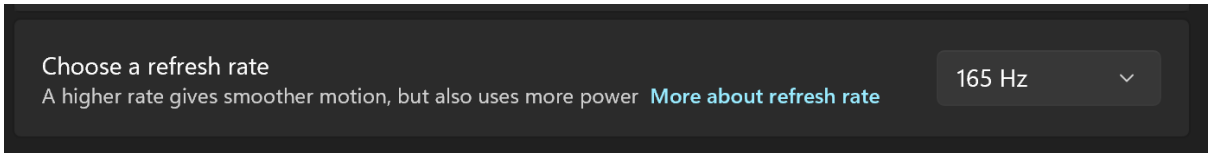
```
1710933906993                    Simulation.js:20
1710933917213          ParticleProjectionSim.js:26
```

Calculating time elapsed:

```
> (1710933917213-1710933906993)/1000
< 10.22
```

As expected, exactly 10.22 seconds.

Second test:



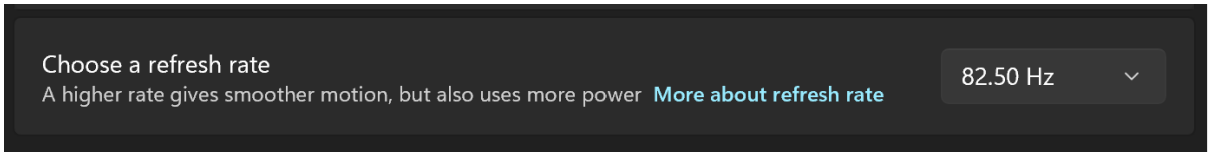
Refresh rate: 165 Hz FPS: ~165

Resulting values:

```
1710959161147                    Simulation.js:20
1710959171369          ParticleProjectionSim.js:26
> 1710959171369-1710959161147
< 10222
```

10222 milliseconds which is 10.222 seconds, off just by 2 milliseconds.

Third test:



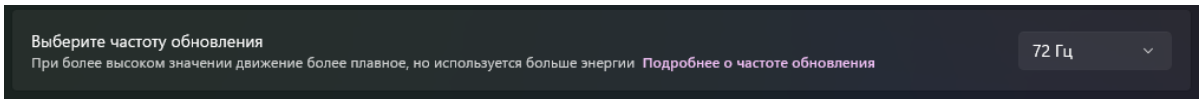
Refresh rate: 82.5 Hz FPS: 82 to 83

Resulting values:

```
1710959390905                    Simulation.js:20
1710959401135          ParticleProjectionSim.js:26
> (1710959401135-1710959390905)/1000
< 10.23
```

10.23 seconds

Fourth test:



Refresh rate: 75 Hz FPS: ~75

Resulting values:

10.2 seconds

For a wide range of refresh rates, simulation proves to keep up with real time very accurately. This covers all points mentioned in success criteria.

Success criteria was fully met.

Nevertheless, stakeholders emphasise the need for speed and angle input for velocity that was not eventually added. This is one of the first potential improvements to the system.

The text boxes currently do not accept empty input. It might be beneficial to set the default value of the text box to 0 as it is a popular type of input. This is not significant, however might save users a bit of time.

5.2.5 Maths equations for input boxes

This feature was omitted by the end of design. I viewed other areas of solution as more important. Nevertheless, this is a very useful feature, and it makes program more usable (as was suggested by the stakeholders multiple times). It would be one of the first ones to be developed in future.

Shunting yard algorithm can be used to parse user input expressions given in infix notation (regular notation, e.g., 2*(5/3)) into postfix notation, also known as Reverse Polish Notation (RPN), or an Abstract Syntax Tree (AST), which can be then computed. After that a simple algorithm can be used to compute the result of the expression.

For example, using RPN:
- iterate through the infix notation from left to right;
- if it is a number, push it on the stack;
- if it is an operator, pop two numbers from the stack and perform operation on them, pushing result back on the stack;
- continue till the end of the notation.

The algorithm would need support trigonometric functions and brackets in expressions as they are very commonly used in mechanics problems.

Other than that, the expression passed by the user will need to be validated. For instance, division by zero, invalid use of brackets or operators, etc.
Preferably, the user will need to receive a prompt on what part of their expression is not valid and why.

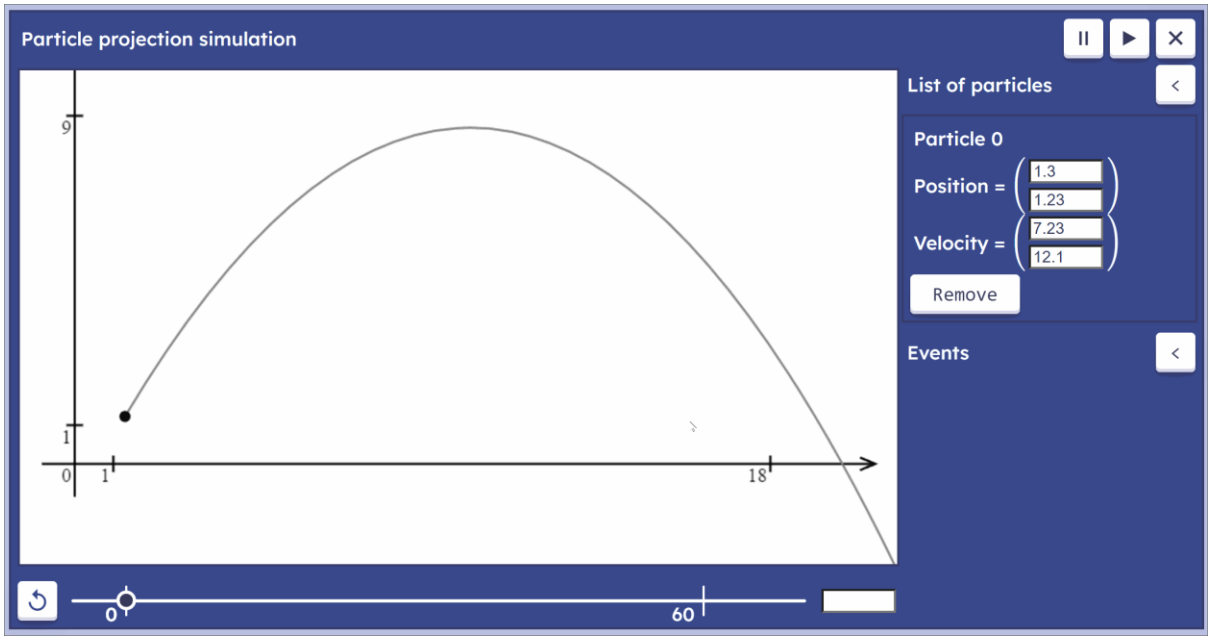
Mykyta Osovskyi
Dronfield Henry Fanshawe School

Success criteria was not met.

5.2.6 Time control system

An important part of the time control system is to keep up with real time when simulation is active and do so regardless of frame rate. It was tested in previous section and tests proved this part to be successful.

Buttons:

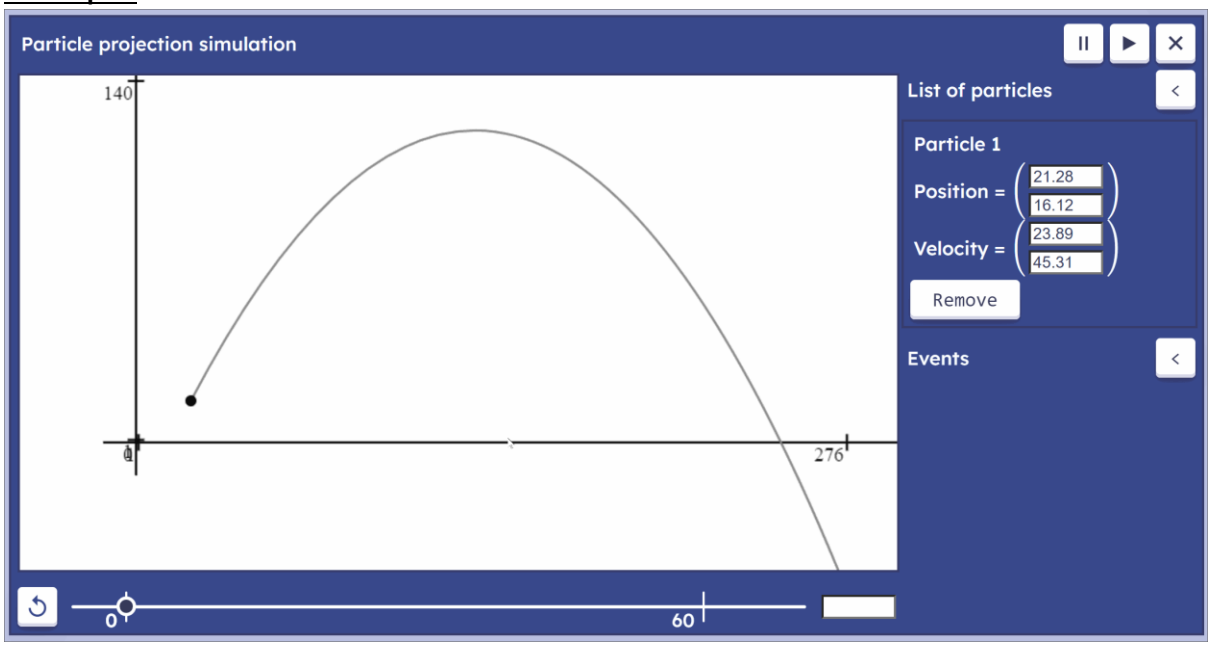


This GIF shows that buttons function properly. The callback is triggered on the button release. Throughout development, the reset button was altered slightly. It not only resets simulation time, but also resets translation and scale. To keep a way for a user to reset only time specifically, I left a “Backspace” key input that does exactly that.

Success criteria was fully met.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

Text input:

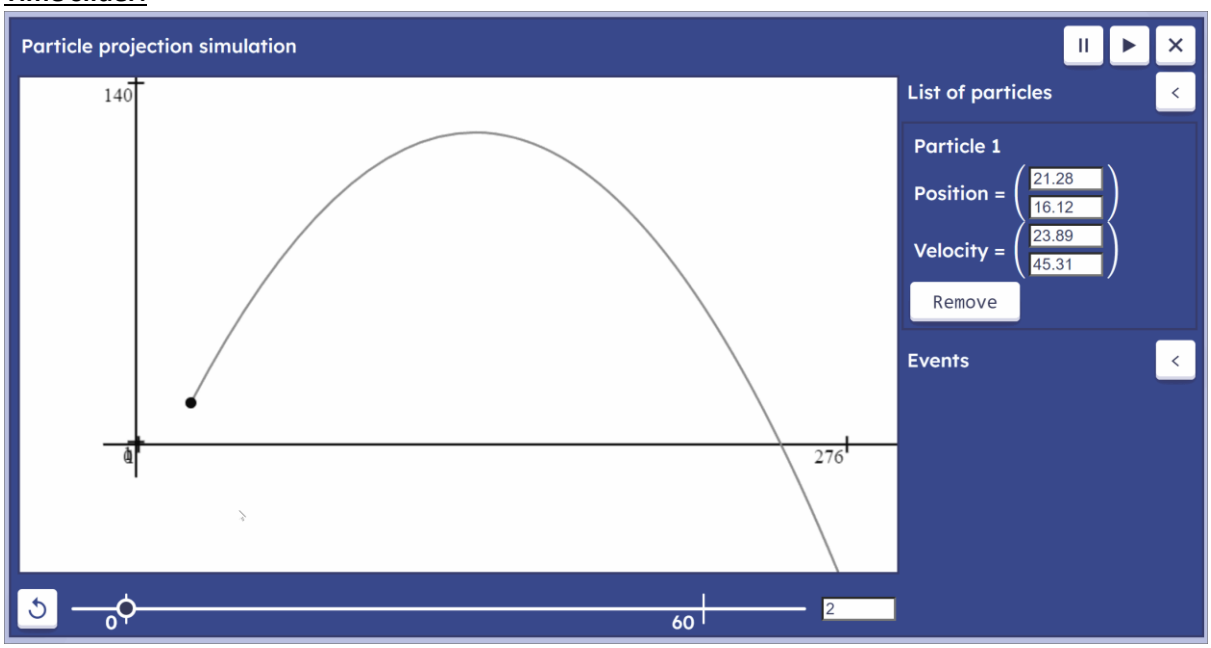


The GIF shows that text input is received correctly. Changes immediately apply to simulation and time slider as the time is changed. Negative time input is permitted for the case when a user wants to see how the particle might have behaved before it reached the starting point.

However, erroneous and outbound input is not prompted to the user as it happened with input for particle parameters. There is also no label to indicate that this text box is for time input. This might mislead the user and it must be addressed in future.

Success criteria was partially met.

Time slider:



The GIF shows how the simulation time is altered when the thumb is moved by the user. The label under the thumb changes dynamically and moves with the thumb keeping synchronised with current

time in simulation. If the time goes beyond the slider, the thumb remains just at the end of the slider, never going beyond it:

The thumb sticks to major tick marks of 0 and 60 when it is close to them.

Success criteria was fully met.

The major drawback of the slider is that it has a static range of -5 to 70 seconds. In most simulations, the model of a particle is unlikely to be used for that long. Therefore, it makes it hard for users to specify exact time they want with the slider (text input must be used).

A future improvement will be to make the range of a slider dynamic. When the particle is created, simulation will estimate the largest amount of time the model will work for and adjust values of the slider accordingly. Alternatively, the user will specify the time they want to appear as a second tick mark.

The label of current time overlaps with the reset button when it is on the left and with tick mark labels when it is close to them. This is also not a huge problem, but it would be good to avoid such collisions of different interface parts. To address that, the position of a thumb label can go on top of the slider when it is near to static labels.

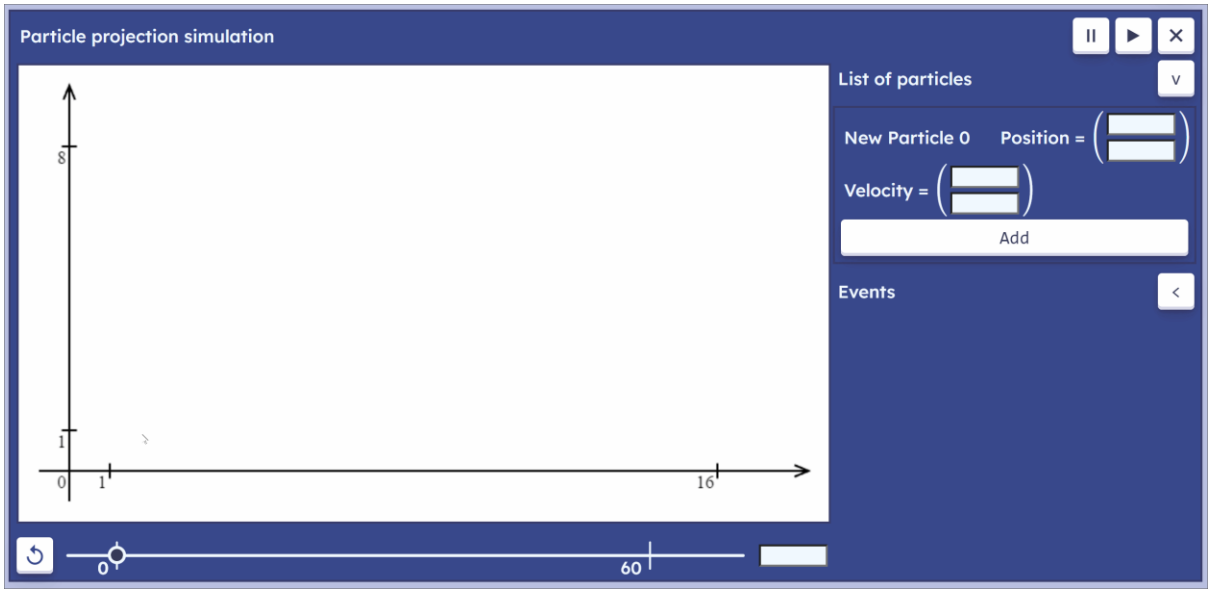
Simulation speed adjustment:
This feature was not implemented entirely.

Success criteria was not met.

The feature would require keeping a constant value in the simulation class which would be multiplied by the real time elapsed between two consecutive updates in the update method. In this way, to make simulation 2 time slower, the constant will be equal to 0.5; to speed up simulation 2 times, the constant will need to be equal to 2 and so on. This value would have been changed by a setter invoked from the IO Handler class which will provide a drop-down menu for a user to choose one of the modes.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

5.2.7 Camera movement



The GIF illustrates how “camera” is being moved using WASD keys. Every object of the simulation obeys the movement. This is achieved with centralised mapping system which uses scale and translation to convert between simulation space and canvas coordinate system. In this way, simulation remains unaffected by user inputs.

During the development process, I also added an ability to speed up movement when holding Shift key. It further extends usability of this feature.

Additionally, the camera is moved when the particle is created because of the automatic scale adjustment. The camera is moved smoothly, so the user understands what changed after the button has been pressed.

Initially defined points from the success criteria are satisfied.

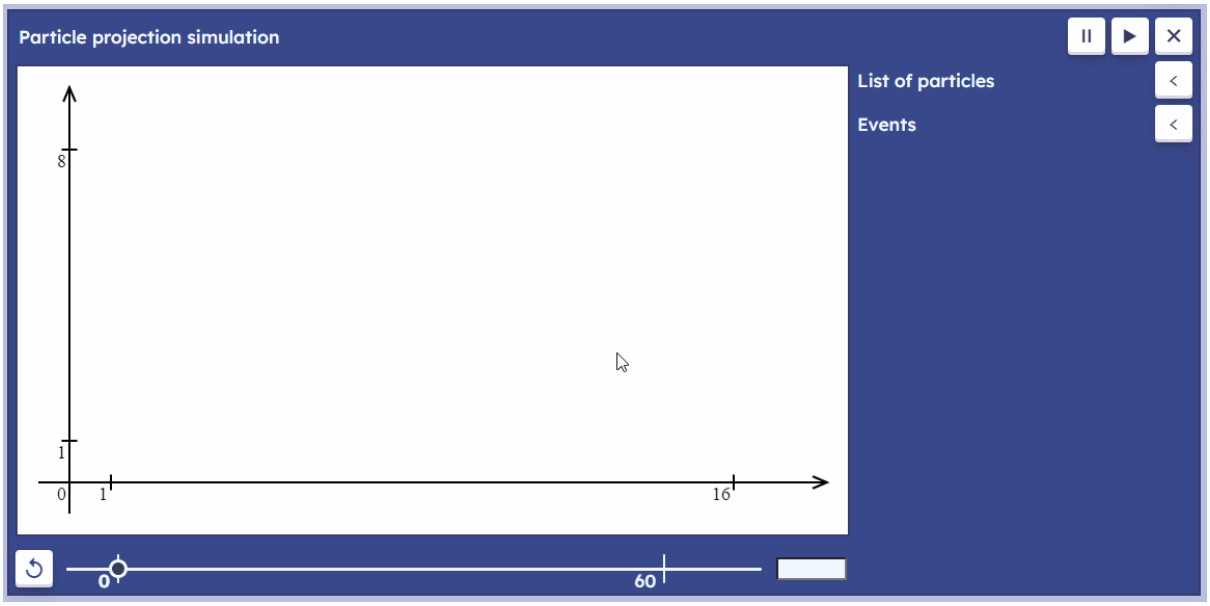
Success criteria was fully met.

Potential improvements:

In current system, if you hold a key, there is a slight delay before the camera is moved. This delay is enforced and helps to prevent multiple events firing unintentionally. For instance, when you press Space key to stop simulation, but the action is executed twice, and simulation is not stopped eventually. This issue can be fixed by separating logic for camera movement button into an update method of IO handler, so there will be no such delay. Other keys like Space for stopping simulation and Backspace for resetting the time must be left as they are.

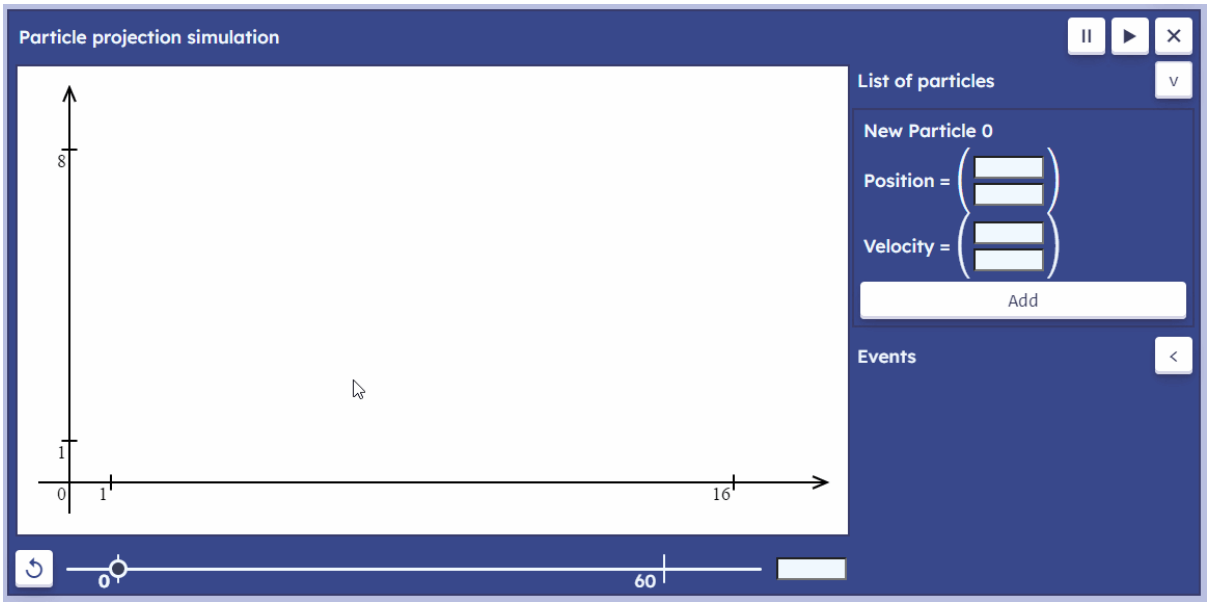
Mykyta Osovskyi
Dronfield Henry Fanshawe School

5.2.8 Scale adjustment



The GIF shows how scale is being adjusted on user input. Development tests validate manual scale adjustment system.
Smooth transition is provided together with smooth camera movement when translation and scale slowly approach their target values.

Apart from manual scale adjustment, which was mentioned in success criteria, I also developed an automatic one, which is executed every time new particle is put in a simulation:



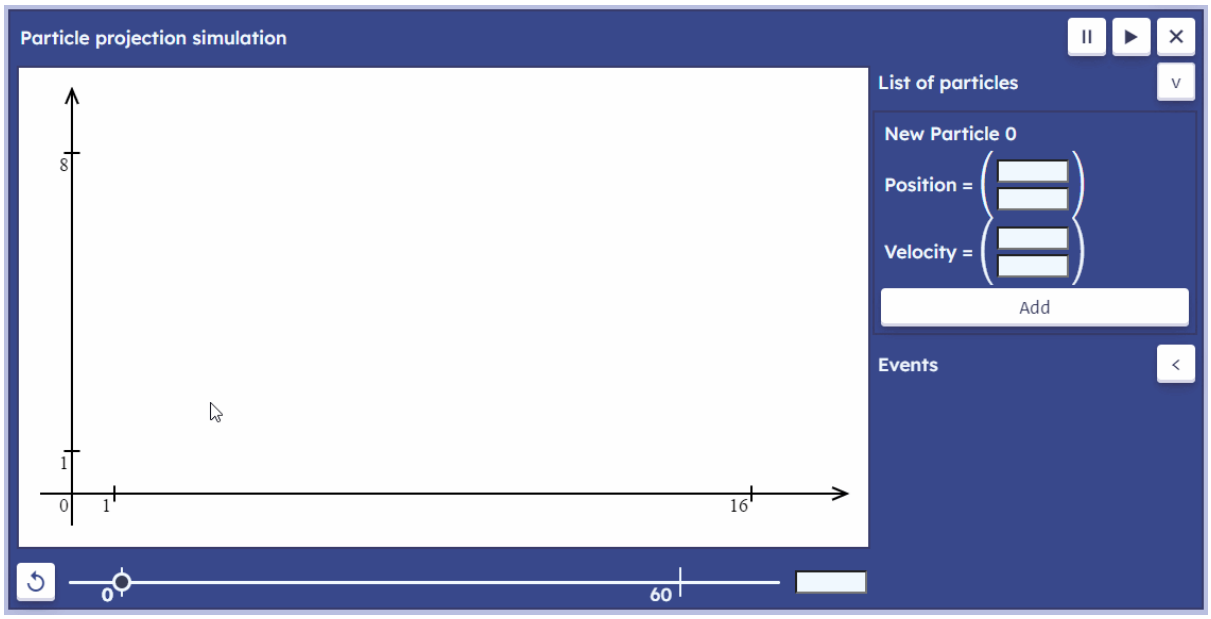
It considers most relevant points on the trajectory of the particle and adjusts scale and translation such that all of them are in the middle of the screen.

Therefore, functionality satisfies the success criteria.

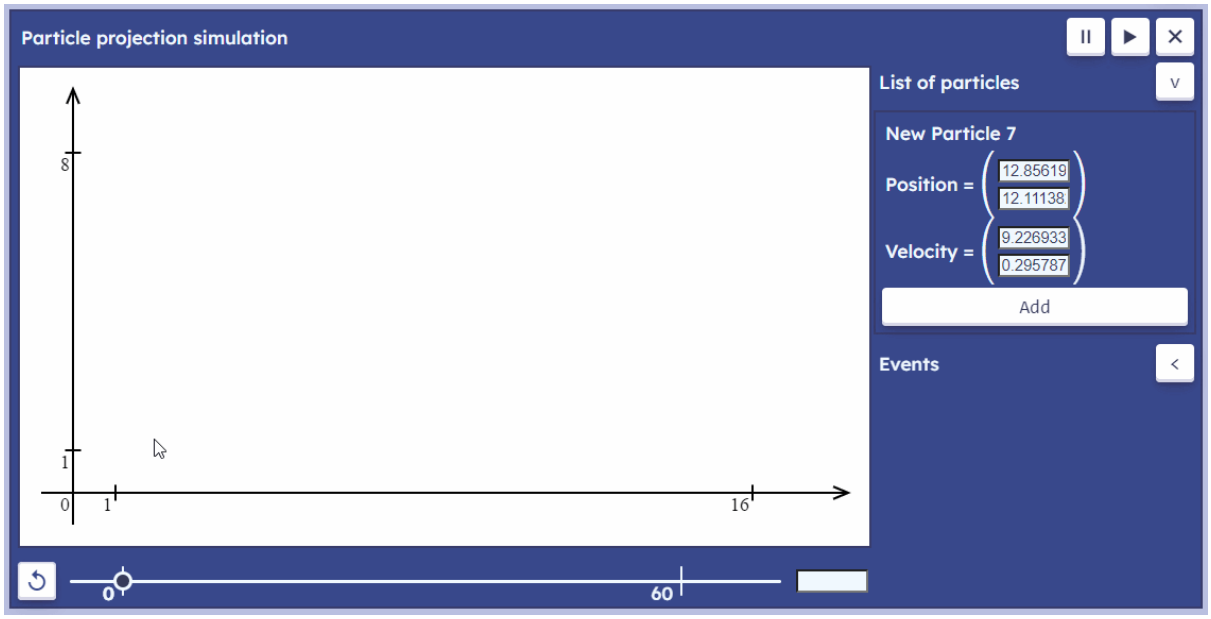
Success criteria was fully met.

Issues and potential improvements:
Potentially, scale can be adjusted using a mouse wheel, zooming in and out on scroll.
Also, a helpful feature will be to zoom into a certain point. The main objective of this operation is to adjust scale and translation such that the point in simulation space, into which the user zooms in, is maintained in the same point on the screen.

5.2.9 Trajectory



The same was demonstrated for other GIFs previously. Trajectories are created correctly, and particles follow them exactly.

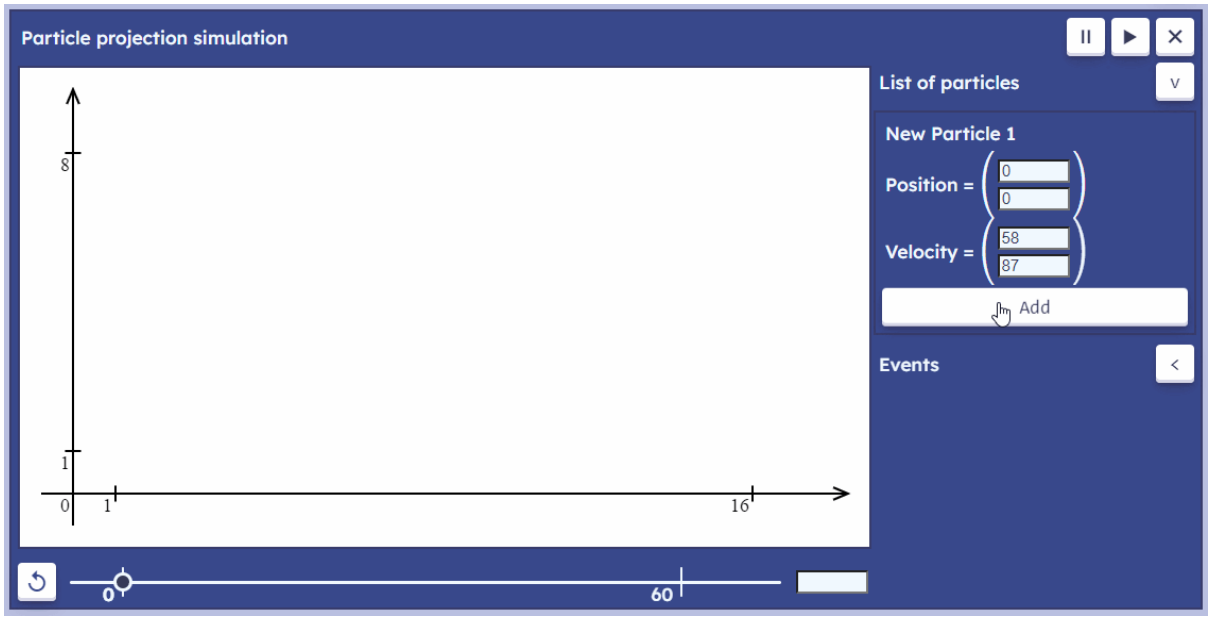


The GIF shows that the trajectory objects is being removed together with the particle. This part of success criteria is provided by the “deleteById()” method of the ViewSim class. Trajectory and particle have the same identifiers and get deleted together.

Success criteria was fully met.

However, there is a range of edge cases that present a problem to trajectory drawing.

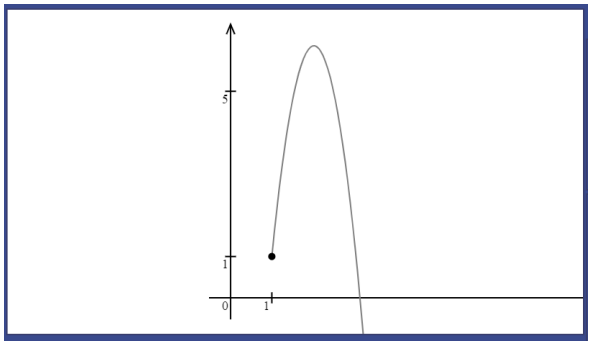
When the trajectory object is created, it considers current value of the scale to determine the last point on the trajectory. Because of that, trajectories that are created at smaller scales are also smaller. And this does not affect usability if the particle is created using GUI input. At the same time, actual users are more likely to create particles with text input. And if the values they input are much larger in relation to the current value of the scale, the following problem occurs:



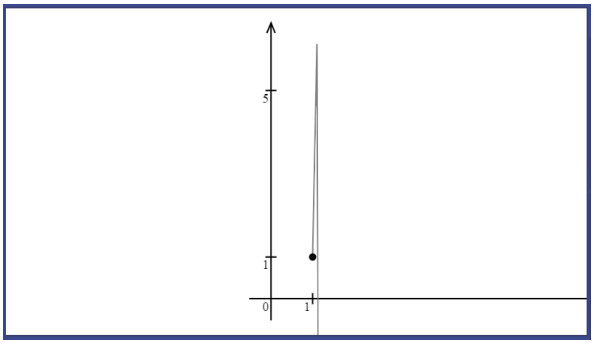
In this case, the initial scale value is 40. The user creates a particle with a velocity of (58, 87). However, simulation makes a prediction based on the value of scale which is much different from the actual scale after it has been adjusted. Trajectory ends up being too short. This problem can be solved very easily by rearranging a couple of lines of code such that the trajectory is created after the scale is adjusted. In this way, new value of the scale will be used, resulting in a more appropriate prediction.

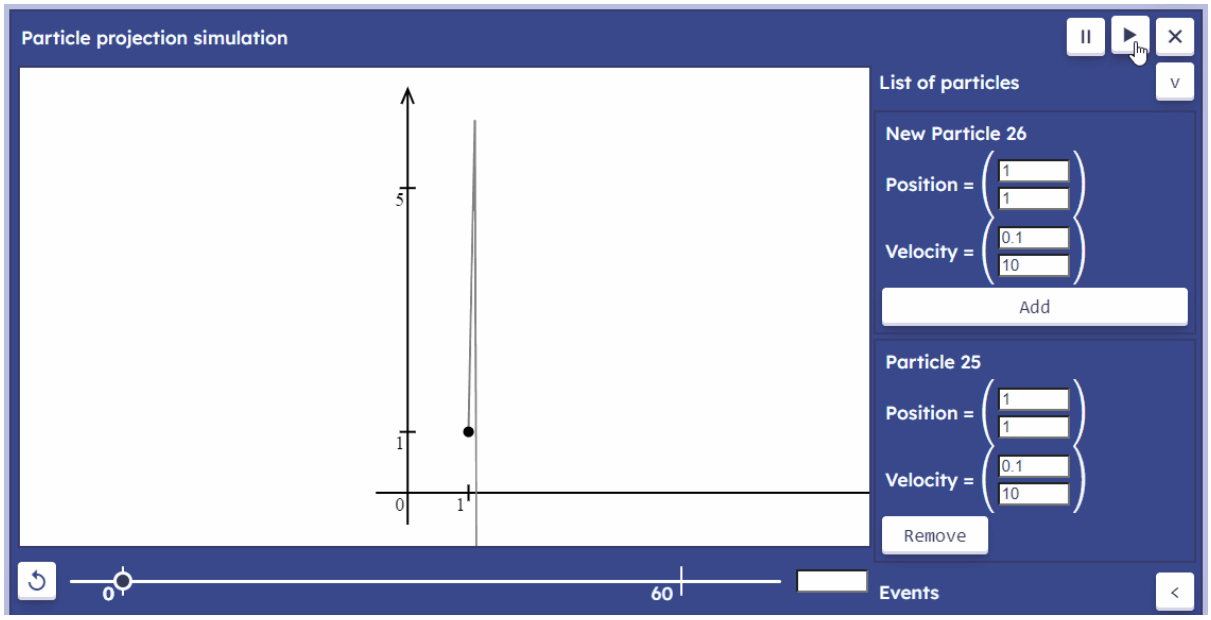
The sharp edge problem, that was discussed in the tests and corrections section of the trajectory class implementation turned out to be only partially solved.

Particle with velocity (1, 10)



Particle with velocity (0.1, 10)





The particle quickly derivates from the trajectory object. This is more apparent for smaller scales and especially when particle's speed by x is much smaller than speed by y. I do not think this might arise as a problem when actual users will solve problems with my software. The angle of projection in this case is 89.43 degrees, which is unreasonably close to 90 and there no maths problems that use such angles.

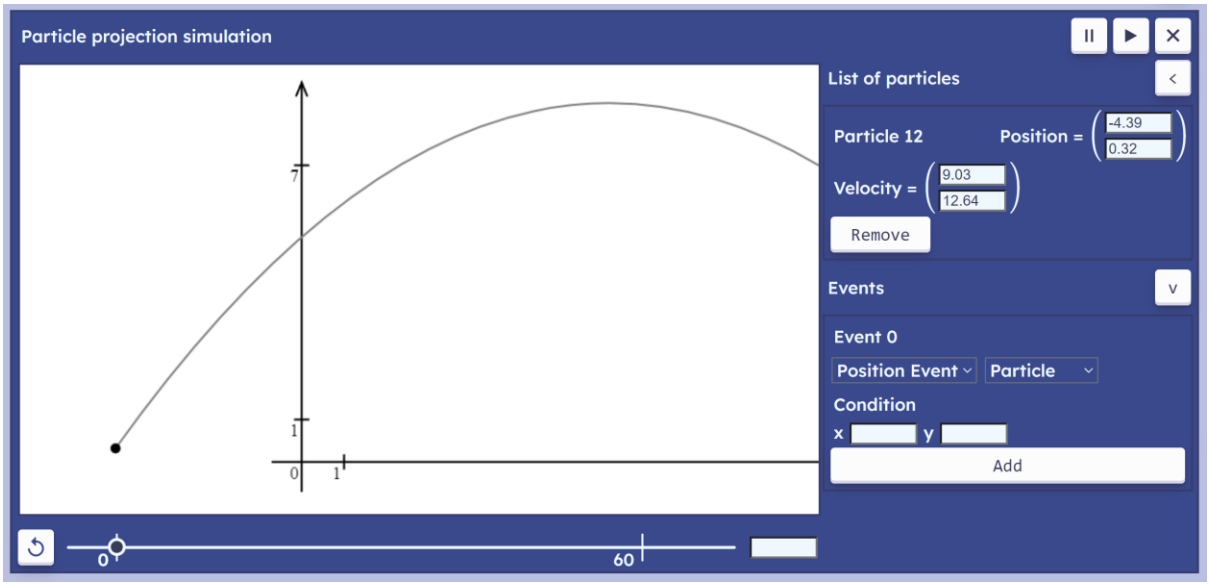
Nevertheless, this problem must be considered and addressed at later stages of post development. One of the possible ways to solve it is to create special case for when the angle of projection is over a certain limit, for instance, 85 degrees. It would change the second point for a trajectory to appear earlier so the Bezier curve is drawn properly.

5.2.10Event system

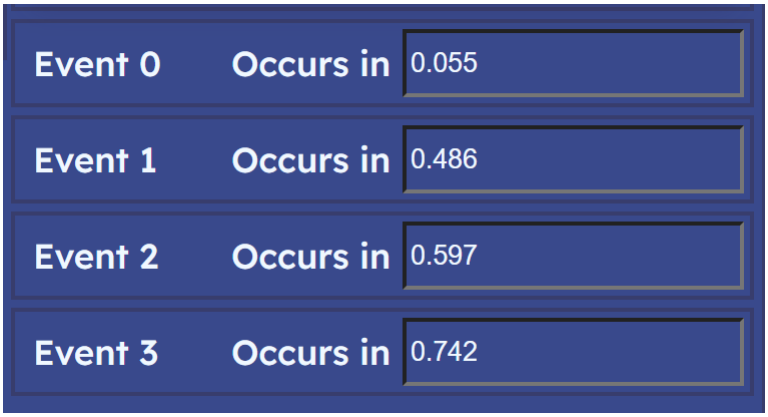
The GIF shows how event area is opened. The event type drop-down provides a choice between three event types. Once the event type is chosen, correct input fields are revealed. The user can also choose a particle they want to work with.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

Carry out a test for a particle.

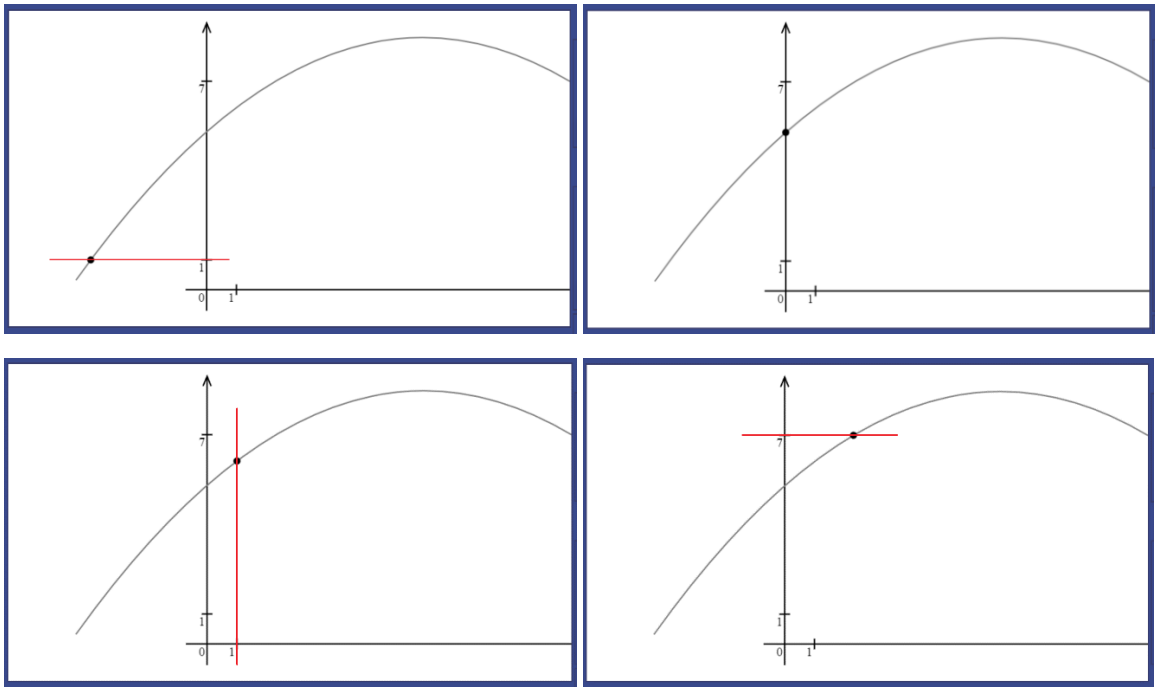


Coordinate axes give a range of reference points to validate if the particle stops at the right position. The first one will occur when the particle is at position 1 by y. Then the particle reaches 0 by x; then 1 by x; and finally, 7 by y.



Initialised these events in this order (the queue of events is on the screenshot to the right).

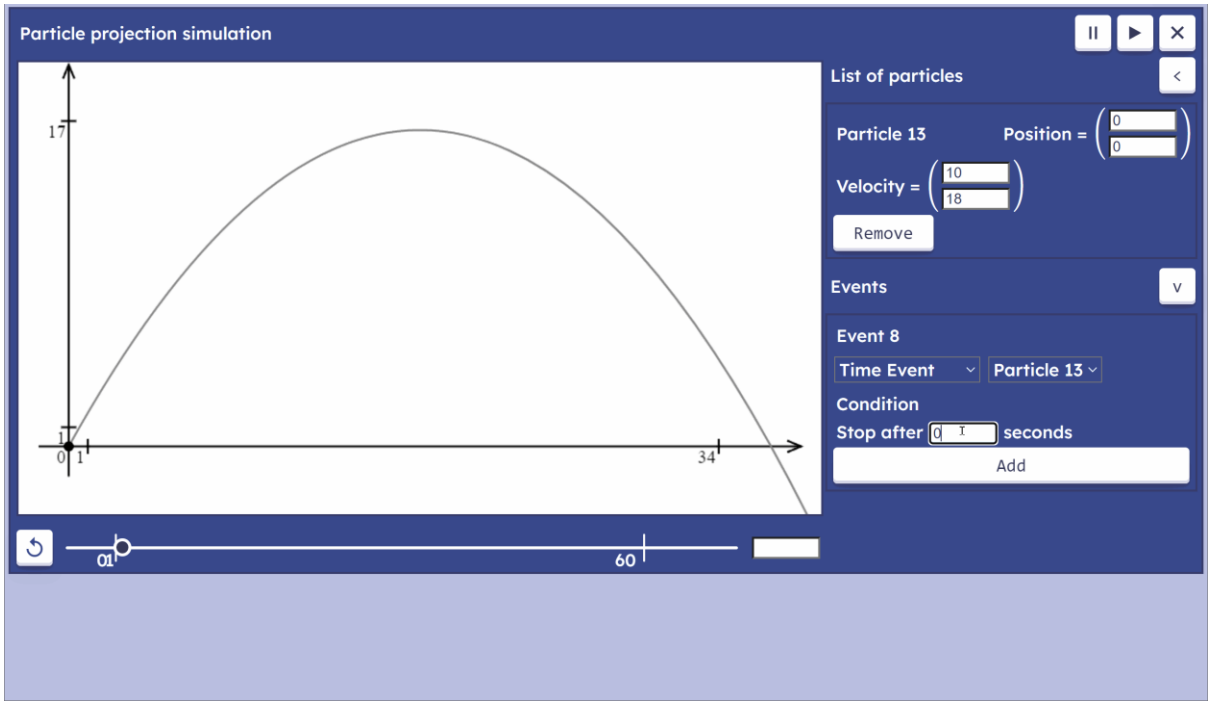
Now start simulation and check if the particle is at the right position:



These screenshots represent simulations states when each event occurs. The red line goes from the tick mark on the axis to the particle. As can be seen from screenshots, each position event is calculated correctly.

With Velocity Event it is harder to show such obvious examples that prove the fact that the system works correctly. During development (in corresponding section), I carried out more rigorous tests that involved matching calculations carried out by hand to the program output.

The Time Event is the simplest of all three. To test it by observation, start simulation with a particle at (0, 0) and velocity (10, 18). In 0.5 seconds, its position by x will be 5, in 1 seconds: 10; in 2 seconds: 20; in 5 seconds: 100.



The GIF shows a process of creating 4 of these time events. Position can be observed on the IO area on the right: 0 -> 5 -> 10 -> 20 -> 50, as expected.

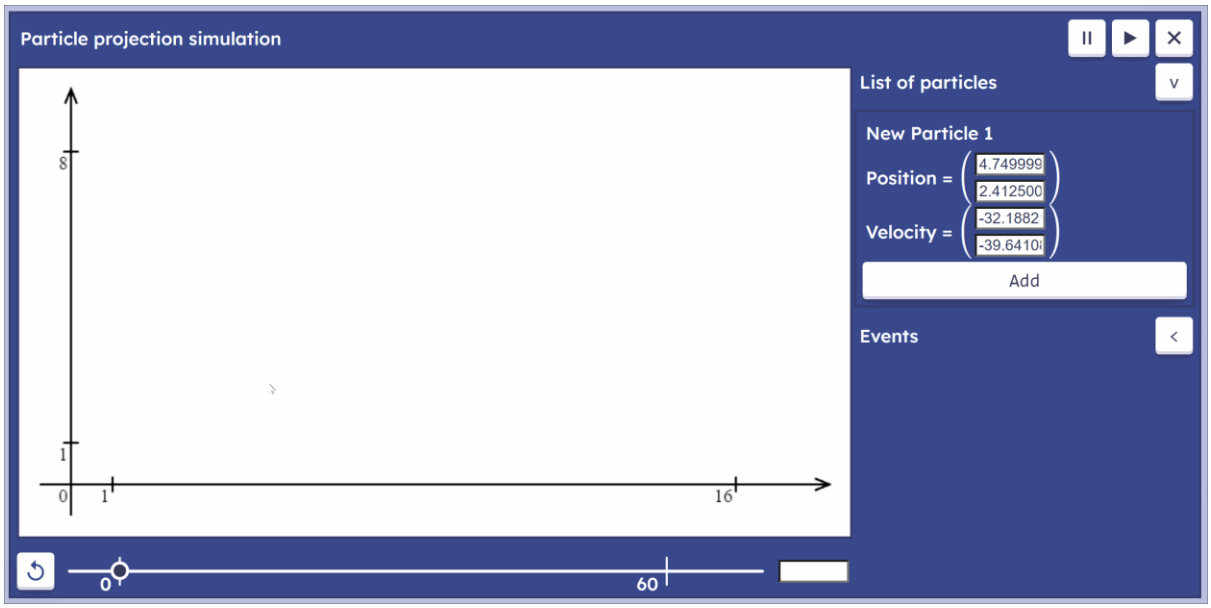
Success criteria was fully met.

*A major part of this feature's success is how helpful it is in solving problems. Success of the usability aspect is proven down below in the usability testing section (potential improvements are given in conclusions to the useability testing).

5.2.11 Coordinate axes

The coordinate axes object appeared on all previous tests, as it is always on the screen when simulation just starts. Initially, it is positioned such that the origin is next to the bottom left corner.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

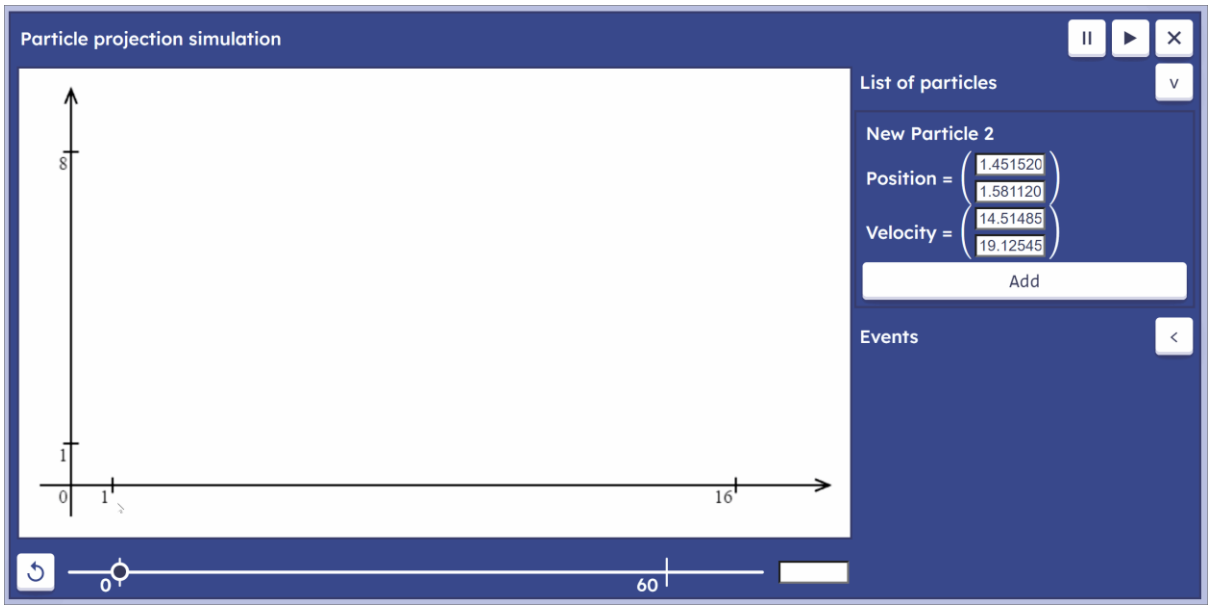


The GIF shows how numerical labels adjust their position as the value of the scale changes. Axes themselves have fixed length regardless of scale. It allows them to be usable for any type of particle, not depending on whether it moves in a couple of metres per second or in thousands. The scale will adjust and numerical labels along with it.

The correctness of numerical labels is tested here by creating particles with positions matching to the values on the scale and observing result on the screen. In each case, the tick mark, and the label exactly matched position of the particle.

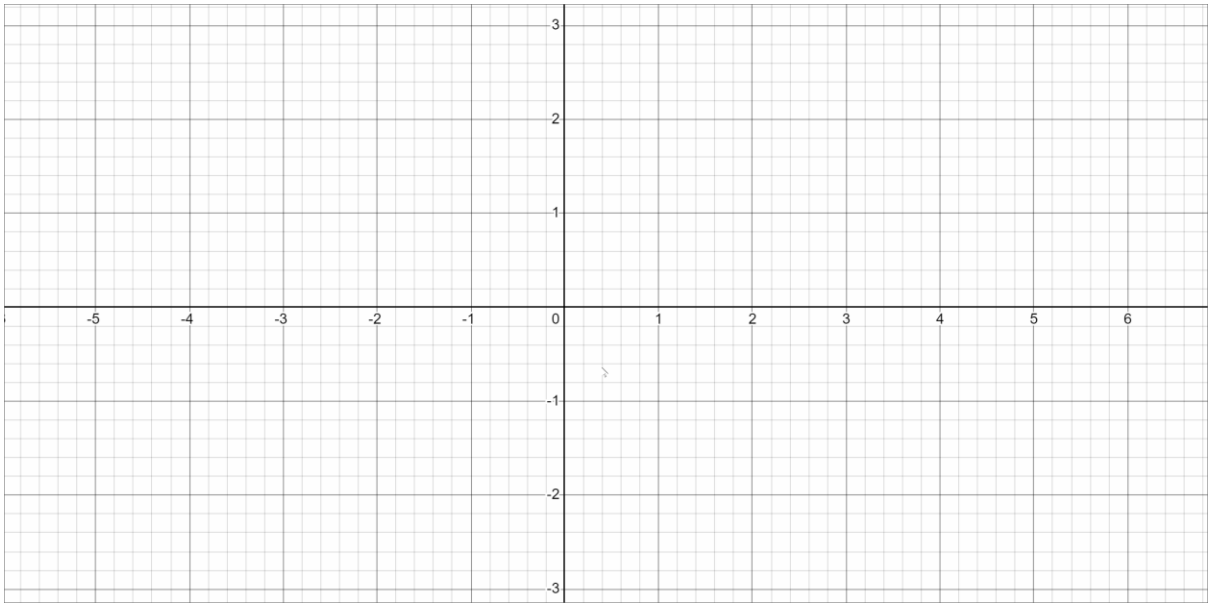
Success criteria was fully met.

The system, however, still has some minor issues or unaddressed edge cases that might need an improvement in future.
One problem is the for cases when the particle is located below or to the left of the origin, a major part of axes disappears from the screen:



In each of these cases the axes are not much usable.

Other problem was discussed at the end of the testing section in development, the labels and tick marks for 1 unit of simulation space are not very usable at very large or low scales.
To address this problem, the value of the smaller tick mark, which is now constant 1 for every scale can be adjusted to the scale as well as the larger value. For instance, for lower scale, it will be 0.1 / 0.01 / 0.001 …; for larger scale: 10 / 100 / 1000 … It will make this tick mark a good reference point for users regardless of scale. Similar approach can be seen on Desmos graphing calculator:



This GIF was recorded on https://www.desmos.com/calculator

Incorporating this technique would solve both problems.
As not everybody would prefer coordinate axes across their screen, a check box can be used to switch it.

5.3 Usability testing

While acceptance testing covers success criteria and identifies whether a feature was correctly implemented or not, usability testing will evaluate how much does the solution meet needs of the users.

I will be working with a group of 7 people, consisting of 2 maths teachers and 5 Y13 students. I will interview each of them separately.
In the first part of the interview, I will present each member of the group with my website and give them some time to read the instructions, making a note of how much time they needed. Then I will ask them to complete some tasks like “create a particle at position (0, 0) with velocity (10, 0)”. These tests must show how well they can navigate my website after just reading instructions.
In the seconds part, I will show stakeholders some mechanics problems and ask them to use my program to get answers.

5.3.1 Interface usability

5.3.1.1 Plan

- Give interviewee time to read instructions and make a note of how much they take.

- Explain interviewee that I will ask them to complete some tasks (be explicit on the fact that it is my program that is being tested, but not them)
- 1. Put Particle Projection Simulation on the screen.
- 2. Open a menu to create new particle.
- 3. Create new particle with initial position at (4, 3) and velocity (4, 12)
- 4. Start simulation.
- 5. Stop simulation.
- 6. Delete this particle.
- 7. Stop simulation and reset time.
- 8. Create a new one using a graphical interface.
- 9. Create an event with y condition of 0.
- Ask them whether they understand what this event represents.
- Start simulation and see what happens.
- Ask interviewee for what they think about the interface; what could be better; how good is the colour scheme; how intuitive is it when first using it; what would they wish to see.
- Ask interviewee for any additional comments they have.

5.3.1.2 First interviewee (student)

Took 1 minute 25 seconds to read and acknowledge instructions. Every initial task was completed without further explanations. Suggested to substitute the instructions with video tutorials displayed when the page is first opened.

It was not intuitive that they need to select the particle before creating an event. Might be helpful to automatically put a particle there.

While working with a particle, the interviewee clicked on the canvas which changed the position input for the particle (the interviewee did not notice that). Afterwards, they deleted the particle and wanted to create the same one again. They clicked on the add button, and completely different particle was created. The system of creating particles clicking on the canvas is not only inconvenient to use, but it also interferes with the regular particle creation system.

5.3.1.3 Second interviewee (teacher)

The interviewee took 2 minutes to read instructions and proceeded to successfully create Particle Projection Simulation. It was not clear what buttons open menus for particle / event creation.

Once the interviewee created a particle, they multiple times confused the menu with actual particle in simulation until I explicitly pointed it out.

A reset button on the bottom was confused with simulation start button.

The interviewee commented that it takes a little to figure out how to navigate the page, but once you get used to it, all controls are very convenient.

The interviewee also expected that position and velocity of already created particles can be changed on the Input Output area, which was not yet possible.

5.3.1.4 Third interviewee (student)

Took 2 minutes to read instructions. Once inputted values into a new particle menu, proceeded to use events before actually adding a particle by clicking on the “Add” button.

The WASD controls appeared unpleasantly abrupt, because of a small delay before translation is changed when the user holds one of the W/A/S/D keys.

The id system for particles and events was confusing.

Empty fields were expected to be 0 by default.

Particle creation by clicking on interface was confusing, the same problem as the one mentioned by the first interviewee.

When there is only one particle in simulation, there is no need for a drop-down menu selection of a particle, it would be better if it happened automatically. In the same way, when creating the first simulation, there is no need for a user to move a cursor to the “add” button, it can happen automatically.

When the page is first loaded, there is a bunch of free space in the middle and bottom parts of the screen. It can be filled with videos / gifs that demonstrate software in use and go through different ways to interact with the system, covering some nuances, like the fact that to use keyboard controls, you first need to click on the canvas.

5.3.1.5 Others

The problems mentioned previously are recurring and largely overlap across all interviewees. In this section (and the same section for Problem Application) I will list problems concerning other stakeholder (1 teacher and 3 other students), improvements they suggest and other comments they have.

Interviewees took from one minute to a minute and a half to read instructions.

The id system was confusing again. The event ids different to particle ids did not make sense. Most interviewees struggled to identify a button that opens a menu to create new particles.

A button that will allow to remove events will be helpful.

Units of measurements will be a helpful but not necessary addition. Especially for the “occurs at” label in event queue. Not all interviewees understood that it represents time.

More information on events in the form of textual / video explanations or hints.

Use of the word “drop-down menu” in two different meanings (actual drop-down menu to select simulation type and drop-down menu to add new particles) in brief instructions confused some interviewees.

Assign different colours for different particle. When hovering over the particle on the canvas, put its position and velocity on the screen or highlight its corresponding output block. Background of the output block can match colour of the particle.

Difference parts of IO area blend into one because of the same colours. Outline that separates them is barely noticeable.

5.3.2 Problem application

5.3.2.1 Plan

First problem:

Example 1

A particle is projected horizontally at 25 m s^{-1} from a point 78.4 metres above a horizontal surface. Find:

a the time taken by the particle to reach the surface

b the horizontal distance travelled in that time.

An interviewee will have to create a particle at (0, 78.4) with velocity of (25, 0). They will then need to add a Position Event with condition on y of 0 (reaching a surface). It will enqueue an event that occurs in 4 seconds (the answer to part (a)). They will then need to start a simulation and wait four seconds for an event to occur. Simulation will stop and position of the particle will be (100, 0). 100 metres is the answer to part (b).

Second problem:

Example 2

A particle is projected horizontally with a velocity of 15 m s^{-1}. Find:

a the horizontal and vertical components of the displacement of the particle from the point of projection after 3 seconds

b the distance of the particle from the point of projection after 3 seconds.

An interviewee will need to delete the first particle and create a new one at (0, 0) with initial velocity of (15, 0). They will then need to create a Time Event that occurs in 3 seconds. They will run simulation and observe how event occurs in 3 seconds. At that moment its position will be (45, -44.1) which is the answer to part (a). For part (b), they will need to use calculator to take square root of a sum of squares of particle's position components (this is irrelevant to usability testing, though).

The other option for them is to use time input text box and input a value of 3 to observe position of a particle at that moment in time. It will be up to interviewee which method to use.

Third problem:

"A particle is projected from the ground with velocity $\vec{\boldsymbol{U}} = (3\boldsymbol{i} + 5\boldsymbol{j})\ ms^{-1}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the horizontal and vertical directions respectively. Find the greatest height above the ground reached by the particle".

They will need to create a particle at position (0, 0) and initial velocity of (3, 5). They will need to create a Velocity Event with y condition of 0 (when particle is at its highest point on trajectory, its velocity consists only of horizontal component). The answer is the y component of particle's position once it reaches the event.

*First two problems are taken from Year 2 Statistics and Mechanics book by Pearson Edexcel.

5.3.2.2 First interviewee (student)

Every problem was solved successfully with no prompts and directions; however, some features of the interface were a bit confusing.

From problem to problem, the interviewee found it easier to close simulation rather than delete particles and reset the scale.

The interviewee kept clicking on reset button expecting it to reset only time and was confused when position of the particle on the screen changed.

They found it inconvenient to use keyboard controls as they required a click on the canvas. Several times, they attempted to use keyboard control not focusing on the canvas (not clicking on it).

5.3.2.3 Second interviewee (teacher)

Once they were familiar with the interface, it took them around 10 seconds to read the first problem and start correctly inputting position of the particle (however, for one of the problems, they confused x and y components).

Several times, they forgot to put 0 as an input value, presumably assuming that the default value was 0. However, they promptly realised the mistake and corrected themselves. They emphasised how helpful were error messages.

After successfully completing three problems, they went on to explore possibilities of the program and accidentally discovered a bug. They tried to look at vertical projection of the particle which resulted in a simulation to completely break (the same happened before when the particle was projected from the origin horizontally). The problem occurs because of automatic scale adjustment. As the particle is projected vertically, the width of a rectangle of significant points becomes equal to zero which results in division by zero and breaks simulation. This is a simple to fix bug which must be addressed in further development. One of the ways to ensure that this does not happen is to enforce a minimum value for width and height of the rectangle.

Event system was very well understood. However, the “occurs at” label did not clearly indicate that the event is due to occur 4 seconds after it is created. The interviewee also accidentally created multiple events and wanted to delete one of them but did not have such option.

5.3.2.4 Third interviewee (student)

The interviewee kept coming back to the canvas and moving camera with WASD button such that it is focused back on the coordinate axes. They suggested to remove automatic translation and scale adjustment, adding a separate button to focus on the particle instead of it being initial option after a particle is created. Instead, let the camera remain focused on coordinate axes when the particle is created.

The events disappearing without any notice were confusing to the interviewee, might be helpful to notify the user or leave events with an ability for the user to delete it when they want.

The interviewee suggested a label for time input. Accepting both dot and comma as a separator for whole and decimal parts will be useful.

Suggested a feature of showing particle’s name / id when hovering over it. The output block for corresponding particle can be highlighted.

5.3.2.5 Others

Add two separate buttons to reset time and scale. Most interviewees wanted used the reset button to reset time only and it also reset the scale.

Interviewees were very confused when they created a particle not at the start of a simulation and when they pressed reset button, they were no longer able to find exact moment in time when they created the particle. It can be fixed by resetting simulation internal time back to zero every time there are no more particles in the simulation.

An ability to move around interface using “drag” system. So, instead of WASD keys, hold left mouse button and move it some distance on the interface.

Coordinate axes were important for most interviewees, which again emphasises the point on changing the default scale adjustment to a different one.

Some suggested to change I/O for scale adjustment to +/-.

Camera following the particle will be useful as it gets hard to adjust simulation with WASD and IO keys every time. Ability to zoom in on the particle specifically will be largely beneficial.

All interviewees were visibly enjoying the process of solving problems with my software, once they understood how the system works. It indicates success of the event system. However, there are still a lot of improvements to be made to it.

5.3.2.6 Special note

Apart from my notes, I also received a separate feedback email from one of the maths teachers (specialising on mechanics) at my school:

“”“

Hello Mykyta,

I think your programme is excellent. As I said to you while I was trying it, I would be happy with it if it was a commercial product that the school had paid money for.

The simulation has a lot of functionality. I could not think of anything else that I would need that it did not have. The one exception was simulating a 1D problem (ball thrown vertically).

It is difficult to keep the interface simple when you have so much functionality, but you have managed this very well. It does take a little getting used to, but the red messages that appear when you make an error are very clear. As we discussed, different colouring of live and non-live elements may help. Maybe also the simulation could default to one particle and one event open?

One more thing I thought of was that maybe the particle and the time should stop when it reaches the border of the window? We also discussed automatic scaling of the time slider.

Features I really like and are useful to teaching:

+The flightpath is shown and the particle is animated along it

+It scales automatically

+You can move the time slider back and forth.

+The visual design of the interface is simple and clear.

+So much functionality!

Mykyta Osovskyi
Dronfield Henry Fanshawe School

Possible improvements:

- Different colouring of live/dormant particles and events

-1D problems

- Scaling of the time slider

- Ability to edit parameters of a particle after starting the simulation

It really is a very impressive project. You deserve every success!

Best wishes,

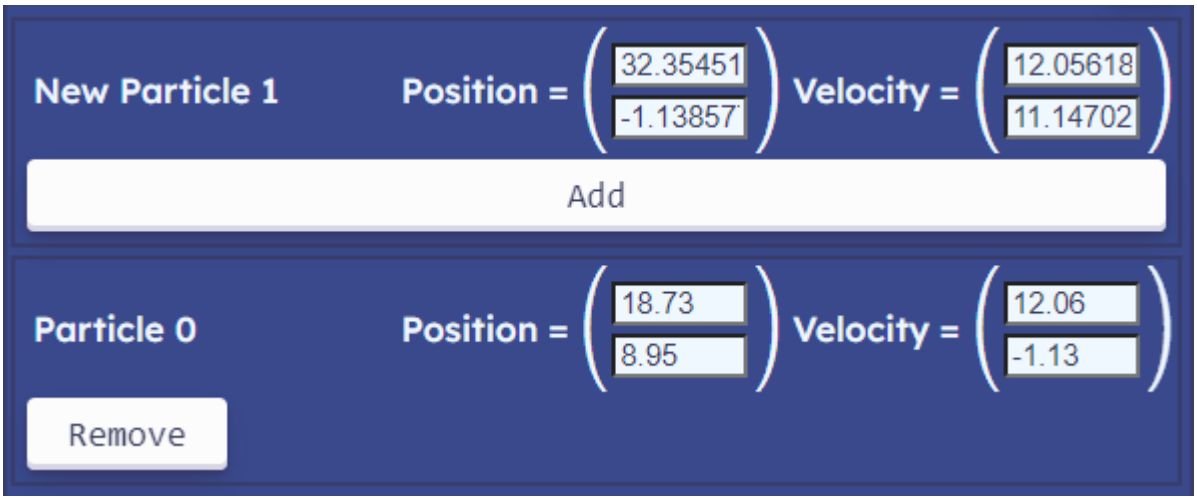
Mr Waggott

“”””

Most points from the email were already covered.

5.3.3 Conclusions

- The colour scheme was accepted across all interviewees.
- Average time to read instructions was under two minutes as planned, which satisfies another point of the interface success criteria.
- Instructions on how to use simulations at the start of the page are better to be substituted with a short video explanation on all essential controls.
- The reset button is better to be changed to reset time solely. Can add the other button to reset all of time, scale, and translation.
- Change particle selection on the event area such that if there is only one particle, it is automatically selected.
- Ability to change particle’s position after it has been created will be helpful. Use input boxes for particle’s position and velocity to alter values of the particle’s position dynamically. The trajectory object for a particle will also need to be changed.
- Confusion in using IO area to create particles occurs because the menu for creating new particles looks the same as output blocks for already created particles:



It was suggested that menus can be of a different colour / different shade of blue; or menu can have different arrangement of Input field.
The other option is to return to the menu being pop up when the plus button is clicked and disappear when the particle is added.

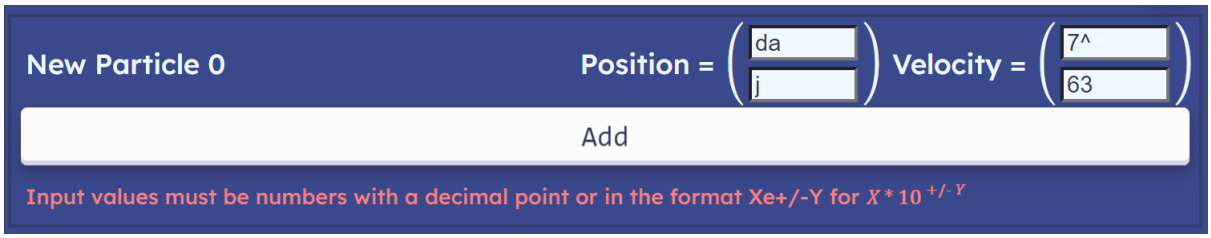
- Particle creation with clicks on a GUI must be completely reworked. The user must receive a clear indication when their clicks on the canvas define a new particle. For instance, if the user clicks on the canvas while holding an “A” key, they will enter a “new particle mode”, where they will define position and velocity for a new particle. There should also be a sign on the canvas that new particle is being added. This system must be separated from the regular text input menu.
- “Drag” camera movement in addition to WASD keys is desirable. Also, will be helpful to add the same functionality for arrow keys as for WASD keys, as they might be more familiar to wider audience.
  On of the ways to approach this is to let the user drag camera using the left mouse button and create new particles with the right mouse button.
- Buttons to open menus for adding new particles or events are located too far from the main area of the page and most users do not find their functionality intuitive. This can be changed by simply moving the buttons closer to titles.
- Column vectors are not the most universal way for representing position and velocity of the particle. Other countries are more familiar with (x, y) notation. To accommodate for that, there should be labels next to the input boxes for column vectors for more clear instructions.
- Empty input fields must have default value of zero as most users find it intuitive and convenient. It is worth noting that an input field might accidentally be left blank and defaulted to zero, preventing the error message and the user will not notice that straight away.
- Event system must have more explanations to it. Events must not disappear from the queue when they occur as the user might want to come back to them later. Instead, there should be a delete button. The value of “occurs at” must be followed by units (commonly seconds). Particle selection is not necessary when there is only one particle in the system. All interviewees skipped the part of choosing a particle and went straight to adding the event.
- Very handy feature will be a button that empties all input fields.
- WASD control abrupt movement can be fixed by moving the key dictionary check from button click callback to IO handler update method. This will allow to start required action

straight when the user presses the button with no need to wait for a page to fire other Mouse Events (this is the cause of the delay).

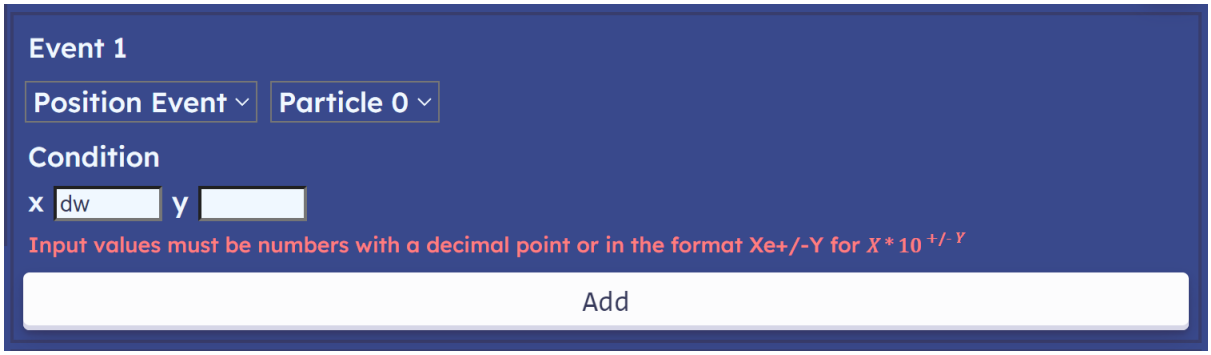
- The default scale adjustment must leave coordinate axes at its original position, while adapting value of the scale so the particle is in sight. This is because most interviewees found it more comfortable to work when the axes are at their most familiar position: origin at the left bottom corner. Apart from that, the particle's position can be represented with two additional tick marks on coordinate axes, one on y axis and the other one on x axis with the label showing value of the corresponding component of the position.
- The user should have an option to stick camera to the particle, so when it goes off from the screen, camera will follow it by adjusting scale and translation according to the particle's movement.
- The bug for 1D (vertical) projections must be fixed (discussed here).
- Change the outline of different areas on input output part of the interface. It must be more distinguishable from the background.

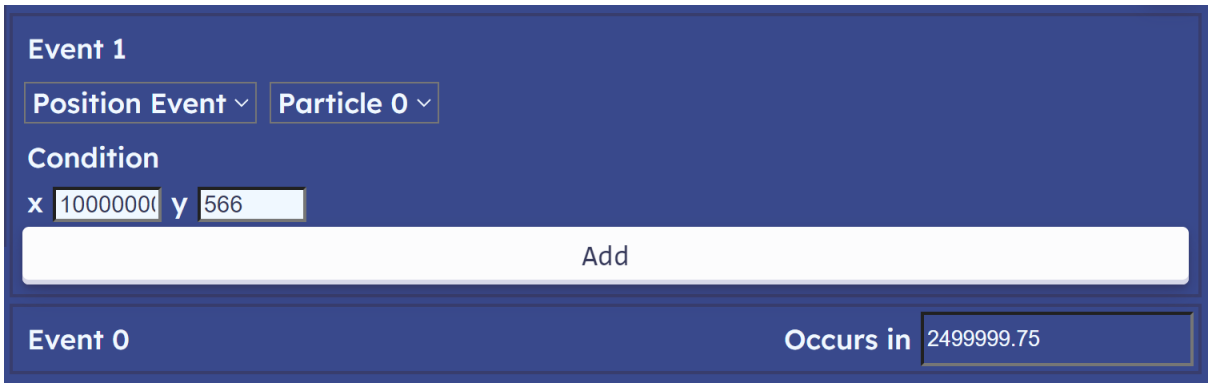
5.3.4 Robustness

5.3.4.1 Extreme and Erroneous input

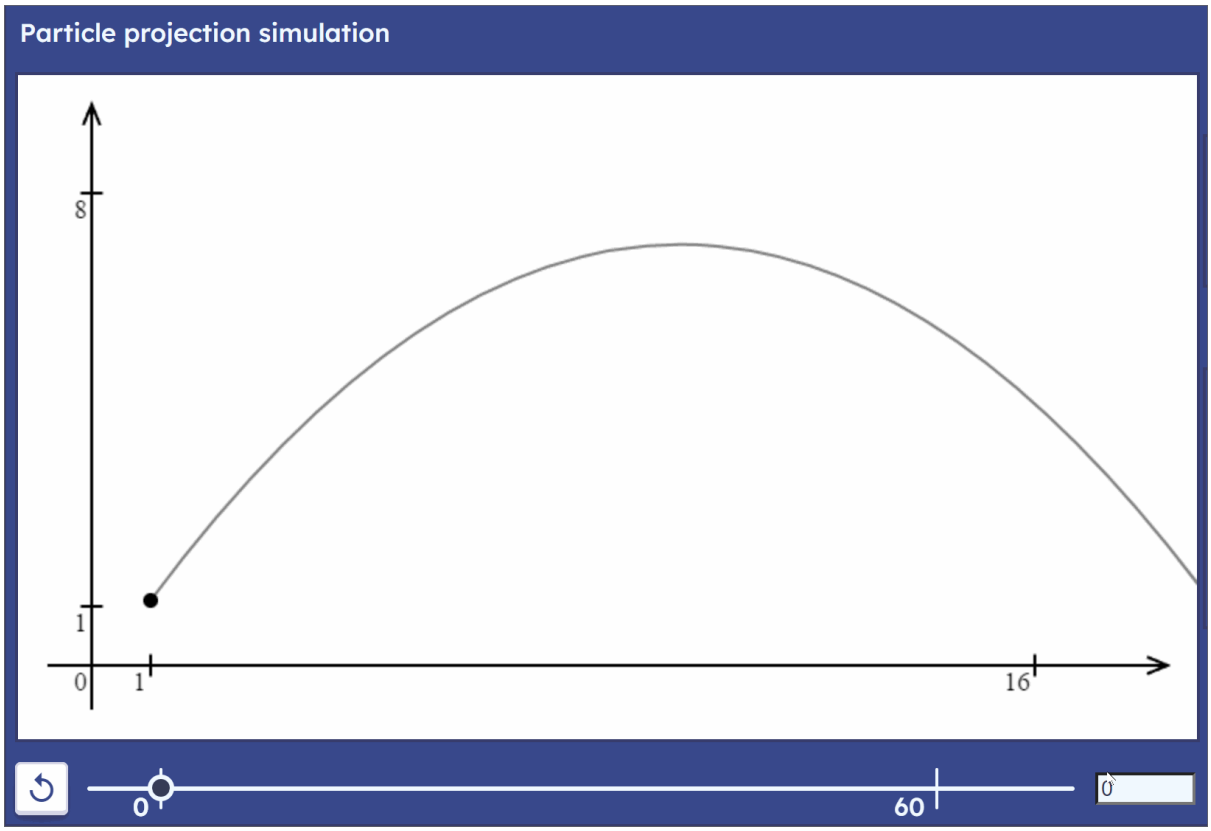


An IO handler captures erroneous or outbound input and outputs appropriate error messages (most stakeholders found it helpful).



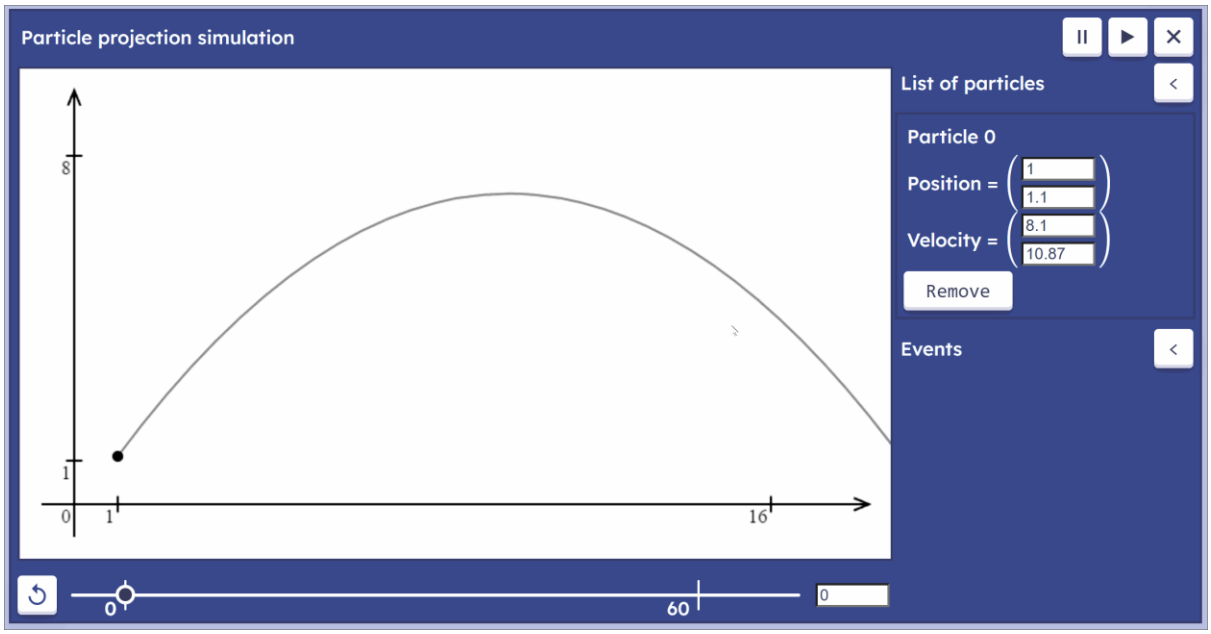


Events, on the other side, do not enforce validations for extreme input values, because the event does not immediately affect the system. For example, if the user wants to see when the particle reaches 10000 by x, the program will output the time when it happens, but the system will not be affected. Invalid input like nonnumeric characters give the same error message as for particle input.



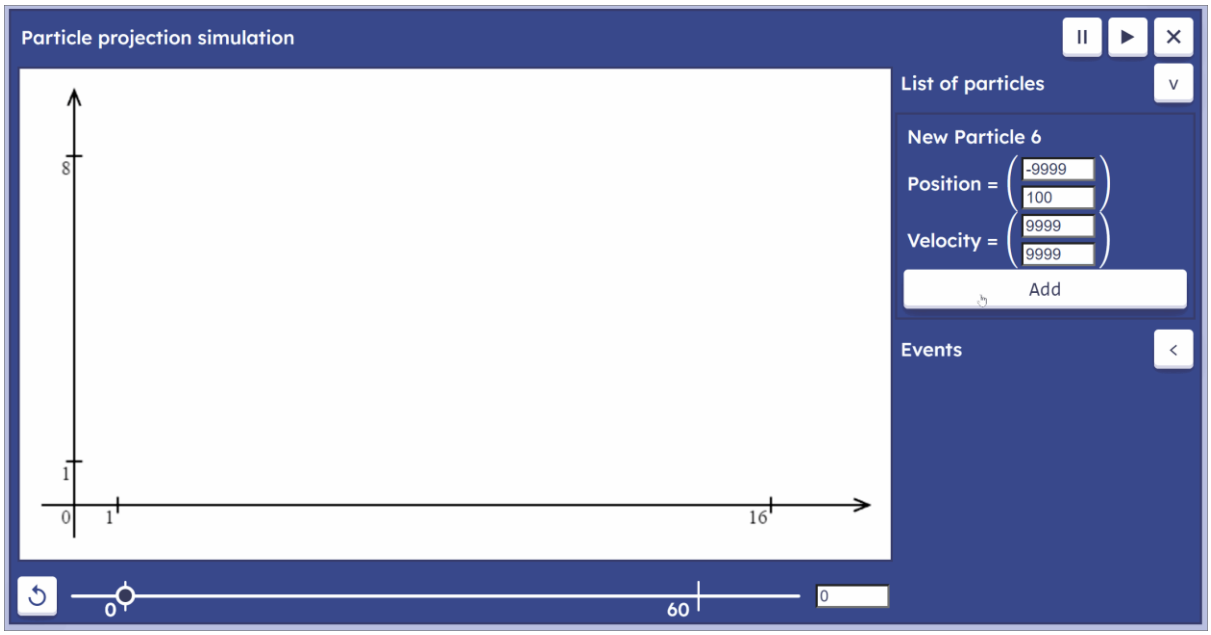
The text time input is the least robust element of the interface, it has no error messages and no limits for input. But erroneous values are not accepted and do not affect the system. Robustness is maintained as the user can always set the time to 0 and previous input will have no effect on the system.
Also, for extreme values, the thumb of the slider does not leave its range. Its position is clamped such that it is always located between -5 and 70 on the slider.

Mykyta Osovskyi
Dronfield Henry Fanshawe School



The scale can also be increased/decreased indefinitely using I/O keys. The user can always come back to initial value by pressing the reset button. The same applies to translation.
Because the scale and translation are separated from the model (actual simulation that calculates positions of particles, advances simulation in time, etc.), change in scale and translation does not affect the particle as can be seen from the output block for “Particle 1” on the right.

Now consider the case of creating a particle with boundary values for position and particle:



When the particle is first created, the scale and translation are adjusted to fit its trajectory into screen. However, the trajectory itself is very small. This is because the trajectory creating routine uses the scale value before it has been adjusted. This problem was discussed in acceptance testing. Creating the same particle afterwards gives perfectly fine results. Even though values at the ends of coordinate axes are huge (18946833), simulation functions as intended, the user can move camera around simulation and change scale.

5.3.4.2 Overloading

Now, I will test how my program behaves when the user attempts to create a lot of particles or simulations at the same time.

To assist my testing, I will calculate framerate as I change parameters. I will keep an array of the difference in time between the past 100 consecutive frames.

```
this.counter = 0;
this.fps_list = new Array(100);
```

The counter keeps current position of a new element to be added into array.

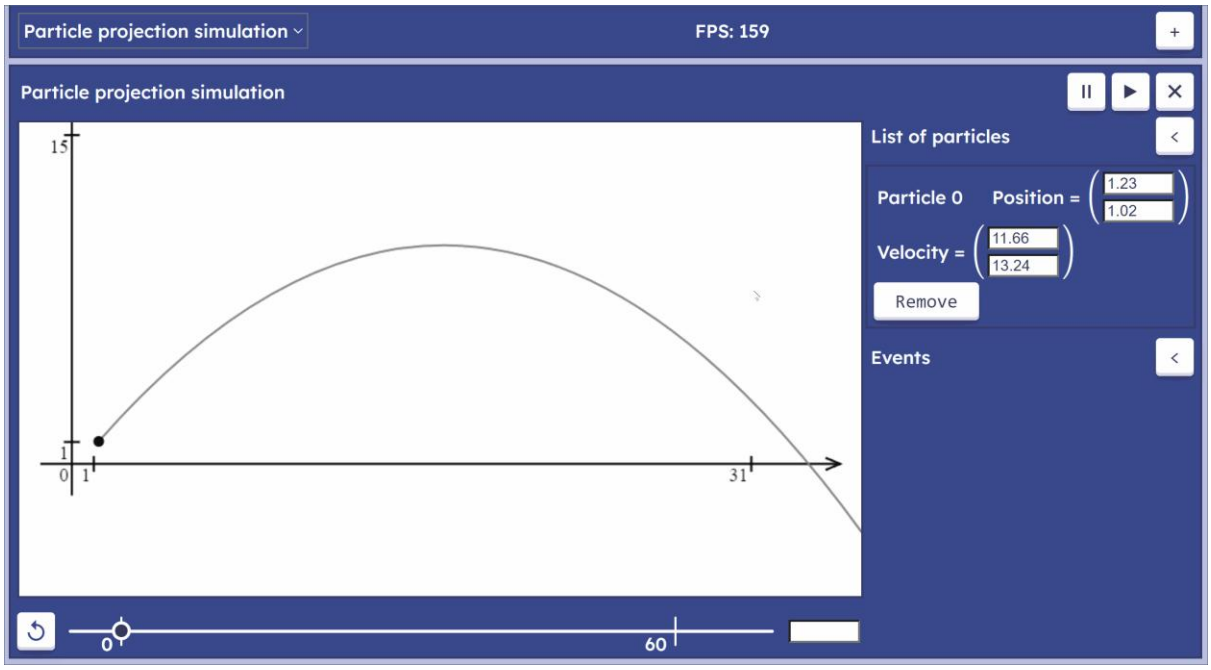
```
this.fps_list[this.counter++ % 100] = dt;
this.output_fps();
```

The counter wraps around each time it reaches a 100 (achieved by taking a remainder of counter divided by 100). Then the fps is outputted:

```
output_fps() {
    let sum = 0;
    for (let i = 0; i < 100; i++)
        sum += this.fps_list[i];
    fps.innerHTML = "FPS: " + Math.round(1000/sum * 100);
}
```

The fps is calculated as 1000 milliseconds divided by a sum of time differences and multiplied by 100 (because the data is gathered across 100 frames).

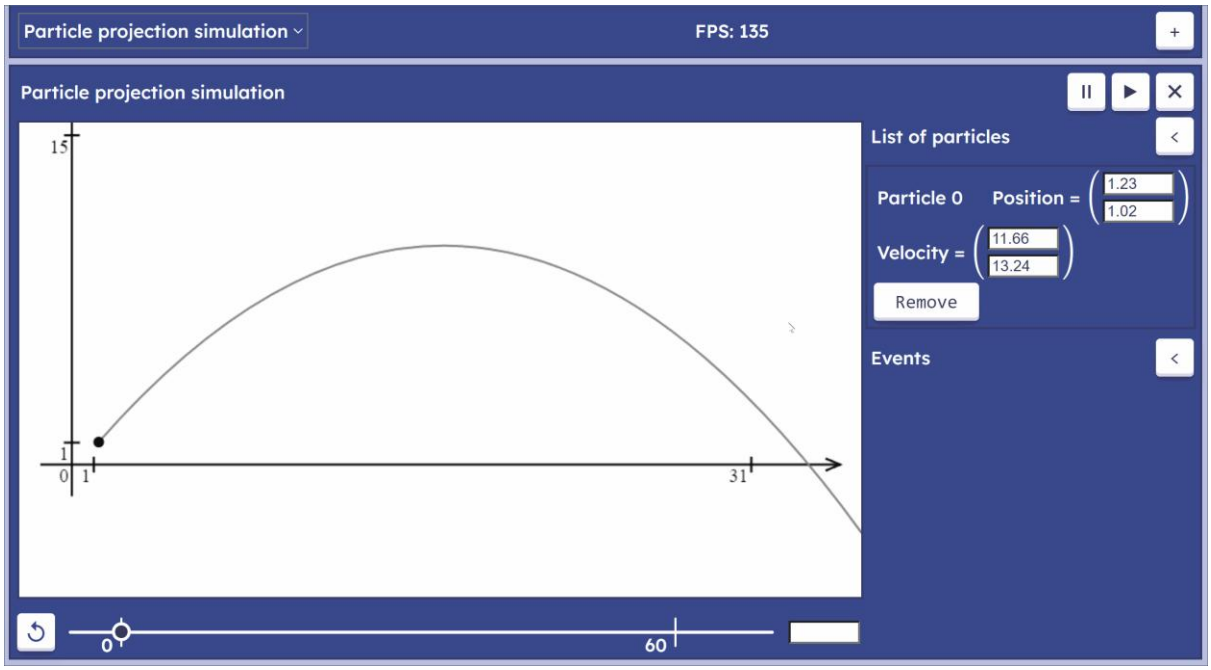
1 Simulation | 1 Particle:



Fps varies from 130 to 165.

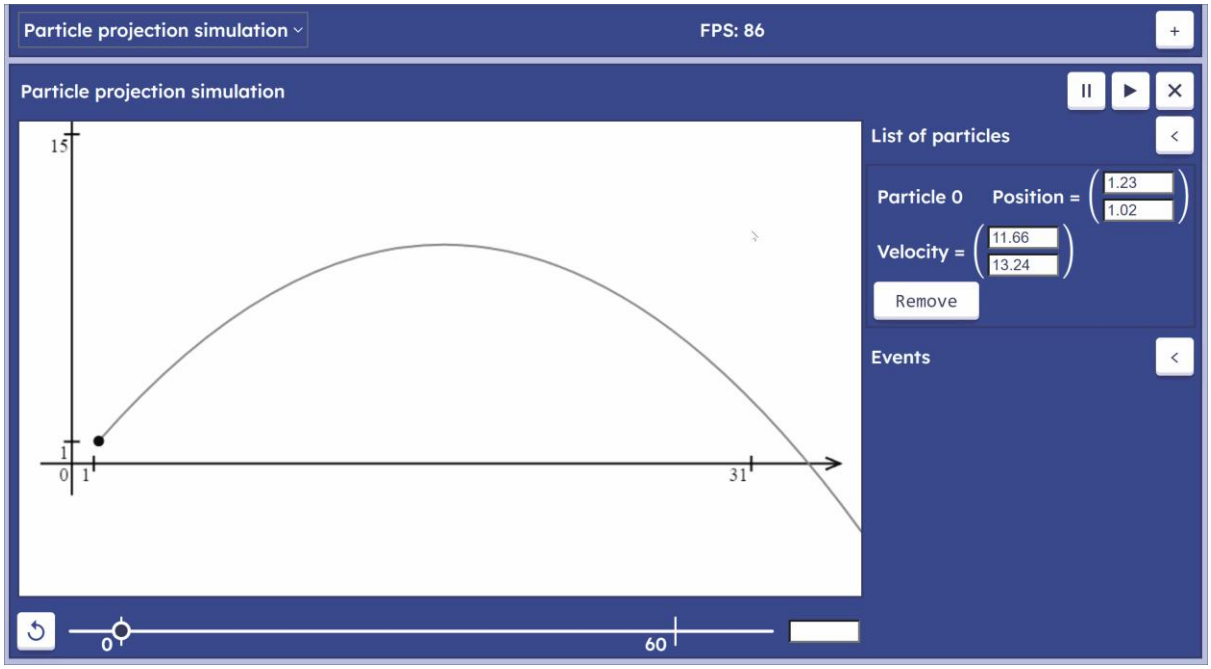
Mykyta Osovskyi
Dronfield Henry Fanshawe School

10 Simulations:



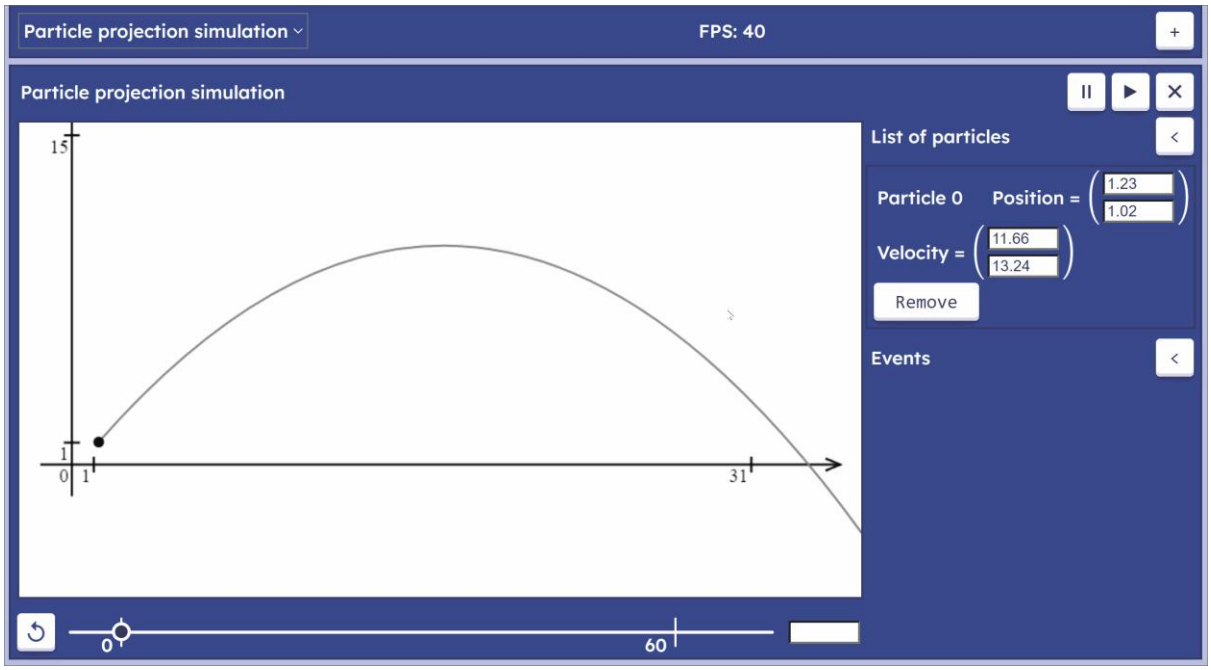
Still high fps, not much difference.

50 Simulations:



A noticeable drop to 60 – 80.

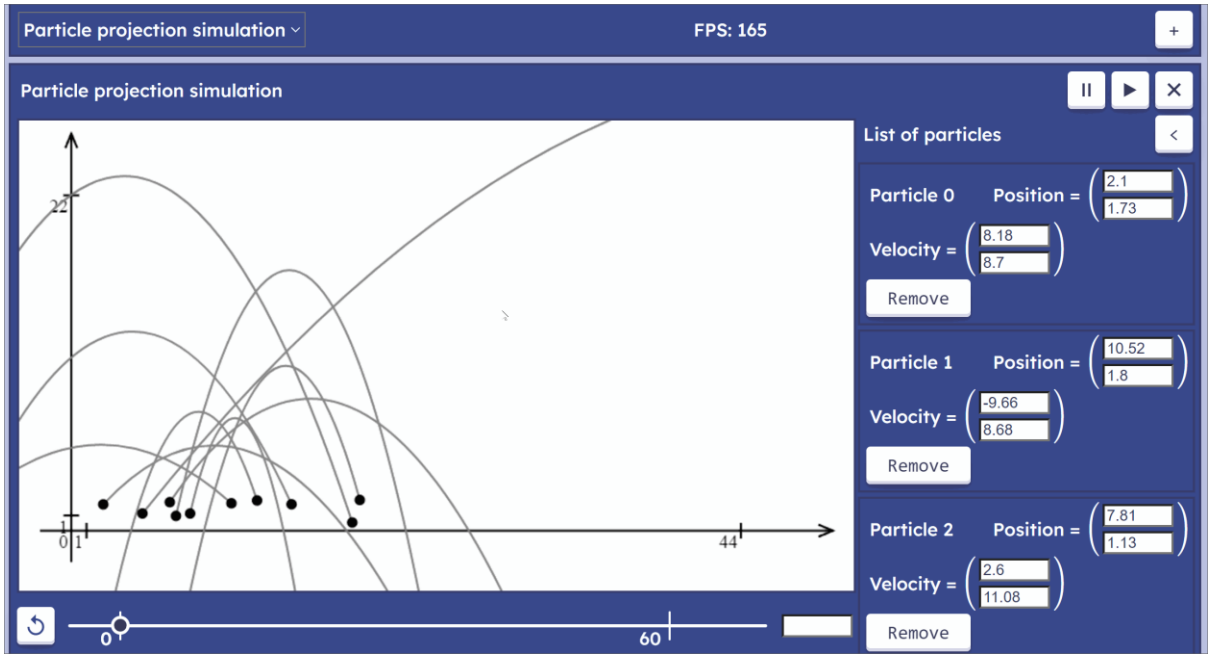
100 Simulations:



Larger drop to 40 fps.

Even though these scenarios are unrealistic for real usage scenarios, it may be worth limiting the total number of simulations to 10-30 to maintain reasonable refresh rate.

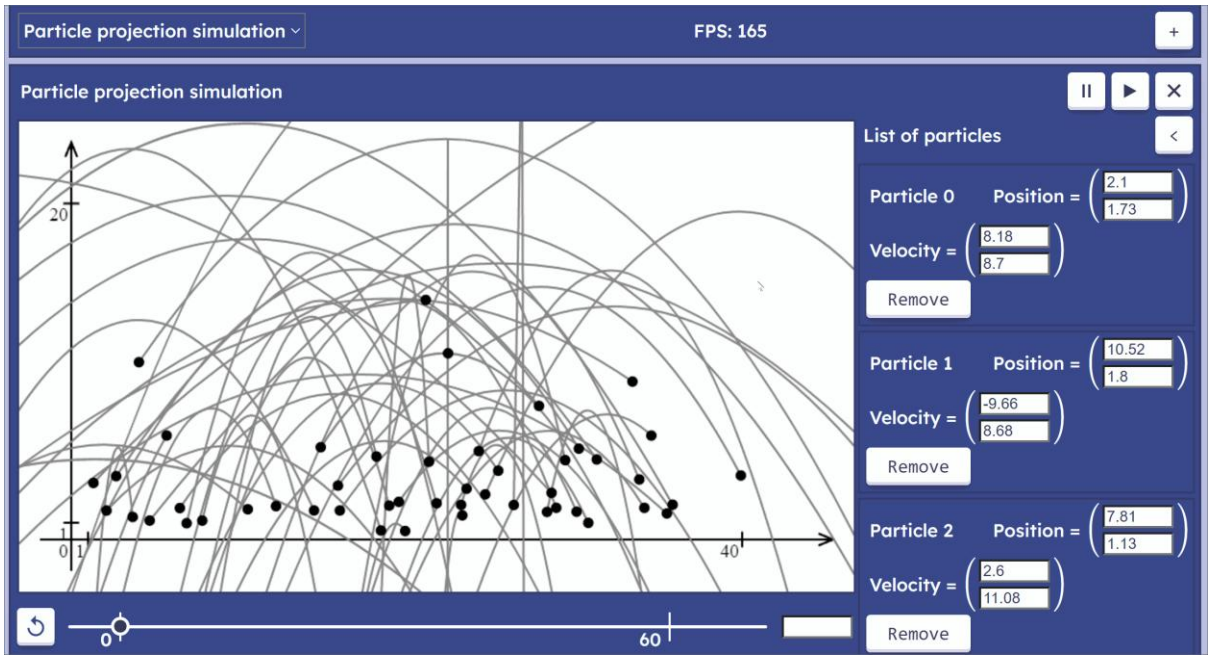
10 Particles:



Minor drop to ~150 fps. However, it is hard to imagine scenarios where the user will actually need more than 2-3 particles at a time on one simulation.

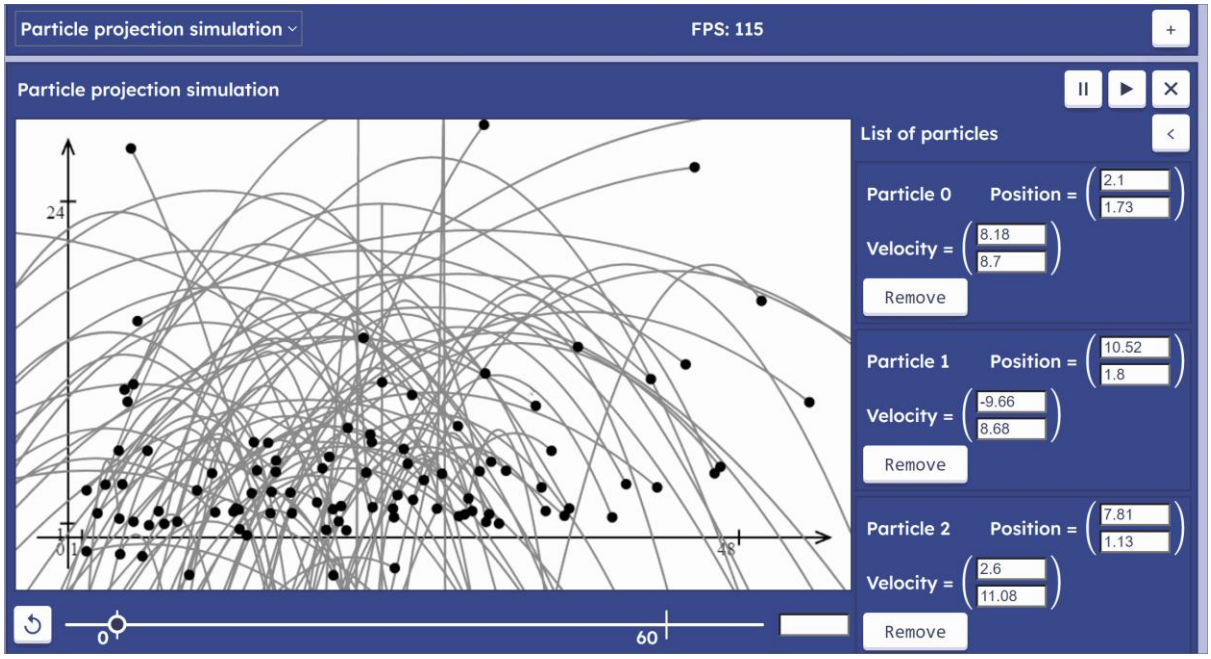
Mykyta Osovskyi
Dronfield Henry Fanshawe School

50 Particles:



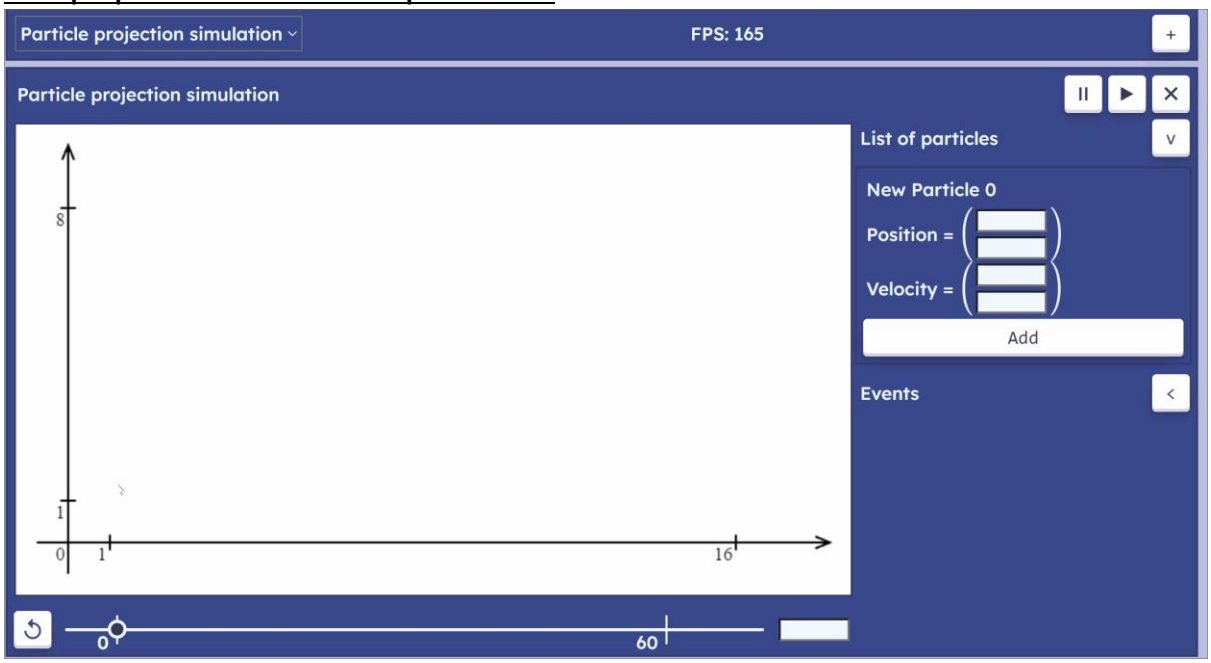
Larger drop to 80-90 fps. Simulation remains usable.

100 Particles:



Fps drops to about 50.

Multiple particles with the same parameters:



The frame rate drops gradually, more particles create more identical trajectory objects which overlap each other.

Simulation does not break because of the larger number of particles, however the frame rate decreases. A user can always delete the simulation, refresh the page, or delete unwanted particles and simulation performance will become better.

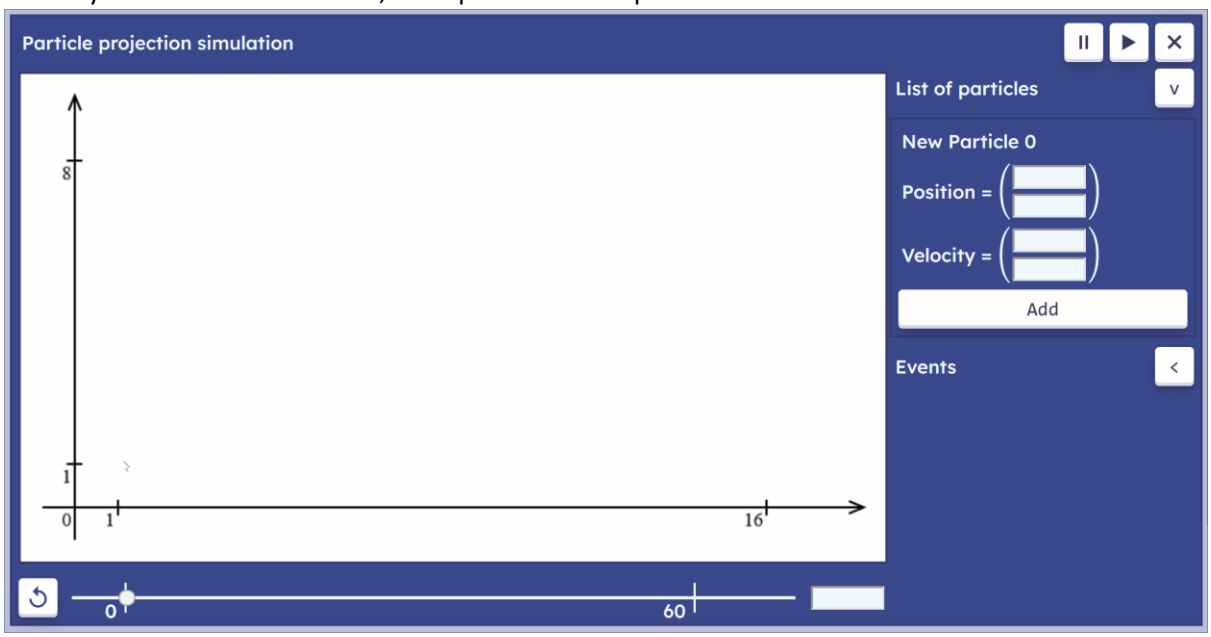
Individual results may vary depending on the specific machine the program runs on.

5.3.4.3 Accessibility

Program accessibility was not one of the central considerations during development. The website does not use too many elements that are not supported for some browsers.
Mainly, the time slider styling relies on browsers that use WebKit as a browser engine. Some examples are Safari and Google Chrome (which now changed to Blink but still supports -webkit-prefixed CSS). Most popular other browsers still support WebKit features for compatibility reasons, even if they use a different browser engine.

Mykyta Osovskyi
Dronfield Henry Fanshawe School

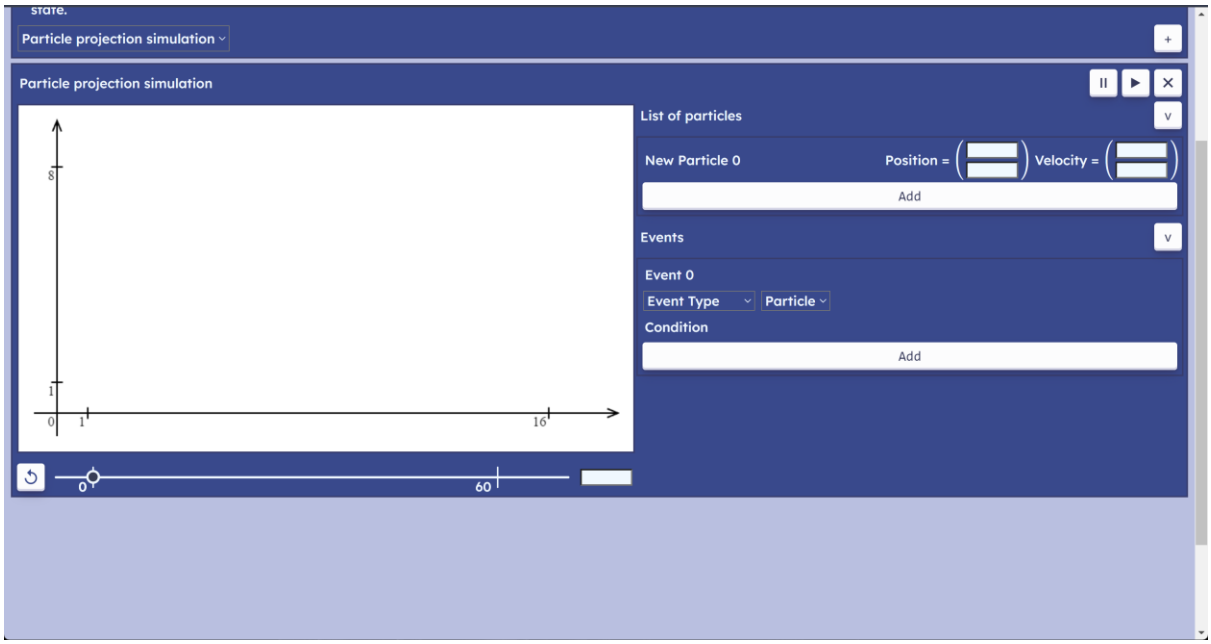
I ran my website on Firefox 124, example of a Gecko powered web browser:



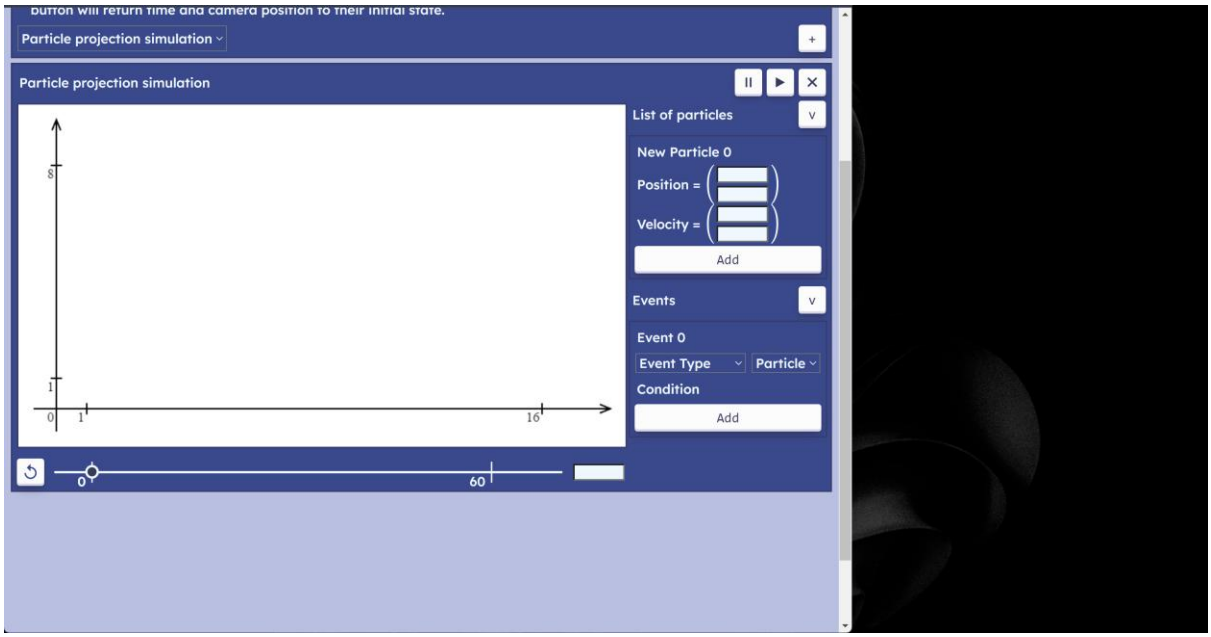
The only element that does not function as intended is the thumb of the slider. It does not have an outline; it is positioned incorrectly, and the mouse pointer does not become a pointer when hovering over it. It indicates that none of the "::-webkit-slider-thumb" CSS styling was applied. It can be easily fixed by defining the same style for "::-moz-range-thumb", a mozilla firefox alternative.

Testing how the website behaves for varying sizes of interface:

Wide interface:

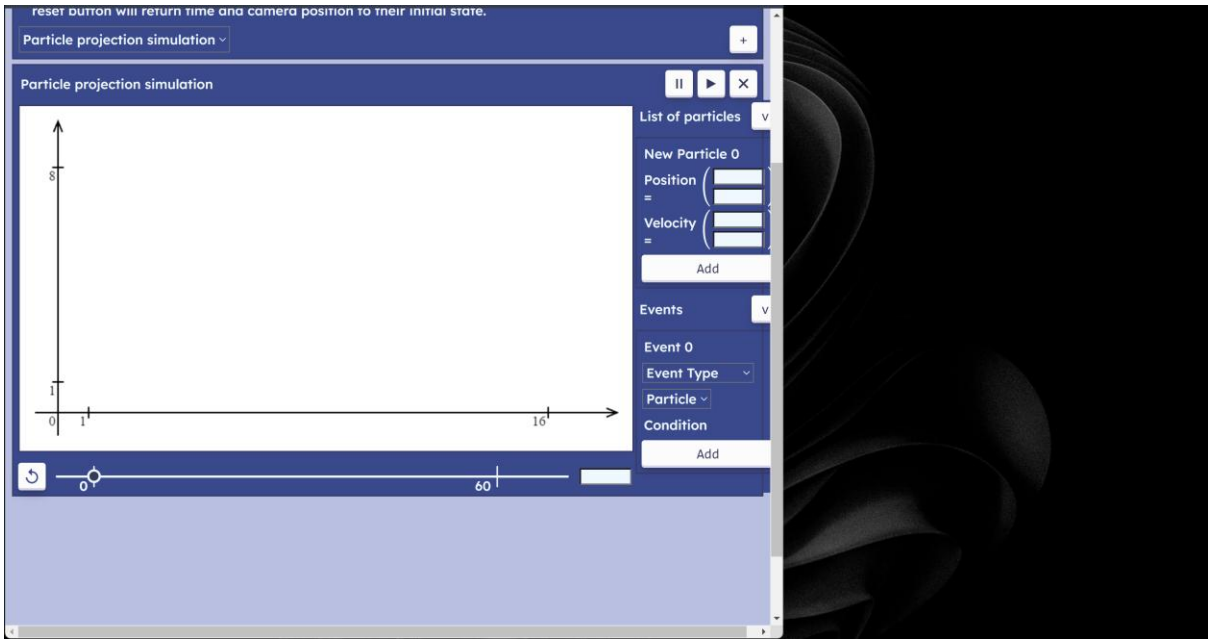


Mykyta Osovskyi
Dronfield Henry Fanshawe School



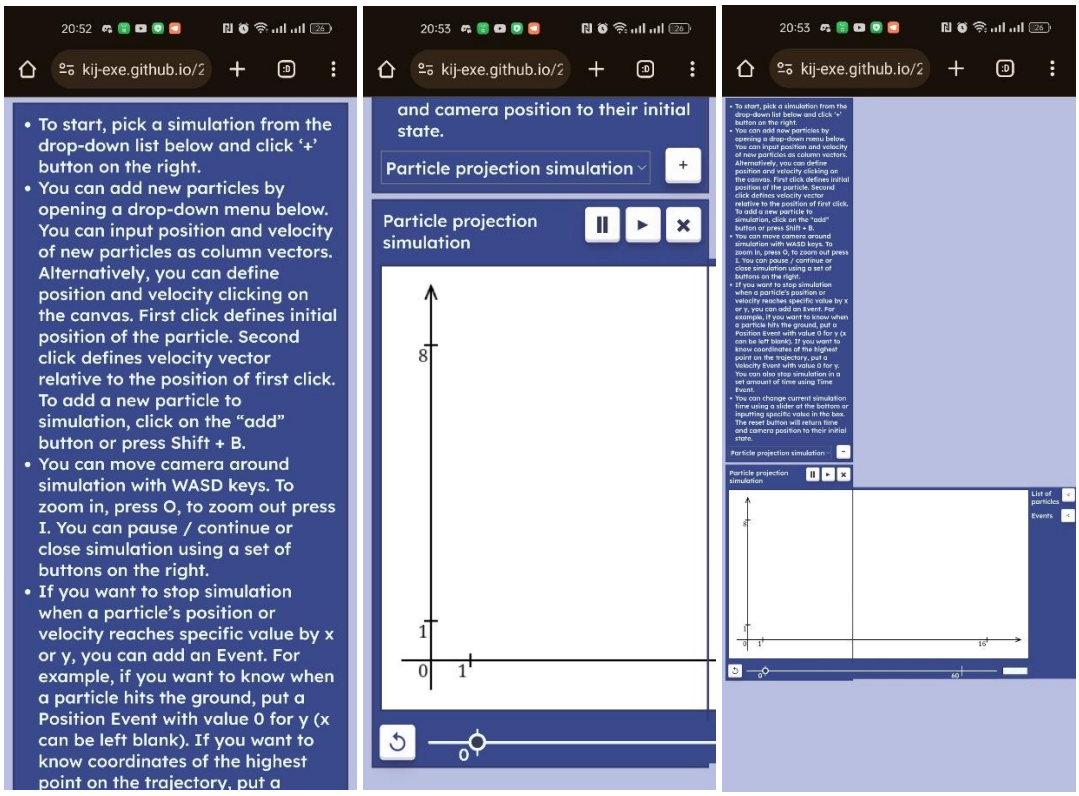
Interface elements wrap around to the point when each singular component on IO area is in a column. Font or individual element sizes do not change. Canvas retains its position regardless of how the user interacts with the website.

Mykyta Osovskyi
Dronfield Henry Fanshawe School



After that point, the IO area goes beyond the window.

Mobile devices:



When the website is first opened, it looks like that (first two screenshots on the top). When you open a simulation, it allows you to zoom out, but overall, the main page size remains unchanged. You can create particles and use all the same functionality as on desktops and laptops (apart from keyboard control).

There is a lot of work to be done before it is possible to use the program effectively on mobile devices. The page needs to adapt to the phone's orientation. The View and Model parts of the code

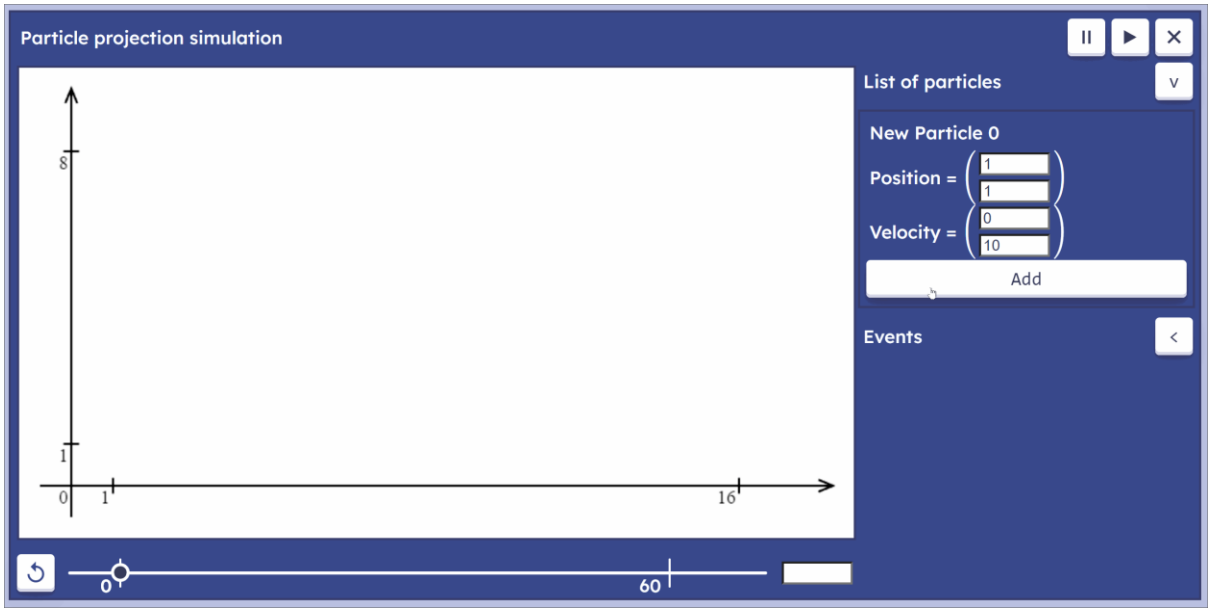
will not experience any changes because of that (however, initial translation and scale will be dictated by the size of the canvas). The change in size of all elements will be managed by IO handler, CSS, and HTML code.

5.3.4.4 Breaking cases

Even though the solution is overall very robust, there are two cases that were discovered during the tests that completely break simulation.

1) In some reasons, if General-Purpose Simulation is created, an error occurs that breaks the page completely and all new simulations would not work at all unless a user reloads the page. The General-Purpose Simulation has no functionality and should have been removed at all from the initial drop-down list.

2) 1D Projection that were discovered when I interviewed a mechanics teacher who attempted to launch a particle vertically:



Here I show an attempt to create a particle with velocity (0, 10). The canvas goes white and does not show anything unless a new simulation is created. I believe this problem occurs because the scale is set to 0 because of unhandled edge cases in evaluation of a rectangle of "significant" points. The same problem was discussed here.

5.4 Limitations

Main limitation of my program is a narrow range of problems that can be solved. The only problems that are currently covered by my program are basic top-down problems that require a couple of substitutions into formulas or solving a quadratic equation, at most. Examples of these problems are:

- find position / velocity of a particle after a certain amount of time;
- sketch trajectory of a particle;
- find position of the highest point on the trajectory;
- find position of the point where the particle collides with vertical / horizontal wall / hits the ground;
- find the time when the particle will be moving with the speed of 1 ms^{-1} in vertical direction.

However, much more problems are more complex and require students to first construct and equation and then solve it. For example, when initial velocity is unknown, but position of the highest point on trajectory is given. Students will still be able to use my program but only to validate their calculations by creating the same particle with velocity they calculated and checking if the highest point on trajectory is correct.

This process is tedious, and it is easier for students to open a solution bank to check if they got correct answers. Nevertheless, it can be helpful for teachers when they first present such problems to their students. My program can be helpful to show that their answers match reality by modelling this particle.

One of the ways to broaden the number of problems covered by the program is to introduce a so called “suvat” model, it can be used to solve most of the problems. Students commonly construct it for a particle moving from one point to the other. “s” is the distance moved from one point to the other, “u” is the velocity at the first point, “v” is the velocity at the second point, “a” is the acceleration, “t” is the time elapsed. Normally, 3 of these variables are required to calculate the other 2. This process can be easily formalised into an algorithm.

The other limitation that users are not yet able to input mathematical expressions. Importance of this feature was discussed before. Users will need to use calculator and copy results to input boxes. They will need to copy them to a lot of significant figures or model will not be accurate. It is very important to add this feature in future and as early as possible.
Angle and speed input for velocity goes under the same category. It would make it much easier for users to work from what they have in the question.

One of the weak points in my program is accessibility. There should be more alternative texts to some elements and more support for a screen reader where possible; a separate colour palette should be designed for colour blind people; their accessibility must be tested with colour-blind people who might use the software. As well as general accessibility, there is more to be done on cross browser compatibility, as can be seen from the robustness tests, some features of the program do not work as intended on some other browsers (one of which is Mozilla Firefox, one of the most popular browsers). It also applies to support for mobile devices. It is very relevant for students as they do not always have their computers available when they do maths homework.

Lastly, usability will largely benefit from including video explanations or virtual assistants to the page, who will guide the user the first time they load the page. As pointed out by one of the stakeholders, when the page is first opened, there is a lot of unoccupied space, which can be filled with videos of usage scenarios.

5.5 Maintenance

Throughout evaluation I highlighted a lot of points that will form a basis for future maintenance. It would need to start from fixing most obvious bugs and inputs that break the program (like 1D projection problem).

I tried to keep code self-commenting, but still left a lot of comments that clarify some algorithms. As stated in Pseudocode / Code conventions, most variable names are descriptive, for instance, “canvas_container” or “initial_position”. However, there are a lot of variables that are named “x” or “vx”, these are mainly used to implement formulas and they are typically followed by a sequence of comments.

The project is broken down into series of files where each file represents a single class. These files are split up into 5 folders by their purpose. 3 of them are Model, View and Controller parts of the solution according to Model-View-Controller architecture. The other one is “utility”, which is a collection of helper classes, like Vector and Priority Queue, and a static library of functions (for now, it has only functions for calculating quadratic and linear equations). The last one contains unit tests for Vector and Priority Queue classes.

This approach indeed helps to make modular code, however, the single largest file in all the program is “ParticleProjectionIO.js” which contains 1081 lines of code. This is because all HTML DOM elements are created, positioned, and styled manually:

```
let container = document.createElement("div");
container.style.display = "flex";
//   flex container for a button to extend
container.appendChild(add_button);
particle_input.appendChild(container);
```

This is how an element is created and added on the page.

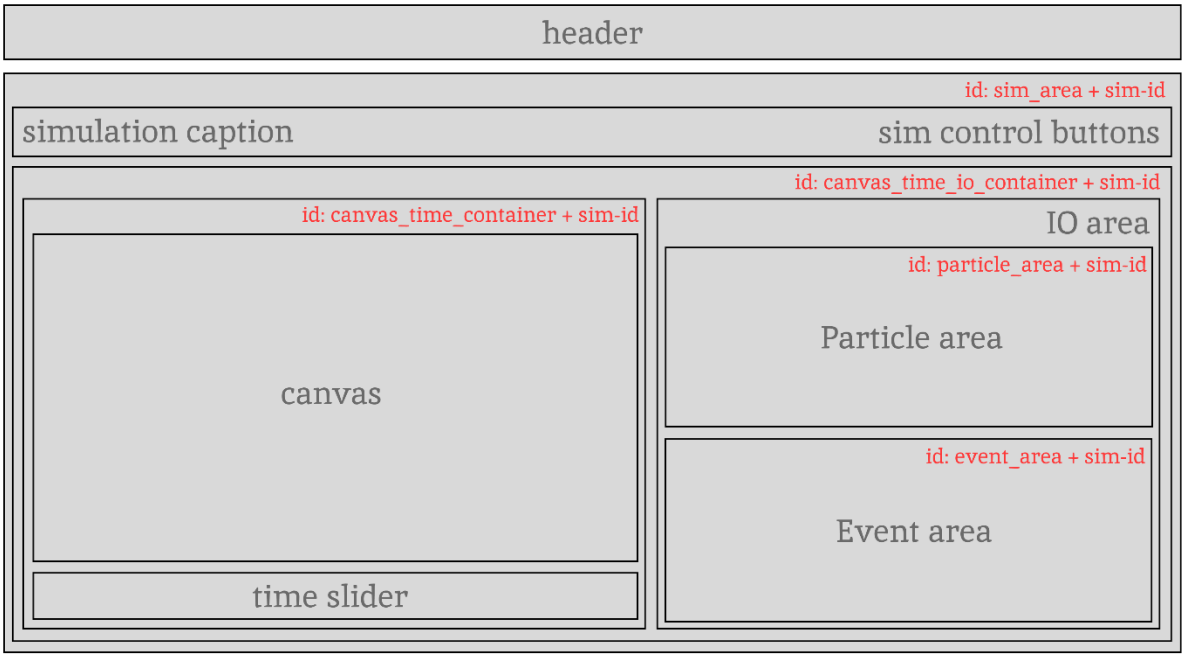
“document.createElement()” method has been used 72 times and “appendChild()” 74 times. A better, long-term practice would be to use a front-end framework like Angular or React. They provide possibilities to create interface components in much more abstract from HTML DOM way writing much less code. In future, I would need to transition to one of the dedicated frameworks.

Angular framework, for instance, is based on TypeScript instead of JavaScript. This is a transpiler of JavaScript. It offers static variable typing, enforcing explicit type declaration. It makes it much easier to catch syntax errors earlier during development. Even though there is more code to write, it is more readable. IDEs like Visual Studio code can straight away understand which class or interface a variable content belongs to and automatically suggest appropriate methods or give hints when a non-existent method is called.

All these features assist future maintenance of the project.

The Util library, that now contains static functions for solving quadratic and linear equations, can also be used in the Event class when calculating the time event occurs at. It simplifies the code. Furthermore, the Util library should contain validation functions and error messages, instead of storing them as constants inside of the IO Handler. For instance, have a function “isInRange()” that would check if a given value is in range of -10000 to 10000, “isNumeric()” that would check if a given string represents a number.
As an alternative to transitioning to one of the frontend frameworks, the Util library can contain several functions that create, position and style HTML elements. For instance, a function “createFlexBox(containter)” which will create a flex HTML element, put it inside of the container, passed as a parameter, and return the element. Or a “createParagraph(text)” method, which would create a <p> HTML element and set its initial innerHTML text to the value of the “text” parameter.

A large part of the IO handler is an ID system. Initial ID system was designed in this section. An id system allows to retrieve and change HTML elements after they have been created, not passing them as a parameter. Throughout the development, the initial ID system and overall layout changed a lot. In the next diagrams I will explain a conceptual layout of the page:

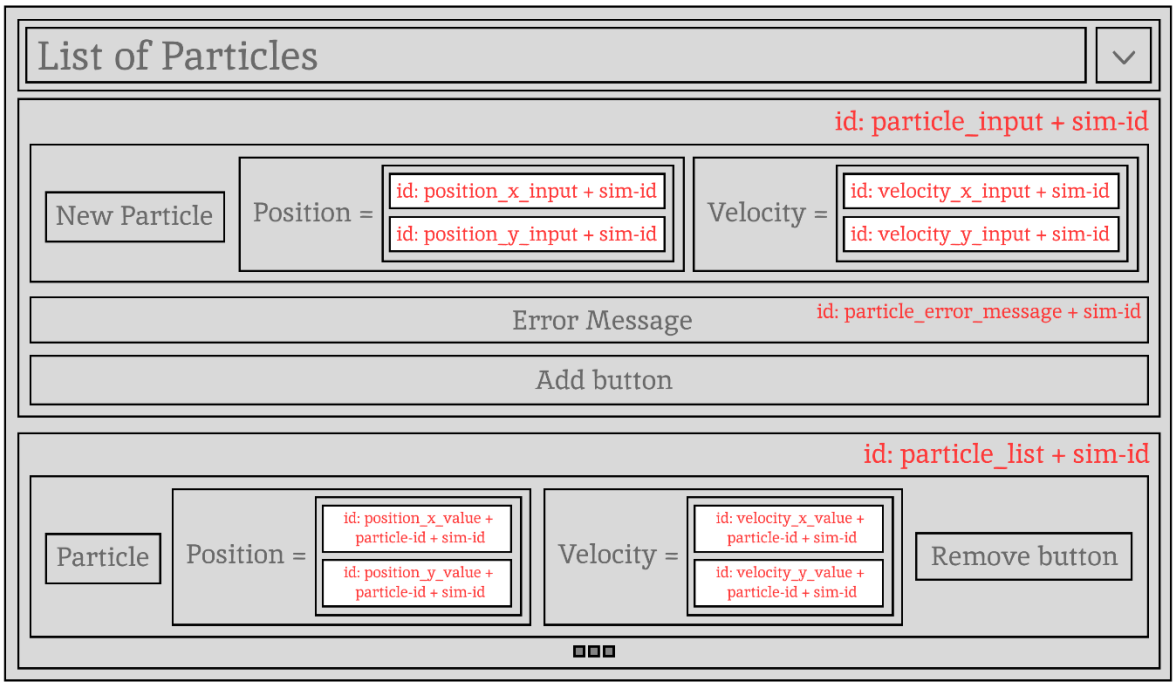


This is a current layout of the page, the header includes brief instructions text, simulation drop-down list and a button to add new particles. Simulation layout is below the header. All simulations (Particle Projection Simulations) share the same layout, except the “sim-id” component to every id. It starts from 0 for the first simulation and gets incremented by one for each consecutive simulation.

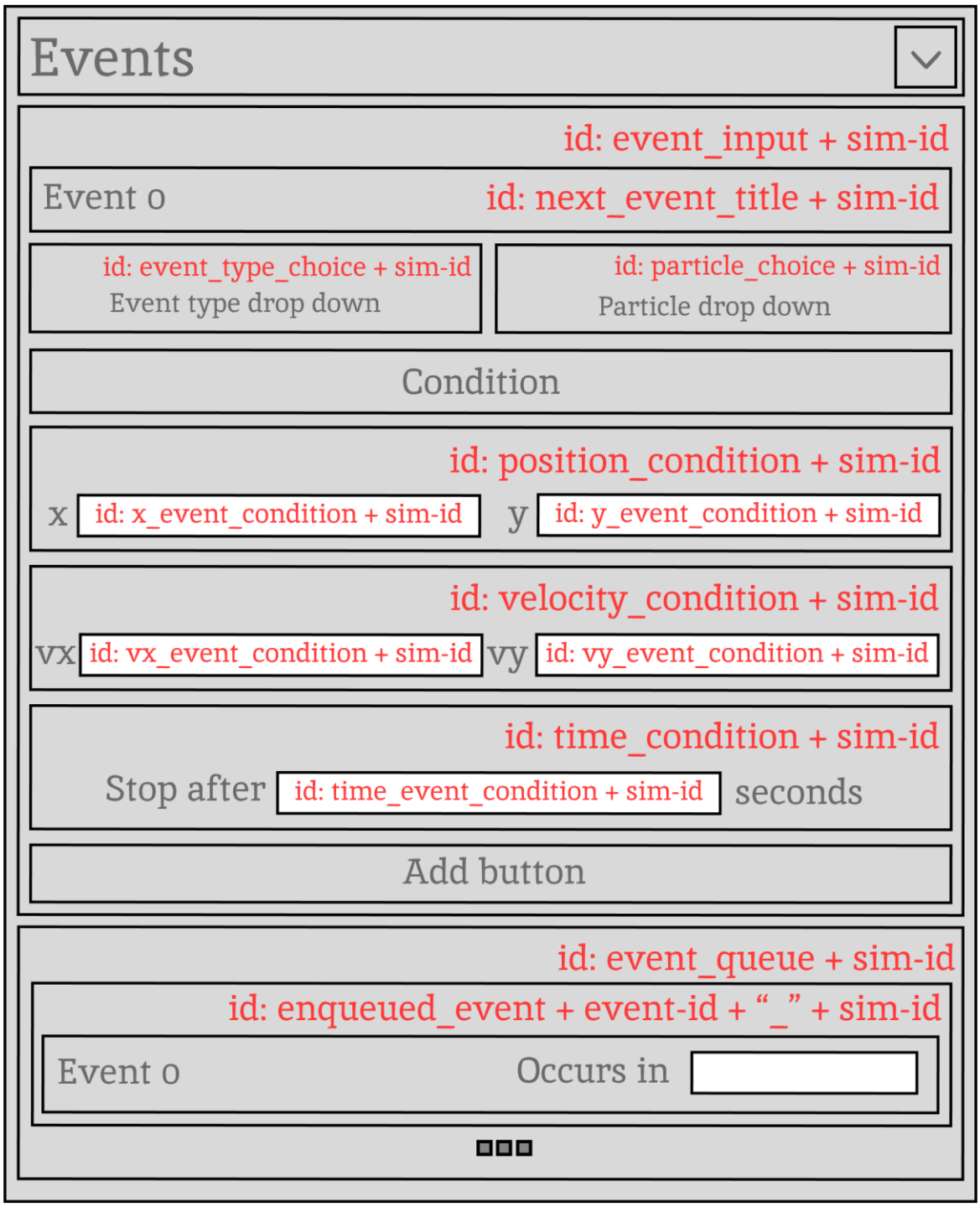
There is a further breakdown for time slider, particle area and event area:

A time slider (more appropriate to name it a time control area) consists of three components, the reset button, the time slider, and the text time input.

Mykyta Osovskyi
Dronfield Henry Fanshawe School



The particle input is toggled by clicking on “v” button, an error message becomes visible only when the user passes incorrect input and is hidden when the user correctly creates new particle. The particle list gets appended with new particles increasing an id of the next particle. Position and velocity outputs need ids to update the values when simulation is active.



Position / Velocity / Time condition blocks are not visible on the screen at the same time, but they are still on the page. When the user changes the event type, the block corresponding to the chosen value becomes visible.

Multilevel nested containers may be confusing, but they allow the page to be structured in a specific way. Therefore, it is an essential part of maintenance to note any changes to the ID system, especially because it is very hard to infer ID structure from the code.

The UML diagram from the design section is also very important to update because it experiences a lot of changes throughout development.

And lastly, a very important part of future maintenance is the consistent use of the code repository. Initially, when it was only me working on the project, I was able to keep in mind all decisions I made in implementing one or the other feature and if something broke, I was able to manually backtrack to the state of my project before it happened. As the number of features increases, it will become harder and harder. Implementation of each separate feature can be encapsulated to a separate git branch, where a person working on the feature will gradually implement it using a consistent commit convention, so a code reviewer will be able to track changes.

6 Code appendix

6.1 index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>2d Simulations</title>
    <link rel="stylesheet" href="style.css">

    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Lexend:wght@100..900&display=swap"
rel="stylesheet">

    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
</head>
<body>
    <header class="box" id="header" style="display: flex; flex-wrap: wrap;">
        <ul style="width: 100%;">
            <li>
                To start, pick a simulation from the drop-down list below and click ‘+’ button on the
right.
            </li>
            <li>
                You can add new particles by opening a drop-down menu below.
                You can input position and velocity of new particles as column vectors.
                Alternatively, you can define position and velocity clicking on the canvas.
                First click defines the initial position of the particle.
                Second click defines the velocity vector relative to the position of first click.
                To add a new particle to simulation, click on the “add” button or press Shift + B.
            </li>
            <li>
                You can move camera around simulation with WASD keys.
                To zoom in, press O, to zoom out press I.
                You can pause / continue or close simulation using a set of buttons on the
right.
            </li>
            <li>
                If you want to stop simulation when a particle’s position or velocity reaches specific
value by x or y, you can add an Event.
                For example, if you want to know when a particle hits the ground, put a Position Event
with value 0 for y (x can be left blank).
                If you want to know coordinates of the highest point on the trajectory, put a Velocity
Event with value 0 for y.
                You can also stop simulation in a set amount of time using Time Event.
            </li>
            <li>
                You can change current simulation time using a slider at the bottom or inputting
specific value in the box.
                The reset button will return time and camera position to their initial state.
            </li>
        </ul>
```

```
        <datalist id="marks">
            <option>0</option>
            <option>60</option>
        </datalist>

    </header>

    <script type="module">
        import Controller from "./src/controller/Controller.js"

        let controller = new Controller();
    </script>
</body>
</html>
```

6.2 style.css

```
html {
    background-color: #b9bfe0;
    font-size: 13pt;
    font-family: "Lexend", sans-serif;
    font-optical-sizing: auto;
    font-weight: 500;
    font-style: normal;
    min-height: 150%;
}

body * {
    background-color: #39498C;
    color:aliceblue;
}

button {
    margin: 3px;
    align-items: center;
    appearance: none;
    background-color: #FCFCFD;
    border-radius: 4px;
    border-width: 0;
    box-shadow: rgba(45, 35, 66, 0.4) 0 2px 4px,rgba(45, 35, 66, 0.3) 0 7px 13px -3px,#D6D6E7 0 -3px 0
inset;
    box-sizing: border-box;
    color: #36395A;
    cursor: pointer;
    display: inline-flex;
    font-family: "JetBrains Mono", monospace;
    height: 48px;
    justify-content: center;
    line-height: 1;
    list-style: none;
    overflow: hidden;
    padding-left: 16px;
    padding-right: 16px;
    position: relative;
    text-align: left;
    text-decoration: none;
    transition: box-shadow .15s,transform .15s;
    user-select: none;
    -webkit-user-select: none;
    touch-action: manipulation;
    white-space: nowrap;
    will-change: box-shadow,transform;
    font-size: 18px;
    width: 36px;
    height: 36px;
}

button:focus {
  box-shadow: #D6D6E7 0 0 0 1.5px inset, rgba(45, 35, 66, 0.4) 0 2px 4px, rgba(45, 35, 66, 0.3) 0 7px
13px -3px, #D6D6E7 0 -3px 0 inset;
}
```

```
button:hover {
  box-shadow: rgba(45, 35, 66, 0.4) 0 4px 8px, rgba(45, 35, 66, 0.3) 0 7px 13px -3px, #D6D6E7 0 -3px 0
inset;
  transform: translateY(-2px);
}

button:active {
  box-shadow: #D6D6E7 0 3px 7px inset;
  transform: translateY(2px);
}

p, label, ul {
    margin: 3px;
    padding: 3px;
    align-self: center;
}

ul {
    padding-left: 18px;
}

select {
    font-family: inherit;
    font-size: inherit;
    font-weight: inherit;
    margin: 4px;
}

.box {
    margin: 8px 3px 8px 3px;
    padding: 3px;
    outline: 2px solid #383D6D;
}

canvas {
    background-color: white;
    outline: 2px solid #383D6D;
    margin: 8px 3px 8px 5px;
    padding: 0;
    /* space before io_area */
}

.error_message {
    font-size: 10pt;
    color: #ff7e7e;
}

mtd {
    padding: 2px;
}

math, mrow, mi, mo, msup {
    color: inherit;
    font-weight: 600;
}

input[type="text"] {
    font-size: 11pt;
    width: 60px;
    height: 16px;
    align-self: center;
    color: #36395A;
    background-color: aliceblue;
}

input[type="range"] {
    -webkit-appearance: none;
    appearance: none;
    align-self: center;
    height: 3px;
    background-color: aliceblue;
    border-radius: 3px;

    align-self: center;
```

```
    margin: 0;
    z-index: 1;
}

input[type="range"]::-webkit-slider-thumb {
    -webkit-appearance: none;
    appearance: none;
    height: 12px;
    width: 12px;
    border-radius: 50%;
    outline: 3px solid aliceblue;
    background: #373d56;
    cursor: pointer;
    margin-top: -16px;

    transition: all .2s ease-in-out;
}

input[type="range"]::-webkit-slider-thumb:hover {
    transform: scale(1.1);
}

.tickmark {
    height: 28px;
    width: 2px;
    background-color: aliceblue;
    position: absolute;

    align-self: center;
    border-radius: 3px;
}

.ticktext {
    position: absolute;
    background-color: rgb(0, 0, 0, 0);
}
```

6.3 Controller

6.3.1 Controller.js

```
import ParticleProjectionSim from "../model/particle-projection/ParticleProjectionSim.js";
import GeneralPurposeSim from "../model/general-purpose/GeneralPurposeSim.js";
import ParticleProjectionIO from "./ParticleProjectionIO.js";
import GeneralPurposeIO from "./GeneralPurposeIO.js";
import ViewSim from "../view/ViewSim.js";


class Controller {
    constructor() {
        this.sim_list = [];
        this.view_list = [];
        this.io_list = [];
        //   lists of simulations, views and input/output handlers, the
        //   simulation, its view and io handler are stored by the same
        //   index in the list

        this.sim_names = ["Particle projection simulation"
                          /*"General-purpose simulation"*/];
        this.sim_classes = ["ParticleProjection"/*"GeneralPurpose"*/];
        //   lists of simulation names and their class identifiers
        //   can be easily changed if a new simulation is added

        this.createSimulationsDropDown();

        this.next_id = 0;
        //   id of the next simulation

        this.createAddSimButton();

        this.startUpdateLoop();
```

```
    }

    createSimulationsDropDown() {
        this.select = document.createElement("select");
        this.select.style.marginRight = "auto";
        //   creating a select element for a dropdown menu

        let option = document.createElement("option");
        option.value = -1;
        option.innerHTML = "Choose a simulation";
        this.select.appendChild(option);
        //   add a default option to the drop-down menu that prompts the
        //   user to choose a simulation

        for (let i = 0; i < this.sim_names.length; i++) {
            let option = document.createElement("option");
            //   creating an option
            option.value = i;
            //   value to associate an option with a simulation
            option.innerHTML = this.sim_names[i];
            //   fills option with the name of the simulation
            this.select.appendChild(option);
            //   add the option to the dropdown menu
        }
        //   value of each option will correspond to the index of
        //   corresponding simulation in the sim_list and to the
        //   name of its group of classes

        let header = document.getElementById("header");
        //   retreive the body tag from the page
        header.appendChild(this.select);
        //   add the drop-down on the page
    }

    createAddSimButton() {
        let add_sim_button = document.createElement("button");
        //   instantiating a button that creates a new simulation
        add_sim_button.innerHTML = "+";

        add_sim_button.onclick = () => {
            this.addSim();
        }
        //   assigning the function of adding a simulation to the button
        //   click on the button will invoke an add_sim function
        //   of the controller with associated index that is selected

        let header = document.getElementById("header");
        //   retreive the body tag from the page
        header.appendChild(add_sim_button);
    }

    startUpdateLoop() {
        requestAnimationFrame((timestamp) => {
            this.prev_timestamp = timestamp;
            //   the first instance when the update method is called
            this.update(timestamp);
            //   start the update loop
        });
    }

    update(timestamp) {
        let dt = timestamp - this.prev_timestamp;
        //   calculating time elapsed from the last frame
        this.prev_timestamp = timestamp;
        //   for the next update call to calculate dt

        for (let i = 0; i < this.sim_list.length; i++) {
            if (this.sim_list[i].isActive()) {
                //   update and redraw every active simulation
                this.sim_list[i].update(dt);
            }

            this.view_list[i].redraw();
            this.io_list[i].update();
```

```
        }

        requestAnimationFrame((timestamp) => {
            this.update(timestamp);
        });
    }

    addSim() {
        let index = parseInt(this.select.value);
        //   temporary variable for the option on the dropdown menu
        if (index === -1)
            return
        //   if no option is chosen (value of -1), no simulation is added

        let class_group = this.sim_classes[index];
        //   name of the group of classes the added simulation belongs to

        let sim_area = this.createSimArea();

        let sim = eval("new " + class_group + "Sim()");
        this.sim_list.push(sim);
        //   instantiating sim object for the new simulation and
        //   adding it to the list

        this.createSimHeader(sim, sim_area, index);

        let canvas_time_io_container = document.createElement("div");
        canvas_time_io_container.id = "canvas_time_io_container" + this.next_id;
        canvas_time_io_container.style.display = "flex";
        sim_area.appendChild(canvas_time_io_container);
        //   container for io area, canvas and time control

        let canvas_time_container = document.createElement("div");
        // canvas_time_container.style.display = "flex";
        // canvas_time_container.style.flexDirection = "column";
        canvas_time_container.id = "canvas_time_container" + this.next_id;
        canvas_time_io_container.appendChild(canvas_time_container);
        //   container for canvas and time control

        let view = new ViewSim(this.next_id, canvas_time_container);
        this.view_list.push(view);
        let io = eval("new " + class_group + "IO(this.next_id, sim, view, canvas_time_io_container)");
        this.io_list.push(io);
        //   creating ViewSim and IOHandler objects

        this.next_id++;
        //   increment id for the next simulation;
    }

    createSimArea() {
        let sim_area = document.createElement("div");
        sim_area.id = "sim_area" + this.next_id.toString();
        //   assigning an identifier to the new simulation area

        sim_area.classList.add("box");
        sim_area.style.marginTop = "10px";
        sim_area.style.marginBottom = "10px";

        //   sim area styling

        let body_element = document.getElementsByTagName("body")[0];
        //   retrieve the body element
        body_element.appendChild(sim_area);

        return sim_area;
    }

    createSimHeader(sim, sim_area, index) {
        let container = document.createElement("div");
        container.style.display = "flex";
        container.style.width = "100%";

        let title = document.createElement("p");
        title.innerHTML = this.sim_names[index];
```

```
        title.style.marginRight = "auto";
        //   creating a title paragraph html tag and setting its
        //   content to the name of the new simulation being added
        container.appendChild(title);

        this.createSimButtons(sim, container);

        sim_area.appendChild(container);
        //   adding buttons and the title on the sim_area
    }

    createSimButtons(sim, container) {
        let terminate_sim_button = document.createElement("button");
        terminate_sim_button.innerHTML = "&#10006;";
        terminate_sim_button.classList.add("sim_area_button");

        let pause_sim_button = document.createElement("button");
        pause_sim_button.innerHTML = "&#10073;&#10073;";
        pause_sim_button.classList.add("sim_area_button");

        let continue_sim_button = document.createElement("button");
        continue_sim_button.innerHTML = "&#9654;";
        continue_sim_button.classList.add("sim_area_button");
        //   instantiating buttons, labeling them and assigning
        //   a sim_area_button CSS class

        terminate_sim_button.onclick = () => {
            this.terminateSim(sim);
        }

        pause_sim_button.onclick = () => {
            sim.pause();
        }

        continue_sim_button.onclick = () => {
            sim.continue();
        }
        //   adding buttons that terminate, pause and continue the simulation
        //   and assigning to them corresponding function using clousers

        container.appendChild(pause_sim_button);
        container.appendChild(continue_sim_button);
        container.appendChild(terminate_sim_button);
    }

    terminateSim(sim) {
        for (let i = 0; i < this.sim_list.length; i++) {
            if (this.sim_list[i] == sim) {
                //   i will represent an index of the simulation in the lists
                let id = this.view_list[i].getId();
                //   getting an id of the simulation
                let sim_area = document.getElementById("sim_area" + id);
                sim_area.remove();
                //   remove entire simulation area

                this.sim_list.splice(i, 1);
                this.view_list.splice(i, 1);
                this.io_list.splice(i, 1);
                //   remove all references to the simulation from the controller lists

                return;
                //   stop the function as the simulation was found
            }
        }
    }
}


export {Controller as default}
```

6.3.2 GeneralPurposeIO.js

```
Class GeneralPurposeIO {
    constructor() {
        console.log("GeneralPurposeIO instantiated");
    }
}


export {GeneralPurposeIO as default}
```

6.3.3 IOHandler.js

```
class IOHandler {
    constructor(id, sim, view, io_canvas_container) {
        this.id = id;
        this.sim = sim;
        this.view = view;

        this.createIOarea(io_canvas_container);
    }

    createIOarea(io_canvas_container) {
        this.io_area = document.createElement("div");
        this.io_area.classList.add("container");
        // this.io_area.style.display = "flex";
        this.io_area.style.flexGrow = "1";
        //   create an io_area

        io_canvas_container.appendChild(this.io_area);
        //   put io_area on the page
    }

    initialize() {

    }
}

export {IOHandler as default};
```

6.3.4 ParticleProjectionIO.js

```
import IOHandler from "./IOHandler.js";
import PointMass from "../model/particle-projection/PointMass.js";
import Vector from "../utility/Vector.js";
import ViewPointMass from "../view/ViewPointMass.js";
import PositionEvent from "../model/particle-projection/PositionEvent.js"
import VelocityEvent from "../model/particle-projection/VelocityEvent.js"
import TimeEvent from "../model/particle-projection/TimeEvent.js"
import Trajectory from "../view/Trajectory.js";


class ParticleProjectionIO extends IOHandler {
    constructor(id, sim, view, io_canvas_container) {
        super(id, sim, view, io_canvas_container)

        this.next_particle_id = 0;
        this.next_event_id = 0;

        this.body_list = [];

        this.initialize();
    }

    initialize() {
        this.INCORRECT_TYPE_ERROR = `
        Input values must be numbers with a decimal point or in the format Xe+/-Y for
        <math xmlns="http://www.w3.org/1998/Math/MathML">
            <mrow>
                <mi>X</mi>
```

```
                <mo>*</mo>
                <msup>
                    <mi>10</mi>
                    <mrow>
                        <mo>+/-</mo>
                        <mi>Y</mi>
                    </mrow>
                </msup>
            </mrow>
        </math>
        `;
        this.OUT_OF_RANGE_VALUES  = "Input values must be between -10000 and 10000";
        this.INVALID_EVENT_ERROR = "Such event will never occur";
        this.PARTICLE_NOT_CHOSEN = "Choose the particle first";

        this.createParticleArea();
        this.createEventArea();
        this.createTimeControl();
        this.createKeyboardInput();
    }

    createParticleArea() {
        let particle_area = document.createElement("div");
        particle_area.id = "particle_area" + this.id;
        particle_area.style.flexGrow = "1";

        this.io_area.appendChild(particle_area);

        let header_container = document.createElement("div");
        header_container.style.display = "flex";
        //   header container
        particle_area.appendChild(header_container);

        let title = document.createElement("p");
        title.style.marginRight = "auto";
        title.innerHTML = "List of particles";
        header_container.appendChild(title);
        //   first the title is created

        this.createParticleMenuButton(header_container);
        //   then the button is created

        this.createParticleInput(particle_area);
        //   create an input area

        let particle_list = document.createElement("div");
        particle_list.id = "particle_list" + this.id;
        particle_area.appendChild(particle_list);
        //   then the particle list area
    }

    createParticleMenuButton(header_container) {
        let button = document.createElement("button");
        button.innerHTML = "<";

        button.onclick = () => {
            let particle_input = document.getElementById("particle_input" + this.id);

            if (particle_input.style.display == "none") {
                //   change button state to show that the menu is opened
                button.innerHTML = "v";
                //   reveal the menu
                particle_input.style.display = "block";

                //   start canvas input chain
                this.view.getCanvas().onclick = (event) => {
                    this.canvasPositionInput(event);
                }
            }
            else {
                //   change button state to show that the menu is closed
                button.innerHTML = "<";
                //   hide the menu
                particle_input.style.display = "none";
```

```
            //   remove canvas input
            this.view.getCanvas().onclick = null;

        }
        //   switches particle_input display
        //   between none and block
    }

    header_container.appendChild(button);
}

getPositionFromClickEvent(event) {
    //   Get the target (canvas element itself)
    const target = event.target;
    //   Get the bounding rectangle of target
    const rect = target.getBoundingClientRect();
    const x = event.clientX - rect.left;
    const y = event.clientY - rect.top;
    //   x and y relative to the top left corner
    //   with y axis pointing downwards
    return this.view.toSimSpace(new Vector(x, y));
}

//   input from the canvas
canvasPositionInput(event) {
    let position = this.getPositionFromClickEvent(event);

    document.getElementById("position_x_input" + this.id).value = position.getX();
    document.getElementById("position_y_input" + this.id).value = position.getY();

    this.view.getCanvas().onclick = (event) => {
        this.canvasVelocityInput(event, position);
    }
}

canvasVelocityInput(event, position) {
    let nextClick = this.getPositionFromClickEvent(event);
    let velocity = nextClick.subtracted(position);
    velocity.multiply(Math.sqrt(this.view.getScale()) / 4);

    document.getElementById("velocity_x_input" + this.id).value = velocity.getX();
    document.getElementById("velocity_y_input" + this.id).value = velocity.getY();

    this.view.getCanvas().onclick = (event) => {
        this.canvasPositionInput(event);
    }
}

createParticleInput(particle_area) {
    let particle_input = document.createElement("div");
    particle_input.classList.add("box");
    particle_input.style.display = "none";
    particle_input.id = "particle_input" + this.id;
    //   particle input area

    particle_area.appendChild(particle_input);

    let title = document.createElement("p");
    title.style.marginRight = "auto";
    title.id = "particle_input_title" + this.id;
    title.innerHTML = "New Particle " + this.next_particle_id;
    //   title for the area
    //   it must be updated every time new Particle is added

    let top_container = document.createElement("div");
    top_container.style.display = "flex";
    top_container.style.flexFlow = "wrap";
    top_container.appendChild(title);
    //   container for the title and inputs
    //   it separates them from the button

    particle_input.appendChild(top_container);
```

```
        this.createPositionInput(top_container);
        this.createVelocityInput(top_container);
        //   creating inputs for position and velocity

        this.createAddParticleButton(particle_input);
        //   button that adds new particles

        let error_message = document.createElement("p");
        error_message.id = "particle_error_message" + this.id;
        error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
        error_message.style.display = "none";
        error_message.classList.add("error_message");
        particle_input.appendChild(error_message);
        //   element to indicate an input error
    }

    createPositionInput(top_container) {
        let position_container = document.createElement("div");
        position_container.style.display = "flex";
        //   a container for all position input

        let position_title = document.createElement("p");
        position_title.innerHTML = "Position =  ";

        let column = this.createColumn(
            "position_x_input" + this.id,
            "position_y_input" + this.id
        );
        //   column input

        position_container.appendChild(position_title);
        position_container.appendChild(column);

        top_container.appendChild(position_container);
    }

    createVelocityInput(top_container) {
        let velocity_container = document.createElement("div");
        velocity_container.style.display = "flex";

        let velocity_title = document.createElement("p");
        velocity_title.innerHTML = "Velocity = ";

        let column = this.createColumn(
            "velocity_x_input" + this.id,
            "velocity_y_input" + this.id
        )

        velocity_container.appendChild(velocity_title);
        velocity_container.appendChild(column);

        top_container.appendChild(velocity_container);

    }

    createColumn(top_id, bottom_id) {
        let column = document.createElement("div");
        column.innerHTML = `
        <math xmlns="http://www.w3.org/1998/Math/MathML">
            <mrow>
                <mo>(</mo>
                <mtable>
                    <mtr>
                        <mtd><mi><input id="${top_id}" type="text" autocomplete="off"/></mi></mtd>
                    </mtr>
                    <mtr>
                        <mtd><mi><input id="${bottom_id}" type="text" autocomplete="off"/></mi></mtd>
                    </mtr>
                </mtable>
                <mo>)</mo>
            </mrow>
        </math>
        `
        return column;
```

```
    }

    createAddParticleButton(particle_input) {
        let container = document.createElement("div");
        container.style.display = "flex";
        //   flex container for a button to extend

        let add_button = document.createElement("button");
        add_button.innerHTML = "Add";
        add_button.style.flexGrow = "1";
        //   allows button to take up all space
        add_button.onclick = () => {
            this.addParticle();
        }

        container.appendChild(add_button);

        particle_input.appendChild(container);
    }

    addParticle() {
        let position_x = parseFloat(document.getElementById("position_x_input" + this.id).value);
        let position_y = parseFloat(document.getElementById("position_y_input" + this.id).value);
        let velocity_x = parseFloat(document.getElementById("velocity_x_input" + this.id).value);
        let velocity_y = parseFloat(document.getElementById("velocity_y_input" + this.id).value);
        //   retrieve values from inputs

        let error_message = document.getElementById("particle_error_message" + this.id);

        for (let value of [position_x, position_y, velocity_x, velocity_y]) {
            if (isNaN(value)) {
                error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
                error_message.style.display = "block";
                return;
            }

            if (value < -10000 || value > 10000) {
                error_message.innerHTML = this.OUT_OF_RANGE_VALUES;
                error_message.style.display = "block";
                return;
            }
        }
        //   validations

        let particle = new PointMass(
            new Vector(position_x, position_y),
            new Vector(velocity_x, velocity_y),
            new Vector(0, -9.8),
            this.sim.getTime(),
            this.next_particle_id++ //  get and increment at the same time
        );
        //   initialise new particle

        this.sim.addBody(particle);

        let view_particle = new ViewPointMass(particle, "black");
        this.view.addBody(view_particle);

        this.view.adjustMapping(particle);

        let trajectory = new Trajectory(particle, this.view.getScale());
        this.view.addBody(trajectory);

        this.body_list.push(particle);
        //   add the particle to io_list's own body_list

        this.createParticleOutput(particle);
        //   create an output window for the particle

        this.resetParticleChoice();
        //   reset the select element for the events
        //   because new particle has been added

        let particle_input_title = document.getElementById("particle_input_title" + this.id);
```

```
        particle_input_title.innerHTML = "New Particle " + this.next_particle_id;
        //   update title for the next particle

        error_message.style.display = "none";
        //   hide error message when particle is added successfully
    }

    createParticleOutput(particle) {
        let particle_list = document.getElementById("particle_list" + this.id);

        let output_container = document.createElement("div");
        output_container.classList.add("box");
        //   container for an output block

        let top_container = document.createElement("div");
        top_container.style.display = "flex";
        top_container.style.flexFlow = "wrap";
        //   container for a title and button

        let title = document.createElement("p");
        title.style.marginRight = "auto";
        title.innerHTML = "Particle " + particle.getId();
        top_container.appendChild(title);

        this.createPositionOutput(top_container, particle.getId());
        this.createVelocityOutput(top_container, particle.getId());
        //   creating outputs for each property

        output_container.appendChild(top_container);

        particle_list.appendChild(output_container);

        this.createRemoveButton(output_container, particle.getId());
        //   button to remove the particle after it is added
    }

    createPositionOutput(top_container, particle_id) {
        let position_output = document.createElement("div");
        position_output.style.display = "flex";
        //   container for an output block

        let position_title = document.createElement("p");
        position_title.innerHTML = "Position = ";

        let column = this.createColumn(
            "position_x_value" + particle_id + "_" + this.id,
            "position_y_value" + particle_id + "_" + this.id
        );

        position_output.appendChild(position_title);
        position_output.appendChild(column);

        top_container.appendChild(position_output);
    }

    createVelocityOutput(top_container, particle_id) {
        let position_output = document.createElement("div");
        position_output.style.display = "flex";

        let velocity_title = document.createElement("p");
        velocity_title.innerHTML = "Velocity = ";

        let column = this.createColumn(
            "velocity_x_value" + particle_id + "_" + this.id,
            "velocity_y_value" + particle_id + "_" + this.id
        );

        position_output.appendChild(velocity_title);
        position_output.appendChild(column);

        top_container.appendChild(position_output);
    }

    update() {
```

```
        for (let particle of this.body_list) {
            if (particle == null)
                continue;
            this.updatePositionOutput(particle);
            this.updateVelocityOutput(particle);
        }
        if (this.sim.isActive())
            this.updateThumb();
    }

    updatePositionOutput(particle) {
        let x_value = document.getElementById("position_x_value" + particle.getId() + "_" + this.id);
        let y_value = document.getElementById("position_y_value" + particle.getId() + "_" + this.id);
        //   retrieve html elements that represent output

        let position = particle.getPosition();

        x_value.value = Math.round(position.getX() * 100) / 100;
        y_value.value = Math.round(position.getY() * 100) / 100;
        //   assign values rounded to 2 dp
    }

    updateVelocityOutput(particle) {
        let x_value = document.getElementById("velocity_x_value" + particle.getId() + "_" + this.id);
        let y_value = document.getElementById("velocity_y_value" + particle.getId() + "_" + this.id);
        //   retrieve html elements that represent output

        let velocity = particle.getVelocity();

        x_value.value = Math.round(velocity.x * 100) / 100;
        y_value.value = Math.round(velocity.y * 100) / 100;
        //   assign values rounded to 2 dp
    }

    createRemoveButton(output_container, particle_id) {
        let button = document.createElement("button");
        button.style.width = "100px";
        //   fixed small width to avoid accidental clicks
        button.innerHTML = "Remove";

        button.onclick = () => {
            let particle = this.body_list[particle_id];
            this.sim.deleteBody(particle);
            this.view.deleteById(particle.getId());

            this.body_list[particle_id] = null;
            //   remove particle from sim, view and io

            this.resetParticleChoice();
            //   reset a particle drop down

            output_container.remove();
            button.remove();
            //   remove particle from HTML
        }

        output_container.appendChild(button);
        //   add button on the page
    }

    createEventArea() {
        let event_area = document.createElement("div");
        event_area.id = "event_area" + this.id;

        let container = document.createElement("div");
        container.style.display = "flex";

        let title = document.createElement("p");
        title.innerHTML = "Events";
        title.style.marginRight = "auto";

        container.appendChild(title)

        let occurred_events = document.createElement("div");
```

```
        occurred_events.id = "occurred_events" + this.id;

        event_area.append(container);
        event_area.appendChild(occurred_events);

        this.io_area.appendChild(event_area);

        this.createEventInput(event_area);

        this.createEventMenuButton(container);

        let event_queue = document.createElement("div");
        event_queue.id = "event_queue" + this.id;
        event_area.appendChild(event_queue);
    }

    createEventMenuButton(container) {
        let button = document.createElement("button");
        button.innerHTML = "<";

        button.onclick = () => {
            let event_input = document.getElementById("event_input" + this.id);

            if (event_input.style.display == "none") {
                event_input.style.display = "block";
                button.innerHTML = "v";
            }
            else {
                event_input.style.display = "none";
                button.innerHTML = "<";
            }
        }

        container.appendChild(button);
    }

    createEventInput(event_area) {
        let event_input = document.createElement("div");
        event_input.style.display = "none";
        event_input.classList.add("box");
        event_input.id = "event_input" + this.id;

        let title = document.createElement("p");
        title.id = "next_event_title" + this.id;
        title.innerHTML = "Event " + this.next_event_id;
        event_input.appendChild(title);

        this.createEventTypeChoice(event_input);
        //   drop-down menu for the event type

        let particle_drop_down = document.createElement("select");
        particle_drop_down.id = "particle_choice" + this.id;
        event_input.appendChild(particle_drop_down);
        //   create the drop-down for a particle

        let condition_title = document.createElement("p");
        condition_title.innerHTML = "Condition";
        event_input.appendChild(condition_title);

        event_area.appendChild(event_input);

        this.resetParticleChoice();
        //   reset particle drop-down


        this.createPositionEventInput(event_input);
        this.createVelocityEventInput(event_input);
        this.createTimeEventInput(event_input);

        let error_message = document.createElement("p");
        error_message.id = "event_error_message" + this.id;
        error_message.style.display = "none";
        error_message.classList.add("error_message");
```

```
        event_input.appendChild(error_message);

        this.createAddEventButton(event_input);
    }

    createEventTypeChoice(event_input) {
        let drop_down = document.createElement("select");
        drop_down.id = "event_type_choice" + this.id;

        let option = document.createElement("option");
        option.value = -1;
        option.innerHTML = "Event Type";
        drop_down.appendChild(option);
        //   default option

        let event_types = ["Position Event", "Velocity Event", "Time Event"];
        //   defines the order where
        //   0 - Position; 1 - Velocity; 2 - Time;

        for (let i = 0; i < event_types.length; i++) {
            let option = document.createElement("option");
            //   creating an option
            option.value = i;
            option.innerHTML = event_types[i];
            drop_down.appendChild(option);
            //   add the option to the dropdown menu
        }

        event_input.appendChild(drop_down);

        drop_down.onchange = () => {
            this.toggleCondition(drop_down);
        }
    }

    resetParticleChoice() {
        let drop_down = document.getElementById("particle_choice" + this.id);
        drop_down.innerHTML = "";
        //   empty previous contents

        let option = document.createElement("option");
        option.value = -1;
        option.innerHTML = "Particle";
        drop_down.appendChild(option);
        //   default option

        for (let i = 0; i < this.body_list.length; i++) {
            if (this.body_list[i] == null)
                continue;
            let option = document.createElement("option");
            //   creating an option
            option.value = i;
            option.innerHTML = "Particle " + i;
            drop_down.appendChild(option);
            //   add the option to the dropdown menu
        }
    }

    toggleCondition(drop_down) {
        let choice = drop_down.value;
        let position_condition = document.getElementById("position_condition" + this.id);
        let velocity_condition = document.getElementById("velocity_condition" + this.id);
        let time_condition = document.getElementById("time_condition" + this.id);

        for (let condition of [position_condition, velocity_condition, time_condition])
            condition.style.display = "none";
        //   turn off every condition input

        let add_button = document.getElementById("add_event_button" + this.id);

        switch (choice) {
            case "-1":
                add_button.onclick = null;
                break;
```

```
            case "0":
                //   Position Event
                position_condition.style.display = "block";
                add_button.onclick = () => {
                    this.addPositionEvent();
                }
                break;
            case "1":
                //   Velocity Event
                velocity_condition.style.display = "block";
                add_button.onclick = () => {
                    this.addVelocityEvent();
                }
                break;
            case "2":
                //   Time Event
                time_condition.style.display = "block";
                add_button.onclick = () => {
                    this.addTimeEvent();
                }
                break;
        }
    }

    resetNextEventTitle() {
        let title = document.getElementById("next_event_title" + this.id);
        title.innerHTML = "Event " + this.next_event_id;
    }

    createPositionEventInput(event_input) {
        let condition_container = document.createElement("div");
        condition_container.id = "position_condition" + this.id;
        condition_container.style.display = "none";

        let x_label = document.createElement("label");
        x_label.innerHTML = "x";

        let x_input = document.createElement("input");
        x_input.type = "text";
        x_input.id = "x_event_condition" + this.id;

        condition_container.appendChild(x_label);
        condition_container.appendChild(x_input);

        let y_label = document.createElement("label");
        y_label.innerHTML = "y";

        let y_input = document.createElement("input");
        y_input.type = "text";
        y_input.id = "y_event_condition" + this.id;

        condition_container.appendChild(y_label);
        condition_container.appendChild(y_input);

        event_input.appendChild(condition_container);
    }

    createVelocityEventInput(event_input) {
        let condition_container = document.createElement("div");
        condition_container.id = "velocity_condition" + this.id;
        condition_container.style.display = "none";

        let x_label = document.createElement("label");
        x_label.innerHTML = "vx";

        let x_input = document.createElement("input");
        x_input.type = "text";
        x_input.id = "vx_event_condition" + this.id;

        condition_container.appendChild(x_label);
        condition_container.appendChild(x_input);

        let y_label = document.createElement("label");
        y_label.innerHTML = "vy";
```

```
        let y_input = document.createElement("input");
        y_input.type = "text";
        y_input.id = "vy_event_condition" + this.id;

        condition_container.appendChild(y_label);
        condition_container.appendChild(y_input);

        event_input.appendChild(condition_container);
    }
    //   x -> vx, y -> vy

    createTimeEventInput(event_input) {
        let condition_container = document.createElement("div");
        condition_container.id = "time_condition" + this.id;
        condition_container.style.display = "none";

        let label_start = document.createElement("label");
        label_start.innerHTML = "Stop after";

        let label_finish = document.createElement("label");
        label_finish.innerHTML = "seconds";

        let time_input = document.createElement("input");
        time_input.type = "text";
        time_input.id = "time_event_condition" + this.id;

        condition_container.appendChild(label_start);
        condition_container.appendChild(time_input);
        condition_container.appendChild(label_finish);

        event_input.appendChild(condition_container);
    }

    createAddEventButton(event_input) {
        let button_container = document.createElement("div");
        button_container.style.display = "flex";

        let add_button = document.createElement("button");
        add_button.style.flexGrow = "1";
        add_button.id = "add_event_button" + this.id;
        add_button.innerHTML = "Add";
        //   onclick is defined when event type is chosen

        button_container.appendChild(add_button);

        event_input.appendChild(button_container);
    }

    addPositionEvent() {
        let x_value = document.getElementById("x_event_condition" + this.id).value;
        let y_value = document.getElementById("y_event_condition" + this.id).value;
        //   retrieving user input for conditions

        let error_message = document.getElementById("event_error_message" + this.id);
        error_message.style.display = "none";

        let particle_id = parseInt(document.getElementById("particle_choice" + this.id).value);
        if (particle_id == -1) {
            error_message.innerHTML = this.PARTICLE_NOT_CHOSEN;
            error_message.style.display = "block";
            return;
        }
        //   validating particle choice
        let particle = this.body_list[particle_id];
        //   retrieving the required particle

        let event = new PositionEvent(particle, this, this.next_event_id);

        if (!isNaN(x_value) && !(x_value === ""))
            event.setXtime(parseFloat(x_value));
            //   validating x_value
        else if (!isNaN(y_value) && !(y_value === ""))
            event.setYtime(parseFloat(y_value));
```

```
            //   validating y_value
        else {
            error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
            error_message.style.display = "block";
            return;
        }

        if (!event.isValid()) {
            error_message.innerHTML = this.INVALID_EVENT_ERROR;
            error_message.style.display = "block";
            return;
        }

        this.sim.addEvent(event);

        this.enqueueEvent(event, this.sim);

        this.next_event_id++;
        this.resetNextEventTitle();
    }

    addVelocityEvent() {
        let x_value = document.getElementById("vx_event_condition" + this.id).value;
        let y_value = document.getElementById("vy_event_condition" + this.id).value;
        //   retrieving user input for conditions

        let error_message = document.getElementById("event_error_message" + this.id);
        error_message.style.display = "none";

        let particle_id = parseInt(document.getElementById("particle_choice" + this.id).value);
        if (particle_id == -1) {
            error_message.innerHTML = this.PARTICLE_NOT_CHOSEN;
            error_message.style.display = "block";
            return;
        }
        //   validating particle choice
        let particle = this.body_list[particle_id];
        //   retrieving the required particle

        let event = new VelocityEvent(particle, this, this.next_event_id);

        if (!isNaN(x_value) && !(x_value === ""))
            event.setXtime(parseFloat(x_value));
            //   validating x_value
        else if (!isNaN(y_value) && !(y_value === ""))
            event.setYtime(parseFloat(y_value));
            //   validating y_value
        else {
            error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
            error_message.style.display = "block";
            return;
        }

        if (!event.isValid()) {
            error_message.innerHTML = this.INVALID_EVENT_ERROR;
            error_message.style.display = "block";
            return;
        }

        this.sim.addEvent(event);

        this.enqueueEvent(event, this.sim);

        this.next_event_id++;
        this.resetNextEventTitle();
    }

    addTimeEvent() {
        let time = document.getElementById("time_event_condition" + this.id).value;

        let error_message = document.getElementById("event_error_message" + this.id);
        error_message.style.display = "none";

        let particle_id = parseInt(document.getElementById("particle_choice" + this.id).value);
```

```
        if (particle_id == -1) {
            error_message.innerHTML = this.PARTICLE_NOT_CHOSEN;
            error_message.style.display = "block";
            return;
        }
        //   validating particle choice
        let particle = this.body_list[particle_id];

        let event = new TimeEvent(particle, this, this.next_event_id);

        if (isNaN(time) || (time === "")) {
            error_message.innerHTML = this.INCORRECT_TYPE_ERROR;
            error_message.style.display = "block";
            return;
        }

        event.setTime(parseFloat(time), this.sim.getTime());
        if (!event.isValid()) {
            error_message.innerHTML = this.INVALID_EVENT_ERROR;
            error_message.style.display = "block";
            return;
        }

        this.sim.addEvent(event);

        this.enqueueEvent(event, this.sim);

        this.next_event_id++;
        this.resetNextEventTitle();
    }

    enqueueEvent(event, sim) {
        let event_queue = document.getElementById("event_queue" + this.id);

        let enqueued_event = document.createElement("div");
        enqueued_event.id = "enqueued_event" + event.getId() + "_" + this.id;
        enqueued_event.classList.add("box");
        enqueued_event.style.display = "flex";
        enqueued_event.style.flexFlow = "wrap";

        let title = document.createElement("p");
        title.innerHTML = "Event " + event.getId();
        title.style.marginRight = "auto";
        enqueued_event.appendChild(title);

        let occurs_in_label = document.createElement("label");
        occurs_in_label.innerHTML = "Occurs in";

        let occurs_in_value = document.createElement("input");
        let time = (event.getTime() - sim.getTime()) / 1000;
        //   calculating the time before event occurs and converting it to seconds
        occurs_in_value.value = Math.round(time * 1000) / 1000;

        enqueued_event.appendChild(occurs_in_label);
        enqueued_event.appendChild(occurs_in_value);

        event_queue.appendChild(enqueued_event);
    }

    executeEvent(event) {
        let enqueued_event = document.getElementById("enqueued_event" + event.getId() + "_" +
this.id);
        enqueued_event.remove();
    }

    createTimeControl() {
        let canvas_time_container = document.getElementById("canvas_time_container" + this.id);

        let time_control_container = document.createElement("div");
        time_control_container.style.display = "flex";

        canvas_time_container.appendChild(time_control_container);

        this.createResetButton(time_control_container);
```

```
        this.createTimeSlider(time_control_container);
        this.createTimeInput(time_control_container);
    }

    createResetButton(time_control_container) {
        let button = document.createElement("button");
        button.innerHTML = '<i class="material-icons" style="background-color: rgba(0, 0, 0, 0);
color: #39498C">&#xe042;</i>';

        button.onclick = () => {
            this.sim.resetTime();
            this.updateThumb();
            //   update positions
            this.view.resetScale();
            this.view.resetTranslation();
            //   reset mapping of simulation space on the screen
        }

        time_control_container.appendChild(button);
    }

    createTimeSlider(time_control_container) {
        let slider_container = document.createElement("div");
        slider_container.style.display = "flex";
        slider_container.style.flexGrow = 1;
        slider_container.style.position = "relative";
        slider_container.style.marginLeft = "10px";
        slider_container.style.marginRight = "10px";

        let slider = document.createElement("input");
        slider.id = "time_slider" + this.id;
        slider.type = "range";
        slider.style.flexGrow = 1;

        slider.min = -5;
        slider.max = 70;
        slider.value = 0;
        slider.step = "any";

        let datalist = document.createElement("datalist");
        for (let i of [0, 60]) {
            let option = document.createElement("option");
            option.innerHTML = i;
            datalist.appendChild(option);
        }
        slider_container.appendChild(datalist);
        slider.setAttribute("list", "marks");

        slider.oninput = () => {
            this.sim.pause();
            this.sim.resetTime(parseFloat(slider.value) * 1000);

            this.updateThumb();
        }

        time_control_container.appendChild(slider_container);
        slider_container.appendChild(slider);

        this.createTickMarks(slider_container);

        let thumb_text = document.createElement("span");
        thumb_text.id = "thumb_text" + this.id;
        thumb_text.classList.add("ticktext");
        thumb_text.style.top = "22px";
        slider_container.appendChild(thumb_text);
        this.updateThumb();
    }

    createTickMarks(slider_container) {
        let tick0 = document.createElement("div");
        tick0.classList.add("tickmark");
        tick0.style.left = `calc(6px + 5 * (100% - 12px) / 75 - 1px)`;
        slider_container.appendChild(tick0);
```

```
        let ticktext0 = document.createElement("span");
        ticktext0.innerHTML = 0;
        ticktext0.classList.add("ticktext");
        ticktext0.style.top = "22px";
        ticktext0.style.right = `calc(6px + 70 * (100% - 12px) / 75 + 8px)`;
        slider_container.appendChild(ticktext0);

        let tick60 = document.createElement("div");
        tick60.classList.add("tickmark");
        tick60.style.left = `calc(6px + 65 * (100% - 12px) / 75 - 1px)`;
        slider_container.appendChild(tick60);

        let ticktext60 = document.createElement("span");
        ticktext60.innerHTML = 60;
        ticktext60.classList.add("ticktext");
        ticktext60.style.top = "22px";
        ticktext60.style.right = `calc(6px + 10 * (100% - 12px) / 75 + 8px)`;
        slider_container.appendChild(ticktext60);
    }

    updateThumb() {
        let slider = document.getElementById("time_slider" + this.id);
        let thumb_text = document.getElementById("thumb_text" + this.id);

        let value = this.sim.getTime() / 1000;
        let clamped_value = Math.max(-5, Math.min(value, 70));
        slider.value = clamped_value;

        thumb_text.innerHTML = Math.round(value);
        thumb_text.style.right = `calc(6px + ${70 - clamped_value} * (100% - 12px) / 75 + 8px)`

    }

    createTimeInput(time_control_container) {
        let input = document.createElement("input");
        input.type = "text";
        input.style.marginRight = "4px";
        input.style.marginLeft = "4px";

        input.onchange = () => {
            let value = input.value;
            if (!isNaN(value)) {
                this.sim.pause();
                this.sim.resetTime(parseFloat(value) * 1000);
                this.updateThumb();
            }
        }

        time_control_container.appendChild(input);
    }

    createKeyboardInput() {
        let canvas = this.view.getCanvas();
        let key_dictionary = {};

        canvas.onkeyup = canvas.onkeydown = (event) => {
            const is_pressed = event.type == "keydown";
            key_dictionary[event.code] = is_pressed;

            if (key_dictionary["KeyW"]) {
                this.view.increaseTranslationBy(new Vector(0, 2 + event.shiftKey*10));
            }
            if (key_dictionary["KeyA"]) {
                this.view.increaseTranslationBy(new Vector(2 + event.shiftKey*10, 0));
            }
            if (key_dictionary["KeyS"]) {
                this.view.increaseTranslationBy(new Vector(0, -2 - event.shiftKey*10));
            }
            if (key_dictionary["KeyD"]) {
                this.view.increaseTranslationBy(new Vector(-2 - event.shiftKey*10, 0));
            }
            if (key_dictionary["KeyI"]) {
                this.view.increaseScaleBy(1/1.2);
            }
```

```
            if (key_dictionary["KeyO"]) {
                this.view.increaseScaleBy(1.2);
            }
            if (key_dictionary["Space"]) {
                event.preventDefault();
                this.sim.toggle();
            }
            if (key_dictionary["Backspace"]) {
                this.sim.resetTime();
                this.updateThumb();
                //   update positions
            }
            if (key_dictionary["KeyB"] && event.shiftKey) {
                this.addParticle();
            }
        }
    }
}

export {ParticleProjectionIO as default}
```

6.4 Model

6.4.1 general-purpose

6.4.1.1 Disc.js

```
class Disc {

}
```

6.4.1.2 GeneralPurposeSim.js

```
import Simulation from "../Simulation.js";


class GeneralPurposeSim extends Simulation {
    constructor() {
        super();
        console.log("GeneralPurposeSim instantiated");
    }
}


export {GeneralPurposeSim as default};
```

6.4.1.3 Polygon.js

```
import RigidBody from "../general-purpose/RigidBody.js";


class Polygon extends RigidBody {
    constructor(id, vertices) {
        super(id);
        this.vertices = vertices;
    }

    getVertices() {
        return this.vertices;
    }
}

export {Polygon as default};
```

6.4.1.4 RigidBody.js

```
import Body from "../Body.js";
```

```
class RigidBody extends Body {
    constructor(id) {
        super(id);
    }
}

export {RigidBody as default};
```

6.4.2 particle-projection

6.4.2.1 Event.js

```
class Event {
    constructor(body, io_handler, id) {
        this.body = body;
        //   assigning a body to the event
        this.occurs_at = -1;
        //   an attribute representing the time that event occurs at
        this.io_handler = io_handler;
        //   reference to the input output handler of this simulation
        this.id = id;
        //   an id of this particular event (will be assigned by the
        //   IO Handler)
    }

    getTime() {
        return this.occurs_at;
    }

    getId() {
        return this.id;
    }

    isValid() {
        return this.occurs_at > 0;
        //   return true if the time the event occurs is greater than
        //   0 (returns false if occurs_at is -1 or 0, meaning that event
        //   is not valid or not yet specified)

    }

    execute() {
        this.io_handler.executeEvent(this);
        //   provides enough details for an io handler to produce
        //   relevant output on the screen
    }

    static compare(event1, event2) {
        return event2.occurs_at - event1.occurs_at;
        //   will return a positive number whenever the second event
        //   occurs later, in this way priorotising the earlier events
    }
}

export {Event as default};
```

6.4.2.2 ParticleProjectionSim.js

```
import Simulation from "../Simulation.js";
import PriorityQueue from "../../utility/PriorityQueue.js";
import Event from "./Event.js"


class ParticleProjectionSim extends Simulation {
    constructor() {
        super();
        this.event_queue = new PriorityQueue(Event.compare);
    }
```

```
    update(dt) {
        //   parameter dt denotes the change in time from the previous
        //   frame
        this.time += dt;

        let event = this.event_queue.peek();
        //   peek the first event from the queue
        // console.log(dt);

        if (!this.event_queue.isEmpty() && event.getTime() < this.time) {
            //   if the time the event occurs is greater than the
            //   current time the simulation is at
            event.execute();
            this.pause();
            //   execute the procedure defined when event was created
            this.time = event.getTime();
            //   set the time of the simulation to the moment when
            //   event has occured
            this.event_queue.pop();
            //   remove event from the queue
        }

        for (let i = 0; i < this.body_list.length; i++) {
            this.body_list[i].update(this.time);
        }
    }

    addEvent(event) {
        this.event_queue.push(event);
    }
}


export {ParticleProjectionSim as default};
```

6.4.2.3 PointMass.js

```
import Body from "../Body.js";


class PointMass extends Body {
    constructor(initial_position, initial_velocity, acceleration, created_at, id = null) {
        super(id);
        this.initial_position = initial_position;
        this.initial_velocity = initial_velocity;
        this.acceleration = acceleration;
        this.created_at = created_at;

        this.position = initial_position;
        this.velocity = initial_velocity;
    }

    getInitialPosition() {
        return this.initial_position;
    }

    getInitialVelocity() {
        return this.initial_velocity;
    }

    createdAt() {
        return this.created_at;
    }

    //   time in seconds since the particle was created
    calculatePositionAt(time) {
        let ds = this.initial_velocity.multiplied(time);
        //   initialising ds variable that represents change in position
        //   and calculating the first part of the equation

        ds.add(this.acceleration.multiplied(time * time / 2));
```

```
        //   adding the change caused by acceleration

        return this.initial_position.added(ds);
    }

    //   time in miliseconds since simulation started
    update(time) {
        let t = (time - this.created_at) / 1000;

        this.position = this.calculatePositionAt(t);
        //   updating position

        let dv = this.acceleration.multiplied(t);
        //   initialising change in velocity variable
        this.velocity = this.initial_velocity.added(dv);
        //   updating velocity
    }
}

export {PointMass as default};
```

6.4.2.4 PositionEvent.js

```
import Vector from "../../utility/Vector.js";
import Event from "./Event.js";


class PositionEvent extends Event {
    constructor(body, io_handler, id) {
        super(body, io_handler, id);
    }

    calculateTime(value, axis) {
        const a = this.body.getAcceleration().dot(axis) / 2;
        const b = this.body.getInitialVelocity().dot(axis);
        const c = this.body.getInitialPosition().dot(axis) - value;
        //   calculating coefficients for the quadratic equation
        //   from the equation of motion in a form of ax^2+bx+c=0

        if (a == 0 && b == 0) {
            this.occurs_at = -1;
            return;
            //   invalid if both a and b are zero
        }

        if (a == 0) {
            //   if only a is zero, then there might be a solution
            let x = (-1) * c / b;
            if (x <= 0)
                this.occurs_at = -1;
            else
                this.occurs_at = x * 1000 + this.body.createdAt();
            return
        }

        let D = b * b - 4 * a * c;
        //   calculating discriminant

        if (D < 0) {
            this.occurs_at = -1;
            return
            //   end function
        }

        let x1 = ((-1) * b - Math.sqrt(D)) / (2 * a);
        let x2 = ((-1) * b + Math.sqrt(D)) / (2 * a);

        if (x2 < x1) {
            let temp = x2;
            x2 = x1;
            x1 = temp;
            //   swap x1 and x2 so x2 >= x1
```

```
        }
        //   two solutions where x1 <= x2
        console.log(x1, x2);


        if (x2 <= 0) {
            this.occurs_at = -1
            return
        }
        //   if the largest solution is negative, then the particle
        //   never reaches the value

        if (x1 <= 0)
            this.occurs_at = x2 * 1000 + this.body.createdAt();
        else
            this.occurs_at = x1 * 1000 + this.body.createdAt();
        //   if the first value is positive, then the particle first
        //   reaches the value after x1 seconds and the second time
        //   after x2 seconds
    }

    setXtime(value) {
        this.calculateTime(value, new Vector(1, 0));
    }

    setYtime(value) {
        this.calculateTime(value, new Vector(0, 1));
    }
}


export {PositionEvent as default};
```

6.4.2.5 TimeEvent.js

```
import Event from "./Event.js";


class TimeEvent extends Event {
    constructor(body, io_handler, id) {
        super(body, io_handler, id);
    }

    setTime(time, current_time) {
        this.occurs_at = time * 1000 + current_time;
    }
}

export {TimeEvent as default};
```

6.4.2.6 VelocityEvent.js

```
import Vector from "../../utility/Vector.js";
import Event from "./Event.js";


class VelocityEvent extends Event {
    constructor(body, io_handler, id) {
        super(body, io_handler, id);
    }

    calculateTime(value, axis) {
        let b = this.body.getAcceleration().dot(axis);
        let c = this.body.getInitialVelocity().dot(axis) - value;
        //   calculating coefficients for the equation
        //   bx + c = 0

        if (b == 0) {
            this.occurs_at = -1;
            return
```

```
            //   invalid if b is zero
        }

        let x = (-1) * c / b;
        if (x <= 0)
            this.occurs_at = -1;
        else
            this.occurs_at = x * 1000 + this.body.createdAt();
    }

    setXtime(value) {
        this.calculateTime(value, new Vector(1, 0));
    }

    setYtime(value) {
        this.calculateTime(value, new Vector(0, 1));
    }
}


export {VelocityEvent as default};
```

6.4.3 Body.js

```
import Vector from "../utility/Vector.js";


class Body {
    constructor(id) {
        this.position = new Vector();
        this.velocity = new Vector();
        this.acceleration = new Vector();

        this.id = id;
    }

    getPosition() {
        return this.position;
    }

    getVelocity() {
        return this.velocity;
    }

    getAcceleration() {
        return this.acceleration;
    }

    getId() {
        return this.id;
    }
}


export {Body as default};
```

6.4.4 Simulation.js

```
class Simulation {
    constructor() {
        this.is_active = false;
        this.body_list = [];
        this.time = 0;
    }

    isActive() {
        return this.is_active;
    }

    pause() {
```

```
        this.is_active = false;
    }

    toggle() {
        this.is_active = !this.is_active;
    }

    resetTime(value=0) {
        this.time = value;
        this.update(0);
    }

    continue() {
        // console.log("Time start: " + new Date().getTime());
        this.is_active = true;
    }

    getTime() {
        return this.time;
    }

    addBody(body) {
        this.body_list.push(body);
    }

    deleteBody(body_to_delete) {
        for (let i = 0; i < this.body_list.length; i++) {
            if (this.body_list[i] == body_to_delete)
                this.body_list.splice(i, 1);
        }
    }

    update() {
    }
}


export {Simulation as default};
```

6.5 tests

6.5.1 PriorityQueue.test.js

```
import PriorityQueue from "../utility/PriorityQueue.js"


//   function for initialising the basic priority queue:
//   [null, 30, 25, 20, 10, 3, 12, 18, 2, 4]
//   used for both stage 1 and 2 tests
function initializeQueue() {
    let pq = new PriorityQueue((a, b) => a - b);
    //   priority queue with the comparator that
    //   prioritises the larger element
    for (let element of [30, 25, 20, 10, 3, 12, 18, 2, 4]) {
        pq.push(element);
    }
    return pq;
}


describe("Stage 1 testing", () => {
    let pq = initializeQueue();

    test("test (0)", () => {
        expect(pq.heap).toEqual([null, 30, 25, 20, 10, 3, 12, 18, 2, 4]);
    });

    test("test (1)", () => {
        pq.push(12)
        expect(pq.heap).toEqual([null, 30, 25, 20, 10, 12, 12, 18, 2, 4, 3]);
    });
```

```
    test("test (2)", () => {
        pq.push(29)
        expect(pq.heap).toEqual([null, 30, 29, 20, 10, 25, 12, 18, 2, 4, 3, 12]);
    });

    test("test (3)", () => {
        pq.push(-10)
        expect(pq.heap).toEqual([null, 30, 29, 20, 10, 25, 12, 18, 2, 4, 3, 12, -10]);
    });

    test("test (4)", () => {
        pq.push(42)
        expect(pq.heap).toEqual([null, 42, 29, 30, 10, 25, 20, 18, 2, 4, 3, 12, -10, 12]);
    });
});

describe("Stage 2 testing", () => {
    let pq = initializeQueue();

    test("test (1)", () => {
        expect(pq.pop()).toEqual(30);
        expect(pq.heap).toEqual([null, 25, 10, 20, 4, 3, 12, 18, 2]);
        expect(pq.peek()).toEqual(25);
    });

    test("test (2)", () => {
        expect(pq.pop()).toEqual(25);
        expect(pq.heap).toEqual([null, 20, 10, 18, 4, 3, 12, 2]);
        expect(pq.peek()).toEqual(20);
    });

    test("test (3)", () => {
        expect(pq.pop()).toEqual(20);
        expect(pq.heap).toEqual([null, 18, 10, 12, 4, 3, 2]);
        expect(pq.peek()).toEqual(18);
    });

    test("test (4)", () => {
        expect(pq.pop()).toEqual(18);
        expect(pq.heap).toEqual([null, 12, 10, 2, 4, 3]);
        expect(pq.peek()).toEqual(12);
    });

    test("test (5)", () => {
        expect(pq.pop()).toEqual(12);
        expect(pq.heap).toEqual([null, 10, 4, 2, 3]);
        expect(pq.peek()).toEqual(10);
    });

    test("test (6)", () => {
        expect(pq.pop()).toEqual(10);
        expect(pq.heap).toEqual([null, 4, 3, 2]);
        expect(pq.peek()).toEqual(4);
    });

    test("test (7)", () => {
        expect(pq.pop()).toEqual(4);
        expect(pq.heap).toEqual([null, 3, 2]);
        expect(pq.peek()).toEqual(3);
    });

    test("test (8)", () => {
        expect(pq.pop()).toEqual(3);
        expect(pq.heap).toEqual([null, 2]);
        expect(pq.peek()).toEqual(2);
    });

    test("test (9)", () => {
        expect(pq.pop()).toEqual(2);
        expect(pq.heap).toEqual([null]);
        expect(pq.peek()).toEqual(null);
    });
```

```
    test("test (10)", () => {
        expect(pq.pop()).toEqual(null);
        expect(pq.heap).toEqual([null]);
        expect(pq.peek()).toEqual(null);
    });
});
```

6.5.2 Vector.test.js

```
import Vector from "../utility/Vector.js"

describe("addition / subtraction", () => {
    test("(2,9) + (4,8)", () => {
        expect(new Vector(2,9).added(new Vector(4,8))).toEqual(new Vector(6,17));
    })

    test("(2,9) - (4,8)", () => {
        expect(new Vector(2,9).subtracted(new Vector(4,8))).toEqual(new Vector(-2,1));
    })

    test("(-10,-11) + (3,3)", () => {
        expect(new Vector(-10,-11).added(new Vector(3,3))).toEqual(new Vector(-7,-8));
    })

    test("(-10,-11) - (3,3)", () => {
        expect(new Vector(-10,-11).subtracted(new Vector(3,3))).toEqual(new Vector(-13,-14));
    })

    test("(3,3) - (-10,-11)", () => {
        expect(new Vector(3,3).added(new Vector(-10,-11))).toEqual(new Vector(-7,-8));
    })

    test("(3,3) - (-10,-11)", () => {
        expect(new Vector(3,3).subtracted(new Vector(-10,-11))).toEqual(new Vector(13,14));
    })
});

describe("Multiplication / division", () => {
    test("(6,12) * 3", () => {
        expect(new Vector(6,12).multiplied(3)).toEqual(new Vector(18,36));
    })

    test("(6,12) / 3", () => {
        expect(new Vector(6,12).divided(3)).toEqual(new Vector(2,4));
    })

    test("(-7,200) * 4", () => {
        expect(new Vector(-7,200).multiplied(4)).toEqual(new Vector(-28,800));
    })

    test("(-7,200) / 4", () => {
        expect(new Vector(-7,200).divided(4)).toEqual(new Vector(-1.75,50));
    })

    test("(0,90) * -10", () => {
        expect(new Vector(0,90).multiplied(-10)).toEqual(new Vector(-0,-900));
    })

    test("(0,90) / -10", () => {
        expect(new Vector(0,90).divided(-10)).toEqual(new Vector(-0,-9));
    })
})

describe("dot product", () => {
    test("(0,10) . (-7,9)", () => {
        expect(new Vector(0,10).dot(new Vector(-7,9))).toEqual(90);
    })

    test("(2,2) . (-1,2)", () => {
        expect(new Vector(2,2).dot(new Vector(-1,2))).toEqual(2);
    })
```

```
    test("(0,10) . (-7,9)", () => {
        expect(new Vector(0,10).dot(new Vector(-7,0))).toEqual(0);
    })
})

describe("length", () => {
    test("length of (3,4)", () => {
        expect(new Vector(3,4).length()).toEqual(5);
    })

    test("length of (12,-5)", () => {
        expect(new Vector(12,-5).length()).toEqual(13);
    })
})


describe("lengthSquared", () => {
    test("squared length of (2,3)", () => {
        expect(new Vector(2,3).lengthSquared()).toEqual(13);
    })

    test("squared length of (6,-10)", () => {
        expect(new Vector(6,-10).lengthSquared()).toEqual(136);
    })
})

describe("normalisation", () => {
    test("normalise (2,3)", () => {
        expect(new Vector(3,-4).normalized()).toEqual(new Vector(0.6,-0.8));
    })

    test("normalise (80,-60)", () => {
        expect(new Vector(80,-60).normalized()).toEqual(new Vector(0.8,-0.6));
    })
})

describe("rotation", () => {
    function rotatedRounded(vector, angle, axis = new Vector()) {
        let rotated_vector = vector.rotatedBy(angle, axis);
        rotated_vector.x = Math.trunc(rotated_vector.x * 1000) / 1000;
        rotated_vector.y = Math.trunc(rotated_vector.y * 1000) / 1000;
        return rotated_vector;
    }

    test("(5,8) rotated by 1 radian around (0,0)", () => {
        expect(rotatedRounded(new Vector(5, 8), 1)).toEqual(new Vector(-4.030, 8.529));
    })

    test("(-20, 8.1) rotated by 13.8 radian around (0,0)", () => {
        expect(rotatedRounded(new Vector(-20, 8.1), 13.8)).toEqual(new Vector(-14.260, -16.194));
    })

    test("(4, 7) rotated by -Pi/2 radian around (0,0)", () => {
        expect(rotatedRounded(new Vector(4, 7), -Math.PI/2)).toEqual(new Vector(7, -3.999));
    })

    test("(3, 3) rotated by -Pi/2 radian around (2, 1)", () => {
        expect(rotatedRounded(new Vector(3, 3), -Math.PI/2, new Vector(2, 1))).toEqual(new Vector(4,
0));
    })

    test("(5,2) rotated by Pi/4 radian around (3,4)", () => {
        expect(rotatedRounded(new Vector(5, 2), Math.PI/4, new Vector(3, 4))).toEqual(new
Vector(5.828, 3.999));
    })
})
```

6.6 utility

6.6.1 PriorityQueue.js

```
class PriorityQueue {
    constructor(compare) {
        this.heap = [null];
        //   a list that is used to store the heap, the first element is
        //   null because the heap starts at index 1
        this.next_index = 1;
        //   index of the next free space in the list
        this.compare = compare;
    }

    isEmpty() {
        return this.next_index == 1;
    }

    rise(index) {
        while (index > 1) {
            //   while the current item is not in the first node
            //   (because the root node cannot rise anymore)
            let parent_index = Math.floor(index / 2);
            //   getting index of the parent element
            let item = this.heap[index];
            let parent_item = this.heap[parent_index];
            if (this.compare(item, parent_item) > 0) {
                //   if the item is larger than its parent, perform a
                //   swap
                this.heap[index] = parent_item;
                this.heap[parent_index] = item;

                index = parent_index;
                //   the next node to consider is now a parent of
                //   the previous node
            }
            else {
                //   if the item is less than its parent, then the
                //   property is satisfied
                break
                //   exit the loop
            }
        }
    }

    push(item) {
        this.heap[this.next_index] = item;
        //   add item to the heap
        this.rise(this.next_index);
        //   fix the property of the heap
        this.next_index += 1;
        //   increment the index for the next element
    }

    sink(index) {
        let largest_child_index;
        //   declaring variable
        while (2 * index < this.next_index) {
            //   index of the left child of the node is 2 * index
            //   therefore this condition is satisfied as long as there
            //   at least one child of the node with this index
            //   if there are no children, then the node is already
            //   on the correct place

            if (2 * index + 1 < this.next_index) {
                //    if there is right child
                let left_item = this.heap[2 * index];
                let right_item = this.heap[2 * index + 1];

                if (this.compare(right_item, left_item) > 0) {
                    //   if the right child is larger
                    largest_child_index = 2 * index + 1;
                }
```

```
                else {
                    largest_child_index = 2 * index;
                }
            }
            else {
                largest_child_index = 2 * index;
            }
            //   previous block of code finds the largest child of
            //   the current node

            let parent_item = this.heap[index];
            let largest_child_item = this.heap[largest_child_index];

            if (this.compare(largest_child_item, parent_item) > 0) {
                //   if the largest child is larger than its parent
                //   perform a swap
                this.heap[largest_child_index] = parent_item;
                this.heap[index] = largest_child_item;

                index = largest_child_index;
                //   the next node to consider is now at this index
            }
            else {
                break;
            }
            //   exit the loop otherwise
        }
    }

    pop() {
        //   if the queue is empty, return null
        if (this.isEmpty())
            return null;

        let item = this.heap[1];
        //   required item temporarily stored

        this.heap[1] = this.heap[this.next_index - 1];
        //   place the last elememt at the top
        this.next_index -= 1;
        //   decrement the next_index attribute

        this.heap.splice(this.next_index, 1)
        //   delete previously last element

        this.sink(1);
        //   make the new first top element of the queue to go down
        //   so the property is maintained
        return item;
    }

    peek() {
        if (this.isEmpty())
            return null;

        return this.heap[1];
    }

}

export {PriorityQueue as default};
```

6.6.2 Vector.js

```
class Vector {
    constructor(x = 0, y = 0) {
        this.x = x;
        this.y = y;
    }

    setX(value) {
        this.x = value;
```

```
    }

    getX() {
        return this.x;
    }

    setY(value) {
        this.y = value;
    }

    getY(value) {
        return this.y;
    }
    //   basic set of getters and setters

    add(vector) {
        this.x += vector.x;
        this.y += vector.y;
    }

    added(vector) {
        let x = this.x + vector.x;
        let y = this.y + vector.y;
        return new Vector(x, y);
    }

    subtract(vector) {
        let x = this.x - vector.x;
        let y = this.y - vector.y;
    }

    subtracted(vector) {
        let x = this.x - vector.x;
        let y = this.y - vector.y;
        //   creating temporary variables for new x and y variables,
        //   original vector is unchanged
        return new Vector(x, y);
    }

    multiply(scalar) {
        this.x *= scalar;
        this.y *= scalar;
    }

    multiplied(scalar) {
        let x = this.x * scalar;
        let y = this.y * scalar;
        return new Vector(x, y);
    }

    divide(scalar) {
        this.x /= scalar;
        this.y /= scalar;
    }

    divided(scalar) {
        let x = this.x / scalar;
        let y = this.y / scalar;
        return new Vector(x, y);
    }

    dot(other_vector) {
        //   returns the dot product between the vector on which the method is called
        //   and the vector passed as a parameter
        //   has a commutative property: a.dot(b) = b.dot(a)
        return this.x * other_vector.x + this.y * other_vector.y;
    }

    lengthSquared() {
        return this.x * this.x + this.y * this.y;
        //   can also be represented as this.dot(this)
    }

    length() {
```

```
        return Math.sqrt(this.lengthSquared());
    }

    normalize() {
        if (this.x === 0 && this.y === 0)
            throw new Error("Zero vector cannot be normalised");
        this.divide(this.length());
    }

    normalized() {
        if (this.x === 0 && this.y === 0)
            throw new Error("Zero vector cannot be normalised");
        return this.divided(this.length());
    }

    rotatedBy(angle, axis = new Vector()) {
        //   defined as anticlockwise rotation by angle
        let rotated_vector = this.subtracted(axis);
        //   first tranlate the axis to the origin

        rotated_vector = new Vector(
            rotated_vector.x * Math.cos(angle) - rotated_vector.y * Math.sin(angle),
            rotated_vector.x * Math.sin(angle) + rotated_vector.y * Math.cos(angle)
        );
        //   rotate vector like that

        rotated_vector.add(axis);
        //   translate the axis back
        return rotated_vector;
    }

    reflectedInX() {
        let new_y = this.y * (-1);
        return new Vector(this.x, new_y);
    }

    reflectedInY() {
        let new_x = this.x * (-1);
        return new Vector(new_x, this.y);
    }

    //   Allows to easily spread vector coordinates using
    //   spread operator
    [Symbol.iterator] = function* () {
        yield this.x;
        yield this.y;
    }
}


export {Vector as default};
```

6.6.3 Util.js

```
class Util {
    //   define as accepting a, b, c for an equation of the form
    //   ax^2 + bx + c = 0
    //   output list of two roots (in ascending order)
    //   if there is only one root, both elements are the same
    //   if there no roots, both elements are NaN
    static solveQuadratic(a, b, c) {
        let D = b * b - 4 * a * c;
        //   calculating discriminant

        if (D < 0) {
            return [NaN, NaN];
        }

        let x1 = ((-1) * b - Math.sqrt(D)) / (2 * a);
        let x2 = ((-1) * b + Math.sqrt(D)) / (2 * a);

        if (x2 < x1) {
```

```
            let temp = x2;
            x2 = x1;
            x1 = temp;
            //   swap x1 and x2 so x2 >= x1
        }
        //   two solutions where x1 <= x2

        return [x1, x2]
    }

    //   b, c for
    //   bx + c = 0
    //   output a single number
    //   if no roots, output NaN
    static solveLinear(b, c) {
        if (b == 0) {
            return NaN;
            //   no roots if b is zero
        }

        let x = (-1) * c / b;
        return x;
    }
}

export {Util as default};
```

6.7 view

6.7.1 CoordinateAxes.js

```
import Vector from "../utility/Vector.js";


class CoordinateAxes {
    constructor() {
    }

    redraw(view) {
        let scale = view.getScale();

        view.renderTextByTopRight(new Vector(-3,-3), "0")

        view.drawArrow(
            new Vector(-30, 0),
            new Vector(730, 0)
        );
        //   x-axis

        view.drawArrow(
            new Vector(0, -30),
            new Vector(0, 380)
        );
        // y-axis

        this.drawHorizontalTickMark(
            new Vector(0, scale),
            "1", view
        )

        this.drawVerticalTickMark(
            new Vector(scale, 0),
            "1", view
        )

        let x_tick_value = Math.floor(650 / scale);
        let y_tick_value = Math.floor(330 / scale);

        this.drawHorizontalTickMark(
            new Vector(0, y_tick_value * scale),
            y_tick_value, view
```

```
        )

        this.drawVerticalTickMark(
            new Vector(x_tick_value * scale, 0),
            x_tick_value, view
        )

        // consider what happens when 1 > 350 / this.scale
        // redisgn such that not the value is rounded but the position of the tickmark is adjusted
        }

        drawHorizontalTickMark(center, text, view) {
            this.drawTickMark(center, text, view, new Vector(7, 0));
        }

        drawVerticalTickMark(center, text, view) {
            this.drawTickMark(center, text, view, new Vector(0, 7));
        }

        drawTickMark(center, text, view, offset) {
            view.drawLine(
                center.subtracted(offset),
                center.added(offset)
            )

            view.renderTextByTopRight(center.added(new Vector(-3, -3)), text);
        }
}

export {CoordinateAxes as default};
```

6.7.2 Trajectory.js

```
import Vector from "../utility/Vector.js";
import ViewBody from "./ViewBody.js";


class Trajectory extends ViewBody {
    constructor(body, scale, color="grey", layer=0) {
        super(body, color, layer);
        this.starting_point = body.getInitialPosition();

        let x0 = this.starting_point.getX();
        let y0 = this.starting_point.getY();
        let vx = body.getInitialVelocity().getX();
        let vy = body.getInitialVelocity().getY();
        let a = -9.8;
        // set of constant parameters

        let f = (x) => {
            return y0 + vy*(x - x0)/vx + a*(x - x0)*(x - x0)/(2*vx*vx);
        }
        // y = f(x)

        let df = (x) => {
            return vy/vx + a*(x - x0)/(vx*vx);
        }
        // derivative of f(x)

        let x2 = x0 + vx * (60/Math.sqrt(scale));
        let y2 = f(x2);
        this.end_point = new Vector(x2, y2);

        let m0 = df(x0);
        let m2 = df(x2);
        //   derivatives of two tangents

        let x1 = (y2 - y0 + m0*x0 - m2*x2) / (m0 - m2);
        //   x coordinate of the intersection of two tangents
        this.control_point = new Vector(x1, y0 + m0 * (x1 - x0));
    }
```

```
    redraw(view) {
        view.drawParabolaBy(this.starting_point, this.control_point, this.end_point, this.color);
    }
}

export {Trajectory as default};
```

6.7.3 ViewBody.js

```
class ViewBody {
    constructor(body, color="black", layer=1) {
        this.body = body;
        this.color = color;
        this.layer = layer;
    }

    getBody() {
        return this.body;
    }

    getId() {
        return this.body.getId();
    }

    getLayer() {
        return this.layer;
    }

    redraw(view) {
    }
}


export {ViewBody as default};
```

6.7.4 ViewDisc.js

```
class ViewDisc {
    constructor() {
    }
}
```

6.7.5 ViewPointMass.js

```
import ViewBody from "./ViewBody.js";


class ViewPointMass extends ViewBody {
    constructor(body, color="black", layer=1) {
        super(body, color, layer);
        //   body is an instance of PointMass in this case
    }

    redraw(view) {
        view.drawPoint(this.body.getPosition(), this.color);
        //   the position of the point mass is its centre
    }
}


export {ViewPointMass as default};
```

6.7.6 ViewPolygon.js

```
import ViewBody from "./ViewBody.js";
```

```
class VeiwPolygon extends ViewBody {
    constructor(body, color="black", layer=2) {
        super(body, color, layer);
    }

    redraw(view) {
        view.drawPolygon(this.body.getVertices(), this.color, false, true);
    }
}


export {VeiwPolygon as default};
```

6.7.7 ViewSim.js

```
import Polygon from "../model/general-purpose/Polygon.js";
import PointMass from "../model/particle-projection/PointMass.js";
import Util from "../utility/Util.js";
import Vector from "../utility/Vector.js";
import CoordinateAxes from "./CoordinateAxes.js";
import ViewPointMass from "./ViewPointMass.js";
import VeiwPolygon from "./ViewPolygon.js";


class ViewSim {
    constructor(id, canvas_time_io_container) {
        this.id = id;
        this.body_list = [];
        // initialising set of attributes to their initial values

        this.canvas = document.createElement("canvas");
        //   creating the canvas html element
        this.canvas.width = 800;
        this.canvas.height = 450;
        //   width and height must be assigned in JS
        //   so the graphics is not stretched to fit the space
        this.canvas.style.alignSelf = "start";
        //   prevent the canvas from stretching
        this.canvas.tabIndex = 1;
        //   make canvas focusable

        canvas_time_io_container.appendChild(this.canvas);
        //   add canvas on the page

        this.ctx = this.canvas.getContext("2d");
        //   getting access to "context" of the canvas for drawing purposes
        this.ctx.lineWidth = 2;

        this.target_scale = 40;
        this.INITIAL_SCALE = 40;
        this.scale = 40;

        this.target_translation = new Vector(0 + 50, this.canvas.height - 50);
        this.INITIAL_TRANSLATION = new Vector(0 + 50, this.canvas.height - 50);
        this.translation = new Vector(0 + 50, this.canvas.height - 50);

        this.coordinate_axes = new CoordinateAxes();

    }

    getId() {
        return this.id;
    }

    getCanvas() {
        return this.canvas;
    }

    getScale() {
        return this.scale;
    }
```

```
    resetScale() {
        this.target_scale = this.INITIAL_SCALE;
    }

    resetTranslation() {
        this.target_translation = this.INITIAL_TRANSLATION;
    }

    redraw() {
        this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
        //   clearing canvas area from the previous frame

        this.scale += (this.target_scale - this.scale) / 15;
        this.translation.add(this.target_translation.subtracted(this.translation).divided(15));

        this.coordinate_axes.redraw(this);

        for (let i = 0; i < this.body_list.length; i++) {
            this.body_list[i].redraw(this);
            //   redrawing each body separately, if bodies are stored by layers,
            //   then first bodies will be below later bodies
        }
    }

    addBody(body) {
        this.body_list.push(body);
        let i = this.body_list.length - 1;
        while (i > 0 && this.body_list[i - 1].getLayer() > this.body_list[i].getLayer()) {
            //   while the end of the list has not been reached
            //   and the body at (i) is on lower layers than body (i+1)
            let temp = this.body_list[i - 1];
            this.body_list[i - 1] = this.body_list[i]
            this.body_list[i] = temp;
            //   swap them
            i--;
        }
    }

    deleteBody(body) {
        for (let i = 0; i < this.body_list.length; i++) {
            if (this.body_list[i] == body || this.body_list[i].getBody() == body) {
                this.body_list.splice(i, 1);
                break;
            }
        }
    }
    // when adding layers, implement insertion sort as the list will be partially sorted

    deleteById(id) {
        for (let i = 0; i < this.body_list.length; i++) {
            if (this.body_list[i].getId() == id) {
                this.body_list.splice(i, 1);
                i--;
            }
        }
    }

    //   converting coordinates from simulation space
    toCanvas(vector) {
        let new_vector = vector.reflectedInX();
        //   reflecting the vector
        new_vector.multiply(this.scale);
        //   scaling relative to the origin
        new_vector.add(this.translation);
        //   translating the vector to a new position
        return new_vector;
    }

    toSimSpace(vector) {
        let new_vector = vector.subtracted(this.translation);
        new_vector.divide(this.scale);
        return new_vector.reflectedInX();
    }
```

```
    scaled(value) {
        return value * this.scale;
    }

    drawPolygon(vertices, color="black", toFill=true, toStroke=false) {
        this.ctx.moveTo(...this.toCanvas(vertices[0]));
        this.ctx.beginPath();
        //   start path at the first vertex
        for (let vertex of vertices) {
            let translated_vertex = this.toCanvas(vertex);
            //   translate to canvas coordinates
            this.ctx.lineTo(...translated_vertex);
            //   line to the next vertex
        }
        //   move to each vertex in order
        this.ctx.closePath();

        if (toStroke) {
            this.ctx.strokeStyle = color;
            this.ctx.stroke();
        }

        if (toFill) {
            this.ctx.fillStyle = color;
            this.ctx.fill();
        }
    }

    drawCircle(center, radius, color="black") {
        this.ctx.fillStyle = color;

        let translated_center = this.toCanvas(center);

        this.ctx.beginPath();
        this.ctx.arc(
            ...translated_center,
            this.scaled(radius) /* radius */,
            0, 2 * Math.PI /* whole circle */
        )

        this.ctx.fill();
        //   fill the shape
    }

    drawPoint(center, color="black", radius=5) {
        let unscaled_radius = radius / this.scale;
        this.drawCircle(center, unscaled_radius, color);
    }

    inversedTranslated(vector) {
        let new_vector = vector.reflectedInX();
        //   reflecting the vector
        new_vector.add(this.translation);
        //   translating the vector to a new position
        return new_vector;
    }

    setScale(scale) {
        this.target_scale = scale;
    }

    drawLine(start, finish, color="black") {
        //   unscaled
        this.ctx.beginPath();
        this.ctx.moveTo(...this.inversedTranslated(start));
        //   starting point
        this.ctx.lineTo(...this.inversedTranslated(finish));
        //   end point
        this.ctx.closePath();
        this.ctx.strokeStyle = color;
        this.ctx.stroke();
    }

    renderTextByTopRight(point, text, color="black") {
```

```
        this.ctx.fillStyle = color;

        this.ctx.font = "18px serif";
        this.ctx.textAlign = "right";
        this.ctx.textBaseline  = "top";
        this.ctx.fillText(
            text,
            ...this.inversedTranslated(point)
        );
    }

    drawArrow(start, finish, color="black") {
        this.ctx.strokeStyle = color;

        let angle = Math.PI / 8;
        let arrow = finish.subtracted(start);
        let reversed_arrow_unit = arrow.multiplied(-1).normalized();
        //   direction of the arrow
        let left_wing = reversed_arrow_unit.rotatedBy(-angle).multiplied(15);
        let right_wing = reversed_arrow_unit.rotatedBy(angle).multiplied(15);

        this.ctx.beginPath();
        this.ctx.moveTo(...this.inversedTranslated(start));
        this.ctx.lineTo(...this.inversedTranslated(finish));
        this.ctx.stroke();
        //   first draw the base completely
        this.ctx.beginPath();
        //   start new path for the head of the arrow
        this.ctx.moveTo(...this.inversedTranslated(finish.added(left_wing)));
        this.ctx.lineTo(...this.inversedTranslated(finish));
        this.ctx.lineTo(...this.inversedTranslated(finish.added(right_wing)));
        this.ctx.stroke();
    }

    increaseTranslationBy(vector) {
        this.target_translation = this.target_translation.added(vector);
    }

    increaseScaleBy(value) {
        this.target_scale *= value;
    }

    drawParabolaBy(starting_point, control_point, end_point, color="black") {
        this.ctx.beginPath();
        this.ctx.moveTo(...this.toCanvas(starting_point));
        this.ctx.quadraticCurveTo(
            ...this.toCanvas(control_point),
            ...this.toCanvas(end_point)
        );

        this.ctx.strokeStyle = color;
        this.ctx.stroke();
    }

    adjustMapping(particle) {
        let initial_position = particle.getInitialPosition();
        let list_of_points = [initial_position];

        let x0 = initial_position.getX();
        let y0 = initial_position.getY();
        let vx = particle.getInitialVelocity().getX();
        let vy = particle.getInitialVelocity().getY();
        let a = -9.8;

        let roots = Util.solveQuadratic(a / (2*vx*vx), vy / vx, y0);
        if (!isNaN(roots[0])) {
            let x1 = roots[0] + x0;
            let x2 = roots[1] + x0;
            list_of_points.push(new Vector(x2, 0));
            list_of_points.push(new Vector(x1, 0));
        }
        else {
            // add a point symmetrically to the starting point to compensate for roots being lost
        }
```

```
        let f = (x) => {
            return y0 + vy * (x - x0) / vx + a * (x - x0) * (x - x0) / (2 * vx * vx);
        }

        let x3 = Util.solveLinear(a / (vx * vx), vy / vx) + x0;
        let y3 = f(x3)
        if (x3 != null)
            list_of_points.push(new Vector(x3, y3));
        //   highest point on trajectory

        let y2 = f(0);
        if (Math.abs(y2 - y0) <= Math.abs(y3 - y0) * 1.5)
            //   add only if it is closer to starting point than
            //   1.5 times distance by y between starting point and highest point
            list_of_points.push(new Vector(0, y2));
        //   y intersection point

        list_of_points.push(
            particle.calculatePositionAt(2)
        )
        // position after first 2 seconds of movement

        let min_x = 0, max_x = 0, min_y = 0, max_y = 0;

        for (let point of list_of_points) {
            min_x = Math.min(min_x, point.getX());
            max_x = Math.max(max_x, point.getX());
            min_y = Math.min(min_y, point.getY());
            max_y = Math.max(max_y, point.getY());
        }

        let width = max_x - min_x;
        let height = max_y - min_y;
        let rect_centre = new Vector((min_x + max_x) / 2, (min_y + max_y) / 2)

        this.target_scale = Math.min(700 / width, 350 / height);

        let screen_centre = new Vector(this.canvas.width / 2, this.canvas.height / 2);
        this.target_translation = screen_centre.added(rect_centre.multiplied(-
this.target_scale).reflectedInX());
    }
}


export {ViewSim as default};
```