Rigidbody Collision and Water Simulator(2D)

A level computer science project by Theodore Castle

Candidate Number: 1000

Centre Name: The Chase Sixth Form

Centre Number: 24245

Contents

Introduction

A breakdown of the solution:

This solution aims to be able to simulate elastic collisions of rigidbodies, and their interactions with water.

A Rigidbody is at its core a perfect representation of a object that does not deform or change shape(in contrast to soft bodies).

Water is to remove any complications or ambiguity with using other fluids with different properties.

A simulation is a simplification of real world environments to be used for various purposes.

Analysis

The target audience:

The target audience is anyone who is interested in studying physics or teaching physics. The project aims to visualise common concepts in physics, such as acceleration due to gravity, before starting the simulation the interface will allow the user to add objects into the simulated space, and be able to enter their starting properties(shape, size, weight, density, position, velocity), the simulators interface will allow the simulation to be paused at any time and step forward and backwards in time, objects can be inspected to view their internal properties, all these features combined will be able to give the user an insight into the basic principles of physics.
Stakeholders will be able to test the program and provide feedback, my stakeholders are: Kevin Tu and Jack Ben-Dyke, both who study Physics and Further Mathematics A levels, they will be able to verify the program's accuracy and provide a statement about usability.

How computational methods will be used:

Physics is very mathematical, and forces are dictated by different laws which can be translated into functions in code, **simulations are also very repetitive and need to perform many checks**, therefore this is a problem very suited for computing. Simulating objects in general space is a difficult task, an accurate simulator where all possible forces are simulated and accounted for would require a lot of computational power.
Abstraction:
I will be using abstraction to remove forces that are insignificant or largely irrelevant, while finding the balance where all essential information is retained, for instance: drag due to fluids can be ignored due to its high complexity and low impact at this scale.
Decomposition:
I will be using decomposition to break down the many aspects of physics, each principle can be written into functions and modules, before integrating all the aspects into the overall simulation.

Physics principles	Mathematic expressions or equations
Gravity	=9.8m*s^-2
Density	=Mass/Size
Buoyancy	= weight of displaced fluid
Velocity	=v+a

▲Table of principles that can each be written as a function
Algorithmic thinking:
Algorithmic thinking is essential as it allows pre planning of sections of the program. Flow charts and Pseudo code will be very useful to create rougher drafts and structural framework for completing my project. Flow charts can visually describe the sequence of the program and will be advantageous when designing the inner workings of the physics calculations;

computation is a major part of the project which emphasises this. Pseudo code allows me to plan the structure of the program. Combined with decomposition, it will allow detailed planning of functions, which can ensure a reliable foundation to build the project upon.

Previous solutions:

Various physics libraries such as:

- Matter.js (https://github.com/liabru/matter-js)
- Plank.js (https://piqnt.com/planck.js)

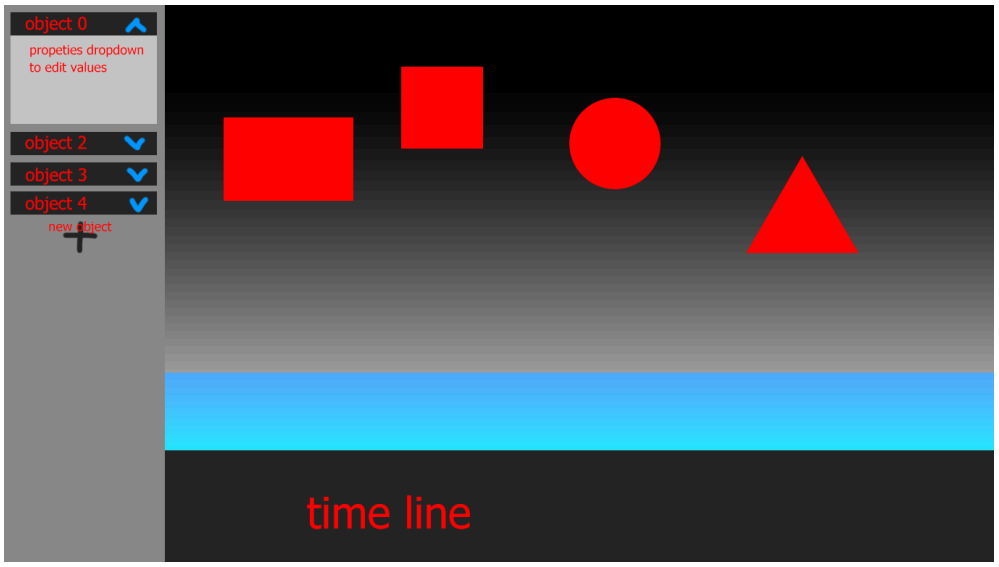
I found examples of what I would like to mimic and expand on, they are pre-existing physics engines or libraries used to only simulate physics.
I would like to mimic its inner workings for my physics simulation and add a UI element for the end user.
I would like to create a solution that is computationally efficient, and package it with an existing graphical library, to create a program with a GUI geared towards learning physics and understanding the inner workings of the program.

Essential components of the program:

A brief overview of the planned features:

1. Display a graphics in 2D
2. Have a GUI for inputs(properties, timeline)
3. Ability to add a rigidbody object through the GUI
4. An input for shape of a new rigidbody
5. An input for size and weight of a new rigidbody
6. An input for position of a new rigidbody(includes x,y and rotation)
7. An input for velocity of a new rigidbody(includes x,y and rotation)
8. Start simulation in
9. Simulate gravity, collision, buoyancy, drag
10. Allow user to pause or move the playhead at any point of simulation
11. Allow the user to inspect the properties of an object
12. Allow changing of properties(position,velocity,size,weight,etc) of any object at any point in time
13. Deletes/pauses objects that are out of view automatically to reduce computational strain
14. End/pause simulation once equilibrium is reached(check if no values are changing)

The program window:



▲Window mockup of program
(2 elements: GUI, simulation window)

The GUI (Graphical user interface):

▲Timeline representation(placeholder)
The timeline section will contain the play/pause button and speed multiplier, the timeline will also enable the user to move forwards and backwards through time

▲GUI mockup
(grey: various GUI elements, red text: title and/or the explanation of function)

<table>
<tr><td rowspan="3">Shape:</td><td colspan="4"><shape></td></tr>
<tr><td>Size:</td><td><size></td><td>Mass:</td><td><Mass></td></tr>
<tr><td>Density:</td><td><density></td><td>(*1)</td><td></td></tr>
<tr><td rowspan="2">Position:</td><td>X coordinate:</td><td>Y coordinate:</td><td>Rotation:</td><td rowspan="2"></td></tr>
<tr><td><x></td><td><y></td><td><r></td></tr>
<tr><td rowspan="4">Velocity:</td><td>X velocity:</td><td>Y velocity:</td><td>Rotation:</td><td rowspan="4"></td></tr>
<tr><td><velox></td><td><veloy></td><td><velor></td></tr>
<tr><td>Velocity</td><td>Direction</td><td rowspan="2">(*2)</td></tr>
<tr><td><scale></td><td><direction></td></tr>
</table>

▲Table represents a potential layout for the properties drop down
(Key: <> is the editable variable displayed)
annotations:
*1: **(auto changes density value when edited/is auto filled after setting mass and size;**
ex: <shape=circle>,<size=10>,<mass=10> to <density=31.4159265359>)
*2: **(auto converts values to/from <velox> and <veloy>;**
ex: <x=1>,<y=1> to/from <scale=1.41421356237>, <direction=45°>)

Other GUI elements:checkbox for turning water on/off

The Simulation Window:

▲ The objects in red can be inspected by clicking on them, enabling the user to see the internal values of the object in the GUI properties drop down
(blue: water in simulation, red: rigidbody objects in simulation)

Success criteria:

Main Criteria:

The UI is easy to navigate:

- ☐ The GUI isn't distracing from the main simulation.
- ☐ The GUI and its function is clear without any tutorial, students should be able to use the inputs just by already understanding basic physics and its mechanics.
- ☐ The representation of values is intuitive, the values in the properties panel are easily understandable and relatable to physics.
- ☐ The timeline is easy to understand and manipulate.

Time manipulation and its related functions are implemented:

- ☐ A play/pause button implemented
- ☐ A play speed multiplier is implemented
- ☐ Including negative multipliers(playing in reverse)
- ☐ Click and drag function on the playhead(similar to youtube)

The simulation is accurate:

- ☐ The simulation does not deviate from real life physics in any significant way, gravity, velocity, and other required calculations are accurate and consistent
- ☐ The water and its related physics are successfully implemented in an accurate and consistent manner.

Optional Criteria:

Implementing various display modes:

- ☐ Display modes such as force vectors can be toggled on/off

More physics variables or aspects implemented:

- ☐ Account for drag

Allow the user to change various constants:

- ☐ Change the gravitational acceleration constant
- ☐ Change the drag coefficient for fluids
- ☐ Change density of liquid

Hardware and Software requirements:

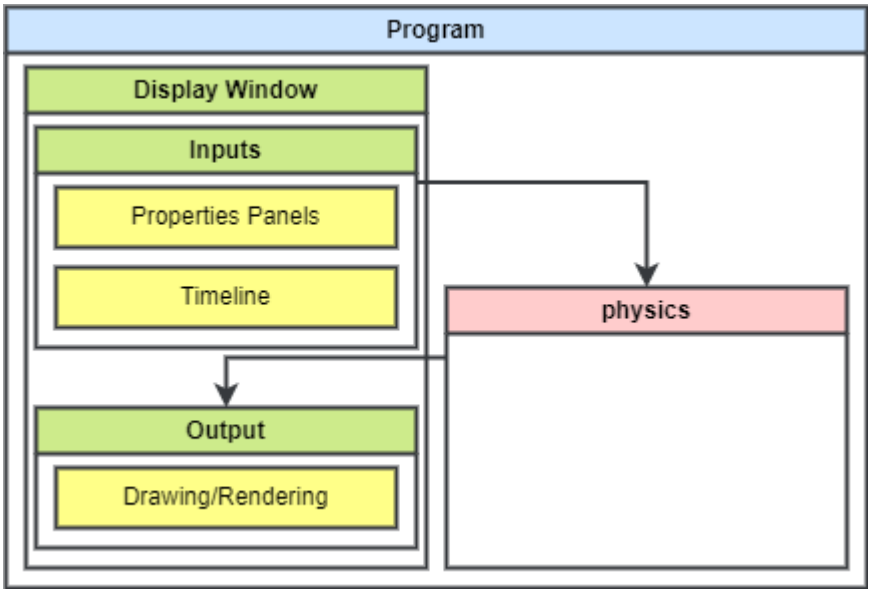
Minimum hardware requirements:

- A 64bit processor
- 2Gb of RAM
- Standard keyboard and mouse
- A qHD(960x540px) monitor capable of 60hz
- Modern web browser with HTML 5, CSS6, JavaScript 2020 (ES13) support

Design

Overview:

A breakdown for the program, each element will have its own further breakdown and flow charts.



▲a diagram of inputs, processes and outputs

Display Window:

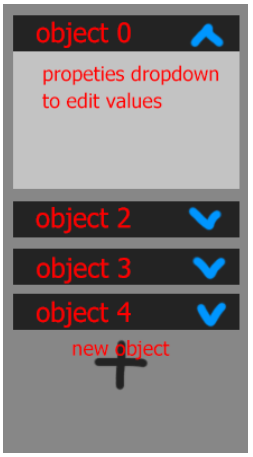
The display window contains 2 large elements: **Input** and **Output**.

Inputs:

There are two main inputs accessible to the user from the **graphical user interface(GUI)**, the **Properties panel**, and the **Timeline**.

Properties panel:

The properties panel is the main interface for numeric values, it will allow the user to easily change the properties of a rigidbody, properties that can be edited are: shape, size, velocity, and position.



Timeline:

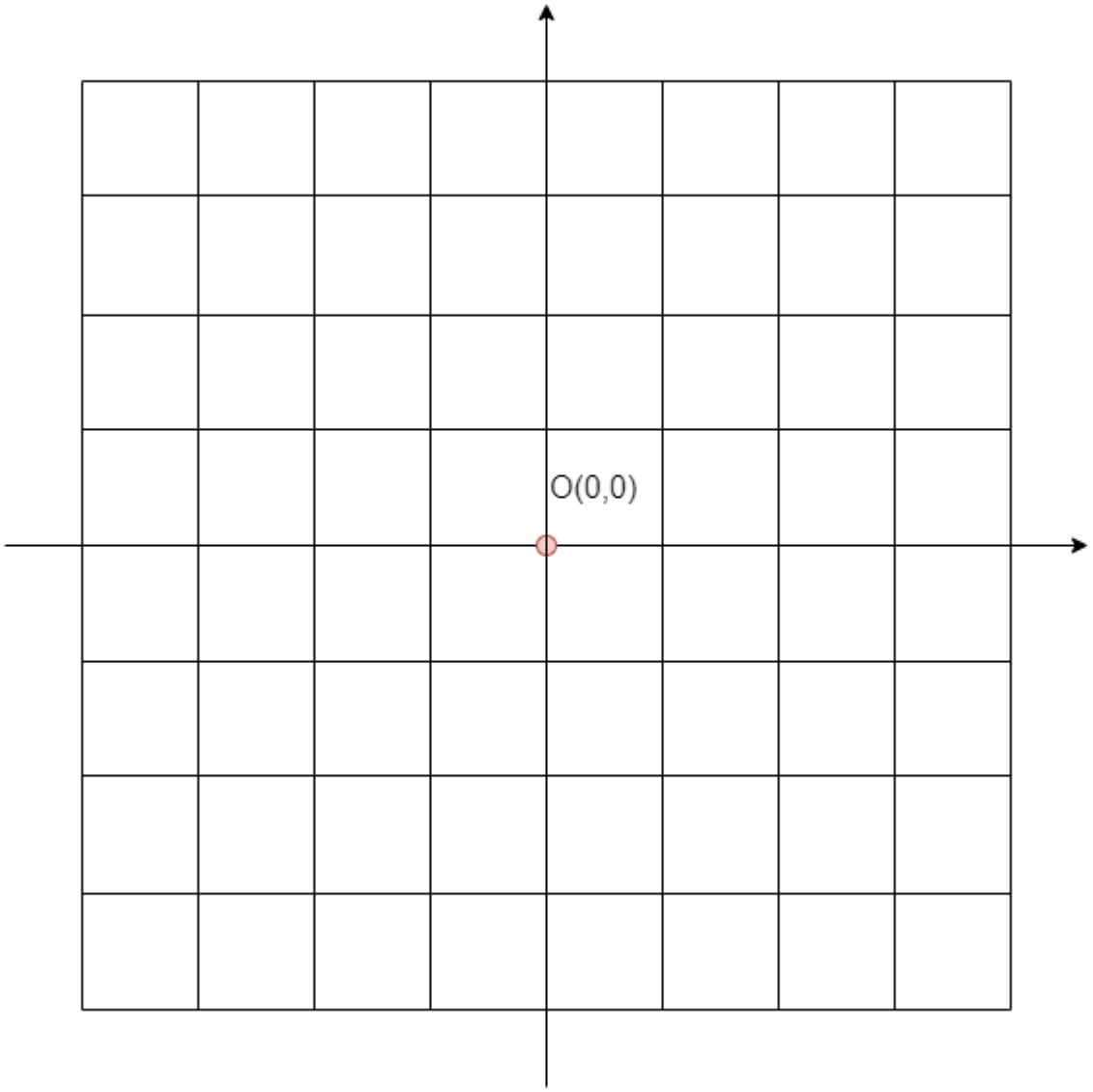
Outputs:

There is only 1 output: **the visual representation of the simulation**, the method that will be translating numeric values of the physics simulation, then interpreting it to a grid of pixels that is visually understandable to the user, the method will be referred to as the “Renderer”, because the method will essentially be Drawing/Rendering an image. The main purpose of the Renderer is to translate the scale of the simulation into a scale that can be displayed on the screen, essentially converting the metric length values to a number of pixels, the secondary purpose of the renderer is to control the rate at which the program refreshes the display, i.e: frame rate, the target frame rate is 60 frames/second, since most computer monitors are able to support this frame rate.

Physics:

Absolute 2D space:

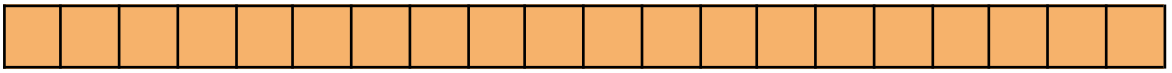
The space that the simulation will be working in, a 2D plane of space is usually represented with an X-axis and Y-axis, with an origin of (0,0), with all rigidbodies movements relative to the origin, typically in computer science the origin is pinned to the top left, with right as the positive direction of X and down as the positive direction of Y, but considering the scientific and mathematical nature of the application, the origin will be pinned to the centre with right being the positive direction of the X-axis and up as the positive direction of the Y-axis, all rigidbodies will exist relatively to the origin, allowing rigidbodies with a negative position value to be represented. An arbitrary scale will be picked to represent the distance between points, for physics purposes it should reflect the metric length measurement system,for a more precise simulation the scale should be able to represent small values.



Tick based time:

Due to time being a core input element, the simulation should mainly accept time as the controlling factor, functions and methods should be written as to include time as the primary input, with other inputs adapting accordingly, time will be implemented with a tick based system, which has been proven very effective and reliable in most circumstances, its is a system of having an internal clock to regulate events, in which 1 second should equal X ticks, where the value of X will be decided during the development stage, the larger the value of X, the more precise the simulation, each tick represents a change in time, ideally the tick rates are also independent from frame rates, which will be useful when viewing the simulation in a sped up state, at which the frame rate does not change, but the tick rate is higher than frame rate.

Tick rate:



Frame rate:

▲ a representation of frame rate and tick rate not in sync (not to scale)

Represented value	Name for variable	Datatype	*notes
Tickrate	TPS	int	*defining the duration of a tick at normal speed *controls the granularity of the simulation *the target tickrate is 100 ticks/s, a perfect tick lasts 10ms
Speed Multiplier	tickScale	float	*controls the amount of ticks/s when manipulating the speed *1 is normal speed *0 is paused simulation *<1 is slowed speed *>1 is accelerated speed *a negative multiplier will result in the simulation to reverse
Real Tickrate	realTPS	float	* =TPS*tickSpeed *is the real speed that the simulation will run at

Psudocode

func Tick(){//does operations}
int TPS = 100;//technically a constant
float tickScale = 1, realTPS = TPS * tickScale;
setInterval(Tick(),realTPS);

*note
*setInterval runs the tick operation on an interval set by realTPS

Rigidbody properties and datatypes:

Any value with an X, Y, or Rotational component will be structured as a Array datatype, with index 0,1,2 being the X,Y,R component for the value respectively

Represented value	Name for variable	Datatype	*notes
Shape	shape	int	*Each value will represent a corresponding shape *all planned shapes are equilaterals or circular
Size (area)	size	float	*normally size would be representing volume, but in 2D it would be representing area
Mass	mass	float	*in conjunction with size, we can calculate density
Position	pos	float[]	*x,y,r components
Velocity	velo	float[]	*x,y,r components
Force	force	float[]	*x,y,r components

Universal functions that apply to objects:

Physics principles that can be written as functions with inputs:

- Gravity (mass)
- Collision (mass)
- Buoyancy (mass, size)

How physics functions should be structured:
A choice between two:

1.Outputting force (in Newtons):

Outputting the resultant force of a principle will have on the rigidbody, then totaling all resultant forces from the functions.
Using gravity as an example, the acceleration due to gravity is **g(**~9.8m/s), the force that gravity adds in newtons: `F = -g * mass`

the total resultant force is all functions that return force added together, such as `totalForce = Gravity + Buoyancy + Collision`, therefore we can calculate the acceleration of the rigidbody: `acceleration = totalForce / mass`

Pseudocode

```
float[] force = {x,y,r};//x,y,r are placeholders to a real value
func gravityF(float mass){
      return mass * -9.8; //gravity acts in the negative y axis
}
func collisionF(){
      //returns force due to collision
}
func buoyancyF(){
      //returns buoyant force
}

force = {force[0] + collisionF(),force[1] + collisionF() +
gravityF() + buoyancyF, force[2] + collisionF()};
```

2.Outputting acceleration (in m/s):

Outputting the resultant acceleration of a principle will have on the rigidbody, then totaling all resultant acceleration from the functions.

Using gravity as an example, the acceleration due to gravity is g(~9.8m/s), the resultant acceleration: `A = -g` ,

or a more general rule for acceleration would be: `A = force / mass`

Pseudocode

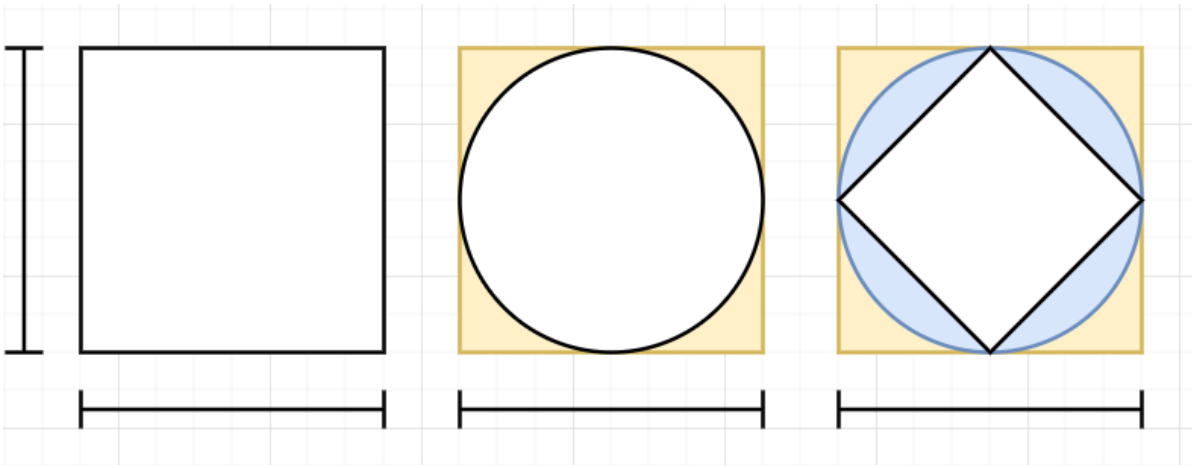
```
float[] velo = {x,y,r};//x,y,r are placeholders to a real value
func gravityA(float mass){
      return -9.8; //gravity acts in the negative y axis
}
func collisionA(){
      //returns acceleration due to collision
}
func buoyancyA(){
      //returns acceleration due to buoyant force
}

velo = {velo[0] + collisionA(),velo[1] + collisionA() + gravityA()
+ buoyancyA, velo[2] + collisionA()};
```

After considering that only the acceleration due to gravity would benefit from being written in a form that outputs acceleration, i have determined that the functions will be structured to output force, as it has an extra benefit of not needing to be calculated or updated every tick, and only needs to be changed once another external force is added, such as collision or contact with water, simplifying the number of operations, hopefully resulting in a higher efficiency, although this is subject to change during development.

Creating an efficient collision function:

Collision is very computationally intensive due to the many complex checks for if a collision has occurred, and calculating what the resultant force will be, below is my attempt to reduce the frequency of complex checks will be performed.

1.How we should think of shapes:

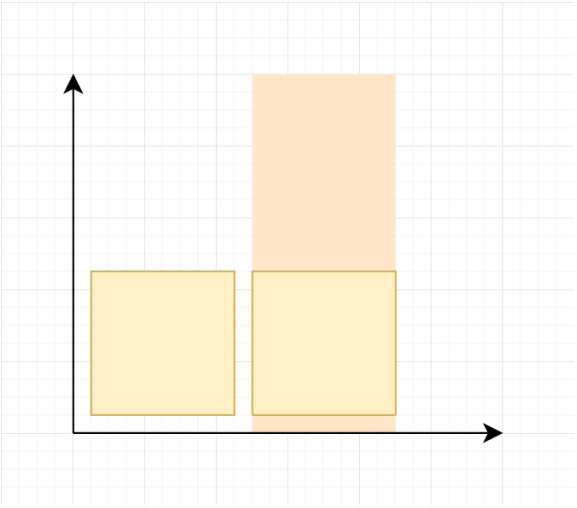
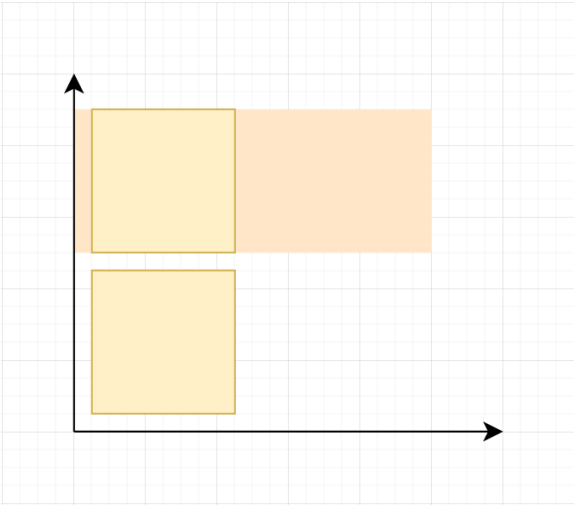


▲ a diagram to assist in explanation

We can think of shapes as containing each other, from outer to inner, in the order of computational intensity, a square (or rectangle) is the most computationally simplistic, and the final shape is the least, therefore we should aim to reduce the amount of times the final shape is accessed.

2.First check:

The outermost square (or rectangle) is called the **bounding box**, the bounding box is the smallest rectangle that the shape can fit into, in the first check we only need to check if the bounding box is overlapping any other bounding box, this is a very simple check, as **we can compare the X coordinates entirely separately from the Y coordinates**, if the shape does not overlap in 1 axis it is impossible for it to overlap at all, however, this does not mean that if 1 of the axis overlaps that a collision has happened.

Case	Diagram
The X coordinates don't overlap, here we can clearly see that despite the Y coordinates completely overlapping, the X coordinates not overlapping means we already know that the shapes haven't collided	
The Y coordinates don't overlap, here we can clearly see that despite the X coordinates completely overlapping, the Y coordinates not overlapping means we already know that the shapes haven't collided	

Represented value	Variable	Datatype
Bounding box	bBox	float[]
Bounding box of other rigidbody	bBox2	

*note
*The values stored in the layout of {posX,posY,halfLengthX, halfLengthY} *posX, posY is the X,Y coordinate of the bounding box centre *halfLengthX, halfLengthY are the half the length of their respective sides

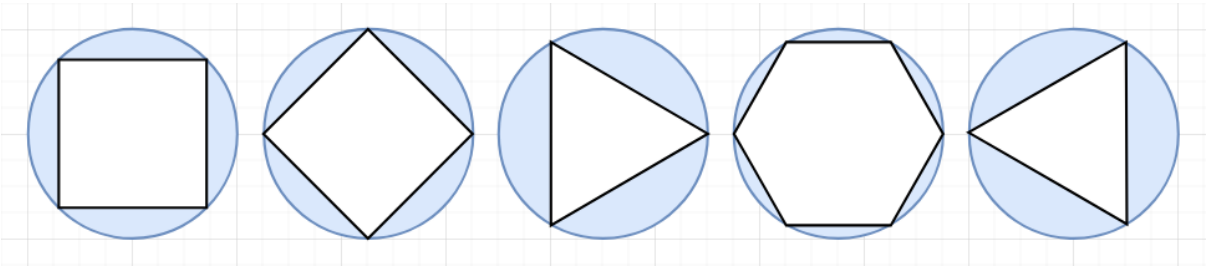
Pseudocode

`func cCheck1(int[] bBox,bBox2){`
`    collision = false`
`    if(math.abs(bBbox[0]-bBox2[0]) > bBox[2]+bBox[2]){`
`        return collision;//x axis check`
`    }else if(math.abs(bBbox[1]-bBox2[1]) > bBox[3]+bBox[3]){`
`        return collision;//y axis check`
`    }`
`    return !collision;`
`}`

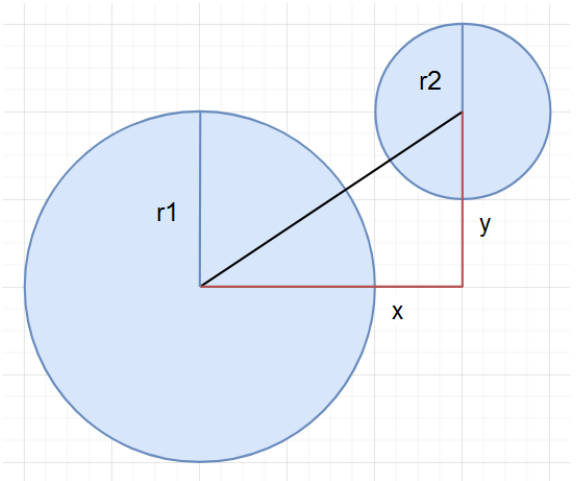
*note
*The pseudocode can be condensed with boolean algebra *math.abs() to ensure no negative lengths

3.Second check:

Since all shapes that are planned to be included are equilaterals (all sides have equal length), thay are all able to be encompassed by a circle, of which all points lie on, we can check if the circles collide, in reality it is a **simple check if the distance between the centre of each circle is smaller than their radiuses combined**, in absolute space we can calculate the distance by using Pythagoras's theorem.



▲all equilaterals points lie on a circle, no matter the rotation

Case	Diagram
$(r1+r2)^2 < x^2 + y^2$, the distance between centre points is greater than the 2 radiuses combined	

Represented value	Variable	Datatype
Bounding circle	bCir	float[]
Bounding circle of other rigidbody	bCir2	

*note
*The values stored in the layout of {posX,posY,radius} *posX, posY is the X,Y coordinate of the centre *radius is the distance from centre to point of the shape

Pseudocode

```
func cCheck2(int[] bCir,bCir2){
    collision = false
    if(math.pow(bCir[0]-bCir2[0],2)+math.pow(bCir[1]-bCir2[1]) >
math.pow(bCir[2]+bCir2[2],2)){
        return collision;
    }
    return !collision;
}
```

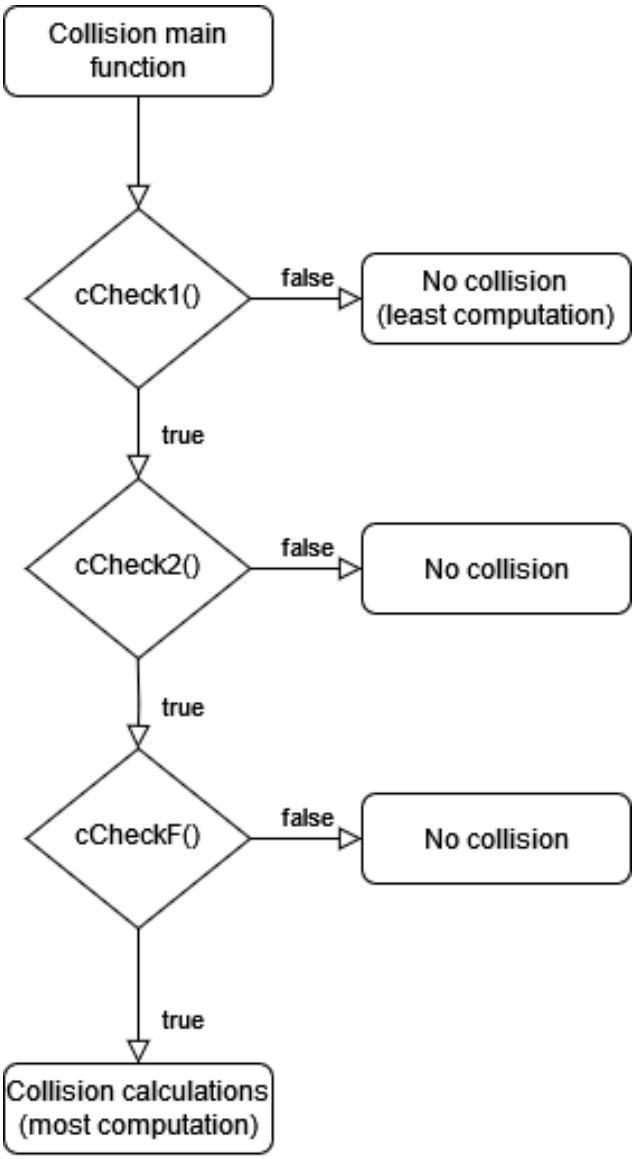
*note
*The pseudocode can be condensed with boolean algebra *math.pow(<value>,2) squares the <value>

4.Final check:

If **all checks before this can't confirm that the rigidbodies won't collide**, this function will give the final verdict if two objects actually collide, this is a very complex function, which is why the previous functions are in place to prevent this function from being used as much as possible.

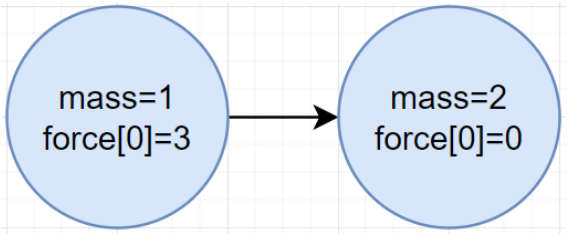
I will be researching the existing solutions to this problem in the development phase, as there is decades of research into optimising and squashing bugs caused by collision detection.

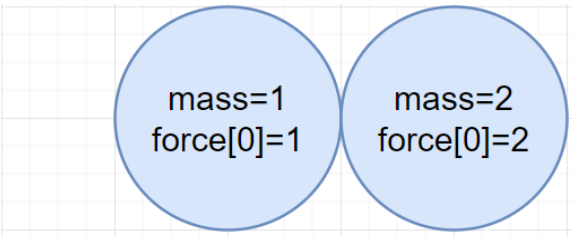
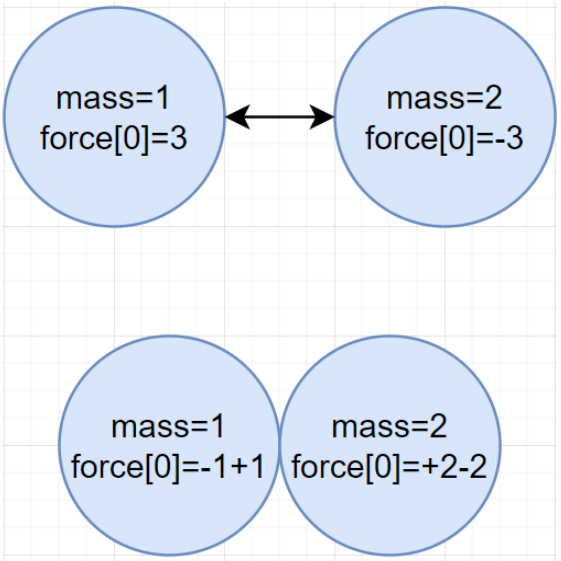
5.Overview of the processes to check for collision:



6.calculating the resultant force:

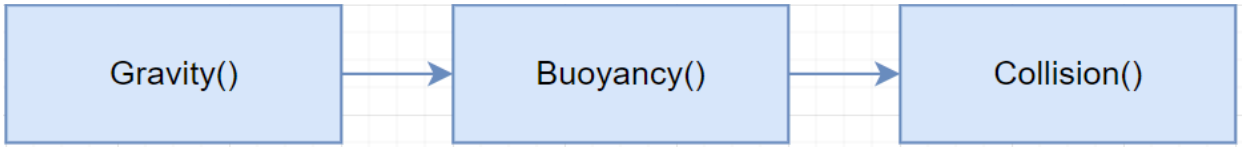
Assuming the collisions are elastic (no force is lost to the environment, https://en.wikipedia.org/wiki/Elastic_collision), we can think of the amount of force passed on as a ratio of masses, as with all forces we can also separate them into their X, Y and Rotational components, and calculate them separately.

Case	Diagram
The object on the left is moving towards the object on the right with a force of 3 (we can calculate the velocity by: `force / mass`), the **ratio of mass** between the 2 objects is **1:2**	

When the object on the left **collides** with the object on the right, it imparts a force of 3 * (⅔) = 2, and retains a force of 3 * (⅓) = 1	
If the 2 rigidbodies are moving towards each other, we simply **perform the steps above for both objects**. We can verify with the example to the left, when 2 rigidbodies with the same force collide, they come to a stop.	

Order of operations for physics:

Any simultaneous, parallel processing or multithreading is completely out of the scope in this project, therefore there must be an order of operations in the physics simulation, we must decide the importance or magnitude of effects of each physics function and/or object, the order may be subject to change during the development phase of this project.

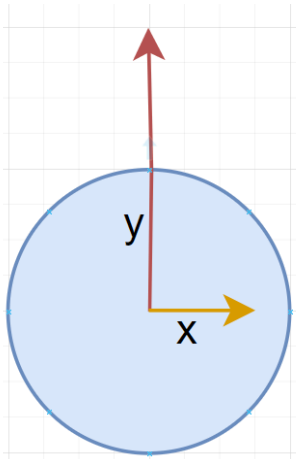


Miscellaneous:

Debugging functions:

Display force vectors:

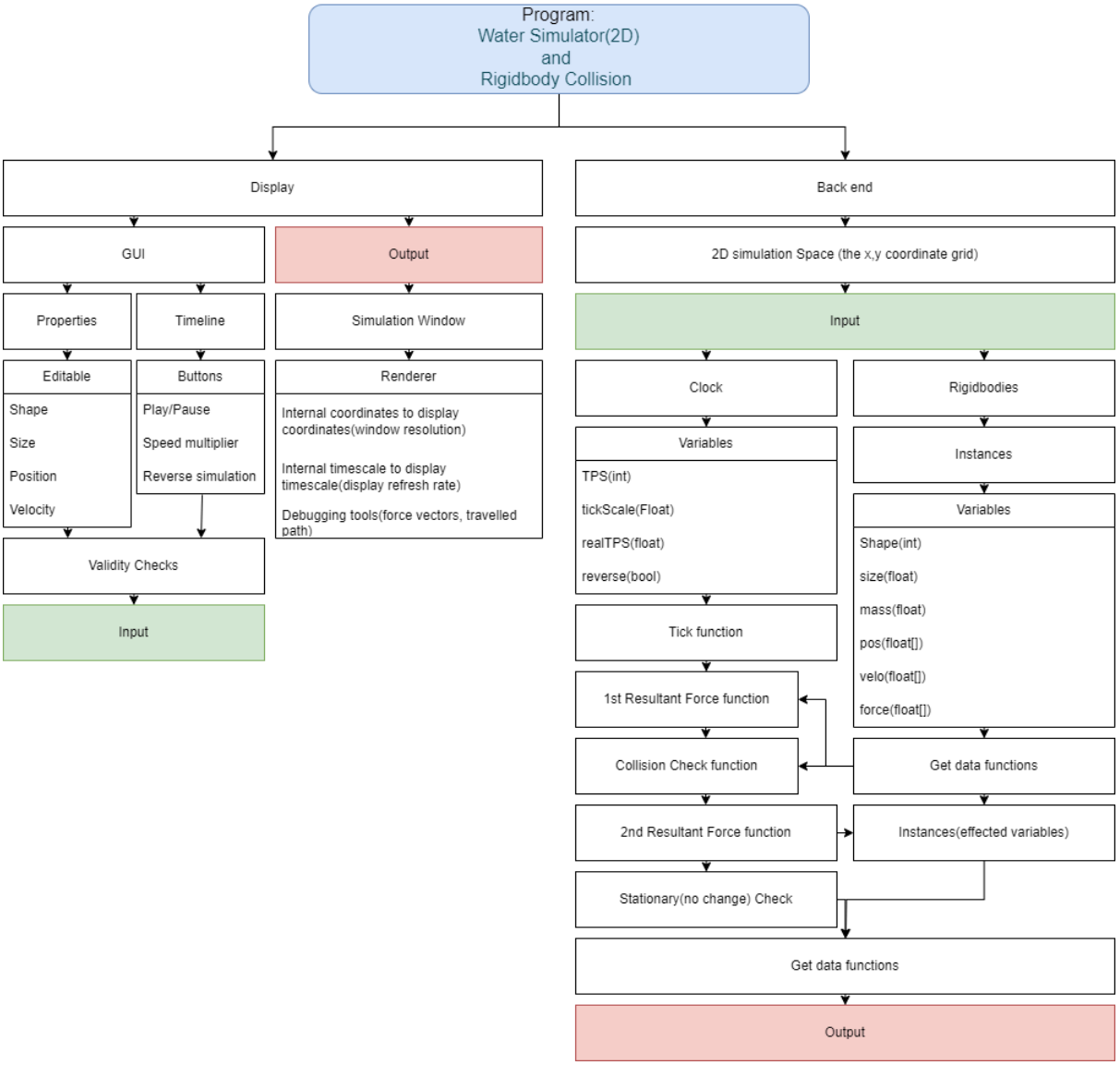
This will help me understand which forces are acting on the rigidbody, and will aid in figuring out any anomalies in movement.



Display hidden variables:

This will allow me to inspect variables that are normally hidden to the user, such as the values for the force.

Summary:


Program:
Water Simulator(2D)
and
Rigidbody Collision
Display
GUI
Output
Properties
Timeline
Simulation Window
Editable
Buttons
Renderer
Shape
Size
Position
Velocity
Play/Pause
Speed multiplier
Reverse simulation
Internal coordinates to display coordinates(window resolution)
Internal timescale to display timescale(display refresh rate)
Debugging tools(force vectors, travelled path)
Validity Checks
Input
Back end
2D simulation Space (the x,y coordinate grid)
Input
Clock
Rigidbodies
Variables
TPS(int)
tickScale(Float)
realTPS(float)
reverse(bool)
Instances
Variables
Shape(int)
size(float)
mass(float)
pos(float[])
velo(float[])
force(float[])
Tick function
1st Resultant Force function
Collision Check function
Get data functions
2nd Resultant Force function
Instances(effected variables)
Stationary(no change) Check
Get data functions
Output

Development

Pre-Development phase:

Tools:

I will be using visual studio code (VScode) by Microsoft to develop this solution, VScode is a lightweight IDE commonly used for web development, its popularity will increase the chance of an existing solution or guidance if i encounter any problems.

Languages:

I will be using:

- HTML
- CSS
- JavaScript (JS)

Justifications and considerations:

A web app can be packaged with Electron.js, which is a framework that allows a web app to packaged to resemble a native application, although this could simplify the final solution for the user, i am unfamiliar with it, and it doesn't seem to add any other benefits over a normally structured web application hosted on a local server, which is what i will be sticking with. Weighing the pros and cons of developing a native application and a locally hosted web application, the pros of the latter are clear:

- Supports many platforms and operating systems
- Is a technology that i'm already familiar with

Since I am running this solution in a web browser, the languages chosen are the most optimal and commonly used. For the **front end**, **HTML and CSS** are well established, and very flexible, allowing me to create a user friendly interface. For the **back(logic) end**, **JS** is also well established, and integrates well with HTML and CSS, it possesses all the necessary paradigms that any other capable programming language such as Java, C++, or Python would have.

A correction on datatypes:

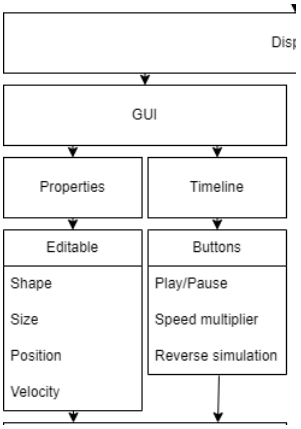
At the “Rigidbody properties and datatypes” section in the design phase i specified all the datatypes for various variables, forgetting that **javascript is an untyped language,** so instead i will be using a **naming convention of using prefixes on variable names**, if it’s a numeric value, the prefix of `int` will be added to the name, if it’s a boolean value, the prefix of `bool` will be added to the name, if its a string value, the prefix `str` will be added, all suffixes will have the first letter capitalised to form **camelCase**, ex: `var strShape = “circle”` , this is needed to maintain clarity throughout the code, so the correct type of data is parsed at all times.

Phase 1:

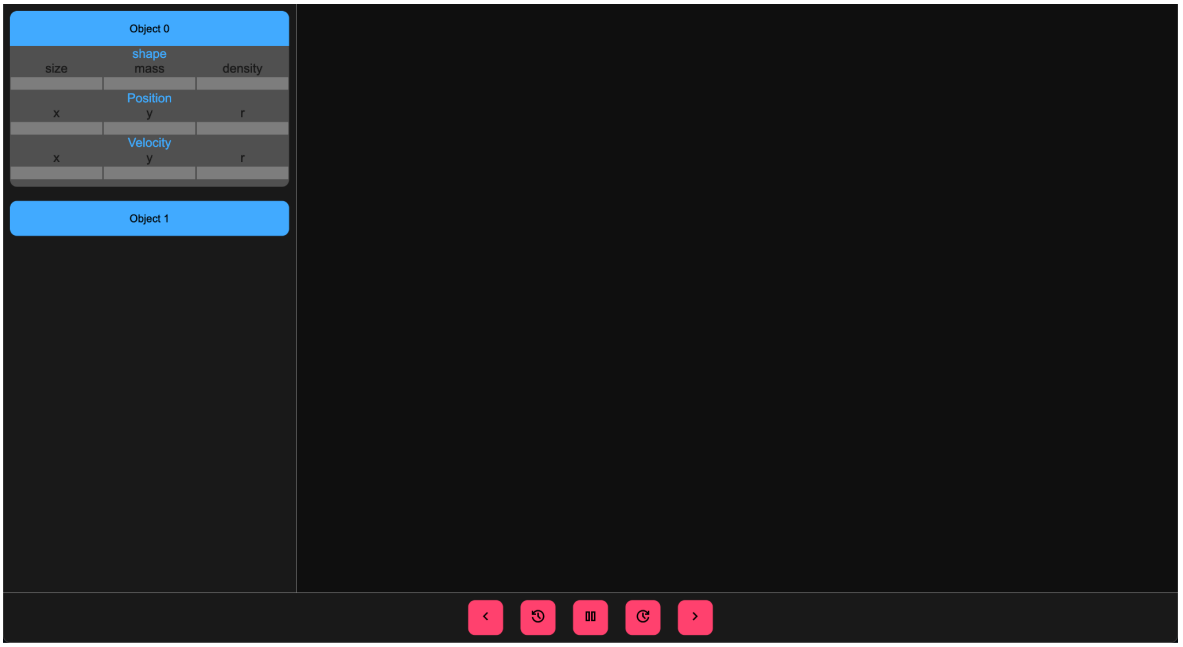
I have decided to start with the GUI, because for the most part the GUI is a static element which we can easily layout at the beginning of the development cycle

The GUI:

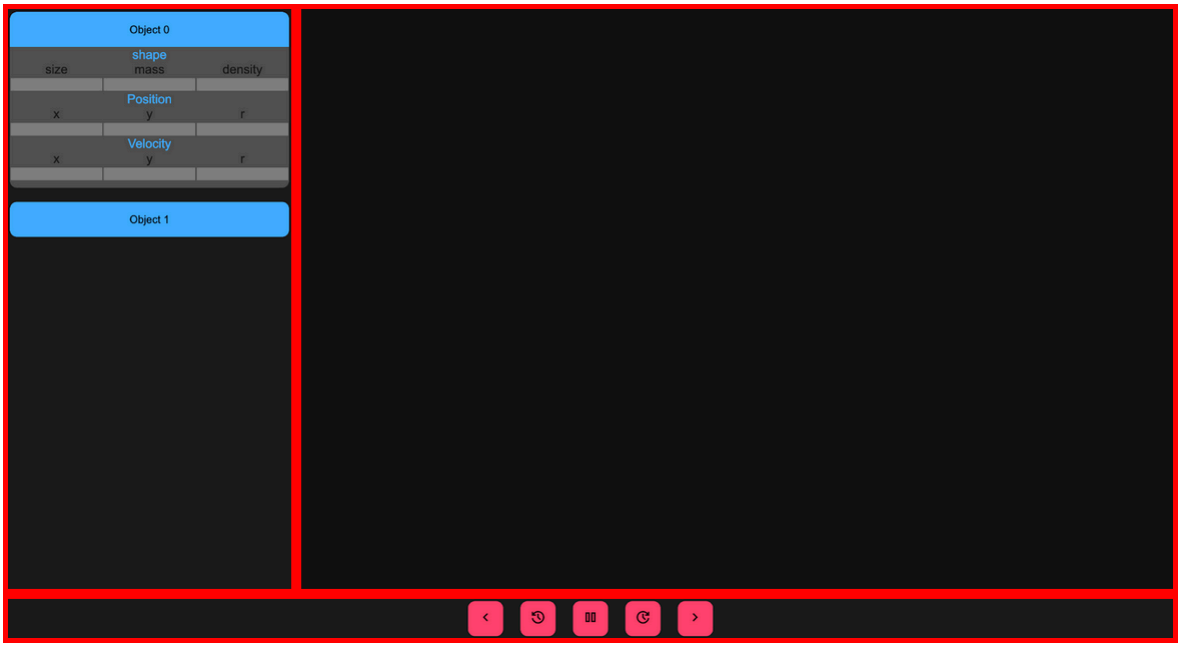
The goal with this phase is to establish the overall layout and design of the user interface, ensuring that the window is able to maintain its layout on various window sizes to maximise compatibility with devices, maintaining the usability of the program.



1st nonfunctioning HTML and CSS mockup:



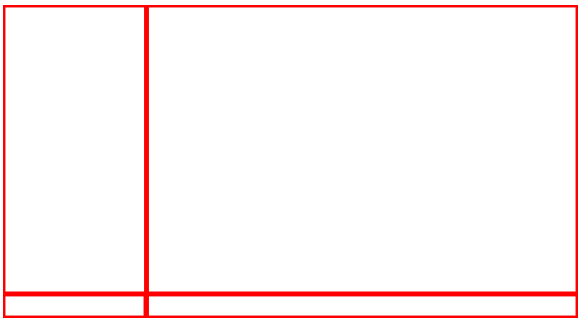
Currently all inputs and dropdown menus are nonfunctional, but the windows aspect ratio and relative placement are all complete



The layout can be first divided into 3 sections, these 3 sections maintain a constant aspect ratio, it is important to keep the simulation space in a constant aspect ratio to prevent stretching and distortion when converting the internal simulation to the display, therefore using the `aspect-ratio` property in CSS was essential, along with other properties to set up the overall windows consistency, after ensuring the windows aspect ratio is 16:9, i use the `display` property with the value `grid` ,and use the following lines to define a evenly split grid, this works because the `fr` keyword divides the grid based on a ratio of available space:

```
grid-template-columns: 1fr 3fr;
grid-template-rows: 8fr 1fr;
```

Resulting in the grid and ratios below



tall row: short row is 8:1
narrow column: wide column is 1:3 (=4:12)

HTML code snippet

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>NEA project</title>
    <link rel="stylesheet" href="style.css">


</head>
<body>

    <div class="container">

        <div id="propperties">…
        </div>

        <div id="simulation">…
        </div>

        <div id="timeline">…
        </div>
    </div>

    <script type="module" src="main.js"></script>
</body>

</html>
```

CSS code snippet

```
*{
    margin:0px;
    padding:0px;
    box-sizing: border-box;
    font-family: arial;
}
.container{
    padding: 0px;
    max-width: 100%;
    max-height: 100%;
    aspect-ratio: 16/9;
    /* min-width: 1280px; */
    /* fallback resolution */
    min-width: 960px;

    display: grid;
    grid-template-columns: 1fr 3fr;
    grid-template-rows: 8fr 1fr;
}
```

Distributing the space

The following sections of CSS define the parts of a grid the div fills, mainly the: `grid-column` and `grid-row` properties, using a slash we can also define a dive to fill a range of columns or rows

*I still do not know why in `#timeline` the property `grid-column:1/3` fills column 1 and 2 whereas `grid-column:1/2` does not, despite documentation suggesting that.

Documentation:
https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Grids#line-based_placement

```
#propperties{
   grid-column: 1;
   grid-row: 1;
   border-right:1px solid grey;
   background-color: #1b1b1b;

   display: flex;
   flex-direction: column;
   justify-content: flex-start;
}
```

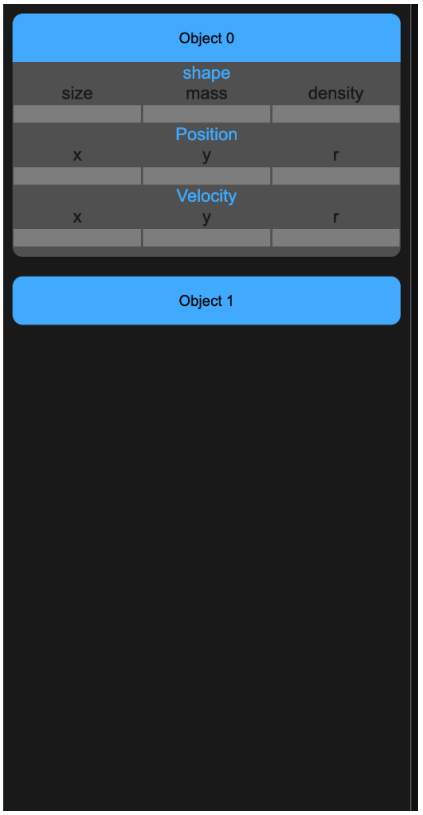
```
#simulation{
   grid-column: 2;
   grid-row: 1;
   background-color: greenyellow;
}
```

```
#timeline{
   grid-column: 1/3;
   grid-row: 2;
   border-top: 1px solid gray;
   background-color: #1b1b1b;
}
```

Properties panel:

As explained before, the properties panel is where values will be entered to control the rigidbodies properties, there is an expanded dropdown above, and a collapsed dropdown below, allowing space for multiple rigidbodies even in small window sizes, which can be common in older teaching equipment or computer hardware.

There are some missing inputs for simplicity this early in the stage, but its modular design will make it easy to add new inputs later on in the development process, with the "`vcat`" class being flexibly expandable as it uses the "`flexbox`" display type, and various classes such as "`hthree`" or "`htwo`" to distribute the input fields horizontally.



HTML code snippet

```
<div id="propperties">


            <div class="expandable">
                <button class="expanded">Object 0</button>
                <div class="dd">
                    <div class="vcat">
                        <div class="category">shape</div>
                        <div class="hthree">
                            <div>size<input></div>
                            <div>mass<input></div>
                            <div>density<input></div>
                        </div>
                    </div>
                    <div class="vcat">
                        <div class="category">Position</div>
                        <div class="hthree">
                            <div>x<input></div>
                            <div>y<input></div>
                            <div>r<input></div>
                        </div>
                    </div>
```

```
                    <div class="vcat">
                        <div class="category">Velocity</div>
                        <div class="hthree">
                            <div>x<input> </div>
                            <div>y<input></div>
                            <div>r<input></div>
                        </div>
                    </div>

                </div>

            </div>

            <div class="expandable">
                <button class="unexpanded">Object 1</button>
            </div>

        </div>
```

CSS code snippet

```
#propperties{
   grid-column: 1;
   grid-row: 1;
   border-right:1px solid grey;
   background-color: #1b1b1b;

   display: flex;
   flex-direction: column;
   justify-content: flex-start;
}
.expandable{
   padding: 10px;
}
.unexpanded{
   border: none;
   width: 100%;
   height: 50px;
   border-radius: 10px;
   background-color: #42adff;
}
.expanded {
   border: none;
   width: 100%;
   height: 50px;
   border-radius: 10px 10px 0 0;
```

```
    background-color: #42adff;
}
.dd {
    background-color: #505050;
    border-radius: 0 0 10px 10px;
    padding-bottom: 10px;
    /* display: grid;
    grid-template-columns: 5fr;
    grid-template-rows: 9fr; */
}
.category{
    color: #42adff;
}
.vcat {
    display: flex;
    flex-direction: column;
}
.htwo{
    display: grid;
    grid-template-columns: repeat(2, 1fr);
}
.hthree{
    display: grid;
    grid-template-columns: repeat(3,1fr);
}
.vthree{
    display: grid;
    grid-template-rows: repeat(3,1fr);
}
input{
    width: 100%;
    border: 1px solid #505050;
    background-color: gray;
}
```

Simulation area:

This is mostly just set up for later use, using <canvas> tag as the simulations rendering space.

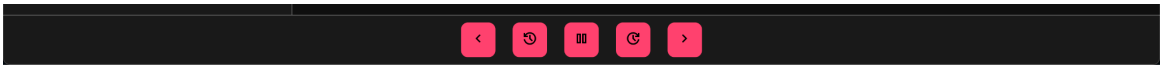
HTML code snippet

```
        <div id="simulation">
            <canvas id="canvas1"></canvas>
        </div>
```

CSS code snippet

```
#simulation{
    grid-column: 2;
    grid-row: 1;
    background-color: greenyellow;
}
#canvas1 {
    background-color: #0f0f0f;
    position: inherit;
    height: 100%;
    width: 100%;
}
```

Timeline:



As explained before, this is the section that will be controlling the playback of the simulation, currently from left to right the controls are:

- Step backwards
- Play in reverse
- Pause
- Play
- Step forwards

The controls can also be expanded with a speed multiplier
*the icon set used is “Google material symbols”

source:
https://fonts.google.com/icons?selected=Material+Symbols+Outlined:history:FILL@0;wght@400;GRAD@0;opsz@24

HTML code snippet

```
<div id="timeline">
            <div id="timecontrol">
                <button class="pb"><img
src="/Assets/chevron_left_FILL0_wght400_GRAD0_opsz24.svg"></button>
                <button class="pb"><img
src="/Assets/history_FILL0_wght400_GRAD0_opsz24.svg"></button>
                <button class="pb"><img
src="/Assets/pause_FILL0_wght400_GRAD0_opsz24.svg"></button>
                <button class="pb"><img
src="/Assets/update_FILL0_wght400_GRAD0_opsz24.svg"></button>
                <button class="pb"><img
src="/Assets/chevron_right_FILL0_wght400_GRAD0_opsz24.svg"></button>
            </div>
        </div>
```

CSS code snippet

```
#timeline{
    grid-column: 1/3;
    grid-row: 2;
    border-top: 1px solid gray;
    background-color: #1b1b1b;
}
#timecontrol{
    padding: 10px;
    display: grid;
    grid-template-columns: repeat(5,50px);
    gap: 25px;
    justify-content: center;
}
.pb{
    background-color: #ff426e;
    height: 50px;
    width: 50px;
    border: none;
    /* padding: 10px; */
    border-radius: 10px;
}
```

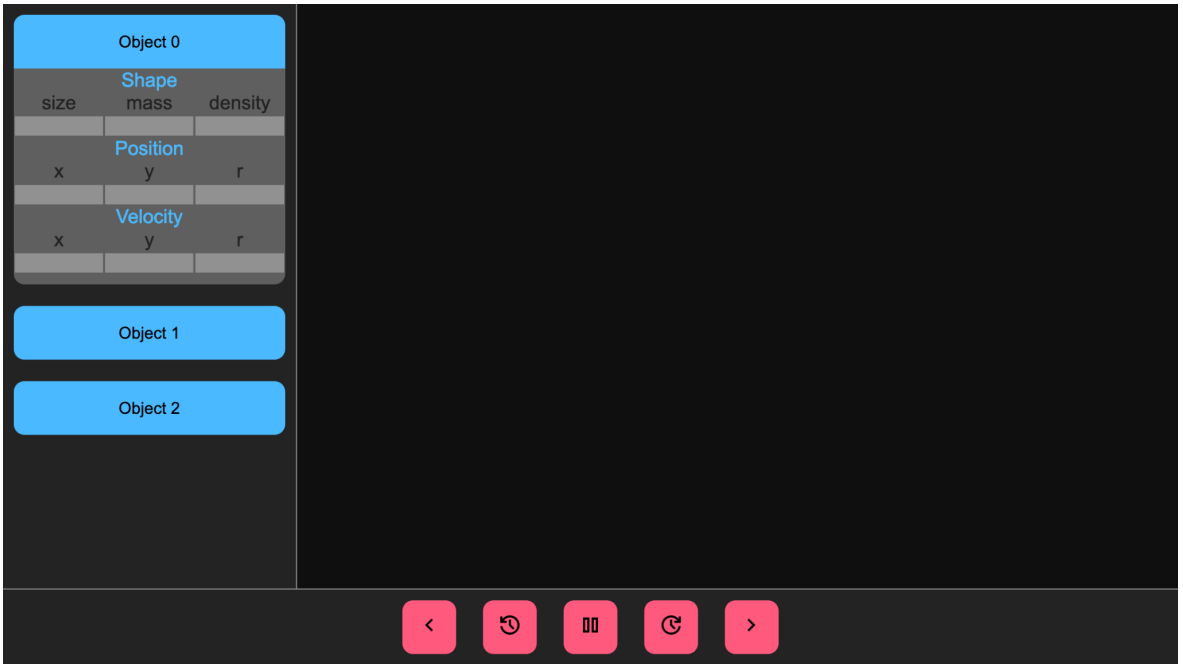
Phase 2:

Continuing work on the GUI, implementing the more dynamic sections of CSS, and incorporating JS modules.

The GUI:

Adding a **colour change on click** to indicate to the user their action has been received by the program, implementing code that enables the **dropdowns to expand or collapse** when clicked, along with automatically only leaving 1 dropdown is expanded at a time.

Colour changes on click:



The buttons now change to a deeper colour when clicked, using the `:active` pseudoclass in CSS
*was unable to capture it in a screenshot due to the nature of having to be clicked

CSS snippet for properties buttons

```
.unexpanded{
   border: none;
   width: 100%;
   height: 50px;
   border-radius: 10px;
   background-color: #42adff;

}
.unexpanded:active{
   border: none;
   width: 100%;
   height: 50px;
```

```
    border-radius: 10px;
    background-color: #507fc5;

}
.expanded {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px 10px 0 0;
    background-color: #42adff;

    display: none;
}
.expanded:active {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px 10px 0 0;
    background-color: #507fc5;
}
```

CSS snippet for playback buttons

```
.pb{
    background-color: #ff426e;
    height: 50px;
    width: 50px;
    border: none;
    /* padding: 10px; */
    border-radius: 10px;
}
.pb:active{
    background-color: #963149;
    height: 50px;
    width: 50px;
    border: none;
    /* padding: 10px; */
    border-radius: 10px;
}
```

Brief explanation of JS modules:

Modules are basically ways to split up one large JS file into multiple smaller JS module files with a main module linking all the modules together. This modular approach is efficient and allows us to **organise the code into separate files in a modules folder**, each module can

be **individually tested or changed** if needed, with the added benefit of making the main.js file a lot **easier to read**. JS modules can have the file name suffix of .mjs or the normal .js .

Example	
main.js	add.mjs

```
import add from "./modules/add.mjs";
console.log(add(1,10));
```

```
function add(a,b){
   return a + b;
}
export default add;
```

The HTML file is then able to load the code

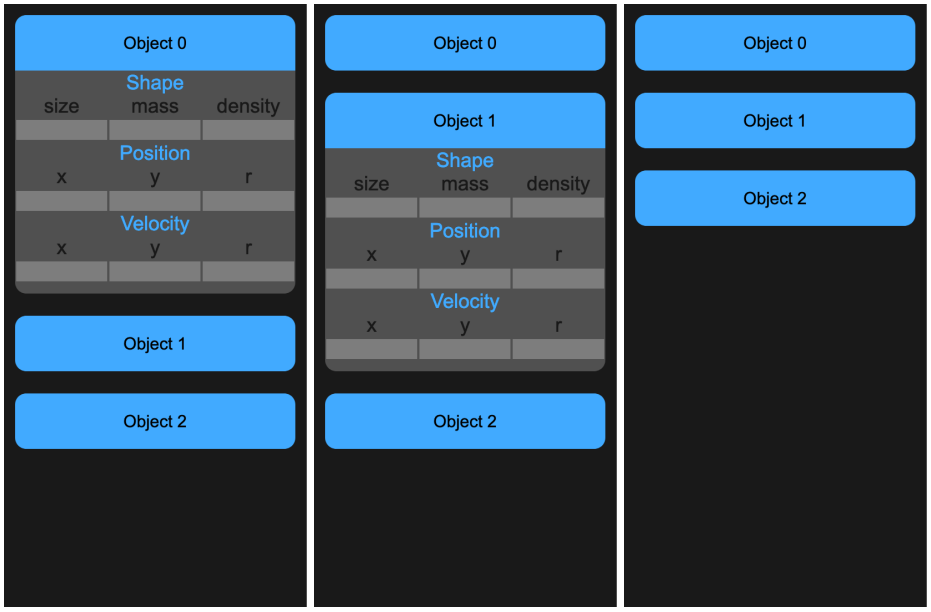
```
<script type="module" src="main.js"></script>
```

Documentation:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Utilising a module to add function to the GUI:

These functions require logical constructs like selection and interaction, therefore JS is required as neither HTML or CSS have general logic functions.

Implementing dropdowns function:



Various states of the dropdowns being expanded and collapsed, it is impossible to have 2 expanded dropdowns, the use of modules below is a perfect example of maintaining readable code using such a feature.

HTML snippet

```
<div class="expandable">
                <button  class="unexpanded">Object 0</button>
                <button class="expanded">Object 0</button>
```

```
            <div class="dd">

                <div class="vcat">
                    <div class="category">Shape</div>
                    <div class="hthree">
                        <div>size<input></div>
                        <div>mass<input></div>
                        <div>density<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Position</div>
                    <div class="hthree">
                        <div>x<input></div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Velocity</div>
                    <div class="hthree">
                        <div>x<input> </div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                    <!-- <div class="htwo">
                        <div>Magitude<input></div>
                        <div>Direction<input></div>
                    </div> -->
                </div>

            </div>
        </div>
```

```
<script type="module" src="main.js"></script>
```

JS snippet (from propertiesGUI.js)

```
function expandable(intIndex,boolDoExpand) {
   var dd = document.getElementsByClassName("dd");
   var expanded = document.getElementsByClassName("expanded");
   var unexpanded = document.getElementsByClassName("unexpanded");
```

```
    //collapses everything
    for (let i = 0; i < dd.length;i++){
        expanded[i].style.display = "none";
        unexpanded[i].style.display = "block";
        dd[i].style.display = "none";
    }

    //clicked dd expands if needed
    if(boolDoExpand){
        expanded[intIndex].style.display = "block";
        unexpanded[intIndex].style.display = "none";
        dd[intIndex].style.display = "block";
    }
}

function propGUIinitiate(){
    //opens first dd on load
    expandable(0,true);

    const expanded = document.getElementsByClassName("expanded");
    const unexpanded = document.getElementsByClassName("unexpanded");

    for (let i = 0; i < expanded.length; i++) {
        expanded[i].addEventListener("click", function () { expandable(i, false)
});
        unexpanded[i].addEventListener("click", function () { expandable(i, true)
});
    }
    console.log("initiated");
}

export default propGUIinitiate;
```

Function	Parameters	Explanation
`expandable`	`intIndex` `boolDoExpand`	1. get the 3 types of elements in HTML (`dd`, `expanded`, `unexpanded`), as there is more than 1 item with the classname, it is stored in an array, when the `<element>.style.display` is set to “`block`” it is displayed, set to “`nothing`” it is hidden 2. collapse all the dropdowns, as no matter what condition the drop downs are in, only a maximum of 1 can be opened

		3. if `boolDoExpand === true`, expand the dropdown of `intIndex`
`propGUIinitiate`		1. `expandable(0,true)` opens the first dropdown on load 2. Get the 2 button elements by their class names (`expanded`, `unexpanded`), it is stored in an array 3. Use a for loop to add an event listener to every item in the arrays, an event listener basically catches the users input on a HTML element, Documentation: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener 4. The function performed is the previous function (`expandable`), the event listener passes in the index for the element that is clicked, the listener for `expanded` passes in `false`, while the listener for `unexpanded` passes in `true`

JS snippet (from main.js)

```
import propGUIinitiate from "./modules/propertiesGUI.js";

propGUIinitiate();
```

We can keep the main module very tidy as seen above, and the `propGUIinitiate` function is called once in `main.js`.

Implementing more GUI functions:

Added “remove rigidbody” button, and “new rigidbody” button, so the user may add and remove rigidbodies, **up to 4 ridgidbodies can be added**, then the “new rigidbody” button automatically hides itself, if there are less than 4 rigidbodies, the “new rigidbody” button automatically unhides itself.

when adding rigidbodies, it will always add the first available number, i.e. if 0 and 2 are present, 1 will be added when “new rigidbody” is clicked, the 0th rigidbody cannot be removed.

HTML & CSS snippet

As I have already laid out the most important features within, going forward I will not be showing HTML and CSS changes, unless they massively impact function.

JS snippet (from main.js)

```
import propGUIinitiate from "./modules/propertiesGUI.js";
import newRigidbodyGUI from "./modules/newrigidbodyGUI.js";
import removeRigidbodyGUI from "./modules/removerigidbodyGUI.js";

propGUIinitiate();
newRigidbodyGUI();
removeRigidbodyGUI();
```

JS snippet (from newrigidbodyGUI.js)

```
function newRigidbodyGUI(){
   const newRB = document.getElementById("newrigidbody");
```

```
    newRB.addEventListener("click", function () { onclick() });
}

function onclick(){
    var expandable = document.getElementsByClassName("expandable");
    var newRBc = document.getElementById("newRBc");
    var dispRB = false;

    //show 1st avalable rigidbody
    for (let i=0;i<4;i++){
        if (expandable[i].style.display == "none"){
            expandable[i].style.display = "block" ;
            break;
        }
    }

    //are max rigid bodies shown
    for(let i=0;i<4;i++){
        if(expandable[i].style.display == "none") {
            dispRB = true;
        }
    }
    //hide button if max rigidbodies shown
    if(!dispRB){
        newRBc.style.display = "none"
    }
}

export default newRigidbodyGUI;
```

Function	Explanation
`onclick`	1. Get the 2 HTML elements (`expandable`, `newRBc`), getting by ID only returns 1 value, so it isn't an array like getting by class previously, `expandable` contains all the elements of the dropdown, so hiding it hides everything within. 2. Variable `dispRB = false` 3. If `expandable[i].style.display == "none"` then the value is changed to "`block`" we only want 1 rigidbody to be added so we `break` out the loop 4. If all items in `expandable[].style.display == "none"` then `dispRB` remains `false` 5. `if(!dispRB)`, which is equivalent to `if(dispRB == false)` then the new rigidbody button is hidden

newRigidbodyGUI	Initialises the GUI in `main.js`

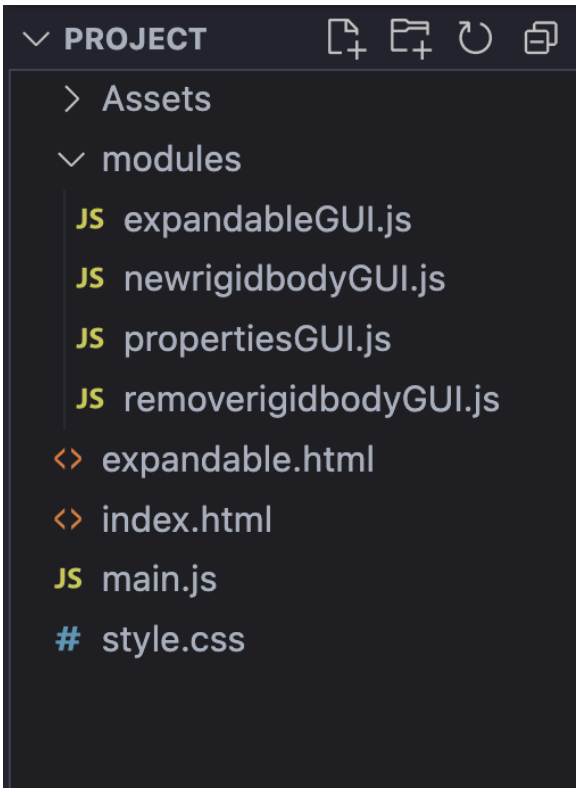
JS snippet (from removerigidbodyGUI.js)

```
function removeRigidbodyGUI() {
    const removeRB = document.getElementsByClassName("remove");

    //disallow deleting first object for now
    removeRB[0].style.display = "none";

    //adding event listeners for all remove rigidbody buttons
    for(let i=0;i<removeRB.length;i++){
        removeRB[i].addEventListener("click", function () { onclick(i) });
    }
}

function onclick(intIndex){
    var expandable = document.getElementsByClassName("expandable");
    var newRBc = document.getElementById("newRBc");

    expandable[intIndex].style.display = "none";
    newRBc.style.display = "block";
}

export default removeRigidbodyGUI;
```

Testing and demonstration of current iteration(1):

The current interaction has laid the foundation for the GUI, implementing buttons, information display, and compatibility with a range screen window resolutions.

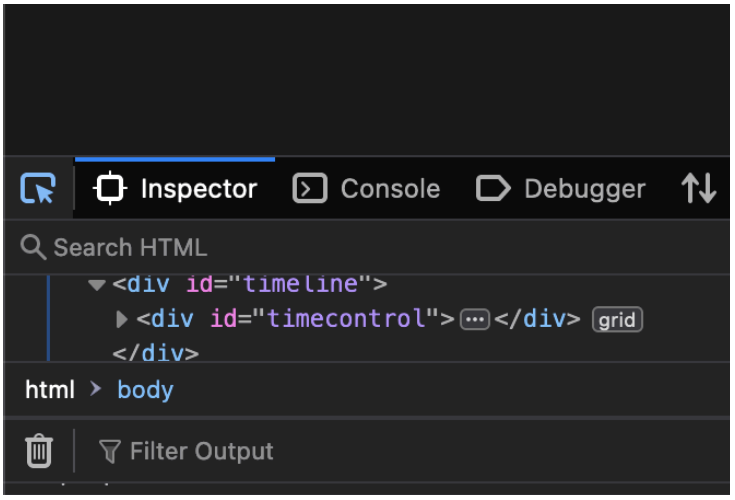
A recap and overview of the current file structure:

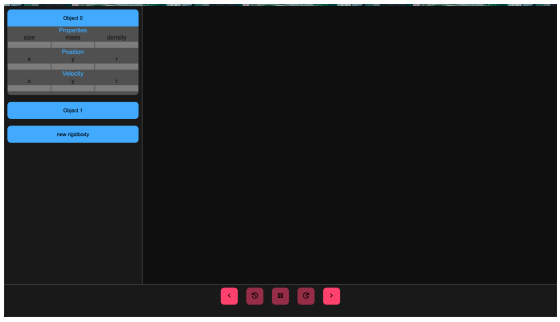
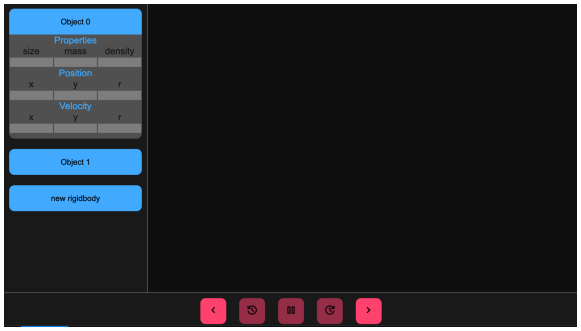
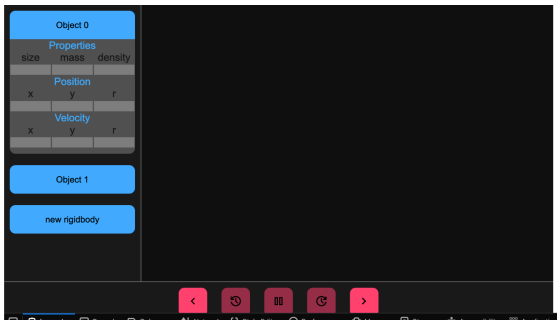
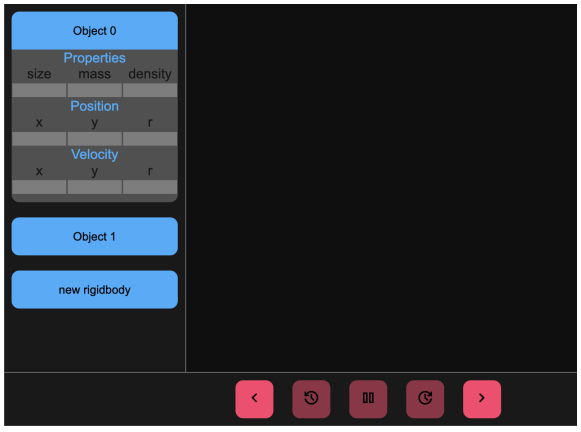
Current file structure:	
By using modules I can **ensure that the project is well organised**, and can **conduct isolated tests**, such as the `expandableGUI.js` module (along with `expandable.html`), which is a failed attempt at implementing the functions that the other files currently perform, by using node structures, as it's a failed attempt i can simply **not import it** into `main.js`, but still **keep record of the attempt** for future use. Documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules Documentation: https://developer.mozilla.org/en-US/docs/Web/API/Document/importNode	

Testing:

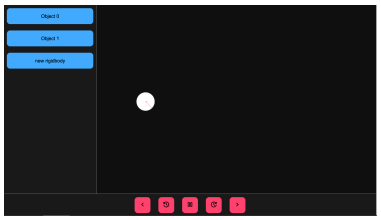
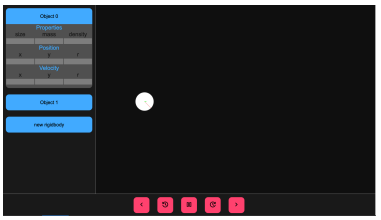
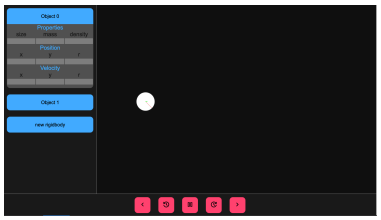
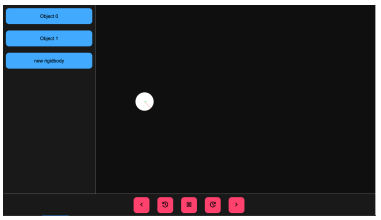
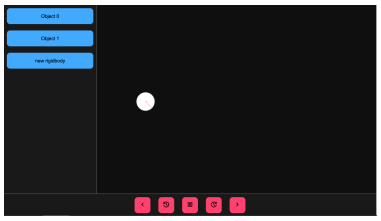
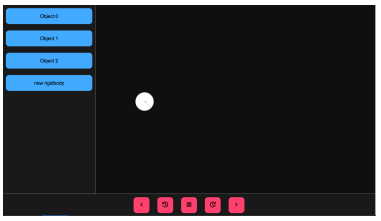
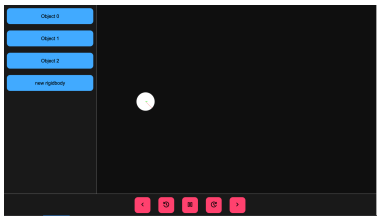
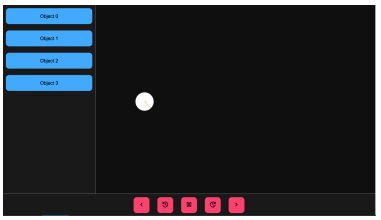
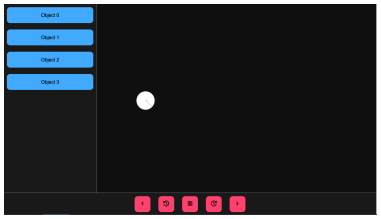
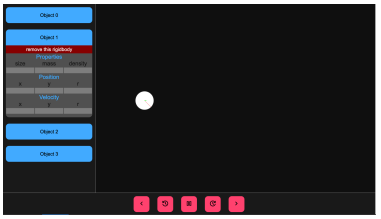
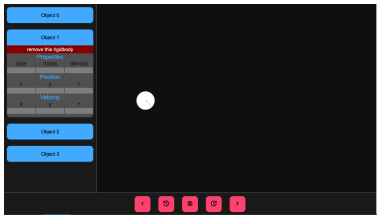
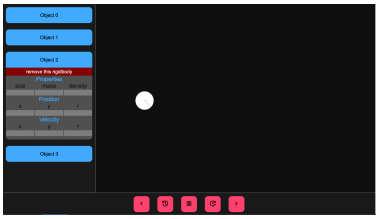
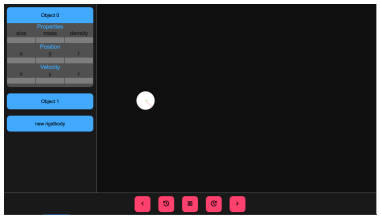
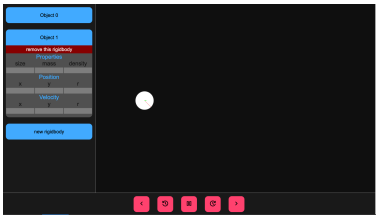
Testing an assortment of resolutions:

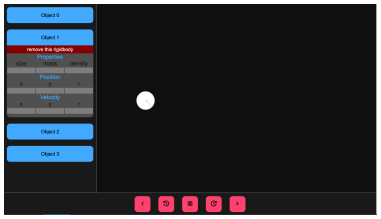
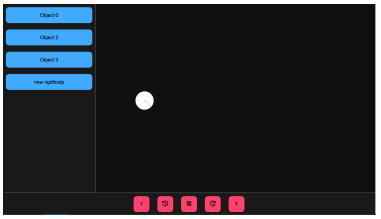
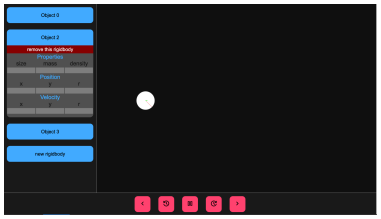
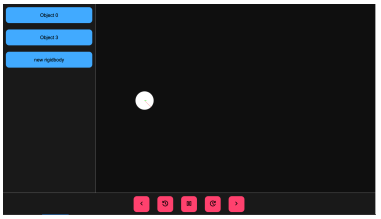
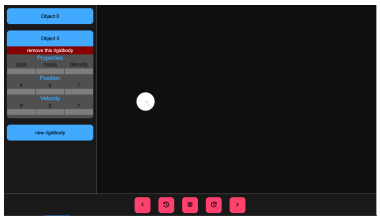
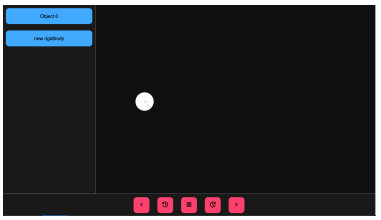
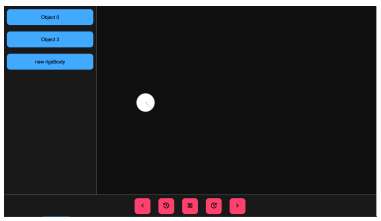
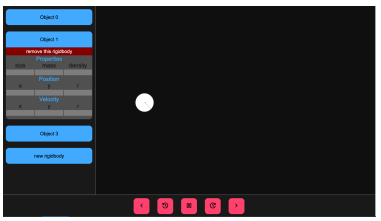
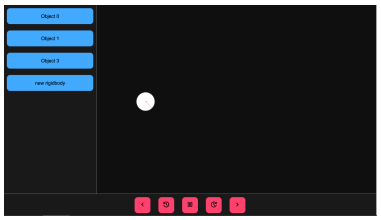
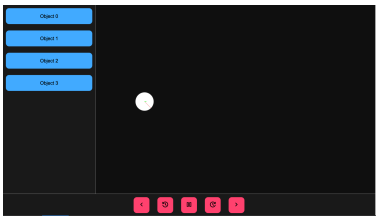
Resolutions are measured using the “pick element from page” tool, icon seen to the left of the “inspector” tab, in the inspector of the browser, the inspector can be show by pressing F12 on the keyboard.



Screenshot	Resolution
	1230*692
	1108*623
	961*541
	Any resolution below 960*540 is cropped, this is expected as the solution has specified the minimum display is 960*540

Testing an assortment of possible cases:

Starting state	Action and expectation	Result state
	Action: click on object 0 Expectation: expand its dropdown	
	Action: click on object 0 Expectation: collapse its dropdown	
	Action: click on new rigidbody Expectation: first available rigidbody(2) added	
	Action: click on new rigidbody Expectation: first available rigidbody(3) added, new rigidbody button hidden	
	Action: click on object 1 Expectation: expand its dropdown	
	Action: click on object 2 Expectation: expand its dropdown, object 1 dropdown automatically closed	
	Action: click on object 1 Expectation: expand its dropdown, object 0 dropdown automatically closed	

	Action: click on remove rigidbody Expectation: remove the rigidbody(1)	
	Action: click on remove rigidbody Expectation: remove the rigidbody(2)	
	Action: click on remove rigidbody Expectation: remove the rigidbody(3)	
	Action: click on new rigidbody Expectation: first available rigidbody(1) added	
	Action: click on new rigidbody Expectation: first available rigidbody(3) added, new rigidbody button hidden	

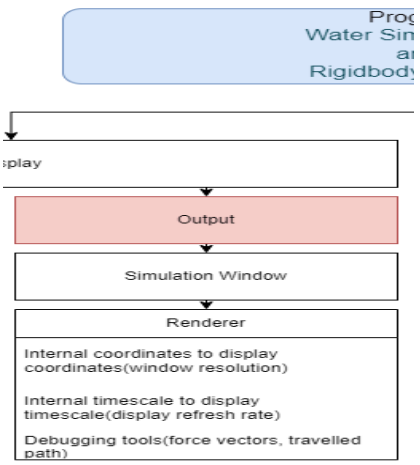
Conclusion:

All test resolutions resulted in the desired outcome, the window is able to be scaled from 960px*540px upward, maintaining the aspect ration of 16:9. I was unable to test higher values as i didn't have a display which supported higher resolutions.
All test actions resulted in the desired outcome, not all combinations of actions were able to be tested as there is approximately 4^4 combinations of state changes.
The GUI currently doesn't yet interact with the simulation, this functionality is planed to be added in a future iteration.

Phase 3:

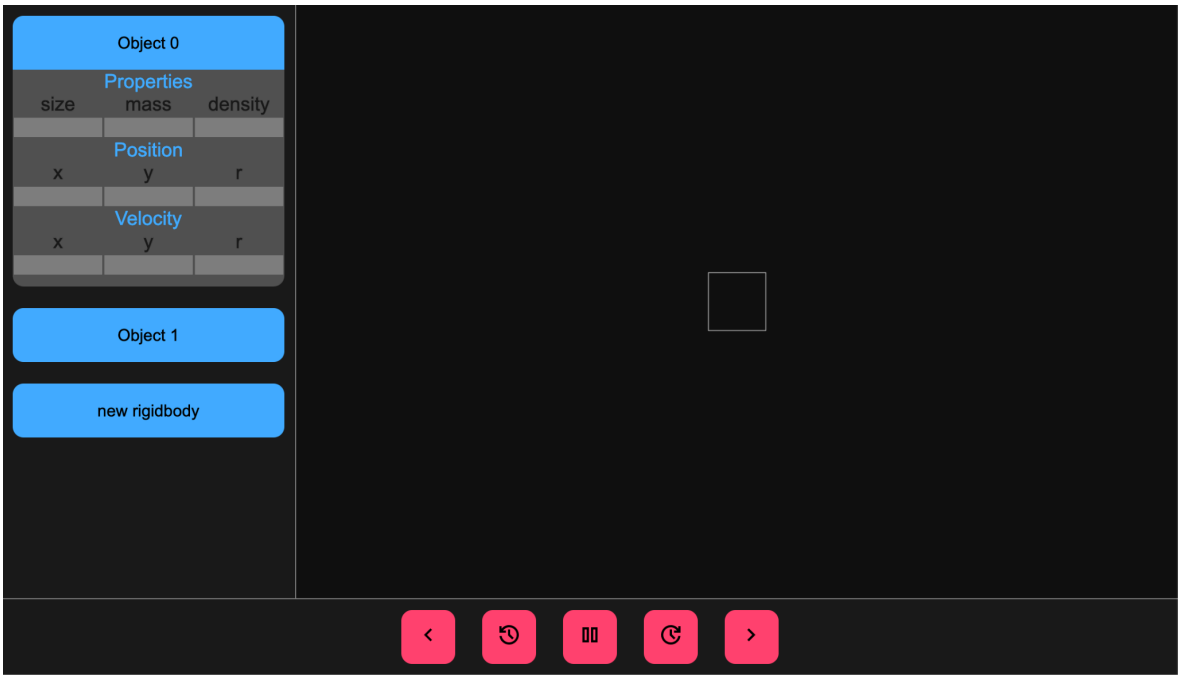
Rendering using the <canvas> element in html.
The goal of this phase is to kick start the development of the renderer, the renderer is responsible for translating the internal numeric values into a visual representation.
By making the renderer first, I can test the reliability of it by manually entering values into code directly.

Documentation:
https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/canvas

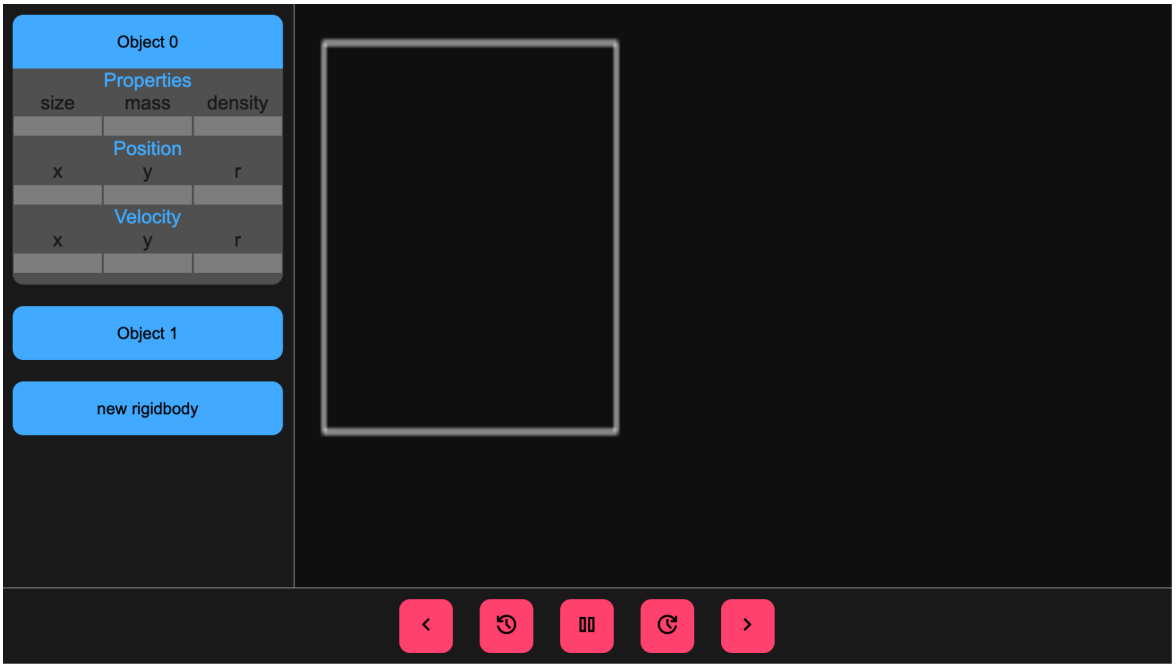


The Renderer:

Render anything test:



Rendering a simple square. During this i had this major set back: the s**caling and resolution were completely inconsistent**, as it turns out, the canvas element's default resolution is 150px*100px, **instead of inheriting from the CSS** that defines its actual size on screen.



To the top was an attempt at drawing a 100px * 100px square, with the code below.

```
ctx.strokeStyle = "white";
ctx.strokeRect(10,10,100,100);
```

It is clearly not a square. The solution was actually setting the resolution in JS, using a base unit, multiplying appropriate values based on the aspect ratio.

```
//canvas w:h = 12:8
var base = 128;
//base unit can be scaled if needed later
var cwidth = base * 12;
var cheight = base * 8;

canvas.width = cwidth;
canvas.height = cheight;
```

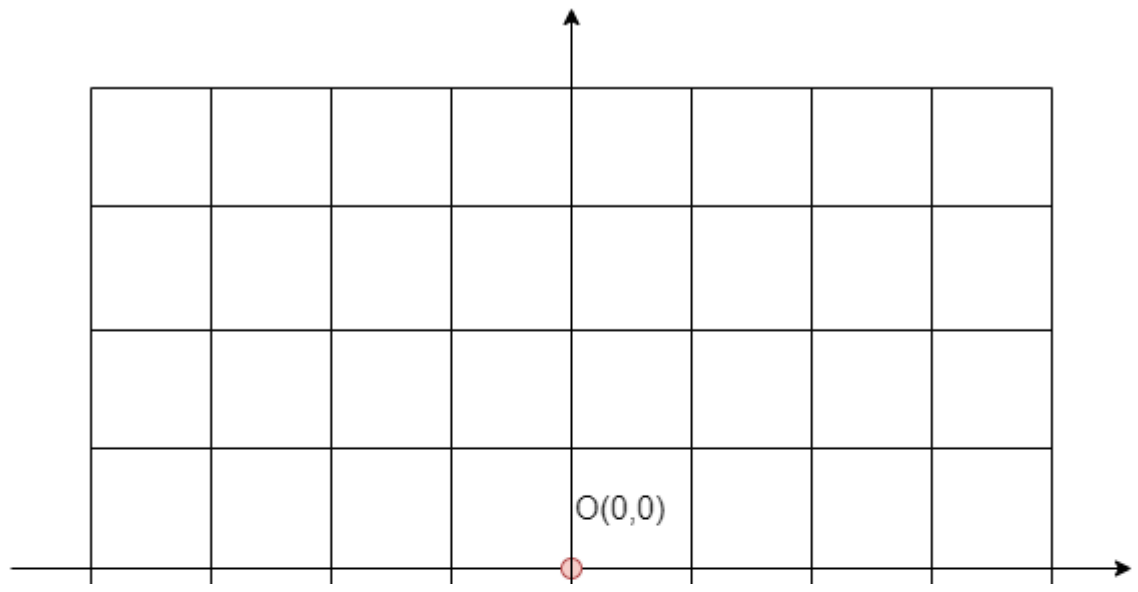
JS snippet (main.js)

```
import propGUIinitiate from "./modules/propertiesGUI.js";
import newRigidbodyGUI from "./modules/newrigidbodyGUI.js";
import removeRigidbodyGUI from "./modules/removerigidbodyGUI.js";
import testDraw from "./modules/testrenderer.js";

propGUIinitiate();
newRigidbodyGUI();
removeRigidbodyGUI();
testDraw();
```

JS snippet (testrenderer.js)

```
function testDraw(){
   var canvas = document.getElementById("canvas1");
   //var sim = document.getElementById("simulation");

   var ctx = canvas.getContext("2d");
   //console.log(sim.offsetWidth);
   //console.log(sim.offsetHeight);

   //canvas w:h = 12:8
   var base = 128;//base unit can be scaled if needed later
   var cwidth = base * 12;
   var cheight = base * 8;

   canvas.width = cwidth;
   canvas.height = cheight;

   // console.log(cwidth);
   // console.log(cheight);

   //draw rect
   //ctx.beginPath();
   ctx.strokeStyle = "white";
   ctx.strokeRect(cwidth/2-50,cheight/2-50,100,100);
   //ctx.stroke();
}
export default testDraw;
```

Remapping the canvas origin:



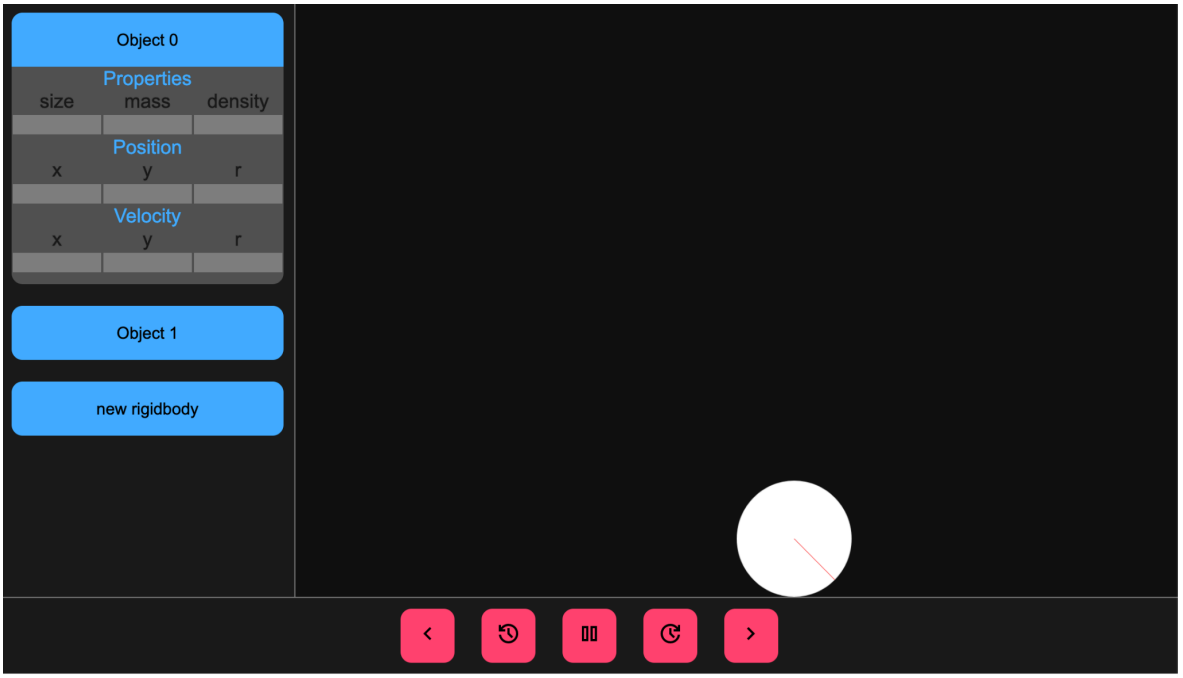
I would like the canvas origin to be in the bottom centre, as most mathematicians are used to setting the ground as y=0, the translate function is able to set the origin by moving by a specified amount from the true origin, however, **i wasn't able to make the up direction positive y**, therefore i will simply **multiply the y position value by -1 upon rendering a rigidbodies position**, essential making the renderer a mirror image of the actual coordinate grid used internally from top to bottom.

Documentation:
https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/translate

JS snippet (canvas.js)

```
function canvas() {
   var canvas = document.getElementById("canvas1");

   var ctx = canvas.getContext("2d");
   //console.log(sim.offsetWidth);
   //console.log(sim.offsetHeight);

   //canvas w:h = 12:8
   var base = 128;//base unit can be scaled if needed later
   var cwidth = base * 12;
   var cheight = base * 8;

   canvas.width = cwidth;
   canvas.height = cheight;

   ctx.translate(cwidth/2,cheight);

   // console.log(cwidth/2);
   // console.log(cheight);

   //draw rect to test
   /*
   ctx.fillStyle = "white";
   ctx.fillRect(100,-200, 100, 100);
   */
}
export default canvas;
```

Drawing a circle with line:



Drawing a circle with a line to **indicate the rotation of the object**, the circle is by far the easiest shape to draw, because it is **completely symmetrical**, using inbuilt Math trigonometry functions, we can easily draw diagonal lines.

Other shapes will be implemented in a future iteration, after physics for circles is added. In a future iteration I plan on changing the function to accept an object rather than multiple parameters, but it is currently clearer to just use arrays.

JS snippet (main.js)

```
var RB1pos = [100,100,45];
var RB1size = 100;
circle(RB1pos,RB1size);
```

JS snippet (shapesRenderer.js)

```
function circle(intPos,intSize){
   //circle and line to indicate rotation angle
   var canvas = document.getElementById("canvas1");
   var ctx = canvas.getContext("2d");

   intPos[1] *= -1;//flip y axis

   ctx.fillStyle = "white";
   ctx.arc(intPos[0],intPos[1],intSize,0,2*Math.PI);
   ctx.fill();

   ctx.beginPath();
   ctx.strokeStyle = "red";
```

```
    ctx.moveTo(intPos[0],intPos[1]);
    //trig to draw line at angle
    ctx.lineTo(intPos[0] + intSize * Math.cos(intPos[2]*Math.PI/180), intPos[1] +
intSize * Math.sin(intPos[2]*Math.PI/180));
    ctx.stroke();
}
function square(intPos, intSize){
    //square stuff, has identicle phases every 90deg
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    intPos[1] *= -1;//flip y axis

    if(intPos[2]%90===0){
        ctx.fillStyle = "white";
        ctx.fillRect(intPos[0]-intSize,intPos[1]-intSize,intSize*2,intSize*2);
    }else{
        //draw diamond shape when rotation angle != n*90
    }
}
//other shapes later
export {circle,square};
```

Drawing the circle

```
ctx.arc(intPos[0],intPos[1],intSize,0,2*Math.PI);
```

The `arc()` function takes in 6 parameters:

- X coordinate of centre
- Y coordinate of centre
- The radius
- Starting angle
- Ending angle
- Counterclockwise (optional)

The starting and ending angle are in radians, we are drawing a circle which is 360° equaling 2π radians. We can use Math.PI to access the value of π then multiply by 2. We can also use a more general equation to convert any angle from degrees to radians:

```
radians = (Math.PI/180)*degrees
```

```
ctx.fill();
```

When 2 ends of a path are joined we can use the `fill()` function to fill the inside of the path with the colour set by the `fillstyle` value.

Drawing the line

```
ctx.moveTo(intPos[0],intPos[1]);
```

The `moveTo()` function takes 2 parameters:

- X coordinate
- Y coordinate

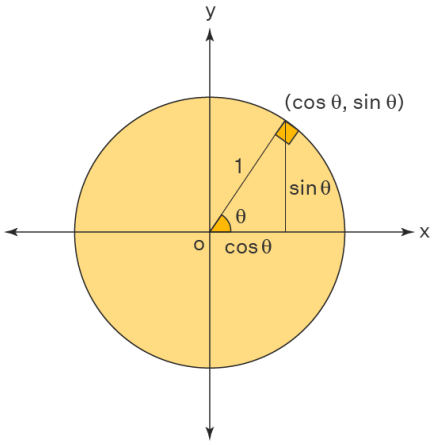
Is used to move to the centre point of circle.

```
ctx.lineTo(intPos[0] + intSize * Math.cos(intPos[2]*Math.PI/180),
intPos[1] + intSize * Math.sin(intPos[2]*Math.PI/180));
```

The `lineTo()` function takes 2 parameters:

- X coordinate
- Y coordinate

Is used to move to the edge of the circle, we can get the **x and y offset** from the centre by using the trigonometric functions sin() and cos(). Multiply the size by cos(angle) to get the horizontal offset from the centre, multiply the size by sin(angle) to get the vertical offset from the centre.
We can use `Math.sin()` and `Math.cos()` to access these functions in JS and they take parameters in radians. The `intPos[2]` holds the value of the rotational position of the rigidbody in degrees, we can use the previously mentioned equation of `radians = (Math.PI/180)*degrees` to convert any angle from degrees to radians.



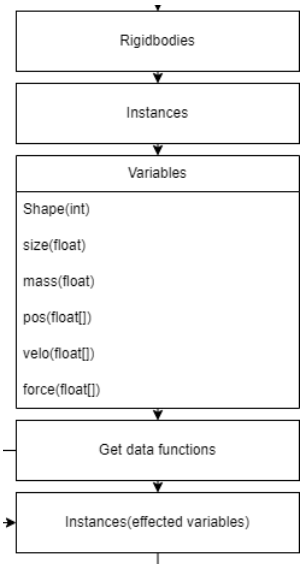
(source: https://www.cuemath.com/geometry/unit-circle/)
Then finally add on the centre coordinate onto the offset.

```
ctx.stroke();
```

We can use `stroke()` function to trace a path with the colour from `strokeStyle value.`

Phase 4:

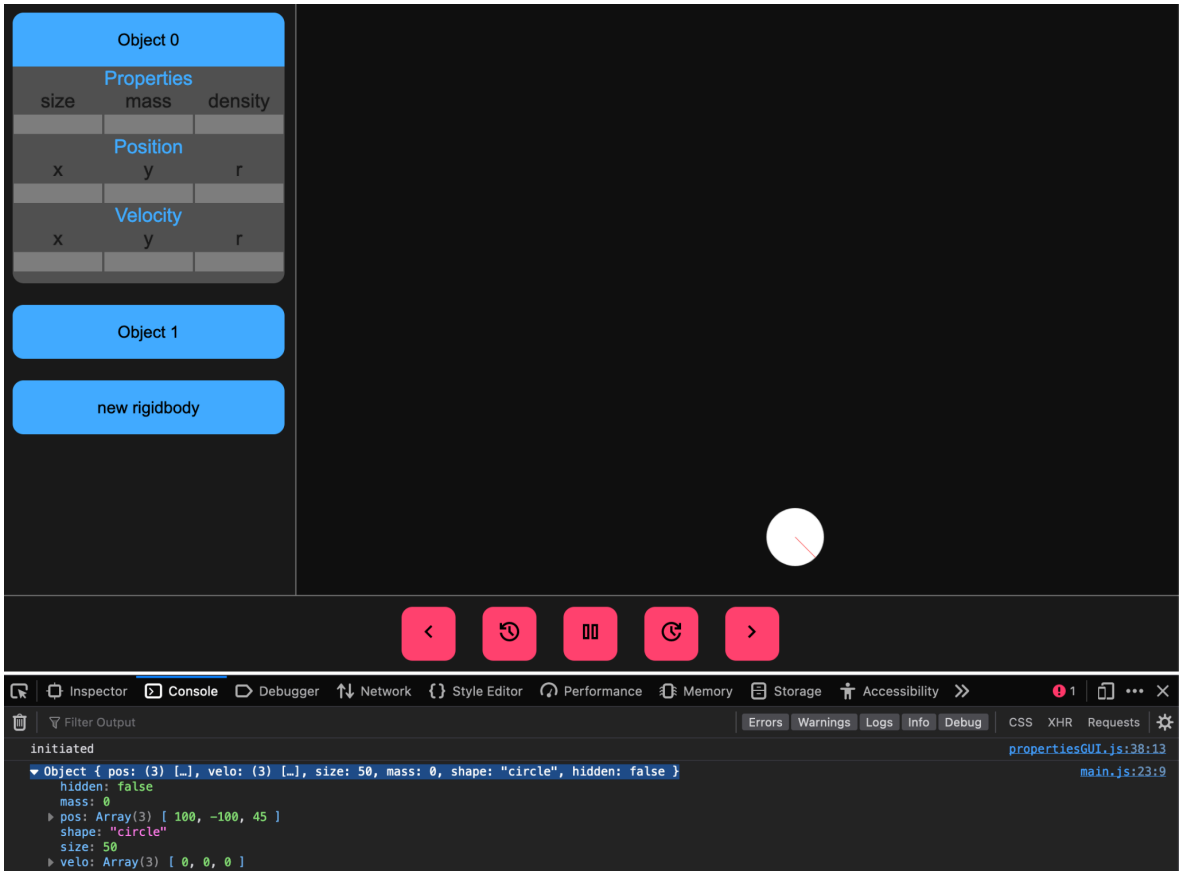
Starting with implementing the skeleton of the **physics engine** to test the most basic parts of the simulation, starting with rigidbodies and their attributes using **object oriented programming**.



The rigidbodies:

While coding, i will frequently use the shorthand “RB” to stand for rigidbodies.

Rigidbody object constructor:



Modifying the existing code to work in terms of **object oriented programming**, this will **add clarity and modularity** to the code, and we only need to pass a single parameter (the rigidbody object) into functions that deal with rigidbodies.

Documentation:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_objects
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

JS snippet (main.js)

```
function rigidBody(intPos,intVelo,intSize,intMass,strShape,boolHidden){
   this.pos = intPos;
   this.velo = intVelo;
   this.size = intSize;
   this.mass = intMass;
   //this.density = mass/size;
   this.shape = strShape;
   this.hidden = boolHidden
}
var RB = new rigidBody([100,100,45],[0,0,0],50,0,"circle",false);

console.log(RB);

circle(RB);
```

JS snippet (shapesRenderer.js)

```
function renderer(RB) {

   //reset canvas
   var canvas = document.getElementById("canvas1");
   var ctx = canvas.getContext("2d");

   //canvas w:h = 12:8
   const base = 128;//base unit can be scaled if needed later
   const cwidth = base * 12;
   const cheight = base * 8;

   canvas.width = cwidth;
   canvas.height = cheight;

   ctx.translate(cwidth / 2, cheight);

   if (RB.shape === "circle") {
       circle(RB);
   } else if (RB.shape === "square") {
       //render square
```

```
    }//more shapes later

}

function circle(RB/*Rigidbody object*/) {
    //circle and line to indicate rotation angle
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    //
    var x = Math.round(RB.pos[0]),
        y = -Math.round(RB.pos[1]),
        r = Math.round(RB.pos[2]),
        size = RB.size;

    ctx.fillStyle = "white";
    ctx.arc(x, y, size, 0, 2 * Math.PI);
    ctx.fill();

    ctx.beginPath();
    ctx.strokeStyle = "red";
    ctx.moveTo(x, y);
    //trig to draw line at angle
    ctx.lineTo(x + size * Math.cos(r * Math.PI / 180), y + size * Math.sin(r *
Math.PI / 180));
    ctx.stroke();
}

function square(intPos, intSize) {
    //square stuff, has identicle phases every 90deg
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    intPos[1] *= -1;//flip y axis

    if (intPos[2] % 90 === 0) {
        ctx.fillStyle = "white";
        ctx.fillRect(intPos[0] - intSize, intPos[1] - intSize, intSize * 2,
intSize * 2);
    } else {
        //draw diamond shape when rotation angle != n*90
    }
}
//other shapes later
```

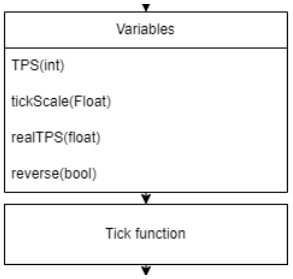
```
export default renderer;
```

Function	Explanation
`renderer()`	resets the canvas and references the relevant shape renderer
`circle()`	Shape renderer for circles, draws a circle shape with a line to indicate rotation
`square()`	Shape renderer for squares, incomplete work in progress function to draw squares

The simulation:

Tick function and module:

This is the first iteration of the tick function, the tick function controls everything that happens within the simulation, the tick module also contains other functions that perform each part of the simulation.
At this point i have decided to not use the force / resultant force to perform my calculations.



JS snippet (Tick.js)

```
function tick(funcRenderer, intTPS, intTickScale, RB) {
   //everything that needs to happen in 1 tick
   console.log("tick");

   if (!RB.hidden) {
       if (intTickScale > 0) {
           normal(intTPS, intTickScale, RB);

       } else if (intTickScale < 0) {
           reverse(intTPS, intTickScale, RB);
       }

       console.log(RB.velo);
       console.log(RB.pos);

       funcRenderer(RB);
   }

   console.log("tock");
```

```
}
function normal(intTPS, intTickScale, RB) {
    //normal forward
    RB = gravity(RB, intTPS, intTickScale);
    RB = move(RB, intTPS, intTickScale);
}

function reverse(intTPS, intTickScale, RB) {
    //reverse backward
    console.log("reverse tick");
    RB = move(RB, intTPS, intTickScale);
    RB = gravity(RB, intTPS, intTickScale);
}

function gravity(RB, intTPS, intTickScale) {
    RB.velo[1] -= 10 / intTPS * intTickScale;
    //9.8m/s = m/tick
    return RB;
}

function move(RB, intTPS, intTickScale) {
    for (let i = 0; i < 3; i++) {
        RB.pos[i] += RB.velo[i] / intTPS * intTickScale;
    }
    return RB;
}

export { tick };
```

Its at this point i realised Javascript passes parameters **by reference** rather than **by value**, so i was extra careful while coding these.
(above statement is wrong and will be addressed in Phase 5)

Function	Explanation
`tick()`	Is referenced when a tick needs to be executed.
`normal()`	A tick moving forwards in time, with the **normal order** of operations
`reverse()`	A tick moving backwards in time, with the **reversed order** of operations
`gravity()`	Adds the acceleration due to gravity to the y velocity of the rigidbody
`move()`	Adds the velocity of itself to its position

The Inputs:

Stepping a tick in time:

The first implementation of an input to the simulation, directly calling the `tick()` function, the next stage of the simulation can be drawn on demand when clicked.

JS snippet (timeGUI.js)

```
function timeGUI() {
}

function stepForward(funcTickOnclick, funcRenderer, intTPS, intTickScale, RB) {
   var stepf = document.getElementById("stepf");
   stepf.addEventListener("click", function () {
       funcTickOnclick(funcRenderer, intTPS, intTickScale, RB)
   });
}

function stepBackward(funcTickOnclick, funcRenderer, intTPS, intTickScale, RB) {
   var stepb = document.getElementById("stepb");
   stepb.addEventListener("click", function () {
       funcTickOnclick(funcRenderer, intTPS, -intTickScale, RB)
   });
}

function playForward(funcOnclick) {
   var playf = document.getElementById("playf");
   playf.addEventListener("click", function () {
       funcOnclick()
   });
}

function playBackward(funcOnclick) {
   var playb = document.getElementById("playb");
   playb.addEventListener("click", function () {
       funcOnclick()
   });
}

function pause(funcOnclick) {
   var pause = document.getElementById("pause");
   pasue.addEventListener("click", function () {
       funcOnclick()
   });
}
```

```
export { timeGUI, stepBackward, stepForward, playBackward, playForward, pause };
```

Function	Explanation
`stepForward()`	Calls the tick() function when the “**step forward**” button is clicked
`stepBackward()`	Calls the tick() function when the “**step backward**” button is clicked, but **multiples the tick scale by -1** to reverse the tick when calling the tick() function
`playForward()`	Unfinished function to play the simulation forwards in time
`playBackward()`	Unfinished function to play the simulation backwards in time
`pause()`	Unfinished function to pause the simulation

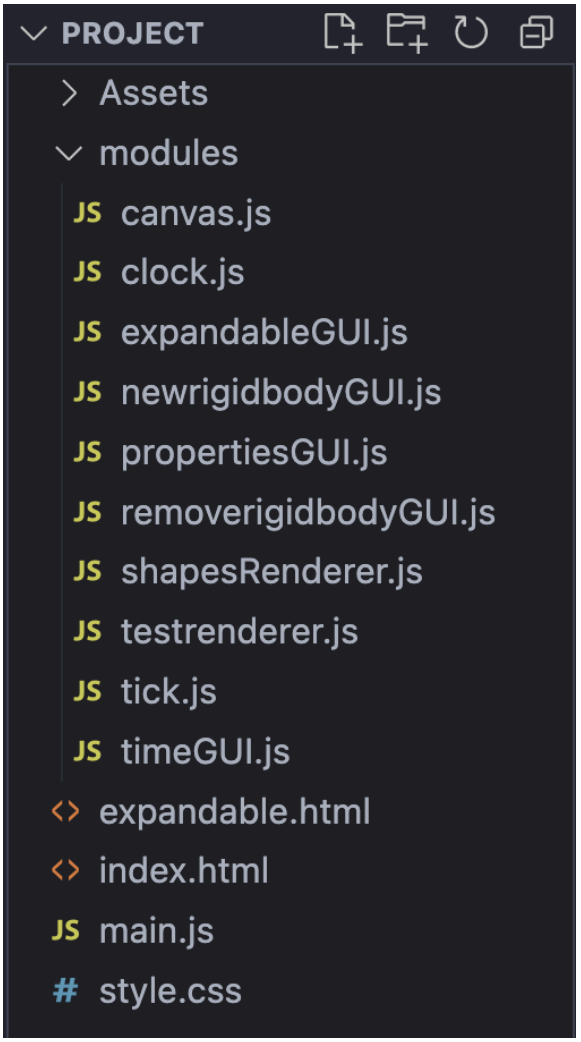
The clock function:

The clock function was planned for this phase, but it is delayed to the next phase after testing/demonstrating this iteration of the project, as it is way more complex than originally thought.

Documentation:
https://developer.mozilla.org/en-US/docs/Web/API/setInterval
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

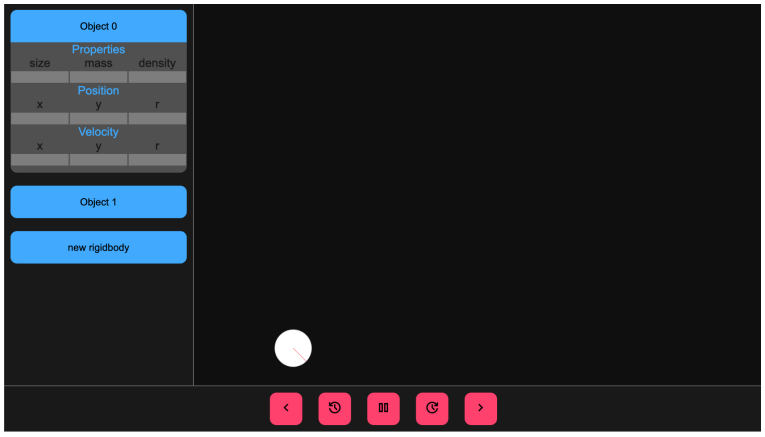
Testing and demonstration of current iteration(2):

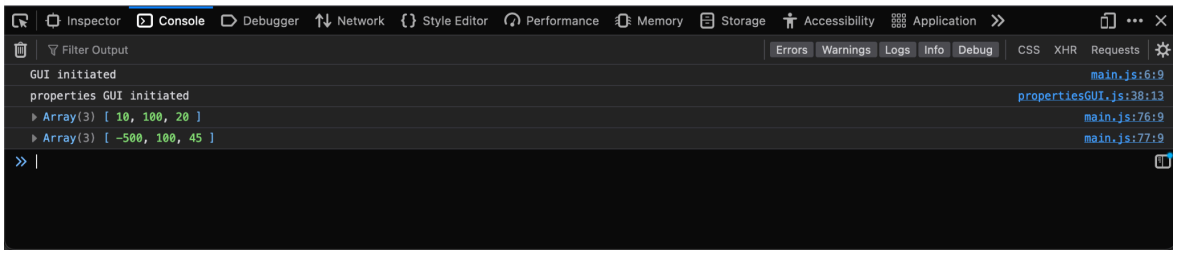
Overview of the current file structure:

Current file structure:	
3 unused files: • clock.js • testrenderer.js • expandable.html	

Demonstration:

1.The initial state of the program when opened:





The GUI is complete, but only the step forward and step backward buttons currently influence the simulation window.

Since we cannot spawn in new rigidbodies from the GUI yet, the rigid body is directly instantiated in `main.js` in code.

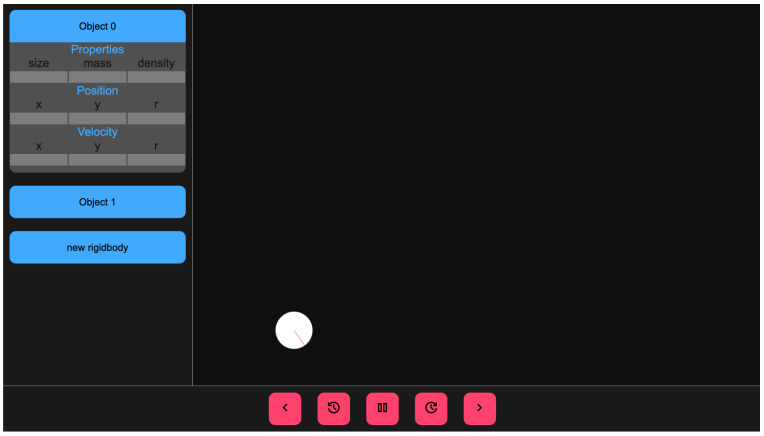
The debug console available in web browsers and is opened by pressing F12, useful for logging any variables or stages in the program.

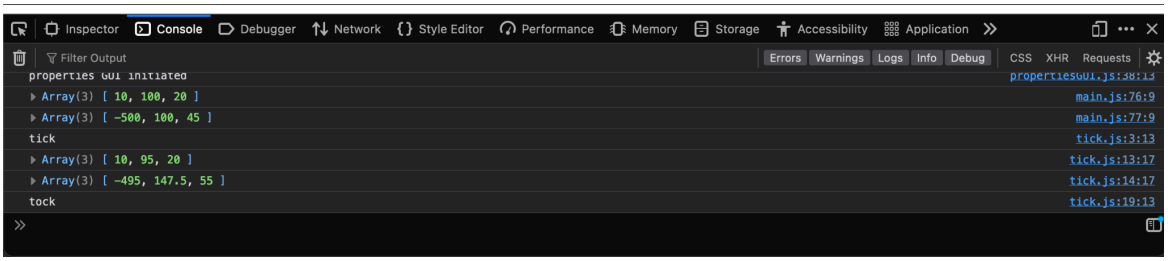
JS snippet (main.js)

```
let RB = new rigidBody([-500, 100, 45], [10, 100, 20], 50, 0, "circle", false);
renderer(RB);
console.log(RB.velo);
console.log(RB.pos);
```

Logs the position and velocity after creation in the console.

2.Click on step forward:





One click on step forward button, `tick()` function is used, moves the rigidbody and renders correctly.

As seen in the debug console, the logged x,y,r positions and velocity is same as expected.

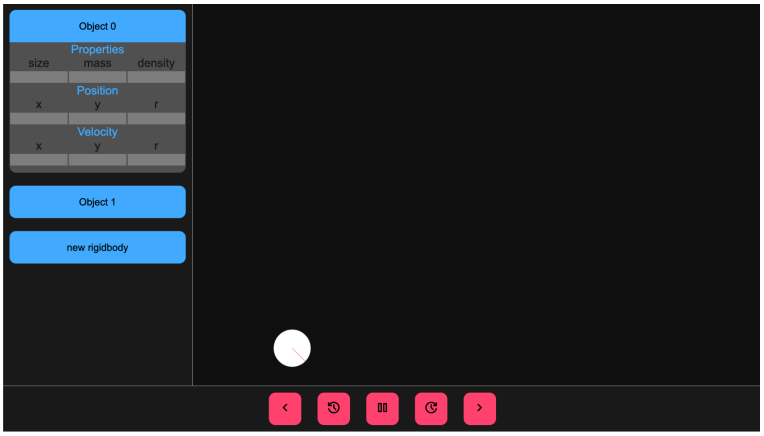
The renderer successfully refreshed the frame, redrawing the rigidbody in the correct position.

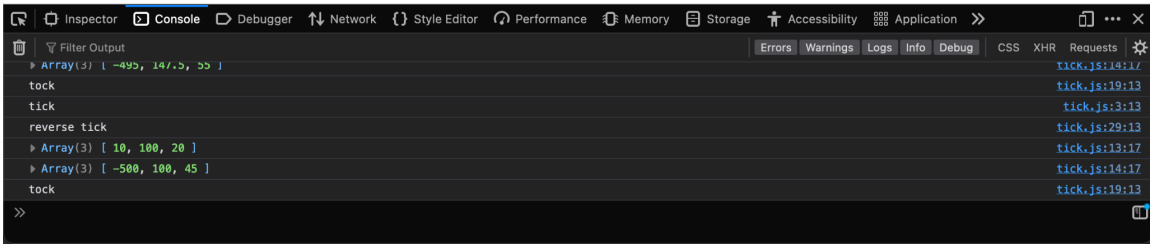
JS snippet (tick.js)

```
function tick(funcRenderer, intTPS, intTickScale, RB) {
   //everything that needs to happen in 1 tick
   console.log("tick");

   if (!RB.hidden) {
       if (intTickScale > 0) {
           normal(intTPS, intTickScale, RB);

       } else if (intTickScale < 0) {
           reverse(intTPS, intTickScale, RB);
       }

       console.log(RB.velo);
       console.log(RB.pos);

       funcRenderer(RB);
   }
```

```
    console.log("tock");
}
```

3.Click on step backwards:





One click on step backward button, `tick()` function is used, moves the rigidbody and renders correctly, rigidbody returns to the state it started in.

In this case the `reverse()` function is used within `tick()`, rather than `normal()`, ensuring the order of operations is consistent.

`reverse()` has the reversed order of operations to `normal()`.

As seen in the debug console, the logged x,y,r positions and velocity is same as expected.

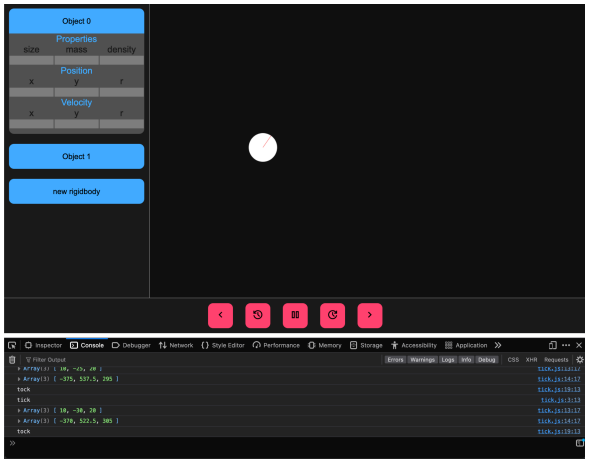
The renderer successfully refreshed the frame, redrawing the rigidbody in the correct position.

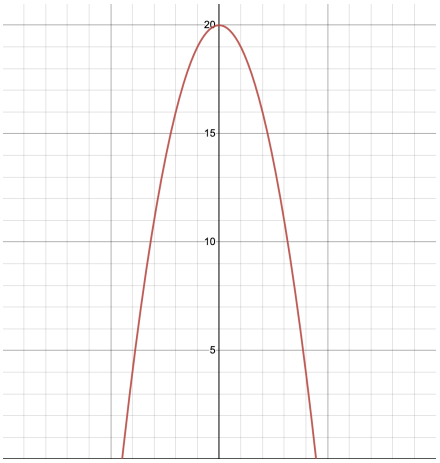
JS snippet (tick.js)

```
function normal(intTPS, intTickScale, RB) {
    //normal forward
    RB = gravity(RB, intTPS, intTickScale);
    RB = move(RB, intTPS, intTickScale);
}

function reverse(intTPS, intTickScale, RB) {
```

```
    //reverse backward
    console.log("reverse tick");
    RB = move(RB, intTPS, intTickScale);
    RB = gravity(RB, intTPS, intTickScale);
}
```

4.Repeated stepping:





Repeated clicking on step forward, the `tick()` function is used repeatedly, rigidbody moves in the path of a parabola, which is the expected motion of a projectile.

Clicking on step backward always reverts the state of the previous state of the simulation.

Simulation is currently consistent in both directions of time.

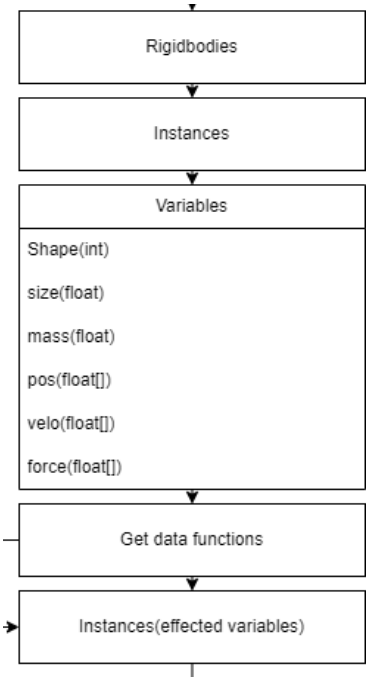
Conclusion:

The basis and framework of the simulation and integration of its most basic control is starting to take shape, however there is still a wide range of features that need to be implemented, and a more organised version of the code needs to be produced with features such as encapsulation, clock cycles, collision checks, or the properties input.

Phase 5:

The rigidbodies:

Encapsulation:
Rewriting the rigidbody class to **properly use encapsulation**, as the previous solution caused some difficulties when only wanting to read a value. also taking methods form the `tick.js` module that will only effect one rigidbody, and moving it in the rigidbody class.



Parameter passing(evaluation strategy):
More research was done in how JavaScript passes parameters, JS **passes values by sharing**, which im not very familiar with.

Call by sharing:
"*is an evaluation strategy that is intermediate between call by value and call by reference. Rather than every variable being exposed as a reference, only a specific class of values, termed "references", "boxed types", or "objects"* "

"*used by many modern languages…JavaScript (objects)*"

Source:
https://en.wikipedia.org/wiki/Evaluation_strategy#Call_by_sharing

JS snippet (main.js)

```
class rigidBody {
    #pos;
    #velo;
    #size;
    #mass;//unused
    #shape;
    #hidden;
    #id;//unused
    constructor(intPos, intVelo, intSize, intMass, strShape, boolHidden, intId) {
        this.#pos = intPos;
        this.#velo = intVelo;
        this.#size = intSize;
        this.#mass = intMass;
        //this.density = mass/size;
        //force = velo*mass
        this.#shape = strShape;
        this.#hidden = boolHidden;
```

```
        this.#id = intId;
    }

    // methods
    gravity(intTPS, intTickScale) {
        this.#velo[1] -= 10 / intTPS * intTickScale;
        //9.8m/s = m/tick
    }

    move(intTPS, intTickScale) {
        for (let i = 0; i < 3; i++) {
            this.#pos[i] += this.#velo[i] / intTPS * intTickScale;
        }
    }
    //gets
    get getPos() {
        return this.#pos;
    }
    get getVelo() {
        return this.#velo;
    }
    get getSize() {
        return this.#size;
    }
    get getHidden() {
        return this.#hidden;
    }
    get getShape() {
        return this.#shape;
    }
    get getMass() {
        return this.#mass;
    }
    get getHidden() {
        return this.#hidden;
    }
    get getId() {
        return this.#id;
    }
    //sets
    set setPos(intPos) {
        this.#pos = intPos;
    }
    set setVelo(intVelo) {
        this.#velo = intVelo;
    }
    set setSize(intSize) {
        this.#size = intSize;
    }
```

```
    set setHidden(boolHidden) {
        this.#hidden = boolHidden;
    }
    set setShape(strShape) {
        this.#shape = strShape;
    }
}
```

- The # symbol indicates a **private property or private method.**
- Private properties **require a get or set function to read or write** outside the scope of the class.
- The `gravity()` and `move()` methods don't interact with other rigidbodies so they are included into the class

The simulation:

Modifying the `tick.js` module to be compatible with encapsulation and private properties.

JS snippet(tick.js)

Removed section

```
/*
function gravity(RB, intTPS, intTickScale) {
    RB.velo[1] -= 10 / intTPS * intTickScale;
    //9.8m/s = m/tick
    return RB;
}

function move(RB, intTPS, intTickScale) {
    for (let i = 0; i < 3; i++) {
        RB.pos[i] += RB.velo[i] / intTPS * intTickScale;
    }
    return RB;
}
*/
```

Changed section

```
function tick(funcRenderer, intTPS, intTickScale, RB) {
    //everything that needs to happen in 1 tick
    console.log("tick");

    if (!RB.getHidden) {
        if (intTickScale > 0) {
            normal(intTPS, intTickScale, RB);
        } else if (intTickScale < 0) {
            reverse(intTPS, intTickScale, RB);
        }
        funcRenderer(RB);
    }
```

```
    console.log("tock");
}

function normal(intTPS, intTickScale, RB) {
    //normal forward
    RB.gravity(intTPS, intTickScale);
    RB.move(intTPS, intTickScale);
}

function reverse(intTPS, intTickScale, RB) {
    //reverse backward
    console.log("reverse tick");
    RB.move(intTPS, intTickScale);
    RB.gravity(intTPS, intTickScale);
}
```

- Removing the methods that have been included in the rigidbody class
- Using the `getHidden` method of the class to read the value of `#hidden`
- Using the methods now inbuilt with the class: `move(), gravity()`

The renderer:

Modifying the rendering module to be compatible with encapsulation and private properties.

JS snippet(shapesRenderer.js)

```
function renderer(RB) {

    //reset canvas
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    //canvas w:h = 12:8
    const base = 128;//base unit can be scaled if needed later
    const cwidth = base * 12;
    const cheight = base * 8;

    canvas.width = cwidth;
    canvas.height = cheight;

    ctx.translate(cwidth / 2, cheight);

    if (RB.getShape === "circle") {
        circle(RB);
    } else if (RB.getShape === "square") {
        //render square
    }//more shapes later

}
```

```
function circle(RB/*Rigidbody object*/) {
    //circle and line to indicate rotation angle
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    //
    var x = Math.round(RB.getPos[0]),
        y = -Math.round(RB.getPos[1]),
        r = Math.round(RB.getPos[2]),
        size = RB.getSize;

    ctx.fillStyle = "white";
    ctx.arc(x, y, size, 0, 2 * Math.PI);
    ctx.fill();

    ctx.beginPath();
    ctx.strokeStyle = "red";
    ctx.moveTo(x, y);
    //trig to draw line at angle
    ctx.lineTo(x + size * Math.cos(r * Math.PI / 180), y + size * Math.sin(r *
Math.PI / 180));
    ctx.stroke();
}
```

Now using the **get methods to read** values and **avoid writing** to these values by accident, examples: `getSize`, `getShape`, `getPos`.

Phase 6:

The clock function:

In this phase my goal is to attempting to implement the clock function, the clock function controls the **repeats of the tick function** in a set interval based on the target tickrate.

Clock function
The clock function has been included in the tick.js file for simplicity.
JS snippet (tick.js)

```
function clock(interval ,funcRenderer, intTPS, intTickScale, RB) {
   var intMSPT = 1000 / intTPS / intTickScale,     //milliseconds per tick
       intRTPS = intTPS * intTickScale,            //real ticks per second
       boolReverse = intTickScale < 0,
       boolPause = (intTickScale == 0);

   if (boolPause) {
       console.log("pls pause");
```

```
        clearInterval(interval);
    } else if (!interval) {
        interval = setInterval(tick, intMSPT, funcRenderer, intTPS, boolReverse,
RB);
    }
}

function tick(funcRenderer, intTPS, boolReverse, RB) {
    //everything that needs to happen in 1 tick
    //console.log("tick");

    if (!RB.getHidden) {
        if (!boolReverse) {
            normal(intTPS, RB);
        } else {
            reverse(intTPS, RB);
        }

        //console.log(RB.getPos);
        //console.log(RB.getVelo);

        funcRenderer(RB);
    }
    //console.log("tock");
}
```

`setInterval()` function in JS takes 2 parameters:

- The function to be executed
- The delay in milliseconds

The `setInterval()` function repeats the function repeatedly with a delay in milliseconds. Using it to call the `tick()` function on a specified interval.

Documentation:
https://developer.mozilla.org/en-US/docs/Web/API/setInterval

JS snippet (main.js)

```
let RBs = [
    new rigidBody(),
    new rigidBody(),
    new rigidBody(),
    new rigidBody(),
];

RBs[0].setHidden = false;
```

```
RBs[0].setPos = [-500, 500, 45];
RBs[0].setVelo = [1, 0, 25];
RBs[0].setDebug = true;

renderer(RBs[0]);

console.log(RBs);

var interval = null;
timeGUI(interval, t.clock, t.tick, renderer, TPS, intTickScale, RBs[0]);
```

```
RBs[0].setHidden = false;
RBs[0].setPos = [-500, 500, 45];
RBs[0].setVelo = [1, 0, 25];
RBs[0].setDebug = true;
```

The first (0th) rigidbody object is setup to have a position, velocity, and debug status.

```
var interval = null;
timeGUI(interval, t.clock, t.tick, renderer, TPS, intTickScale,
RBs[0]);
```

The variable `interval` is storing the ID of the interval that will be set in the `clock()` function. Then the variable is passed into the `timeGUI()` function, which will then pass it into the `clock()` function

The inputs:

JS snippet (timeGUI.js)

```
function timeGUI(interval, funcClock, funcTick, funcRenderer, intTPS,
intTickScale, RB) {

   stepForward(funcTick, funcRenderer, intTPS, RB);
   stepBackward(funcTick, funcRenderer, intTPS, RB);

   playForward(interval, funcClock, funcRenderer, intTPS, intTickScale, RB);
   playBackward(interval, funcClock, funcRenderer, intTPS, -intTickScale, RB);

   pause(interval, funcClock, funcRenderer, intTPS, RB);
}

function stepForward(funcOnclick, funcRenderer, intTPS, RB) {
   var stepf = document.getElementById("stepf");

   stepf.addEventListener("click", function () {
```

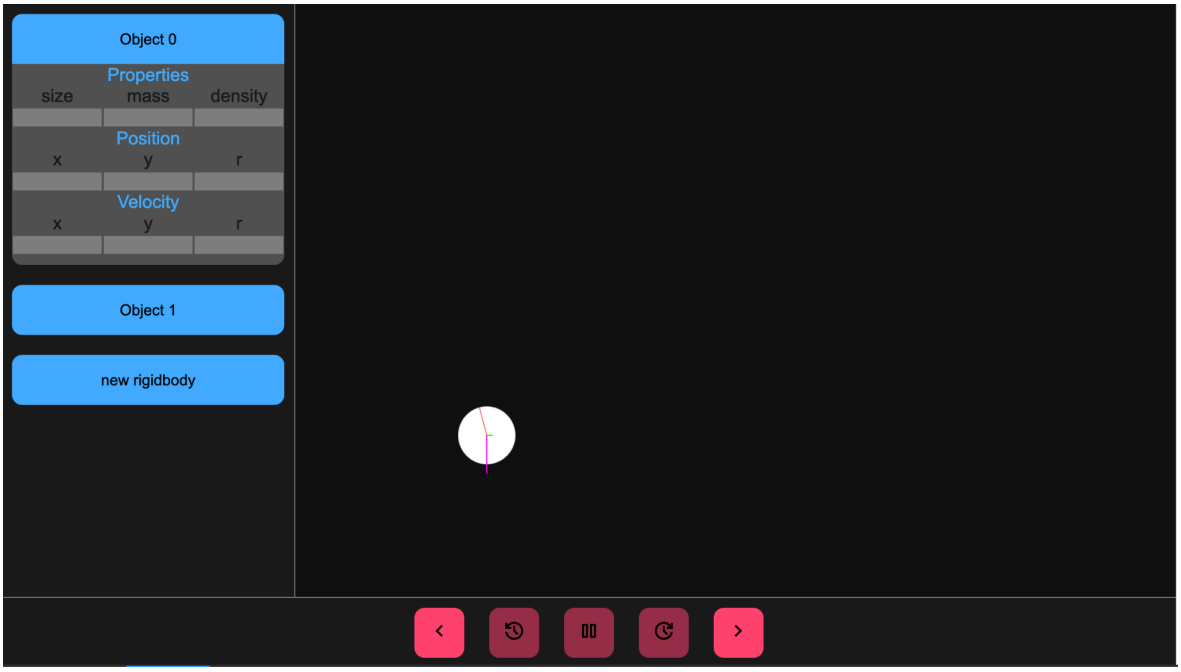
```
        funcOnclick(funcRenderer, intTPS, false, RB);
    });
}

function stepBackward(funcOnclick, funcRenderer, intTPS, RB) {
    var stepb = document.getElementById("stepb");
    stepb.addEventListener("click", function () {
        funcOnclick(funcRenderer, intTPS, true, RB);
    });
}

function playForward(interval, funcOnclick, funcRenderer, intTPS, intTickScale,
RB) {
    var playf = document.getElementById("playf");
    playf.addEventListener("click", function () {
        funcOnclick(interval, funcRenderer, intTPS, intTickScale, RB);
        console.log("playforwards");
    });
}

function playBackward(interval, funcOnclick, funcRenderer, intTPS, intTickScale,
RB) {
    var playb = document.getElementById("playb");
    playb.addEventListener("click", function () {
        funcOnclick(interval, funcRenderer, intTPS, intTickScale, RB);
        console.log("playbackwards");
    });
}

function pause(interval, funcOnclick, funcRenderer, intTPS, RB) {
    var pause = document.getElementById("pause");
    pause.addEventListener("click", function () {
        funcOnclick(interval, funcRenderer, intTPS, 0, RB);
        console.log("pasue");
    });

}
export default timeGUI;
```

Function	Explanation
`stepForward()`	Calls the `tick()` function when the “**step forward**” button is clicked.
`stepBackward()`	Calls the `tick()` function when the “**step backward**” button is clicked, but **modified**

	the parameter `boolReverse` **is set to** `true` to reverse the tick when calling the `tick()` function.
`playForward()`	Calls the clock() function when the "**play forward**" button is clicked.
`playBackward()`	Calls the `clock()` function when the "**play backward**" button is clicked, but `intTickScale` is multiplied by -1 to reverse the tick when calling the `tick()` function.
`pause()`	Calls the `clock()` function when the "**pause**" button is clicked, `intTickScale` is set to 0 to stop the clock.
`timeGUI()`	Initialises the module by calling all the above functions, is exported and called once in `main.js`

Misc:

Adding debug functions



Adding a debugging function to add a display of the velocity using vectors in the x (lime coloured line) and y (magenta coloured line) axis.

JS snippet (main.js)

```
class rigidBody {
    #pos;    //position
    #velo;   //velocity
    #size;   //radius
    #mass;   //mass      //unused
    #shape; //shape
    #hidden;//is it hidden
    #id;     //id        //unused
    #debug;

    constructor(/*intPos, intVelo, intSize, strShape, boolHidden*/) {
        //defaults
        this.#pos = [0, 500, 0];
        this.#velo = [0, 0, 0];
        this.#size = 50;
        this.#shape = "circle";
        this.#hidden = true;
        this.#debug = false;
        /*
        this.#pos = intPos;
        this.#velo = intVelo;
        this.#size = intSize;
        //this.#mass = intMass;
        //this.density = mass/size;
        //force = velo*mass
        this.#shape = strShape;
        this.#hidden = boolHidden;
        */
    }

    // methods
    gravity(intTPS, boolReverse) {
        this.#velo[1] -= 10 / intTPS * (boolReverse ? -1 : 1);
        //9.8m/s (10m/s)
        //RTPS = ticks/second
        //?m/tick
    }

    move(boolReverse) {
        for (let i = 0; i < 3; i++) {
            this.#pos[i] += this.#velo[i] * (boolReverse ? -1 : 1);  //add velo to
pos
        }
        this.#pos[2] %= 360; //reset rotation over 360deg
```

```
        //console.log(this.#pos);
    }

    groundCheck() {
        //TODO: fix this rubbish
        //there is technically inaccuracies here, the higher TPS the lower the
error
        if (this.#pos[1] - this.#size <= 0) {
            console.log("ground contact");
            //add back overshoot into ground
            //this.#pos[1] += this.#pos[1] - this.#size;
            //bounce and reverse y velo
            this.#velo[1] *= -1;
            //this.#velo[1] -= 10;
        }
    }

    //gets
    get getPos() {
        return this.#pos;
    }
    get getVelo() {
        return this.#velo;
    }
    get getSize() {
        return this.#size;
    }
    get getHidden() {
        return this.#hidden;
    }
    get getShape() {
        return this.#shape;
    }
    get getMass() {
        return this.#mass;
    }
    get getHidden() {
        return this.#hidden;
    }
    get getId() {
        return this.#id;
    }
    get getDebug(){
        return this.#debug;
```

```
    }

    //sets
    set setPos(intPos) {
        this.#pos = intPos;
    }
    set setVelo(intVelo) {
        this.#velo = intVelo;
    }
    set setSize(intSize) {
        this.#size = intSize;
    }
    set setShape(strShape) {
        this.#shape = strShape;
    }
    set setHidden(boolHidden) {
        this.#hidden = boolHidden;
    }
    set setDebug(boolDebug) {
        this.#debug = boolDebug;
    }
}
```

Added the #debug private attribute in the rigidbody object class, can be manually set to true when needing to debug by using: `RB.setDebug = true;`

JS snippet(shapesRenderer.js)

```
function renderer(RB) {

    //reset canvas
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    //canvas w:h = 12:8
    const base = 128;           //base unit can be scaled if needed later
    const cwidth = base * 12;
    const cheight = base * 8;

    canvas.width = cwidth;
    canvas.height = cheight;

    ctx.translate(cwidth / 2, cheight);

    if (RB.getShape === "circle") {
        circle(RB);
```

```
    } else if (RB.getShape === "square") {
        //render square
    }//more shapes later

    if (RB.getDebug) {
        renderDebug(RB);
    }
}
function renderDebug(RB) {
    //debug renderer
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    var px = Math.round(RB.getPos[0]),
        py = -Math.round(RB.getPos[1]),
        vx = Math.round(RB.getVelo[0]),
        vy = -Math.round(RB.getVelo[1]);

    ctx.beginPath();
    ctx.strokeStyle = "lime";
    ctx.lineWidth = 2;
    ctx.moveTo(px, py);
    ctx.lineTo(px + vx, py);
    ctx.stroke();

    ctx.beginPath();
    ctx.strokeStyle = "magenta";
    ctx.lineWidth = 2;
    ctx.moveTo(px, py);
    ctx.lineTo(px, py + vy);
    ctx.stroke();

    //console.log([vx, vy]);
}
```

Added the `renderDebug()` function, is called when the #debug attribute of the rigidbody is equal to `true` in the `renderer()` function.

Renders the vertical vector in lime, and the horizontal vector in magenta.

All coordinates are rounded as we cant render half pixels.

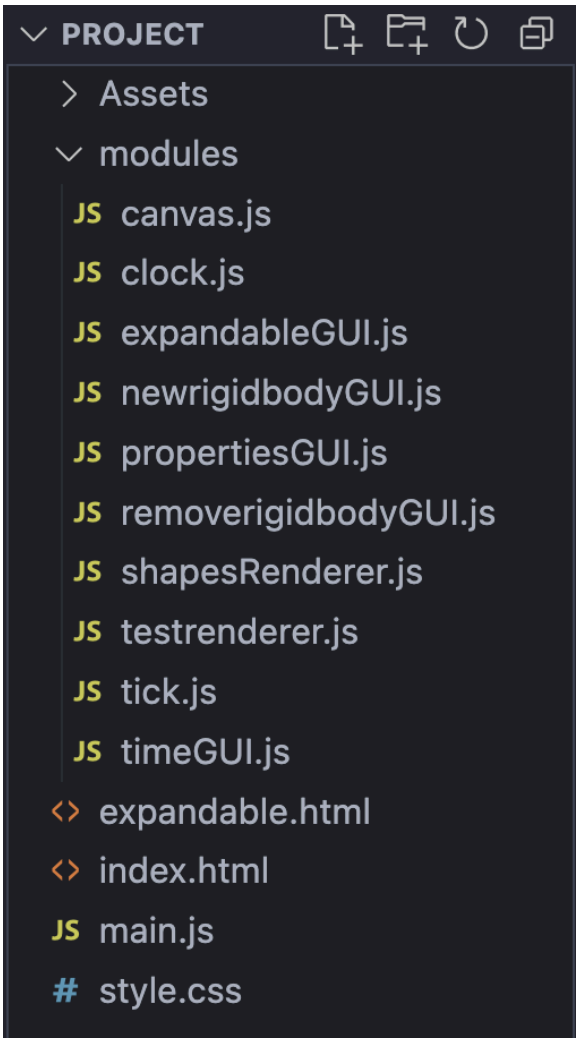
The y coordinates are mirrored because, as mentioned before, the canvas's coordinates are a vertical mirror of the simulators coordinates.

Testing and demonstration of current iteration(Final):

End of development:

This is the end of my development cycle, no more development or iterations will be produced, and i will be conducting the final testing to aid in the evaluation of this project.

Overview of the current file structure:

Current file structure:	
3 unused files: • clock.js • testrenderer.js • expandable.html	

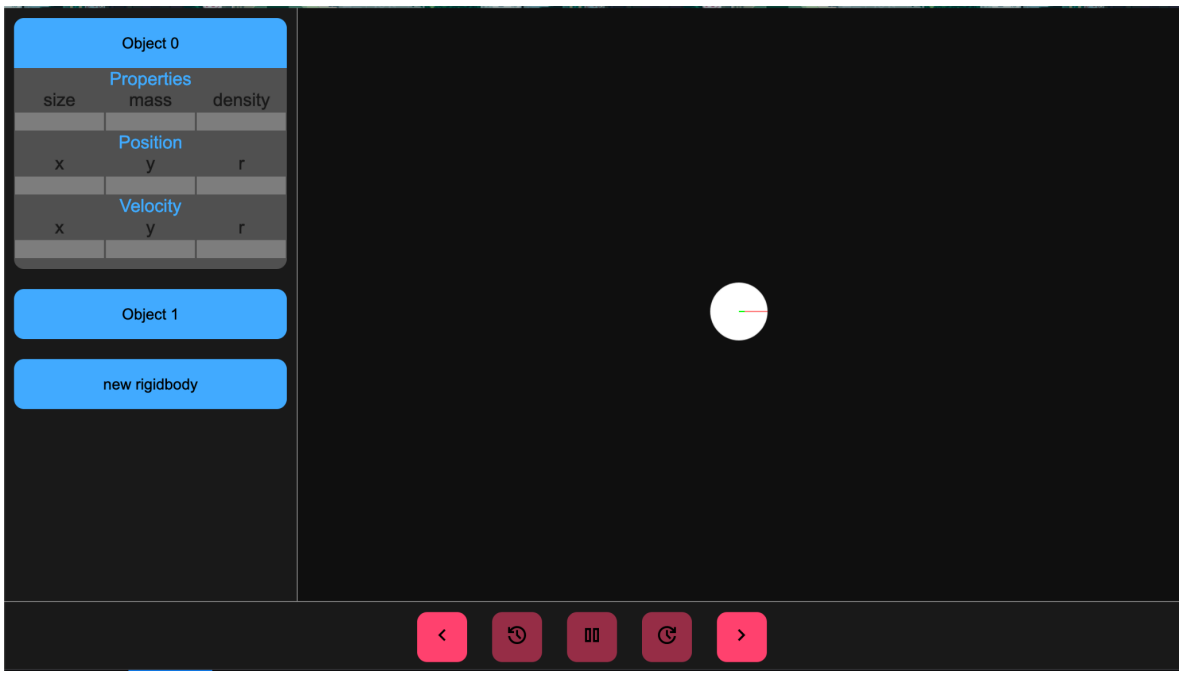
Planed features that are not present:

The input of properties:
I was unable to add the **input system for properties of rigidbodies** due to the incompleteness of the simulation processes. The input of properties would have utilised the `setAttribute()` functions added in phase 5.
The **conversion between different types of units**, ex: heading and magnitude to x and y components and vice versa, or, 2 of size, mass and density to calculate the other.

Sections of the simulation:
I was unable to add the **interactions between rigidbodies** due to the simulation not yet having any **calculations that handle multiple rigidbodies**, and therefore the usage and presence of multiple rigidbodies was forgone entirely.
Calculations not available for any shapes other than the circle.
Collisions, resultant force, and buoyancy are all major components of the simulation that are not implemented in any way.
The `tick()` functions development is incomplete, there were major planned functions planned, such as the: collision functions, resultant velocity function, buoyancy function. These functions would have been incorporated into the the order of operations that the tick function controls.
The `clock()` functions development is also incomplete, the **speed multiplier is not implemented**, due to not having a separate rendering cycle, the increase in simulation speed would be very taxing on the processor.
The rendering functions:
The renderer currently is **only able to handle and render 1 shape**, the circle, more shapes were planned but no more shape renderers were implemented due to the lack of simulations and calculations for other shapes.
The renderer is also not on a separate interval and is tied to the simulation cycles tick rate, this is a waste of computing power since the maximum refresh rate of most computer monitors are around 60hz, but the target simulation tick rate is higher at 100hz, meaning there would be rendered frames that will not be displayed on the monitor.

Demonstration:

1.The initial state of the program when opened:



The GUI is complete, but only the step forward, step backward, play forward, and play backward buttons currently influence the simulation window, i have also darkened the

“play forward”, “play backward” and “pause” buttons, due to some bugs that will be demonstrated later.

Since we cannot spawn in new rigidbodies from the GUI yet, the rigid body is directly instantiated in `main.js` in code.

The debug console available in web browsers and is opened by pressing F12, useful for logging any variables or stages in the program.

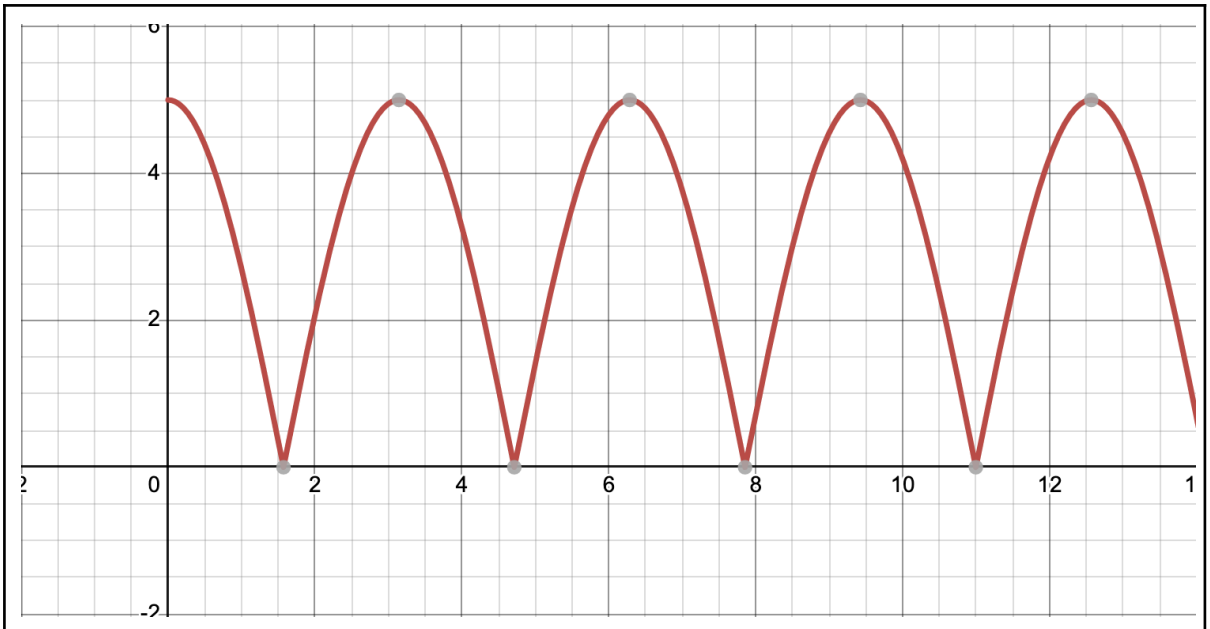
JS snippet(main.js)

```
const TPS = 100;        //ticks per second
```

JS snippet(main.js)

```
let RBs = [
   new rigidBody(),
   new rigidBody(),
   new rigidBody(),
   new rigidBody(),
];

RBs[0].setHidden = false;
RBs[0].setPos = [0, 500, 0];
RBs[0].setVelo = [10, 0, 10];
RBs[0].setDebug = true;

renderer(RBs[0]);

console.log("RB pos [" + RBs[0].getPos + "]");

var interval = null;
timeGUI(interval, t.clock, t.tick, renderer, TPS, intTickScale, RBs[0]);
```

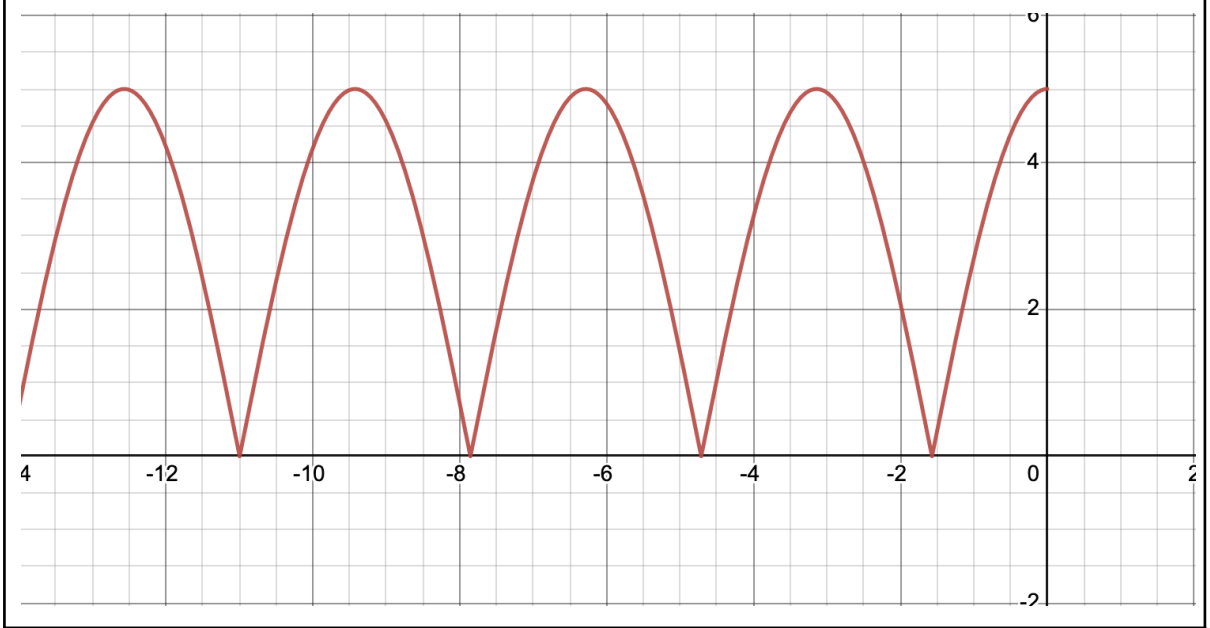
2.Playing forward:



Click on the “play forward” button

The rigidbody moves along the expected path of a parabola and towards the right, repeating as it bounces.

3.Playing backward:



Click on the “play backward” button

The rigidbody moves along the expected path of a parabola and towards the left, repeating as it bounces.

4.The bugs:

The “pause” button doesn't work, i am not entirely sure why, i have a suspicion that it is due to multiple threads being created when using the `addEventListener()` function, and my unfamiliarity of call by sharing causing the intervals’ ID to not be passed when created.

Below is the code that i suspect causes the issue.

JS snippet(tick.js)

```
function clock(interval ,funcRenderer, intTPS, intTickScale, RB) {
   var intMSPT = 1000 / intTPS / intTickScale,     //milliseconds per tick
       intRTPS = intTPS * intTickScale,            //real ticks per second
       boolReverse = intTickScale < 0,
       boolPause = (intTickScale == 0);

   if (boolPause) {
       console.log("pls pause");
       clearInterval(interval);
   } else if (!interval) {
       interval = setInterval(tick, intMSPT, funcRenderer, intTPS, boolReverse,
RB);
   }
}
```

JS snippet(timeGUI.js)

```
function timeGUI(interval, funcClock, funcTick, funcRenderer, intTPS,
intTickScale, RB) {

   stepForward(funcTick, funcRenderer, intTPS, RB);
   stepBackward(funcTick, funcRenderer, intTPS, RB);

   playForward(interval, funcClock, funcRenderer, intTPS, intTickScale, RB);
   playBackward(interval, funcClock, funcRenderer, intTPS, -intTickScale, RB);

   pause(interval, funcClock, funcRenderer, intTPS, RB);
}
```

```
function playForward(interval, funcOnclick, funcRenderer, intTPS, intTickScale,
RB) {
   var playf = document.getElementById("playf");
   playf.addEventListener("click", function () {
       funcOnclick(interval, funcRenderer, intTPS, intTickScale, RB);
       console.log("playforwards");
```

```
    });
}

function playBackward(interval, funcOnclick, funcRenderer, intTPS, intTickScale,
RB) {
    var playb = document.getElementById("playb");
    playb.addEventListener("click", function () {
        funcOnclick(interval, funcRenderer, intTPS, intTickScale, RB);
        console.log("playbackwards");
    });
}

function pause(interval, funcOnclick, funcRenderer, intTPS, RB) {
    var pause = document.getElementById("pause");
    pause.addEventListener("click", function () {
        funcOnclick(interval, funcRenderer, intTPS, 0, RB);
        console.log("pasue");
    });
}
```

JS snippet(main.js)

```
var interval = null;
timeGUI(interval, t.clock, t.tick, renderer, TPS, intTickScale, RBs[0]);
```

Clicking the “play forward” or “play backward” buttons multiple times causes a pseudo speed multiplier, clicking on either seems to create another interval on another thread that manipulates the same rigidbody object.

If “play forward” is clicked twice, there are 2 cooperative intervals both playing forward, causing the simulation to run at 2x speed. This looks to be true for any amount of clicks, until the processor is the bottleneck.

If “play backward” is clicked twice, there are 2 cooperative intervals both playing backward, causing the simulation to run at reverse 2x speed. This looks to be true for any amount of clicks, until the processor is the bottleneck.

If “play forward” and “play backward” are both clicked, there are 2 competing intervals, one playing forward and one playing backward, causing the simulation to pause. However this is not actually paused, the simulation is still using a lot of computing resources. This looks to be true for any equal amount of clicks, until the processor is the bottleneck.

Testing:

The accuracy of gravity and movement:

Setting a rigidbody with below attributes:

- Position at x = 0, y = 500, r = 0
- Velocity of x = 10, y = 0, r = 10

- Shown
- Debug

We can see the exact position of the rigidbody by logging its position value to the console. The debug console available in web browsers and is opened by pressing F12, useful for logging any variables or stages in the program.

Each direction of movement and velocity is self contained, so there is no interference between the directions.

The debug attribute will activate the debugRenderer, which will show the x,y velocity as vectors from the centre of the rigidbody.

After clicking on the “step forward” button, we should expect the rigidbody position to move by the amount of velocity/TPS, because velocity is stored as metres/second and degrees/second, and they need to be converted to metres/tick and degrees/tick.

Every tick the expected value subtracted from the y velocity is: 10/TPS, 10 is a rough approximation of the acceleration due to gravity.

JS snippet(main.js)

```
const TPS = 100;        //ticks per second
```

JS snippet(main.js)

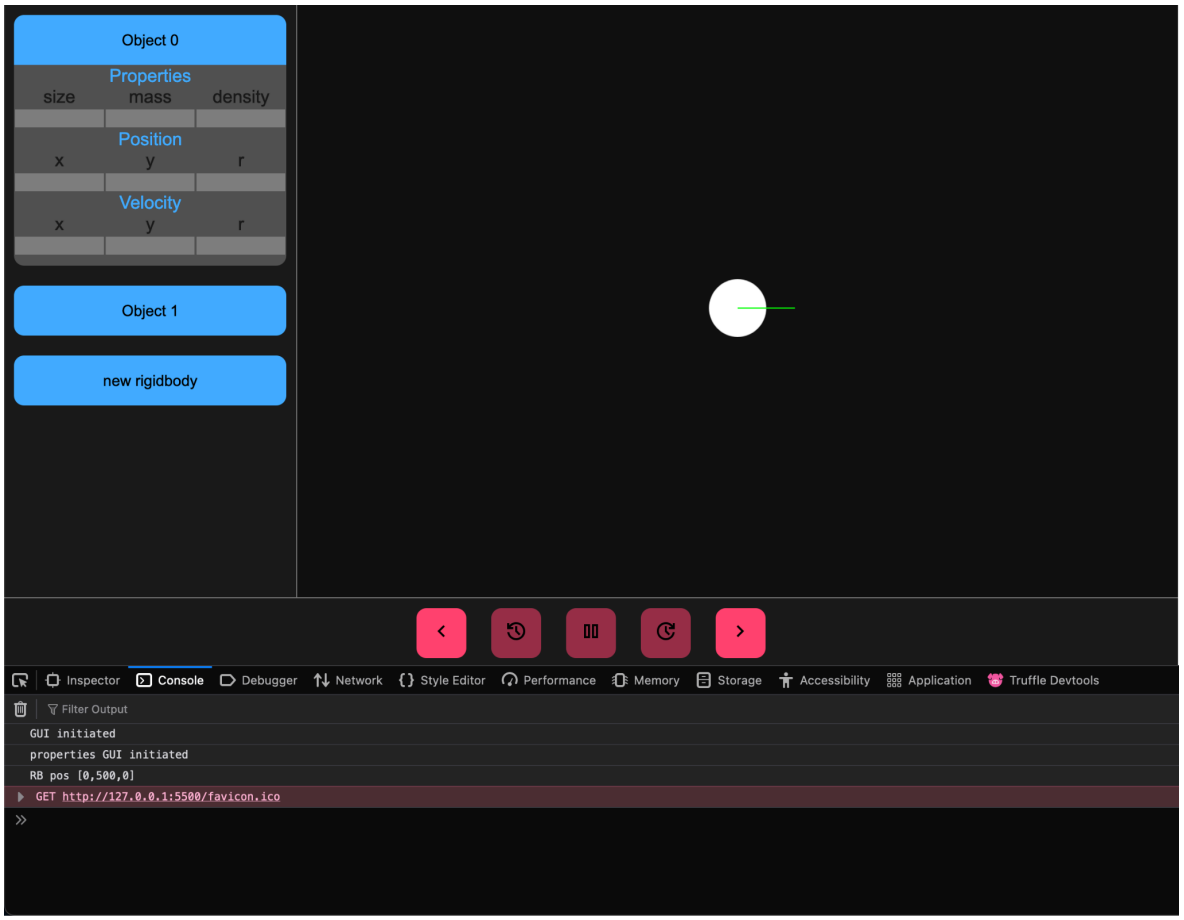
```
RBs[0].setHidden = false;
RBs[0].setPos = [0, 500, 0];
RBs[0].setVelo = [10, 0, 10];
RBs[0].setDebug = true;
```

JS snippet(tick.js)

```
function tick(funcRenderer, intTPS, boolReverse, RB) {
   //everything that needs to happen in 1 tick
   //console.log("tick");

   if (!RB.getHidden) {
       if (!boolReverse) {
           normal(intTPS, RB);
       } else {
           reverse(intTPS, RB);
       }

       console.log("POS"+RB.getPos);
       console.log("VELO"+RB.getVelo);

       funcRenderer(RB);
   }
   //console.log("tock");
```

```
}
```

Initial state of the program



1st click on “step forward” button	2nd click on “step forward” button
POS10,499.9,10 VELO10,-0.1,10	POS20,499.7,20 VELO10,-0.2,10
3rd click on “step forward” button	**4th click on “step forward” button**
POS30,499.4,30 VELO10,-0.30000000000000004,10	POS40,499,40 VELO10,-0.4,10

For the velocity of rotation and x axis, the values are not being divided by TPS (100), therefore the position of the rigidbody is changing too fast.

We can see from above when “step forward” is clicked for a 3rd time, there is a floating point error introduced in the velocity y value, but by the 4th click it is negated somehow.

We can try to retrace the ticks by clicking “step backward” a few times.

1st click on “step backward” button	2nd click on “step backward” button

```
POS30,499.4,30
VELO10,-0.30000000000000004,10
```

```
POS20,499.7,20
VELO10,-0.20000000000000004,10
```

We can see the floating point error persists

And the velocity and position cannot be perfectly retraced, on top of the position changing to much.

Adjusting the code

We can quickly adjust the `move()` function to fix this issue, by adding the `/ intTPS`

JS snippet(main.js)

```
move(intTPS, boolReverse) {
        for (let i = 0; i < 3; i++) {
            this.#pos[i] += this.#velo[i] / intTPS * (boolReverse ? -1 : 1);
//add velo to pos
        }
        this.#pos[2] %= 360; //reset rotation over 360deg
        //console.log(this.#pos);
    }
```

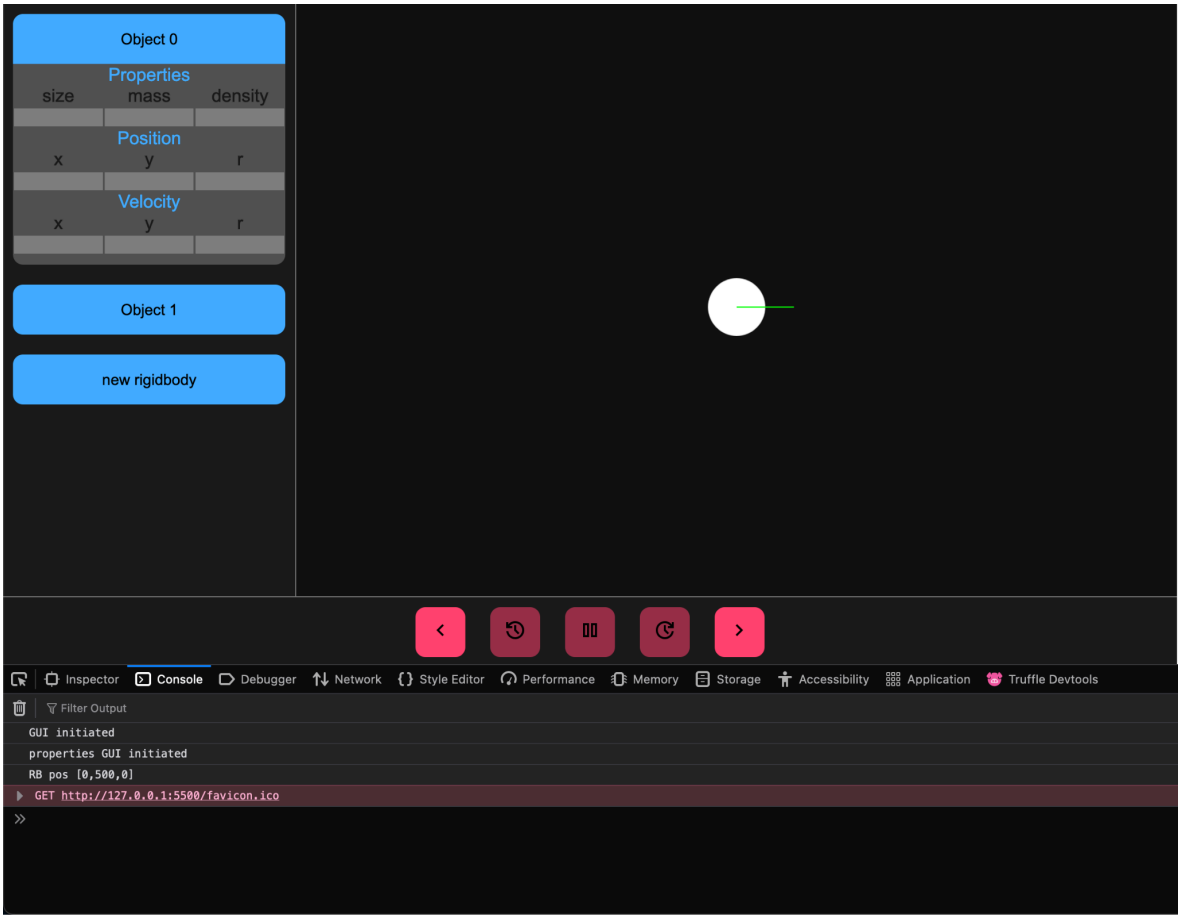
JS snippet(tick.js)

```
function normal(intTPS, RB) {
    //normal forward
    RB.gravity(intTPS);
    RB.groundCheck();
    RB.move(intTPS);
}

function reverse(intTPS, RB) {
    //reverse backward
    console.log("reverse tick");
    RB.move(intTPS,true);
    RB.groundCheck();
    RB.gravity(intTPS, true);
}
```

Repeating the test

Testing using the same starting parameters and instructions.

Initial state of the program



1st click on “step forward” button	2nd click on “step forward” button
POS0.1,499.999,0.1 VELO10,-0.1,10	POS0.2,499.997,0.2 VELO10,-0.2,10
3rd click on “step forward” button	**4th click on “step forward” button**
POS0.30000000000000004,499.994,0.30000000000000004 VELO10,-0.30000000000000004,10	POS0.4,499.99,0.4 VELO10,-0.4,10

The position and velocity are now the same as expected.

At the 3rd click there is a floating point error again, values such as 0.3 cannot be accurately represented in floating point, there is no easy fix for me to do.

We can try to retrace the ticks by clicking “step backward” a few times.

1st click on “step backward” button	2nd click on “step backward” button
reverse tick POS0.30000000000000004,499.994,0.30000000000000004 VELO10,-0.30000000000000004,10	reverse tick POS0.20000000000000004,499.997,0.20000000000000004 VELO10,-0.20000000000000004,10

One floating point error can pollute all the downstream calculations in the simulation, it is impossible to return to the exact state of the 2nd click on “step forward”

The accuracy of the groundCheck() function:

Setting a rigidbody with below attributes:

- Position at x = 0, y = 500, r = 0
- No velocity
- Shown
- Debug

We can see the exact position of the rigidbody by logging its position value to the console. The debug console available in web browsers and is opened by pressing F12, useful for logging any variables or stages in the program.

The debug attribute will activate the debugRenderer, which will show the x,y velocity as vectors from the centre of the rigidbody. This will help us identify the highest point of the bounce, where the velocity would be close to 0.

After clicking on the “play forward” button, we should expect the rigidbody to bounce and return to its original point, repeating this indefinitely.

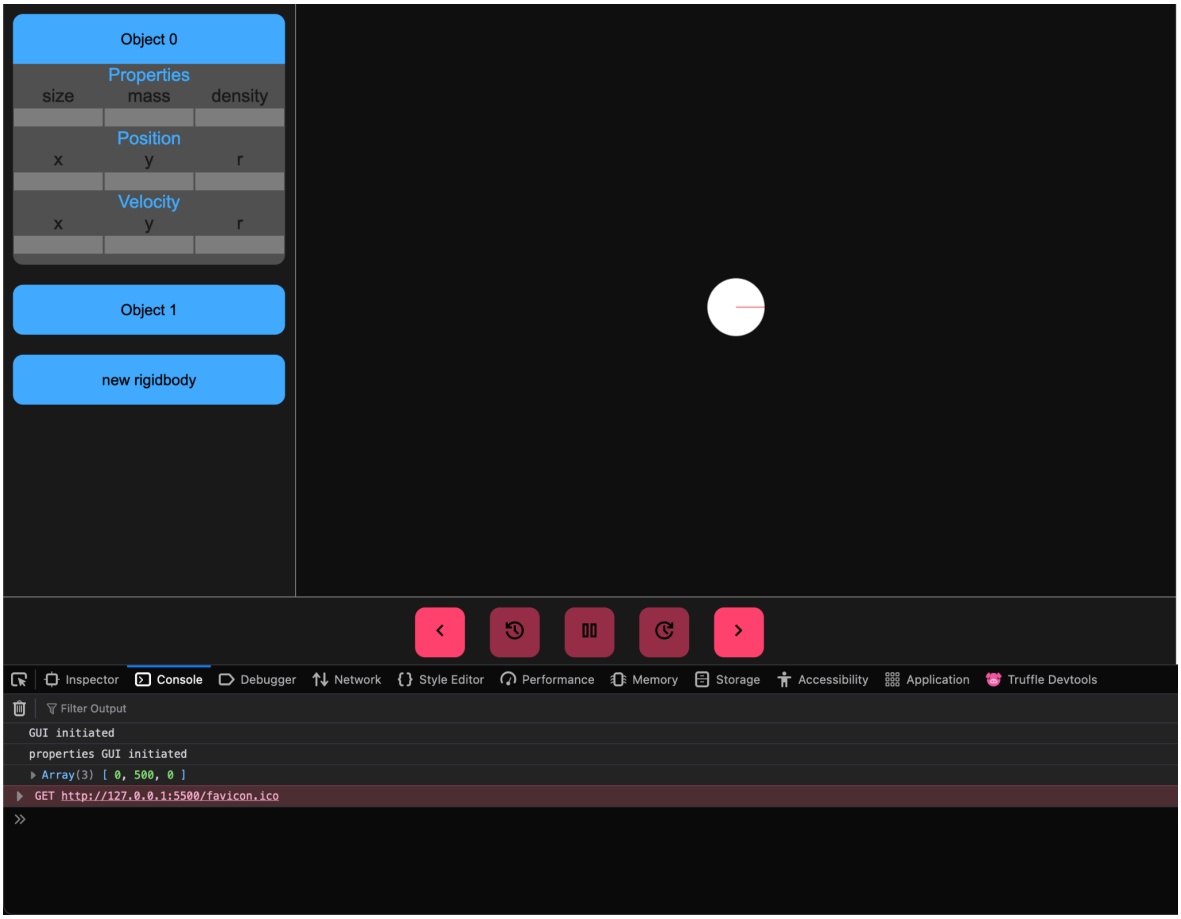
JS snippet (main.js)

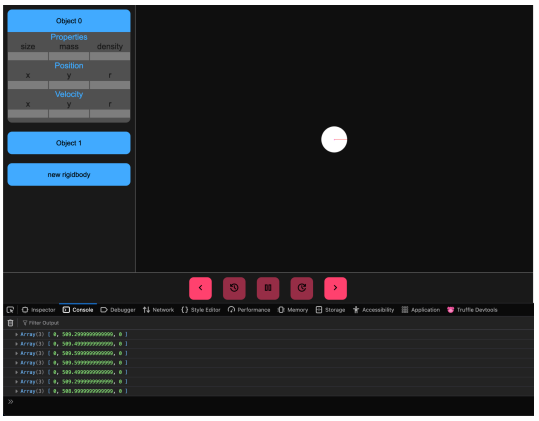
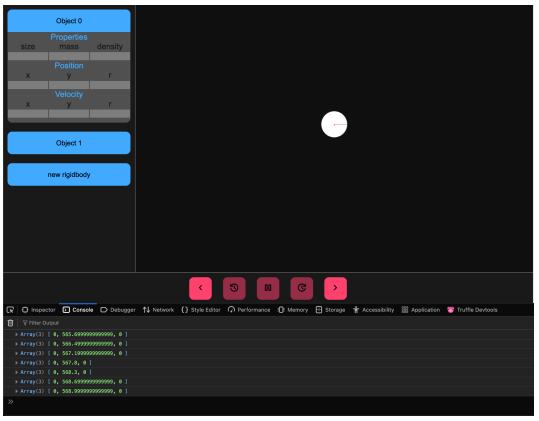
```
RBs[0].setHidden = false;
RBs[0].setPos = [0, 500, 0];
RBs[0].setVelo = [0, 0, 0];
RBs[0].setDebug = true;
```

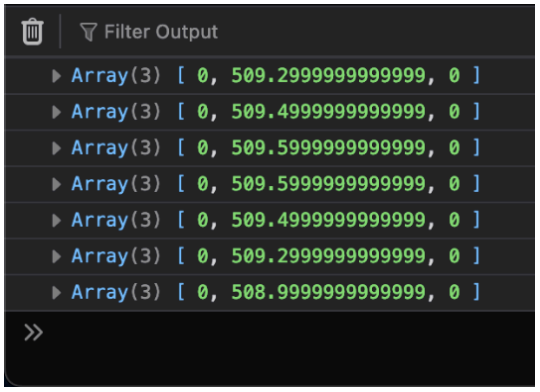
JS snippet (tick.js)

```
function tick(funcRenderer, intTPS, boolReverse, RB) {
    //everything that needs to happen in 1 tick
    //console.log("tick");

    if (!RB.getHidden) {
        if (!boolReverse) {
            normal(intTPS, RB);
        } else {
            reverse(intTPS, RB);
        }

        console.log(RB.getPos);
        //console.log(RB.getVelo);

        funcRenderer(RB);
    }
    //console.log("tock");
}
```

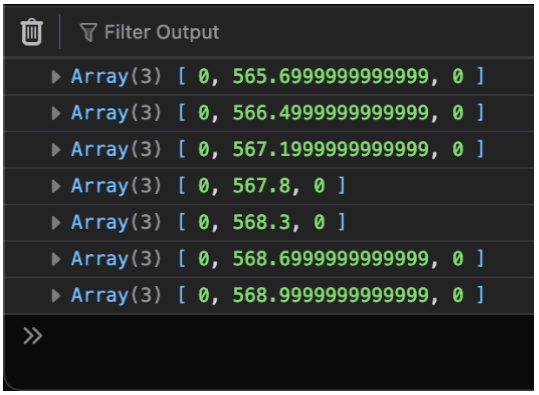
Initial state of the program



After 1 bounce	After multiple bounces
	
Maximum y value	Maximum y value

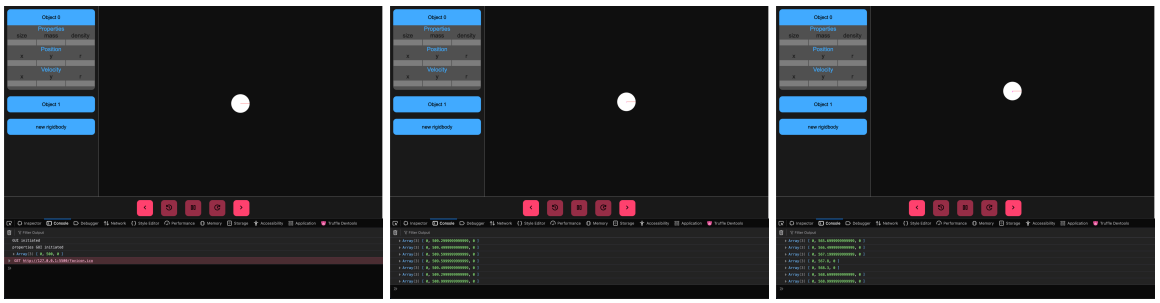


~509.6



~569

A side by side comparison

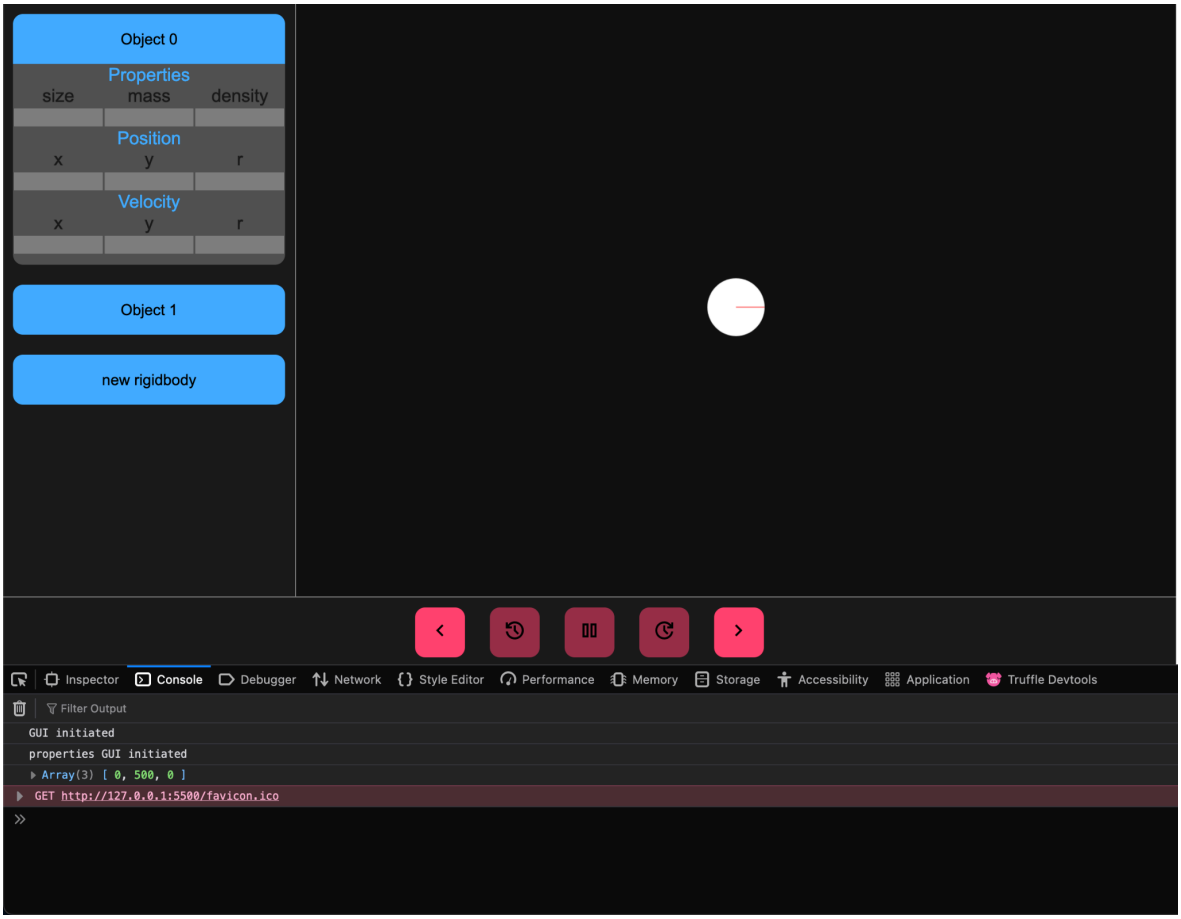


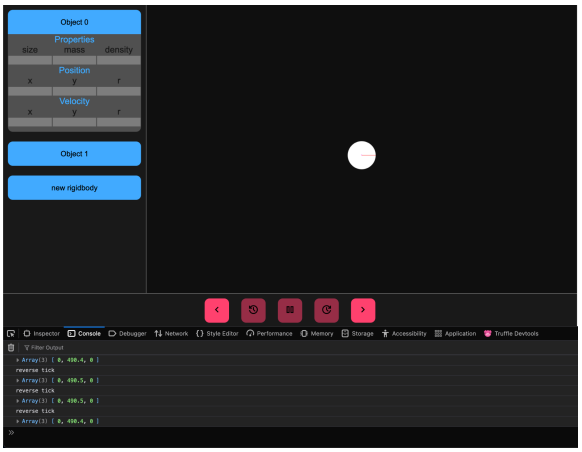
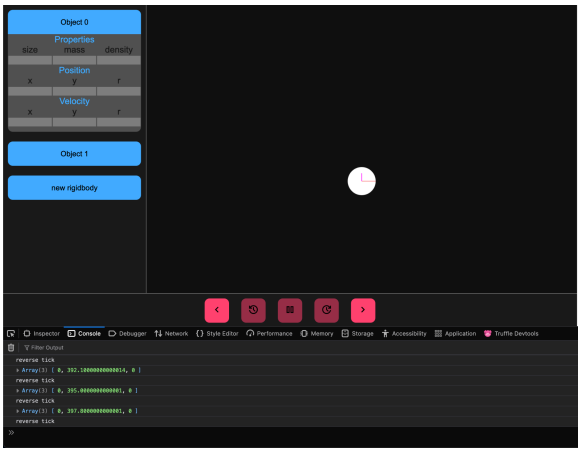
We can see the rigid body position gains height, in the side by side comparison, and by comparing the maximum height achieved after bounces(~509.6,~569).

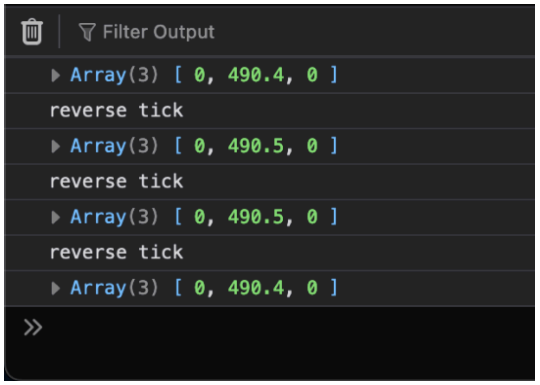
The **rigidbody slowly gains height** from each bounce, this does not accurately reflect the real world, where kinetic energy is lost, or theoretical elastic collisions, where no kinetic energy is lost.

We can change the test by clicking on the “play backwards” button to observe the same parameters but in reverse.

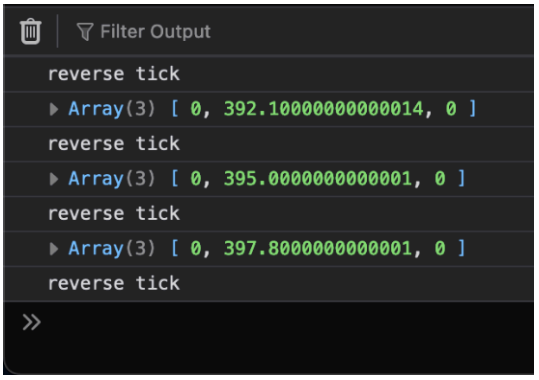
Initial state of the program



After 1 bounce	After multiple bounces
	
Maximum y value	Maximum y value

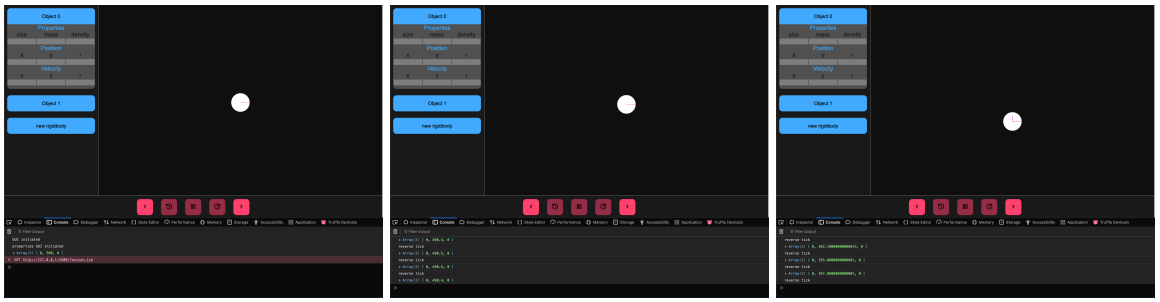


490.5



~397.8

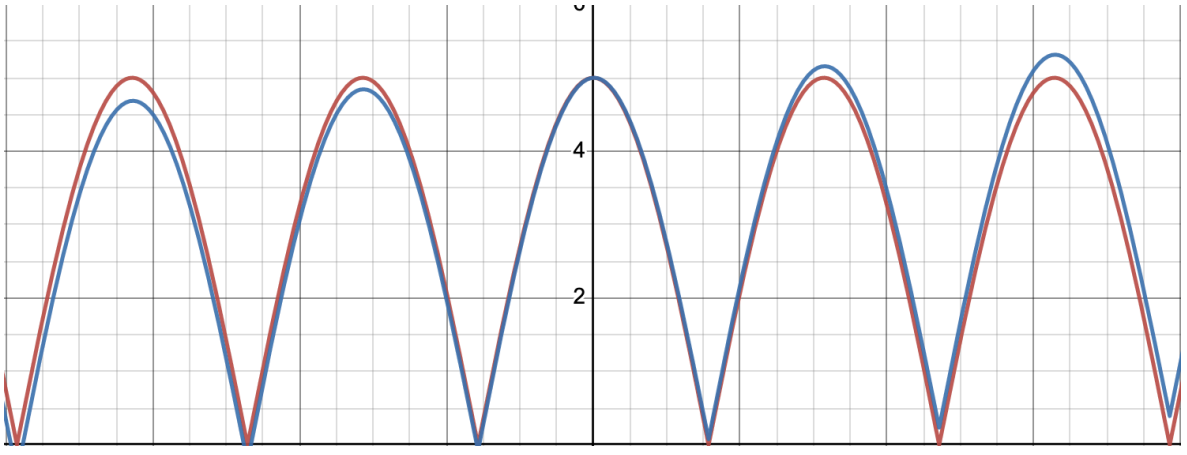
A side by side comparison



We can see the rigid body position lowers in height, in the side by side comparison, and by comparing the maximum height achieved after bounces(490.5,~397.8).

The **rigidbody slowly reduces height** from each bounce, this does not accurately reflect the real world, where kinetic energy is lost, or theoretical elastic collisions, where no kinetic energy is lost.

A graphical representation



In the graph above the horizontal axis represents time(t), and the vertical represents the y position of the rigidbody(y/100).

The blue graph is a representation of the path of the rigidbody in the simulation.

The red graph is a representation of the expected path of the rigidbody in the simulation,

with elastic collisions.

Starting in the centre at t = 0, the y value of the **expected path is identical to the actual path**. If we go towards the **negative t**, the value of y decreases, and is **lower than the expected path**. If we go towards **positive t**, the value of y increases, and is **higher than the expected path**.

Major flaws of the program:

The simulation:
I was unable to negate the floating point error in the simulation, this is important because the program needs to be perfectly deterministic to be able to move both forwards and backwards in time, while perfectly repeating the values of each state of a tick.
The `gravity()` function is inaccurate due to using a very rough approximation of **g** (the acceleration from gravity), using $10m/s^2$, instead of a more accurate approximation such as: $9.780m/s^2$.
The `groundCheck()` function is inaccurate due to being only a simple check if the bottom edge of a rigidbody is below the position y=0, this functions accuracy is highly dependant on the amount of ticks per second(TPS), with a higher TPS greatly reducing the error, with a major trade off of performance. This inaccuracy causes the bounce on the ground to result in extra upwards velocity, further causing the result bounce **height to be higher than original**, implying there is energy gained from the bounce.

Evaluation

Stakeholder statements:

Below are my two stakeholders statements, observations and feedback, both are physics and further maths A level students. Therefore their input and feedback on a physics simulator is trustworthy and valuable to me.

Jack Ben-Dyke:

Jack has knowledge of this project, but hasn't been following the development of it, he is therefore comparable to a beta tester.
“It (the GUI / program) looks good but the controls don't work, it (the rigidbody) looks like its bounced higher than normal, and i cant pause it.” after i reloaded the program to reset it for him, “its pretty cool that you can make the simulation go back (in reverse) though, yeah, the bounce is broken i think.”
He didn't need much guidance to figure out the controls, although he was quick to notice the lack of functionality of a lot of inputs.

Kevin Tu:

Kevin has been more up to date with the development of this project, he is therefore more comparable to a black box or alpha tester, he may have slight awareness of internal workings of the project.
He was already aware that many aspects of this project are unfinished, such as: the inputs for properties and the lack or support for collisions between multiple rigidbodies.
After clicking on the “play forward” button “it looks pretty decent“ then clicking on “pause” button “Why can’t it pause?”
He also independently discovered the bug that allowed the simulation to be sped up by clicking on the “play forward” button multiple times, and the bug that slows the simulation by clicking on the “play backward” button.
“Obviously its not finished, but i think its pretty good as a basic tool, its just not finished enough to learners”

Evaluation of success criteria:

Main Criteria:

(Key: reason for pass, reason for fail)
The UI is easy to navigate:

- ☑ The GUI isn't distracing from the main simulation.
  The GUI is minimal, taking up only a small proportion of the window, and fits comfortably in a variety of window resolutions.
- ☐ The GUI and its function is clear without any tutorial, students should be able to use the inputs just by already understanding basic physics and its mechanics.
  The various controls and inputs are clearly labelled, however the controls are only partially implemented, so most inputs do not have any function to be used.

- ☐ The representation of values is intuitive, the values in the properties panel are easily understandable and relatable to physics.
  No values are shown or made clear externally in the GUI.
- ☑ The timeline is easy to understand and manipulate.
  Timeline uses clear icons that reflect their function.

Time manipulation and its related functions are implemented:

- ☐ A play/pause button implemented.
  Pause button is nonfunctioning, the program needs to be reloaded to end or stop the simulation.
- ☑ A play speed multiplier is implemented.
  The program is able to play the simulation at different speeds.
- ☑ Including negative multipliers(playing in reverse).
  The program is able to play the simulation in reverse.
- ☐ Click and drag function on the playhead(similar to youtube).
  No timeline was implemented, due the way the program extrapolates the simulation, there is not set length the simulation runs for.

The simulation is accurate:

- ☐ The simulation does not deviate from real life physics in any significant way, gravity, velocity, and other required calculations are accurate and consistent.
  The simulation is consistent, however it is not accurate, for example: the simulation consistently causes rigidbodies to gain velocity when bouncing off the ground.
- ☐ The water and its related physics are successfully implemented in an accurate and consistent manner.
  No water or fluid physics was implemented.

Optional Criteria:

Implementing various display modes:

- ☑ Display modes such as force vectors can be toggled on/off
  The “debug” display mode is present and can be activated by setting the “debug” attribute of a rigidbody to “true”, this can be achieved by adding the below line of code: `RB.setDebug = true;`

More physics variables or aspects implemented:

- ☐ Account for drag
  No drag calculations are performed in the simulation.

Allow the user to change various constants:

- ☐ Change the gravitational acceleration constant
  No variable to represent the gravitational constant implemented.
- ☐ Change the drag coefficient for fluids
  No drag calculations are performed in the simulation.
- ☐ Change density of liquid
  No water or fluid physics was implemented.

Closing thoughts:

I feel there are many things i could’ve improved or done better, for one i did not judge the time available to me during the design and conceptualisation phases of this project.

However, i am happy with the magnitude of project io decided to take on, as it was a really enjoyable challenge to learn the skills required to accomplish my goals. Overall i think this project was a great examination of a practical application of my computer science skills.

Resources/Appendix

Resources and tools used in this project:
All code (and most changes) is available at my personal github: https://github.com/Theooutthedore/PhysicsSimulator
Flowcharts and diagrams created with: https://app.diagrams.net/
“Material symbols” Icon library by google: https://fonts.google.com/icons?selected=Material+Symbols+Outlined:history:FILL@0;wght@400;GRAD@0;opsz@24
Graphes created using desmos: https://www.desmos.com/calculator
MDN webdocs for JS documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript

Code from this project:
It was recommended i paste all my code in this document.
index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <title>NEA project</title>
   <link rel="stylesheet" href="style.css">



</head>
<body>

   <div class="container">

       <div id="propperties">

           <div class="expandable">
```

```
            <button  class="unexpanded">Object 0</button>
            <button class="expanded">Object 0

            </button>
            <div class="dd">
                <button class="remove">remove this rigidbody</button>
                <div class="vcat">
                    <div class="category">Properties</div>
                    <div class="hthree">
                        <div>size<input></div>
                        <div>mass<input></div>
                        <div>density<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Position</div>
                    <div class="hthree">
                        <div>x<input></div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Velocity</div>
                    <div class="hthree">
                        <div>x<input> </div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                    <!-- <div class="htwo">
                        <div>Magitude<input></div>
                        <div>Direction<input></div>
                    </div> -->
                </div>

            </div>
        </div>

        <div class="expandable">
            <button class="unexpanded">Object 1</button>
            <button class="expanded">Object 1</button>
            <div class="dd">
```

```
                <button class="remove">remove this rigidbody</button>

                <div class="vcat">
                    <div class="category">Properties</div>
                    <div class="hthree">
                        <div>size<input></div>
                        <div>mass<input></div>
                        <div>density<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Position</div>
                    <div class="hthree">
                        <div>x<input></div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Velocity</div>
                    <div class="hthree">
                        <div>x<input> </div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                    <!-- <div class="htwo">
                                    <div>Magitude<input></div>
                                    <div>Direction<input></div>
                                </div> -->
                </div>

            </div>
        </div>

        <div class="expandable">
            <button class="unexpanded">Object 2</button>
            <button class="expanded">Object 2</button>
            <div class="dd">
                <button class="remove">remove this rigidbody</button>

                <div class="vcat">
                    <div class="category">Properties</div>
```

```
                    <div class="hthree">
                        <div>size<input></div>
                        <div>mass<input></div>
                        <div>density<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Position</div>
                    <div class="hthree">
                        <div>x<input></div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Velocity</div>
                    <div class="hthree">
                        <div>x<input> </div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                    <!-- <div class="htwo">
                                                <div>Magitude<input></div>
                                                <div>Direction<input></div>
                                            </div> -->
                </div>

            </div>
        </div>

        <div class="expandable">
            <button class="unexpanded">Object 3</button>
            <button class="expanded">Object 3</button>
            <div class="dd">
                <button class="remove">remove this rigidbody</button>

                <div class="vcat">
                    <div class="category">Properties</div>
                    <div class="hthree">
                        <div>size<input></div>
                        <div>mass<input></div>
                        <div>density<input></div>
```

```
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Position</div>
                    <div class="hthree">
                        <div>x<input></div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                </div>

                <div class="vcat">
                    <div class="category">Velocity</div>
                    <div class="hthree">
                        <div>x<input> </div>
                        <div>y<input></div>
                        <div>r<input></div>
                    </div>
                    <!-- <div class="htwo">
                                                <div>Magitude<input></div>
                                                <div>Direction<input></div>
                                            </div> -->
                </div>

            </div>
        </div>

        <div id="newRBc">
            <button id="newrigidbody">new rigidbody</button>
        </div>

    </div>

    <div id="simulation">
        <canvas id="canvas1"></canvas>
    </div>

    <div id="timeline">
        <div id="timecontrol">
            <button id="stepb" class="pb"><img
src="./Assets/chevron_left_FILL0_wght400_GRAD0_opsz24.svg"></button>
            <button id="playb" class="pb"><img
src="./Assets/history_FILL0_wght400_GRAD0_opsz24.svg"></button>
```

```
                <button id="pause" class="pb"><img
src="./Assets/pause_FILL0_wght400_GRAD0_opsz24.svg"></button>
                <button id="playf" class="pb"><img
src="./Assets/update_FILL0_wght400_GRAD0_opsz24.svg"></button>
                <button id="stepf" class="pb"><img
src="./Assets/chevron_right_FILL0_wght400_GRAD0_opsz24.svg"></button>
            </div>
        </div>
    </div>

    <script type="module" src="main.js"></script>
</body>

</html>
```

style.css

```
* {
    margin: 0px;
    padding: 0px;
    box-sizing: border-box;
    font-family: arial;
}

.container {
    padding: 0px;
    max-width: 100vw;
    max-height: 100vh;
    aspect-ratio: 16/9;
    /* min-width: 1280px; */
    /* fallback resolution */
    min-width: 960px;

    display: grid;
    grid-template-columns: 1fr 3fr;
    grid-template-rows: 8fr 1fr;
}

/*
========================================
properties input
========================================
*/
div {
    text-align: center;
```

```
    color: #1b1b1b;
}

#propperties {
    grid-column: 1;
    grid-row: 1;
    border-right: 1px solid grey;
    background-color: #1b1b1b;
    display: flex;
    flex-direction: column;
    justify-content: flex-start;
}

#newRBc {
    padding: 10px;
}

.expandable {
    padding: 10px;
}

.unexpanded {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px;
    background-color: #42adff;

}

.unexpanded:active {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px;
    background-color: #507fc5;

}

.expanded {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px 10px 0 0;
```

```
    background-color: #42adff;

    display: none;
}

.expanded:active {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px 10px 0 0;
    background-color: #507fc5;
}

.dd {
    background-color: #505050;
    border-radius: 0 0 10px 10px;
    padding-bottom: 10px;
    /* display: grid;
    grid-template-columns: 5fr;
    grid-template-rows: 9fr; */

    display: none;
}

#newrigidbody {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px;
    background-color: #42adff;
}

#newrigidbody:active {
    border: none;
    width: 100%;
    height: 50px;
    border-radius: 10px;
    background-color: #507fc5;
}

.remove:hover {
    border: none;
    width: 100%;
    height: 25px;
```

```
    background-color: red;
    color: white;
}

.remove {
    border: none;
    width: 100%;
    height: 25px;
    background-color: darkred;
    color: white;
}

.remove:active {
    border: none;
    width: 100%;
    height: 25px;
    background-color: darkred;
    color: white;
}

.category {
    color: #42adff;
}

.vcat {
    display: flex;
    flex-direction: column;
}

.htwo {
    display: grid;
    grid-template-columns: repeat(2, 1fr);
}

.hthree {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
}

.vthree {
    display: grid;
    grid-template-rows: repeat(3, 1fr);
}
```

```
input {
   width: 100%;
   border: 1px solid #505050;
   background-color: gray;
}

.controls {
   background-color: #505050;
   border-left: 10px solid #42adff;
   border-radius: 10px;
}

/*
========================================
simulation disp
========================================
*/

#simulation {
   grid-column: 2;
   grid-row: 1;
   background-color: greenyellow;
}

#canvas1 {
   background-color: #0f0f0f;
   position: inherit;
   height: 100%;
   width: 100%;
   /* aspect-ratio: 12/8; */
}

/*
========================================
time inputs
========================================
*/

#timeline {
   grid-column: 1/3;
   grid-row: 2;
   border-top: 1px solid gray;
   background-color: #1b1b1b;
}
```

```
#timecontrol {
   padding: 10px;
   display: grid;
   grid-template-columns: repeat(5, 50px);
   gap: 25px;
   justify-content: center;
}

.pb {
   background-color: #ff426e;
   height: 50px;
   width: 50px;
   border: none;
   /* padding: 10px; */
   border-radius: 10px;
}

.pb:active {
   background-color: #963149;
   height: 50px;
   width: 50px;
   border: none;
   /* padding: 10px; */
   border-radius: 10px;
}

#pause,#playf,#playb {
   background-color: #963149;
}
```

main.js

```
//GUI or interface
import propGUIinitiate from "./modules/propertiesGUI.js";
import newRigidbodyGUI from "./modules/newrigidbodyGUI.js";
import removeRigidbodyGUI from "./modules/removerigidbodyGUI.js";
import timeGUI from "./modules/timeGUI.js";
console.log("GUI initiated");
//Renderer
import canvas from "./modules/canvas.js";
import renderer from "./modules/shapesRenderer.js";
//Physics
import * as t from "./modules/tick.js";
//import clock from "./modules/clock.js";
```

```
propGUIinitiate();
newRigidbodyGUI();
removeRigidbodyGUI();
canvas();

const TPS = 100;         //ticks per second
var intTickScale = 1;    //speed multiplier

class rigidBody {
   #pos;    //position
   #velo;   //velocity
   #size;   //radius
   #mass;   //mass       //unused
   #shape; //shape
   #hidden;//is it hidden
   #id;     //id         //unused
   #debug;

   constructor(/*intPos, intVelo, intSize, strShape, boolHidden*/) {
       //defaults
       this.#pos = [0, 500, 0];
       this.#velo = [0, 0, 0];
       this.#size = 50;
       this.#shape = "circle";
       this.#hidden = true;
       this.#debug = false;
       /*
       this.#pos = intPos;
       this.#velo = intVelo;
       this.#size = intSize;
       //this.#mass = intMass;
       //this.density = mass/size;
       //force = velo*mass
       this.#shape = strShape;
       this.#hidden = boolHidden;
       */
   }

   // methods
   gravity(intTPS, boolReverse) {
       this.#velo[1] -= 10 / intTPS * (boolReverse ? -1 : 1);
       //9.8m/s (10m/s^2)
       //TPS = ticks/second
```

```
        //(9.8/TPS)m/tick
    }

    move(intTPS, boolReverse) {
        for (let i = 0; i < 3; i++) {
            this.#pos[i] += this.#velo[i] / intTPS * (boolReverse ? -1 : 1);
//add velo to pos
        }
        this.#pos[2] %= 360; //reset rotation over 360deg
        //console.log(this.#pos);
    }

    groundCheck() {
        //TODO: fix this rubbish
        //there is technically inaccuracies here, the higher TPS the lower the
error
        if (this.#pos[1] - this.#size <= 0) {
            console.log("ground contact");
            //add back overshoot into ground
            //this.#pos[1] += this.#pos[1] - this.#size;
            //bounce and reverse y velo
            this.#velo[1] *= -1;
            //this.#velo[1] -= 10;
        }
    }

    //gets
    get getPos() {
        return this.#pos;
    }
    get getVelo() {
        return this.#velo;
    }
    get getSize() {
        return this.#size;
    }
    get getHidden() {
        return this.#hidden;
    }
    get getShape() {
        return this.#shape;
    }
    get getMass() {
        return this.#mass;
```

```
    }
    get getHidden() {
        return this.#hidden;
    }
    get getId() {
        return this.#id;
    }
    get getDebug() {
        return this.#debug;
    }

    //sets
    set setPos(intPos) {
        this.#pos = intPos;
    }
    set setVelo(intVelo) {
        this.#velo = intVelo;
    }
    set setSize(intSize) {
        this.#size = intSize;
    }
    set setShape(strShape) {
        this.#shape = strShape;
    }
    set setHidden(boolHidden) {
        this.#hidden = boolHidden;
    }
    set setDebug(boolDebug) {
        this.#debug = boolDebug;
    }

}

let RBs = [
    new rigidBody(),
    new rigidBody(),
    new rigidBody(),
    new rigidBody(),
];

RBs[0].setHidden = false;
RBs[0].setPos = [0, 500, 0];
RBs[0].setVelo = [10, 0, 10];
RBs[0].setDebug = true;
```

```
renderer(RBs[0]);

console.log("RB pos [" + RBs[0].getPos + "]");

var interval = null;
timeGUI(interval, t.clock, t.tick, renderer, TPS, intTickScale, RBs[0]);
```

expandableGUI.js

```
function expandable(intIndex, boolDoExpand) {
   var dd = document.getElementsByClassName("dd");
   var expanded = document.getElementsByClassName("expanded");
   var unexpanded = document.getElementsByClassName("unexpanded");

   //collapses everything
   for (let i = 0; i < dd.length; i++) {
       expanded[i].style.display = "none";
       unexpanded[i].style.display = "block";
       dd[i].style.display = "none";
   }

   //clicked dd expands if needed
   if (boolDoExpand) {
       expanded[intIndex].style.display = "block";
       unexpanded[intIndex].style.display = "none";
       dd[intIndex].style.display = "block";
   }
}

export default expandable;
```

newRigidbody.js

```
function newRigidbodyGUI(){
   const newRB = document.getElementById("newrigidbody");
   newRB.addEventListener("click", function () { onclick() });
}

function onclick(){
   var expandable = document.getElementsByClassName("expandable");
   var newRBc = document.getElementById("newRBc");
   var dispRB = false;

   //show 1st avalable rigidbody
   for (let i=0;i<4;i++){
```

```
        if (expandable[i].style.display == "none"){
            expandable[i].style.display = "block" ;
            break;
        }
    }

    //are max rigid bodies shown
    for(let i=0;i<4;i++){
        if(expandable[i].style.display == "none") {
            dispRB = true;
        }
    }
    //hide button if max rigidbodies shown
    if(!dispRB){
        newRBc.style.display = "none"
    }
}

export default newRigidbodyGUI;
```

propertiesGUI.js

```
function expandCollapse(intIndex, boolDoExpand) {
    var dd = document.getElementsByClassName("dd");
    var expanded = document.getElementsByClassName("expanded");
    var unexpanded = document.getElementsByClassName("unexpanded");

    //collapses everything
    for (let i = 0; i < dd.length; i++) {
        expanded[i].style.display = "none";
        unexpanded[i].style.display = "block";
        dd[i].style.display = "none";
    }

    //clicked dd expands if needed
    if (boolDoExpand) {
        expanded[intIndex].style.display = "block";
        unexpanded[intIndex].style.display = "none";
        dd[intIndex].style.display = "block";
    }
}

function propGUIinitiate() {
    //hides 2 rigidbodies on load
    const expandable = document.getElementsByClassName("expandable");
```

```
    expandable[2].style.display = "none";
    expandable[3].style.display = "none";

    //opens first dd on load
    expandCollapse(0, true);

    const expanded = document.getElementsByClassName("expanded");
    const unexpanded = document.getElementsByClassName("unexpanded");

    for (let i = 0; i < expanded.length; i++) {
        expanded[i].addEventListener("click", function () { expandCollapse(i,
false) });
        unexpanded[i].addEventListener("click", function () { expandCollapse(i,
true) });
    }
    console.log("properties GUI initiated");
}


export default propGUIinitiate;
//export {expandable};
```

removeRigidbody.js

```
function removeRigidbodyGUI() {
    const removeRB = document.getElementsByClassName("remove");

    //disallow deleting first object for now
    removeRB[0].style.display = "none";

    //adding event listeners for all remove rigidbody buttons
    for (let i = 0; i < removeRB.length; i++) {
        removeRB[i].addEventListener("click", function () { onclick(i) });
    }
}

function onclick(intIndex) {
    var expandable = document.getElementsByClassName("expandable");
    var newRBc = document.getElementById("newRBc");

    expandable[intIndex].style.display = "none";
    newRBc.style.display = "block";
}
```

```
export default removeRigidbodyGUI;
```

shapesRenderer.js

```
function renderer(RB) {

    //reset canvas
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    //canvas w:h = 12:8
    const base = 128;           //base unit can be scaled if needed later
    const cwidth = base * 12;
    const cheight = base * 8;

    canvas.width = cwidth;
    canvas.height = cheight;

    ctx.translate(cwidth / 2, cheight);

    if (RB.getShape === "circle") {
        circle(RB);
    } else if (RB.getShape === "square") {
        //render square
    }//more shapes later

    if (RB.getDebug) {
        renderDebug(RB);
    }
}
function renderDebug(RB) {
    //debug renderer
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    var px = Math.round(RB.getPos[0]),
        py = -Math.round(RB.getPos[1]),
        vx = Math.round(RB.getVelo[0]),
        vy = -Math.round(RB.getVelo[1]);

    ctx.beginPath();
    ctx.strokeStyle = "lime";
    ctx.lineWidth = 2;
    ctx.moveTo(px, py);
    ctx.lineTo(px + vx, py);
```

```
    ctx.stroke();

    ctx.beginPath();
    ctx.strokeStyle = "magenta";
    ctx.lineWidth = 2;
    ctx.moveTo(px, py);
    ctx.lineTo(px, py + vy);
    ctx.stroke();

    //console.log([vx, vy]);
}

function circle(RB) {
    //circle and line to indicate rotation angle
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    var x = Math.round(RB.getPos[0]),
        y = -Math.round(RB.getPos[1]),
        r = Math.round(RB.getPos[2]),
        size = RB.getSize;

    ctx.fillStyle = "white";
    ctx.arc(x, y, size, 0, 2 * Math.PI);
    ctx.fill();

    ctx.beginPath();
    ctx.strokeStyle = "red";
    ctx.moveTo(x, y);
    //trig to draw line at angle
    ctx.lineTo(x + size * Math.cos(r * Math.PI / 180), y + size * Math.sin(r *
Math.PI / 180));
    ctx.stroke();
}

//TODO finish square
function square(intPos, intSize) {
    //square stuff, has identicle phases every 90deg
    var canvas = document.getElementById("canvas1");
    var ctx = canvas.getContext("2d");

    intPos[1] *= -1;//flip y axis

    if (intPos[2] % 90 === 0) {
```

```
        ctx.fillStyle = "white";
        ctx.fillRect(intPos[0] - intSize, intPos[1] - intSize, intSize * 2,
intSize * 2);
    } else {
        //draw diamond shape when rotation angle != n*90
    }
}
//other shapes later
export default renderer;
```

tick.js

```
function clock(interval ,funcRenderer, intTPS, intTickScale, RB) {
    var intMSPT = 1000 / intTPS / intTickScale,     //milliseconds per tick
        intRTPS = intTPS * intTickScale,            //real ticks per second
        boolReverse = intTickScale < 0,
        boolPause = (intTickScale == 0);

    if (boolPause) {
        console.log("pls pause");
        clearInterval(interval);
    } else if (!interval) {
        interval = setInterval(tick, intMSPT, funcRenderer, intTPS, boolReverse,
RB);
    }
}

function tick(funcRenderer, intTPS, boolReverse, RB) {
    //everything that needs to happen in 1 tick
    //console.log("tick");

    if (!RB.getHidden) {
        if (!boolReverse) {
            normal(intTPS, RB);
        } else {
            reverse(intTPS, RB);
        }

        //console.log("POS"+RB.getPos);
        //console.log("VELO"+RB.getVelo);

        funcRenderer(RB);
    }
    //console.log("tock");
}
```

```
function normal(intTPS, RB) {
   //normal forward
   RB.gravity(intTPS);
   RB.groundCheck();
   RB.move(intTPS);
}

function reverse(intTPS, RB) {
   //reverse backward
   console.log("reverse tick");
   RB.move(intTPS,true);
   RB.groundCheck();
   RB.gravity(intTPS, true);
}
/*
function gravity(RB, intTPS, intTickScale) {
   RB.velo[1] -= 10 / intTPS * intTickScale;
   //9.8m/s = m/tick
   return RB;
}

function move(RB, intTPS, intTickScale) {
   for (let i = 0; i < 3; i++) {
       RB.pos[i] += RB.velo[i] / intTPS * intTickScale;
   }
   return RB;
}
*/
export { tick, clock };
```

timeGUI.js

```
function timeGUI(interval, funcClock, funcTick, funcRenderer, intTPS,
intTickScale, RB) {

   stepForward(funcTick, funcRenderer, intTPS, RB);
   stepBackward(funcTick, funcRenderer, intTPS, RB);

   playForward(interval, funcClock, funcRenderer, intTPS, intTickScale, RB);
   playBackward(interval, funcClock, funcRenderer, intTPS, -intTickScale, RB);

   pause(interval, funcClock, funcRenderer, intTPS, RB);
}
```

```
function stepForward(funcOnclick, funcRenderer, intTPS, RB) {
   var stepf = document.getElementById("stepf");

   stepf.addEventListener("click", function () {
       funcOnclick(funcRenderer, intTPS, false, RB);
   });
}

function stepBackward(funcOnclick, funcRenderer, intTPS, RB) {
   var stepb = document.getElementById("stepb");
   stepb.addEventListener("click", function () {
       funcOnclick(funcRenderer, intTPS, true, RB);
   });
}

function playForward(interval, funcOnclick, funcRenderer, intTPS, intTickScale,
RB) {
   var playf = document.getElementById("playf");
   playf.addEventListener("click", function () {
       funcOnclick(interval, funcRenderer, intTPS, intTickScale, RB);
       console.log("playforwards");
   });
}

function playBackward(interval, funcOnclick, funcRenderer, intTPS, intTickScale,
RB) {
   var playb = document.getElementById("playb");
   playb.addEventListener("click", function () {
       funcOnclick(interval, funcRenderer, intTPS, intTickScale, RB);
       console.log("playbackwards");
   });
}

function pause(interval, funcOnclick, funcRenderer, intTPS, RB) {
   var pause = document.getElementById("pause");
   pause.addEventListener("click", function () {
       funcOnclick(interval, funcRenderer, intTPS, 0, RB);
       console.log("pasue");
   });

}
export default timeGUI;
```