**Text

Description automatically generated with low confidence**

**Generic 2D Physics Engine in Javascript**

A dissertation submitted in partial fulfilment of

The requirement for the degree of

MASTER OF SCIENCE in Software Development

In

The Queen’s University of Belfast

By

Jack Kyle

9th of September, 2022

**Declaration of Academic Integrity**

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project   
Handbook

2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)

3. Does not exceed the specified page limit

4. Is clearly presented and proof-read

5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

6. Software and files are submitted via Canvas.

**I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism –**[**http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiaris**](http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiaris)**m/ - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used. I am aware of the disciplinary consequences of failing to abide and follow the School and Queen’s University Regulations on Plagiarism.**

**Name: (BLOCK CAPITALS)  
Student Number:**  
*Student’s signature*  *Date of submission*

**Acknowledgements**

I would like to thank my dissertation supervisor, Dr David Cutting, for his patience with the development of my project, as well as his feedback and advice. The development process would not have run as smoothly without the weekly meetings with him.

I would also like to give thanks to my lecturers for the last year of study, Dr Aidan McGowan, Dr Neil Anderson, Dr Leo Galway, Dr Sanmi Olade and Mr John Busch, for teaching me the valuable skills required to complete this project.

My parents and partner also deserve thanks, for their constant support and belief in me, throughout this entire year and especially since June for this project, it has helped to keep me motivated.

**Abstract**

Insert abstract near end probs

Contents

[Introduction 1](#_Toc112443995)

[Chapter 1: Understanding the Problem 2](#_Toc112443996)

[Chapter 2: User interface design 7](#_Toc112443997)

[Chapter 3: Architecture design and algorithm explanation 10](#_Toc112443998)

[Chapter 4: Implementation 11](#_Toc112443999)

[Chapter 5: Testing 12](#_Toc112444000)

[Chapter 6: Evaluation and Conclusion 13](#_Toc112444001)

[Appendix A](#_Toc112444002)

[Tables of figures A](#_Toc112444003)

[Test Cases B](#_Toc112444004)

[User documentation Z](#_Toc112444005)

[Introduction BB](#_Toc112444006)

[Types of object CC](#_Toc112444007)

[Using the physics engine GG](#_Toc112444008)

[Code blocks KK](#_Toc112444009)

# Introduction

In this dissertation the problem of a 2D physics engine in JavaScript will be

# Chapter 1: Understanding the Problem

The problem presented in this project was one that was rather open ended, as there was no specific level of complexity given for the two-dimensional (“2D”) physics engine that s to be created in JavaScript, thus the level of complexity was decided as the project progressed, according to time constraints, due to this project being undertaken over a period of three months, and the complexity of the mathematical equations required for the simulation of 2D physics.

It was decided that the physics engine library would be one that could be used in the design of simple 2D games and simulations, as shown in the demos of an obstacle course that the user must navigate and a game where the user must round up other balls by pushing them. This choice of design was made with the consideration of the aforementioned time constraints, as well as to allow for the creation of fun and interesting demos to display it.

The target audience for this physics library was decided to be programmers who wish to make simple games or simulations for a website, but do not wish to go to the effort of creating a physics engine of their own, perhaps due to limits of mathematical ability or time constraints of their own when making a website. This project would then allow them to begin creating their own games or simulations using the physics engine written for this project, as they need only create objects within a separate file that references the library in order to create games or simulations of their own, as the physics engine will then take care of the background work when running these games. This makes it a good opportunity for programmers to make games and simulations for websites. This library does, however, limit the programmers to simpler games, that do not reflect reality, as the mathematics behind it is not truly accurate, and is merely designed to emulate effects seen in reality. It was decided the physics engine would not truly emulate reality as the mathematics required behind this would be very complex, and with complex calculations the physics engine would be slower due to more processing time required for running through calculations before refreshing the frame. This would cause the frames per second to drop, causing for a less enjoyable experience for the player of the game.

JavaScript was already the required language for the project, the title being “Generic 2D Physics Engine in JavaScript”, however this was also a well justified choice, as JavaScript is a language that is easy to use in terms of animation, just requiring the use of “requestAnimationFrame()” with a function that moves these objects around and animating this movement each frame, and trying to match these frames to the monitor of the user’s computer’s refresh rate [1]. This, combined with JavaScript’s ability to work with mathematical problems, having comparison and logical operators, made it an excellent choice for a physics engine.

JavaScript can also support object-oriented programming. This meant that a class could be made for each type of object that was desired, holding their characteristics meaning that making objects of each type was easier than making each object and performing different validity checks (e.g. checking that the radius of a circle is zero or greater) each time, actions that were covered by a constructor in a class. It also meant that interaction between objects could be assigned between classes of objects rather than between each individual object, saving complication in the coding (e.g. if an object of class ball collided with an object of class block, it would undergo a collision, changing the velocity of the ball, rather than needing to check what would happen between each object when they were not assigned to a class).

JavaScript is also easy to work with in regards to HTML as it can be run as a script with HTML [2]. This made running code written in JavaScript on a web browser easy, as the code could be applied to an HTML canvas with only a few lines of code required (see Code block 2: index.html, Code block 3: engineConfig.js and Code block 4: physics.js for how the code links the animations of JavaScript to the canvas from HTML).

Another language that could have been used was C++. This language also features object-oriented programming, meaning that classes for each object type could have also been created in this language, same as in JavaScript. C++ also features the ability to work with graphical user interfaces toolkits, meaning it could have been used to animate games like JavaScript does with the physics engine. Both of these features make C++ a suitable language for making a 2D physics engine, however JavaScript was the one chosen for the project.

The requirements for the project also developed as the physics engine was produced due to a lack of initial information. As each feature was decided upon for the physics engine, the requirements list grew, adding requirements that would ensure that the features would work as intended.

These conditions and choices led to the software library produced. These features were decided as the physics engine was developed, some being because they are basics of such a system like the velocity of objects, as well as collisions. Other features were added because they can be used in order to create extra challenges and ideas for games such as the teleporters. Features of the physics engine include:

* Moving balls that can be controlled by the player, have a velocity in X and Y directions with a unit of pixels/loop and an acceleration in X and Y directions with a unit of pixels/loop2
* Canvas is used as grid of cartesian coordinates for drawing, “universal coordinates” also in cartesian coordinate format also used for the position of each object relative to ball controlled by user
* 2D elastic collisions with balls and flat edges
* Magnetic properties for the balls and points in the “background” (i.e. there will be no collision with these magnetic points)
* Squares with friction values also in the “background”
* Squares in the “background” that will “teleport” balls (i.e. the ball will be moved instantly to a point specified in the creation of this square)
* Canvases that will contain all of the objects created by the user with specified height and width and friction for when balls aren’t on top of other friction squares
* Ability for a ball to be controlled by a user using the arrow keys of the keyboard and the other objects will move around on the canvas around the player ball so as to simulate a camera moving centered on the player ball
* If the programmer using the physics engine does not wish for the user to have involvement in the simulation and so no ball is designated as a “player ball”, a “camera ball” will be created and this “camera ball” will also be controlled by the user so that it is possible to view all features included in the program as desired by moving the “camera” to the specific point that the user desires to see.
* Balls controlled by the player, camera or not, will be followed by maintaining it at the centre of the screen and having all objects move around it via two separate coordinate systems, one on the pixels of the screen, one that is “in universe” that will hold the information on where all objects are in relation to the player/camera ball

It was with this list of features that the list of requirements was made:

* The programmer shall import objects desired and the initiate function into their own JavaScript file
* The programmer shall create objects
* The programmer shall program in a canvas with their own specifications
* The programmer shall have the ability to program in balls with their own specifications
* The programmer shall have the ability to program in blocks with their own specifications
* The programmer shall have the ability to program in friction zones with their own specifications
* The programmer shall have the ability to program in magnets with their own specifications
* The programmer shall have the ability to program in teleporters with their own specifications
* The programmer shall end with the initiate function to have the physics engine functions start
* The program shall draw out the objects to the specification of the programmer
* The program shall not draw out objects if they are not on the screen
* The user shall be able to control the ball or camera with the arrow keys
* The program shall emulate the movement of the balls with a velocity of pixels/loop and an acceleration of pixels/loop2
* The program shall resolve collisions between two balls by first removing any penetration to have the balls simply touching
* The program shall emulate 2D elastic collisions between two moving balls
* The program shall emulate 2D elastic collisions between two balls, one moving and one unmoving
* The program shall resolve collisions between a ball and the edge of the canvas by first removing the penetration to have the ball simply touching the edge of the canvas
* The program shall emulate 2D elastic collisions between a ball and the edge of the canvas
* The program shall resolve collisions between a ball and the edge of block by first removing the penetration to have the ball simply touching the edge of the block
* The program shall emulate 2D elastic collisions between a ball and the edge of a rectangular block
* The program shall apply friction values as decided by the programmer for friction zones
* The program shall apply friction values as decided by the programmer for the canvas
* The program shall move a ball when it crosses into a teleporter to the coordinates specified by the programmer
* The program shall hold any player ball in the centre of the screen
* The program shall move objects around the player ball when the player ball is moved to keep focus on the player ball at all times
* The program shall have two systems of coordinates, canvas and universal, to allow for placement of objects to be stored while also allowing for accurate placement drawing
* The program shall emulate magnetic effects between balls when applied by the programmer
* The program shall emulate magnetic effects between magnets and balls when applied by the programmer
* The program shall allow for the camera ball to pass through all objects in order to not disturb what the programmer wishes to watch
* The program shall not allow for multiple player balls
* The program shall not allow for multiple canvases
* The program shall not have balls that have the ghost property as true interact with other objects other than the edges of the canvas

With the features listed above it was then possible for the two demos to be made, one being an obstacle course where a ball is controlled by the player. This game features the use of the player controlled ball, as the player must pilot the ball around the obstacle course to the end. It also features the teleporters to set the player back in the course, magnets to change the player’s velocity towards teleporters and other balls that are stationary to act as barriers and also have magnetism to push the player towards a teleporter. Blocks are also used in the game to act as barriers as well to create narrow paths the player must traverse. The stationary balls, blocks and edges of the canvas will then show off the 2D elastic collisions. “Friction zones” are also used to slow the player down as they get to certain points, as well as speed up the player to make the course more awkward to traverse. There is also the canvas to contain everything and provide the boundaries of the obstacle course in the game.

The other demo is a game where the player must round up other balls by using the collisions with the player ball to move them into a teleporter for them to be trapped. This also uses the player controlled ball, the teleporters, friction zones and blocks. This demo also features balls that are not stationary or controlled by the player, so the ball to ball collision may be shown as the player ball must collide with these other balls in order to move them from their locations to the teleporter. It also shows the magnetism in the balls compared to other balls, as all non-stationary balls (except the player ball) are magnetic and have poles that are “North” while the stationary balls have poles that are “South”. This means that the non-stationary balls will deflect each other, while being attracted to the stationary balls. This is used to the advantage of the player as it will help to hold the balls moved into the teleporter in place after they have been “teleported” as the non-stationary balls will be held against the stationary balls.

These demos are used to show off how these features have been used in order to meet the list of requirements above. It shows not only that each feature works individually but that the features will also work in unison, showing that the physics engine meets requirements.

# Chapter 2: User interface design

Due to the nature of the project, it is required for the user interface to be in some form of Application Programming Interface (API) when the physics engine library is being used, and because of this it shall instead be referred to as the programming interface in this dissertation. The user shall henceforth be known as the programmer as well.

The programming interface was made to be easy to use for the programmer, requiring little interaction with the physics engine, in order for the physics engine to be usable by those with less of an understanding of any mathematics behind the engine. It was then decided that, because of the required ease of programming, the programmer’s interaction with the engine was to be via JavaScript and HTML/PHP files created by the programmer, importing only necessary functions and objects from the physics engine file into the JavaScript file and creating a canvas in the HTML/PHP file. The JavaScript file made by the programmer would then be run as a script by the HTML/PHP file. Files using the physics engine were written in JavaScript as this allowed for easy interaction with the physics engine, also written in JavaScript.

The interface requires the user to first import functions and objects as stated above. This is done with this code (full code available in appendix, under Code block 5: objectCreation.js):

“import {initiate, Ball, Block, Canvas, FrictionZone, Magnet, Teleporter} from ‘./physics.js’;”

Code Snippet : code used for import of important functions and objects into JavaScript of programmer

The code in code snippet 1 is used in order to bring the important function and objects into the JavaScript that the programmer will use as, without importing them from the physics engine, it is impossible for these objects to be created outside of the physics engine and for the initiate function to be called.

After the function and objects are imported, the programmer has free choice in their object creation, however a “Canvas” object must be created, as this holds all of the other objects. After this, all other objects must be made within the bounds of the “Canvas”. An example of a “Canvas” object being made (full code available in appendix, under Code block 5: objectCreation.js):

“let canvas = new Canvas(0,0,1500,3000,0.01,”black”,”orange”);”

Code Snippet : example of code used to create new canvas object

When creating a new “Canvas” object, it has required specifications of X and Y starting points in the cartesian coordinate system with the pixels on screen, a width in pixels, a height in pixels, a friction value, an outline colour and a fill colour. The starting points entered will be the starting point in the universal coordinates. The code in code snippet 2 will create a new “Canvas” object with a start point of (0, 0), a width of 1500 pixels and a height of 3000 pixels. It will have a friction value of 0.01, a black outline and an orange fill.

Every other object will have a similar creation where they must be created as a new object of each class type, with all properties filled in. An example for each type would be (full code available in appendix, under Code block 5: objectCreation.js):

“let ball1 = new Ball (110,450,20,100,5,0,5,0,true,”South”,2,true,false,false,”black”,”red”,false);”

Code Snippet : example of code used to create new ball object

When creating a “Ball” object, the properties specified for it, in order, are the centre points, given as X and Y coordinates in a cartesian coordinate system, the radius, the mass, the velocity vector’s X and Y components (pixels/loop), the maximum velocity this “Ball” can reach by using the arrow keys to move it (pixels/loop), the acceleration scalar that is added to the velocity vector components each loop of the physics engine when an arrow key is held, the magnetic Boolean that decides if it is magnetic or not, the pole of the magnetism, the magnetism value for calculations to do with magnetic properties, the stationary Boolean that decides if it is stationary or not, the player Boolean that decides if it is player controlled or not, the ghost Boolean that decides if it will interact with other objects or not, the colour for the outline, the colour for the fill and the frictionCheck Boolean that is used to check if it has had the friction of the “Canvas” or “FrictionZone” object applied yet each loop. The centre coordinates are in universal coordinates. In the case of this “Ball” object it will have a centre of (110,450), a radius of 20 pixels, a mass of 100, a velocity vector of (5, 0), a maximum velocity of 5, an acceleration scalar of 0, a Boolean value of true for magnetic, a pole that is “South”, a magnetism value of 2, a Boolean value of true for stationary, a Boolean value of false for player, a Boolean value of false for ghost, a black outline, a red fill and a Boolean value of false for frictionCheck.

“let block1 = new Block(600,300,300,300,”black”,”silver”);”

Code Snippet : example of code used to create new block object

When creating a “Block” object, the properties specified for it are the starting points in X and Y coordinates, for the universal coordinate system, the width, the height, the colour of the outline and the colour of the fill. The starting points entered are in the universal coordinates. In the case of code snippet 4, it starts at (600, 300), has a width of 300 pixels, height of 300 pixels, black outline and a silver fill.

“let friction1 = new FrictionZone(500,350,100,100,”black”,”purple”,0.05);”

Code Snippet : example of code used to create new FrictionZone object

When creating a “FrictionZone” object, the properties specified for it are the starting points in X and Y coordinates, for the universal coordinate system, the width, the height, the colour of the outline, the colour of the fill and the friction value for the zone. The starting points entered are in the universal coordinates. In the case of code snippet 3, it starts at (500, 350), has a width of 100 pixels, height of 100 pixels, black outline, purple fill and a friction value of 0.05.

“let magnet1 = new Magnet (550,20,20,20,”South”,”black”,”white”);”

Code Snippet : example of code used to create new Magnet object

When creating a “Magnet” object, the properties for it are the centre coordinates in order of X then Y, for the universal coordinate system, the radius of the “Magnet” in pixels, a magnetism value for the strength of the “Magnet”, the pole of the “Magnet”, the colour of the outline and the colour of the fill. The starting points entered are in the universal coordinates. In the case of the “Magnet” object created in code snippet 5, it has its centre at (550, 20), a radius of 20 pixels, a magnetism value of 20, a pole that is “South”, a black outline and a white fill.

“let teleporter1 = new Teleporter (500,600,150,150,100,100,”black”,”gold”,”silver”);”

Code Snippet : example of code used to create new Teleporter object

When creating a “Teleporter” object, the properties specified for it are the starting points in X and Y coordinates, for the universal coordinate system, the width, the height, the X and Y coordinates for where the “Teleporter” will move objects, the colour of the outline, the colour of the fill and the colour of the circle in the centre. The starting points entered are in the universal coordinates. In this case, the “Teleporter” will start at (500, 600), have a width of 150 pixels, height of 150 pixels, move objects to (100, 100), have a black outline, gold fill and a silver circle in the centre.

This is how the objects are created in the programming interface as it is a function that can be easily followed for each programmer, all that is required is the knowledge of what order the properties are filled in when the object is created so as to avoid these being applied incorrectly when the object data is passed into the physics engine to be displayed. It allows for people with little knowledge of programming or physics to follow instructions to create a program of their own, giving them the possibility to create games and simulations with the physics engine without prior knowledge required outside of creating the JavaScript and HTML files and running them.

It is also important that the programmer can choose to build up the objects within the canvas as desired and so, with the programming interface used to interact with the physics engine, the programmer is able to create objects freely, allowing for a “world” to be built up from scratch in these objects and for the physics engine to be used to draw out this “world” on the HTML canvas and run it.

This ability to create using only the objects is shown in the demos, one being where an obstacle course is built that the player must maneuver a ball round and the other being a game where the player must control a ball to round up other balls. The simplicity of creating these demos is shown in their code, where it is only required to learn how to create the objects properly, and the freedom of the creation is shown as both demos feature very different “worlds” in terms of their structure. This shows that the programming interface has been designed with the ideas of ease and creativity in mind, as it is up to the programmer to create the program they desire.

A screenshot of a computer

Description automatically generated with medium confidence

Figure : example of UI with one of each type of object created

# Chapter 3: Architecture design and algorithm explanation

The architecture of the code was designed with the use of calls to different functions. This is done because it allows for better readability with the code, as well as stopping code from being repeated by calling a function rather than repeating the code that will instead be contained in the function. This makes it a good way to structure the physics engine, as it is important to ensure that a program is readable for those who wish to use it after it has been programmed, in case any additions/edits are to be made to the code. The functions being used instead of repeated code is also helpful as it means that code is not unnecessarily repeated. The use of functions is also helpful as this way it is easier to find errors when one occurs, as it can be pin pointed to a specific method and so it is a smaller amount of code that is to be worked with in order to fix the error, thus making it easier to find and fix. It also allows for the code to be split into functions which are defined by their purpose, so that each function has one distinct purpose, allowing for clarity in what each function does, making it easier for other programmers to understand the code.

The first sections written in the code are the classes for each object and vectors and their functions. These classes were designed in order to hold information on each type of object. The first class created was the vector class. This class was created in order to hold the X and Y components of different variables such as a ball’s velocity, as the movement and positioning of objects are all based on a Cartesian coordinates system, so everything must have X and Y components. This class holds a constructor that takes in the two values for the X and Y components for the vectors. This is used in the creation of vector objects to ensure the initial values are held. The vector class also has functions in it for the calculation of the magnitude, unit and dot product of vectors, called “magnitude” and “dotproduct”. These are both necessary for the resolution of a 2D elastic collision between two balls, which will be further discussed in chapter 4 of this dissertation. The “dotproduct” function will also take in a second vector in the function’s call, as the dot product is calculated between two vectors.

The next class written in the code is “Ball”, the class for the ball objects. This class was created to hold the constructor used to create objects of the ball type. This constructor takes in values for a wide variety of variables: the X and Y components for the centre starting point vector, for the universal coordinate system, the radius of the ball, the mass of the ball, the X and Y components for the starting velocity of the ball (pixels/loop), the maximum velocity of the ball (pixels/loop), the acceleration scalar of the ball, the magnetic Boolean for the ball, the pole of the magnetism for the ball, the value for the magnetism of the ball, the stationary Boolean of the ball, the player Boolean of the ball, the ghost Boolean of the ball, the outline colour for the ball, the filling colour for the ball and the friction check Boolean for the ball. This constructor is used in the creation of the ball objects, and is also used for the validation of the variables entered, correcting certain entries the programmer has made if they are outside of required ranges. The constructor will also set the variables entered into the constructor to the ball so that the variables can be called via each ball. The constructor also adds each ball to an array so that each ball can be called through this array.

The next class in the code is “FrictionZone”, the code for friction zone objects. This class was written to hold the constructor used to create objects of the friction zone type. This constructor will take in values for the X and Y components of the start point vector, for the universal coordinate system, the X and Y components for the length vector, the colour for the outline, the colour for the fill and the value for the friction. This constructor is used for the creation of the friction zone objects and will also run validation of the variables entered, checking that these variables are within the correct ranges, and correcting them to be if outside of these ranges. The constructor will also set the variables entered into the constructor to the friction zone so that the variables can be called via each friction zone. The constructor also adds each friction zone to an array so that each friction zone can be called through this array.

The class “Magnet” is next, holding the code for magnet objects. This class was written to hold the constructor that is used in the creation of objects of the magnet type and this constructor was designed to take in values for the X and Y components of the centre starting point vector, for the universal coordinate system, the radius of the magnet, the magnetism value for the magnet, the pole of the magnet, the colour of the outline and the colour of the fill. The constructor is used in the creation of magnet objects and also runs validation on some of the variables in order to ensure they are within certain ranges. It will also set the variables that are entered into the constructor to the magnet object that is being created so that the variables are able to be called via each magnet created. The constructor also adds each magnet to an array so that each magnet can be called through this array.

After the “Magnet” class is the “Block” class, holding the code for the constructor for the block objects. The constructor is used to create objects of the magnet type, taking in values for X and Y components of the start point vector, for the universal coordinate system, X and Y components for the length vector, the colour of the outline and the colour of the fill of the block. The constructor also runs a validation check on the values for some of the variables entered, ensuring that these variables that are checked are entered within a certain range as required for the variable. The constructor also sets the variables that are entered into the constructor to the block object, so that the block object created will be able to be used for calling the values required from the block. The constructor will also add the block to an array so that each block created can be called via this array.

The next class in the code is the “Canvas” class, which holds the code for the constructor for the canvas object, which is used to create objects of the canvas type. This constructor will take values for the X and Y components of the start point vector, for the universal coordinate system, X and Y components for the length of the canvas, the friction value for the canvas, the colour for the outline and the colour for the fill. The constructor will set these values entered to variables in the canvas object that is created, allowing for these values to be called upon via the canvas created, before adding the canvas to an array so that the canvas may be called through this array when used elsewhere in the code. The constructor will also validate the values entered for some of the variables, in order to ensure they are within the required range for each.

The final class in the code is the “Teleporter” class, which holds the code for the constructor of the teleporter object, as well as the function “drawTeleporter”. The constructor is used to take in values for the variables of the teleporter before assigning them to the teleporter object being created, allowing these values to be called via the object and its variables. These variables are the X and Y components of the starting point vector of the teleporter, for the universal coordinate system, the X and Y components of the length vector, the X and Y components for the vector for where the teleporter moves balls to, the colour for the outline, the colour for the fill and the colour for the circle in the middle. The constructor will also validate the values entered, ensuring these values are within the required range for the variable the value has been assigned to. The teleporter created will then be assigned to the teleporter array, allowing for the specific teleporter to be called via the array when its variables are required. The function for drawing teleporters was made due to the specific drawing requirement of the teleporter having a circle in the middle, therefore a function made only for the teleporter object was made and put in the “Teleporter” class so it can only be used with the teleporter objects. This function will draw the teleporter on the canvas, taking the start point, lengths, colour of the outline, fill and circle to draw it.

The functions come after the classes, and will be gone through in the order that they are used when running through a simulation with the physics engine, meaning that the first function that will be covered is the “initiate” function. This function is used to run initial checks on the objects that have been created by the programmer, ensuring that there is only one player ball and one canvas, as anything else will cause the program to break due to the code being built round the usage of only one of each, as the camera depends on a player ball, and more than one will cause the program to try and follow multiple in one frame, making the frames display incorrectly, and functions involving the canvas have been written with the usage of only one. The function will correct the objects created, removing any canvasses that are made if there are multiple and creating a default canvas, and removing the player status of any other balls and creating a default player ball if multiple are created. It then checks that each other objects are created within the canvas and that the teleporters will not move balls outside of the canvas, as everything must remain within the canvas, or else the code will have issues with movement and display. It will then use “requestAnimationFrame()” with the “mainLoop” function in order to start the main section of the code that will repeat. This function is incredibly important as it runs the initial checks to ensure that each object is created in the correct place and moves these objects if necessary. This was done in the initiate function rather than each constructor so that the programmer does not have to worry about the program breaking if they don’t make the canvas first in their program. The function is also important as the canvas and player ball objects are key to the running of the physics engine, so it is important to ensure that there is only one of each.

The next function called is the “mainLoop” function. This function is used to work out the location of each object relative to the player ball, resolve interactions between objects, animate each section and use “requestAnimationFrame()” with itself to constantly recall this function in a recursive manner. The function will first wipe the canvas, as this will allow for the new objects to be displayed over it in order to display each frame without the original frame being displayed underneath the new frame. After this the function will call functions for the drawing of each object in their correct position. It then cycles through each ball and will run through each function that is applicable to each ball (e.g. all functions for balls that are player controlled and don’t have the ghost property) before finally using “requestAnimationFrame()” with itself for the next cycle through this function.

The first function that is called in the “mainLoop” function is the “universalToCanvasBorder” function. This function takes in the player ball and the canvas. It is used to compare where the edges of the canvas are relative to the player ball on the universal coordinates that record the actual position of each object, and will place these edges in the canvas coordinates compared to the centre (where the player ball always is) based on how far away the canvas edges are from the ball in the universal coordinates. It then calls the “drawRectangle” function with the canvas to draw out the canvas.

The “drawRectangle” function is used in order to draw any rectangular shape, and will take in any object that is to be drawn as a rectangle (friction zones, canvas, and blocks).

After the “universalToCanvasBorder” function is the “universalToCanvasFrictionZones” function. This function takes in the player ball and friction zone objects that are cycled through in a loop from the “FrictionSquareArray” in the “mainLoop” function. It will compare the location of the friction zone to the player ball in the universal coordinates and places the friction zone on the canvas coordinates relative to the centre where the player ball is using this distance calculated. If the friction zone is on the screen then the friction zone will be drawn, using the “drawRectangle” function. If it is not on screen then the friction zone will not be drawn.

The next function called after the “universalToCanvasFrictionZones” function is the “universalToCanvasTeleporters” function. This function takes in the player ball and teleporter objects that are cycled through in a loop in the “mainLoop” function. It will compare the location of the teleporter to the player ball in the universal coordinates and then place it on the canvas coordinates relative to the centre of the screen from this, in order to display the teleporter compared to the player ball which is in the centre. If the teleporter is not on the screen then the object will not be drawn. If it is on the screen, the “drawTeleporter” function from the Teleporter class is called. The function will also check the teleport coordinates universal coordinates against the player ball, adjusting them as appropriate to the canvas coordinates as well.

The next function to be called is the “universalToCanvasMagnets” function which will take in the player ball and the magnet objects that are cycled through in a loop in the “mainLoop” function from the “MagnetArray” where all magnet objects are held. It will compare the location of the magnet to the player ball and place it on the canvas coordinates relative to the player ball, using the distance calculated from the distance in the universal coordinates. If this would place the magnet on screen then the “drawCircle” function shall be called with the magnet, and if it isn’t placed on screen then the function won’t be called.

The “drawCircle” function is used in order to draw any circular object, taking in any object that is to be drawn as a circle (balls, magnets).

The next function to be called after the “universalToCanvasMagnets” function is the “universalToCanvasBlocks” function, which will take in the player ball and the blocks that are cycled through in the “mainLoop” function from the “BlockArray”. This function will compare the position of the block to the player ball in the universal coordinates and set the coordinates of the block relative to the player ball at this distance calculated. If the block is on screen then it will be drawn with the “drawRectangle” function, and if it isn’t then it won’t be drawn.

The last universal to canvas coordinate function to be called is the “universalToCanvasBalls” function that is used to compare all non-player balls to the player ball. The balls are cycled through from the “BallArray” where it will be used in the call of the “universalToCanvasBalls” function along with the player ball. This function will compare the position of the non-player ball to the player ball in universal coordinates and place it on the canvas coordinates at the distance calculated from the player ball. If the non-player ball is on screen then the “drawCircle” function will be called, and if the non-player ball is off screen then the “drawCircle” function won’t be called. This means that the non-player ball will only be drawn when on screen.

The last drawing function to be called is the “drawCircle” function with the player ball to draw it in the centre of the screen.

After this the next function to be called is the “playerControl” function, which will take in only the player ball. This function is used to control the movement of the player ball, checking if the user has pressed an arrow key and setting the acceleration vector as appropriate with the key pressed and the acceleration scalar of the player ball. It also checks if the user has released the key and removes the acceleration vector’s value if so. This function is designed in order to allow the player to have control over the player ball.

The next function is the “checkTeleport” function, which will take place in a loop iterating through each teleporter in the “TeleportArray”, allowing the ball to be checked against each teleporter individually in the function. This function will check if this ball is on top of the teleporter, and if so then move the ball to the teleport points specified in the teleport point vector of the teleporter. This function does not emulate reality in that it features teleportation, something not possible. It does require the centre of the ball to be on the square of the teleporter, in order to emulate the idea that the ball is a 2D representation of a 3D sphere drawn from top down view, meaning that it is only the centre of the ball that touches the “ground” (canvas) at any point, so it is only when this point touches the teleporter that the effect will take place. The teleporter works as soon as the ball touches the teleporter and will move the ball to the location specified without any change in velocity to show that the ball will continue moving in the same direction from this point. This function also comes before the other functions featuring the other objects (other balls, blocks, friction zones, edges) so that the ball’s velocity changes from interactions with these other objects are considered from the new point, again to show that the movement of the ball by teleportation is instant, meaning the interactions with other objects such as friction zone’s will be from its new position.

The next group of functions all take in the ball that is the current one in the loop that is iterating through the “BallArray” and will also take in any balls after this one in the array one by one as well to compare each to the current one. The next function is the “checkCollisions” function. This function will compare the two balls that are entered into the function, and will check if the balls are overlapping. It will resolve any overlapping by moving the balls until they are just touching. This is done so that balls are not inside each other when the collision between the two is resolved so as to better emulate reality, where the balls would not penetrate each other when colliding.

After the “checkCollisions” function is the “ballVectors” function. This comes after the penetration is resolved as this function will resolve the velocities of the balls to ensure that the velocities are correct as of the collision occurring, and so this must be resolved when the balls are just touching and not when there is overlap between the balls. This function will take in the same two balls as the previous function, due to it resolving the collision in terms of the vectors of the velocities. This function will take in these balls and it will perform the maths required for a 2D elastic collision, and set the vectors of the balls’ velocities as necessary afterwards. This is done in order to emulate reality in terms of the collision. This calculation takes into consideration the direction of each ball’s velocities, the place of each ball when colliding and the mass of each ball, to show off how they all affect the resultant velocities. The velocities calculated are added to the balls’ vectors but not added to the actual movement of the balls as this will be done later after each other objects is taken into consideration for the balls.

After this is the final function for ball to ball interaction takes place, the “checkMagneticBalls” function. This function will only be called if both balls have the magnetic property. This function will take in these two balls, same as the last two methods, and it will call the “magnetismCalc” function with the two balls, before then taking the velocity calculated from “magnetismCalc” and making it specifically applicable to the situation of two balls, whether that be two moving balls, or one moving ball and one stationary ball. This method is designed in order to show off magnetic effects between two balls and show how each will affect each other, depending on the pole, magnetism and mass of each ball, in an attempt to emulate reality and how two magnetic balls would interact on a flat surface. The final velocities calculated with mass etc. taken into consideration will be added to the velocity vectors of the balls, again not adding to the position of the ball so that all effects can be taken into consideration. There is also a minimum magnetic velocity that needs to be met in order for the magnetic velocity to be added to the velocities of the balls, again to reflect reality where a negligible magnetic force will not move an object.

The “magnetismCalc” function called in the “checkMagneticBalls” function is used in order to calculate the initial magnetism calculation that is applicable to both two magnetic balls and one magnetic ball and a magnet object, as it only takes the distance and the magnetism values into consideration. This function is used in order to not have this section of the magnetism calculation repeated, avoiding unnecessary repetition of code, thus making it easier to fix as it doesn’t require the code to be fixed multiple times if an error occurs.

After the final ball to ball function comes the “functionGroup” function if the ball is not stationary as these calculations only affect balls that do not have the stationary property, due to all of the functions requiring the ball to move to have these functions take place or affecting the velocity of the ball. This function will take in the ball that is the current iteration of the aforementioned loop. This function’s entire purpose is to hold more functions that are all specifically to do with the ball’s interaction with other objects that aren’t other balls. It is also used to avoid code repetition as these functions are all applicable to the final ball in the array, which needs a section of its own in order to avoid errors from the loop that iterates through each ball and all balls ahead of it in the array, as there are no balls ahead of this ball since it is the last one. Thus that ball will have its own section of statements in the “mainLoop” function, so the “functionGroup” function is needed to avoid the repetition of these function calls when coding, so that again if there is a change in code, it is not needed to be written multiple times, and only once in the “functionGroup” function. The functions the “functionGroup” function holds are the “checkEdges” function, “checkTeleport” function, “checkFriction” function, “checkMagnets” function, “checkBlocks” function, “addAcceleration” function and “addVelocity” function. It also had the code for ensuring the canvas friction is taken into consideration, as this must be checked with each ball that is not on a friction zone, so it is potentially applicable to all balls and so it is best to have in this function to avoid any code repetition if in the “mainLoop” function.

The first function called in the “functionGroup” function, as previously mentioned, is the “checkEdges” function, which will take in the ball going through the “functionGroup” function and the canvas. This function is designed in order to check that if the ball is touching/going through any edges of the canvas then the ball will be moved so that it is just touching the edge of the canvas instead. It will then reverse the velocity of the ball (Y component for the top and bottom of the canvas, X for the left and right of the canvas) in order to emulate the collision of the ball and edge of the canvas, to show it bouncing off the edge. Once again the collision is resolved by first moving the ball so that no penetration is occurring, and this is so that the ball is accurately moving from the edge of the canvas, to emulate reality where the ball would not penetrate the edge. It also stops a bug from occurring where the ball will constantly reverse the appropriate velocity if inside the edge as it will keep detecting the ball is inside the edge if the ball is not moved, thus causing the ball to not correctly resolve the collision. This is to be avoided as the goal is to attempt to emulate reality.

The next function is the “checkFriction” function, which takes in the ball and a friction zone object, and this function will take place in a loop iterating through each friction zone in the “FrictionSquareArray”, allowing the ball to be checked against each friction zone individually in the function. This function will also check if the ball is on top of the friction zone, again considering the ball to be a sphere viewed from a top down view, so that the ball is only considered to be on top of the friction zone when the centre is on the friction zone. This stops any complications with consideration toward rotational velocity with the ball if the ball is considered to be a flat 2D circle where different parts of the ball would have different friction values applied. When the ball is over the friction zone it will have friction applied to it, progressively lowering the values of the velocity vector the longer the ball is on the friction zone, resulting in a slower ball the longer it is on the friction zone to emulate how friction works in reality, with different possible friction values. It will also apply a check to the ball so that the friction of the canvas isn’t applied to the ball as well, in order to avoid the ball being slowed down more than it should, as if a ball is on a friction zone then it should not be considered on the canvas, and thus immune from the effects of its friction. Any values calculated are only applied to the velocity of the ball and not the position in this function, so as to allow the effects of other objects to be added to the velocity before the ball moves.

The next function to be called is the “checkMagnets” function, which will take in the ball if it has the magnetic property and a magnet object, which will come from the loop this takes place in that iterates through the magnet objects in the “MagnetArray”. This function will calculate velocities to be added to the balls at all times, requiring no contact with the ball. The “checkMagnets” function will call the “magnetismCalc” function like the “checkMagneticBalls” function did, calling it to work out an initial value dependent solely on distance and magnetism values. This function will then take the mass of the ball into consideration, which is where it differs from “checkMagneticBalls” which takes the mass of two balls into consideration, and calculate a new magnetic velocity to add to the ball’s velocity vector values. This function is again designed to reflect reality by considering the mass, and magnetism of the magnet and ball into consideration, adding a higher velocity dependent on the values and reversing the direction dependent on the poles of the objects. The function is used in order to show how a magnetic ball would interact with a flat point on the canvas that is magnetic and so the ball can roll over it, allowing for larger values to be added to the velocity than with two balls, as the ball is able to move very close to the centre of the magnet, due to the magnet not stopping the ball from moving closer as it is flat. Any values calculated are only applied to the velocity of the ball and not the position in this function, so as to allow the effects of other objects to be added to the velocity before the ball moves. There is also a minimum magnetic velocity that needs to be met in order for the magnetic velocity to be added to the velocity of the ball, so that the magnetic balls aren’t always moving from the magnetic effect, again to reflect reality where a negligible magnetic force will not move an object.

The next function is the “checkBlocks” function. This also takes place in a loop, this time iterating through the “BlockArray” in order for each block to be compared to the ball. This function will take in the ball and a block, and check for a collision between the ball and the block by checking if there is overlap and which side the ball has collided with. If there is a collision detected and there is penetration then the ball will be moved so that the ball is no longer penetrating the block and just touching the edge of the block. This is done to reflect reality as the ball would not penetrate the block and would instead deflect off it. This deflection is then shown by changing reversing the velocity of the ball, same as what is done with the edge of the canvas. The penetration resolution also occurs in order to stop a bug where the ball will move through the block or constantly reverse velocity as it is trapped in the block. This function is needed for the emulation of the interaction between a ball and a block when the ball collides with the block. It is assumed that the block is immovable due to motion only being applicable to ball objects.

The “checkBlocks” function is the last function that takes in objects other than the ball currently going through the “functionGroup” function. The next function is the “addAcceleration” function. This function is designed in order to add the acceleration vector values to the velocity vector values of the ball, if the acceleration values are over a threshold. This is to have negligible values of acceleration be ignored. The acceleration is added to the velocity once per loop for each ball, as the velocity is added to the position of the ball every loop, and so it must take place at the same rate to show that the acceleration is using the same units (in the case of the velocity this would be pixels/loop, making the acceleration pixels/loop2). It is also ensured that the acceleration doesn’t take the ball beyond its maximum velocity from acceleration, so that the ball doesn’t hit velocities above those set by the programmer, unless an external force acts upon the object such as magnetism or another ball collides with it. This is to reflect where a ball can only reach a maximum velocity under its own volition, and can only achieve a higher velocity with an external force. It also allows for easier control of player balls when a user is in control of one by the keyboard.

The last function in the “functionGroup” function and last function overall is the “addVelocity” function. This function will add the velocity of the ball to the centre of the ball, moving its centre position by that many pixels in this one loop, due to the pixels/loop unit used for velocity. This function comes last as it is adding the final velocity calculated from all of the other functions that have affected the velocity of the ball, and so it is only once the velocity has been calculated with each other factor taken into consideration that the ball’s position will change, so as to show how it would work in reality that all forces are acting at once, even if this is not quite the case in the code, since each step must be worked out in order of coding and only one step can happen at a time, meaning that there is a small delay between each velocity change being worked out, therefore the position should only be changed with this final velocity worked out.

The architecture of the code for this physics engine was designed around the idea of the forces all being worked out before the ball actually experiences any movement, and so it was structured in order to allow for the movement to come at the end of the “mainLoop” function before everything is drawn again for the next frame. It was also structured in order to have the teleporter take priority in the movement, so as to allow forces to act from new positions. It was also structured in order to avoid unnecessary repetition of code and for the ease of the programmer finding errors when they occurred by grouping sections of code into functions, allowing for the code to be more easily split up. It also has classes for each object in order to make it easier to create each object and store each with the variables required for each, as each class has a constructor to make this easier.

# Chapter 4: Implementation

The coding of this physics engine starts with a call to import four variables from the engine configuration file, these variables being “canvas”, “ctx”, “width” and “height”. These four variables all relate to the canvas (not the class, but the actual HTML canvas) and its properties. These variables are imported as each is necessary for the coding of the physics engine, using its properties to actually display any drawing on the canvas, the removal of any drawing from the canvas and for use in deciding which objects are on screen. These are all necessary functions in order for the physics engine to work as required as the physics engine is dependent on the positioning of each object and so it is important each is drawn in the correct place.

After these variables are imported, there is a number of arrays created, “BallArray”, “PlayerBallArray”, “FrictionSquareArray”, “MagnetArray”, “BlockArray”, “CanvasArray” and “TeleportArray”. These seven arrays are created for the holding of each type of object as labelled in each array’s name. These arrays are used in the calling of objects in each, with all but the player ball array and canvas array being cycled through with the usage of “forEach” loops in order to access all objects and compare all balls against every other object to check for potential interactions occurring. In the cases of the player ball array and canvas array there is only one object in each by design, therefore allowing the object in each to be called by calling the first object from the arrays. These arrays are important as otherwise the physics engine is without a way for the objects of each type to be called when needed to check for interactions occurring, so it is important that these arrays are created and have each object of the correct type added to the arrays. The arrays were used as it allows for all objects of each type to be stored in one place, and so it is then possible for the objects to all be accessed from the same place for further use.

The vector class has already been discussed in chapter 3, but the code within will be discussed in detail here, in chapter 4. The constructor will set the X and Y vectors by using:

“this.x\_vector = x\_vector” and “this.y\_vector = y\_vector”

Code Snippet : setting of x and y vectors of vector object in constructor

This sets the X and Y vectors for the object that is created to the X and Y vector values that were entered by the programmer. This means these values can then be accessed via the object created as they are now stored there. The magnitude function will return the square root of the sum of the X component of a vector squared and the Y component of a vector squared, as this is the calculation for the magnitude of a vector, as shown in Equation 1:

Equation : calculation for magnitude of a vector

Where X is the X vector value and Y is the Y vector value.

The magnitude of the vector is defined as “the length of the vector” [3] and so this is calculated in order to work out the distance between the centres of two balls later in the code when checking for a collision between two balls by taking the distance in the X direction and the distance in the Y direction between the two balls and then calculating the magnitude of that vector. The other function of the vector class is the “dotproduct” function, which is used to calculate the dot product of the vector that is being used to call it and another vector that it will be called with. “The dot product between two vectors is based on a projection of one vector onto another” [4] meaning that it is the movement of one vector calculated in the direction of another vector, and this is calculated by multiplying the X components of both vectors together and multiplying the Y components of both vectors together and adding these two results, as shown in Equation 2:

Equation : calculation for dot product of two vectors

Where a is a vector and b is another vector with a being projected onto b, n is the dimension of the two vectors (in this case, second).

This value is used in the calculation of the resultant velocity vectors after the collision of two balls. This is used in order to calculate the velocities of the balls in the normal and tangent directions during the collision. More detail will be given for the uses of the magnitude and dot product functions when discussing the “ballVectors” function.

The next class in the code is the “Ball” class, used in the creation of ball objects. The constructor sets each value entered to the ball object’s variables, same as in the constructor for the Vector class, however there is some additional code beyond setting each variable with:

“this.*variable* = *value entered*”

Code Snippet : generic code for setting variable with value entered into constructor

The first difference is when setting the centre variable of the ball, which is set with a vector object instead:

“this.centre = new Vector(xpoint,ypoint)”

Code Snippet : code for setting centre of ball using vector object

This will set the centre in universal coordinates variable of the ball object being created as a vector object, meaning it will hold two values of its own. This is done this way as the centre has two components in the form of the X and Y coordinates, therefore it will be stored as a vector. This is the same for the ball’s initial “velocity”, “accelerationVector” and “canvasCentre”. These vectors have the “velocityx” and “velocityy” variables entered for the initial velocity variable while the other two have default values entered. The acceleration vector will have (0,0) entered as default as this will be the starting acceleration before the keyboard keys are applied for player balls. The canvas centre will have (0,0) set by default if the ball is not a player ball to be later updated by the “universalToCanvasBalls” function. If the ball is a player ball then it will have the canvas coordinates set as (half the screen width, half the screen height) as this will place the player ball in the centre of the screen, as required for a player ball. These canvas coordinates are used to store where the ball is on the screen and HTML canvas while the other starting point is the position of the ball in the simulation.

The velocity and acceleration mentioned in the previous paragraph will have units of pixel/loop and pixel/loop2. This was decided upon as the position of the ball is adjusted by changing the location of the ball on the canvas (HTML canvas, not class), and the canvas is set out in a grid of coordinates with each coordinate being a specific pixel, therefore meaning the movement of the ball is in pixels, hence this being the measure of length. The measure of time is per loop, as this is how the code is built, to adjust the position once per loop, and the loop potentially being a variable measure of time. This is because “requestAnimationFrame” tries to adjust the amount of callbacks per second to the refresh rate of the monitor of the computer that, in this case, is using the physics engine [5]. Thus the time measure can vary, thus it is easier to set it as per loop.

Certain variables also have conditions for how the variable will be set. One example is the mass of the ball, which requires the value entered to be greater than zero. This requirement is put in place, as, in reality, all objects must have mass and so the value entered for mass must reflect this and so the value must be greater than zero. This is especially important to correct as, in the calculation for the resultant velocities of the collision between two balls, specifically in the final normal for the balls, there is a point where the sum of the masses of both balls is used in the division and so if both balls have a mass of zero then there will be a division by zero, which is impossible. Thus these must be corrected to avoid this occurring, as when the balls have masses of zero the objects that are being tested will be removed from the canvas, due to the arithmetic error. The code will instead set the mass to 0.00001 to give it a very small value instead, as was presumably desired by giving the ball a mass of 0. The magnetism value, used as a measure of strength for the magnetic force the object has, also has a check for ensuring it can not be below zero (i.e. negative) as this will cause magnetic objects to act in an opposite manner to what is expected from the pole of it, therefore if, for example, the programmer wants to have two object repel instead of attract, the poles can be set to the same rather than needing a negative magnetism value. This is to reflect reality where two magnets of opposite poles will not repel each other, so the magnetism values must reflect this by always being positive in order to suit the magnetic pole. Another value that is changed is the acceleration scalar. This value is used as the acceleration value that is applied to the acceleration vector in whichever direction the arrow keys are being pressed (negative of the scalar for left and up). This value is checked to be less than zero and if so then it will be multiplied by negative one. This is because when the acceleration scalar is negative, the ball will go in the opposite direction to the key pressed, and all four directions are covered by the arrow keys anyway that it is not necessary for the keys to be in reverse. It is multiplied by negative one in order to have it act with the same magnitude but in the correction direction. The radius will also have its value changed to one if it is set equal to or below zero if the ball does not have the ghost property. If it has the ghost property then it can be set to zero to allow the player to act as a camera without the ball that is being controlled obscuring the view of the player trying to observe the other objects. The radius is not allowed to be negative either way as when using the .arc method in JavaScript the radius entered must be positive. [6]

There are also if statements before each Boolean value is set. This is because Boolean values can only be true or false, so if anything else is entered then further checks involving these Boolean values will not work as expected. Thus on four of the five Boolean variables, magnetic, stationary, player and ghost, there are if statements to check if the values entered are true or false. If the values are neither of these then the values will be set to false by default before being set with the object being created. If the friction check value is anything but false, it will be set to false before being set for the object variable as the friction value must always be false as this value is used in checking if the ball has been affected by the friction value of a friction zone before it has the canvas’s friction value applied, thus it must be false so that at least one friction value is applied.

There is also an if statement to check if the ball is magnetic before setting the pole and magnetism values entered into the constructor to the object. This is because if a ball is not magnetic then it will not have a pole or a magnetism value, therefore these values will be set to “N/A” and zero respectively, as a non-magnetic will not have these characteristics.

Any variables that do not require any check or are checked later are set to the object without being checked. These variables are “xpoint”, “ypoint”, “velocityx”, “velocityy”, “maxvelocity”, “outline” and “fill”. After all variables are set in the constructor the ball created will be added to the “BallArray”, allowing for it to be called as previously explained.

The next class in the code is the “FrictionZone” class. This is used to create friction zone objects, and also only has a constructor in the class. Same as the Ball class’s constructor, the values entered will be set to the appropriate variables as shown in Code Snippet 9. There will also be variables stored as vector objects, these variables being the “upperpoint”, “canvasStart” and “length” variables, as each has an X and a Y component. These components are the “upperxpoint”, “upperypoint”, “xlength” and “ylength” values for the “upperpoint” and “length” variable vectors, and the “canvasStart” variable has (0,0) entered as default. The canvas start variable will be updated by the “universalToCanvasFrictionZones” function at the start and each loop as the player ball moves. These canvas coordinates are used to store where the friction zone is on the screen and HTML canvas while the other starting point is the position of the friction zone in the simulation.

This constructor also contains if statements to check that values entered are within required ranges, such as for the length variable’s X and Y component values, as these must be greater than zero. This is because the standard for this physics engine is that the starting point of the rectangle will be the top left of the rectangle, as a standard provides clarity for how the object will be drawn, and is also involved within the mathematics of the collision detection. The value must be greater than zero specifically because the object must have some form of length in order to be able to interact with balls, as a friction zone with X and Y lengths of zero will not be counted as beneath the ball as there is no area for the ball to be on top of to have the friction of the friction zone applied. Another value that is checked with an if statement is the friction value entered, and it is checked to see if the ball is greater than one. If the value is greater than one then it will be set to one. This is the case because if the friction value is greater than one, it will start to reverse the values of the velocity vector of the ball on it, doing so once per loop, potentially trapping the ball on the friction zone. This would happen because the friction effect on the velocity is calculated as:

Equation : velocity calculation in friction zone per loop

This means that the friction value must stay at or below 1.

The values that are entered without checks are the X and Y components of the upper point vector as these are checked later in the initiate function, and the outline and fill colours, as there is no easy way to test that the values entered are valid colours with JavaScript. The friction zone is then added to the “FrictionSquareArray” allowing for it to be called as previously explained.

The next class created is the “Magnet” class, which also only contains a constructor. This constructor is for the creation of magnet objects, and features similar elements as the ball and friction zone constructors. This constructor also features variables of the object being created as vector objects, in this case being the centre and canvas centre variables. The centre variable will hold the “xpoint” and “ypoint” values entered as the X and Y components of the vector respectively. The canvas centre variable will be set to (0,0) by default, to be adjusted later by the “universalToCanvasMagnets” function. These canvas coordinates are used to store where the magnet is on the screen and HTML canvas while the other starting point is the position of the magnet in the simulation. These are held as vectors as both have X and Y components.

The magnet class will also check to see if the radius is within a certain range, in this case greater than zero. This is done because the magnet should have a radius in order to be visible, however this does not affect the function of the magnet, only that it will not be visible when drawn out. It is set to one by default if the value is less than one. It must also not be negative for the same reason as balls can’t. The magnetism value will also be checked, ensuring that it is not less than zero. This is done for the same reason as the ball’s magnetism value being checked.

The values that won’t be checked are the “xpoint”, “ypoint”, “pole”, “outline” and “fill”. The X and Y components of the centre vector aren’t checked here as these are checked later in the initiate function. The pole, outline and fill aren’t checked as these are all text and so the programmer is trusted to enter the data correctly as they need it to be. The magnet is then added to the “MagnetArray” allowing for it to be called as previously explained.

The next class in the code is the “Block” class, which also only contains a constructor. Like previous constructors this one will also set variables of the object to vector objects. These variables are the “start”, “canvasStart” and “length” as each variable has X and Y components. The start variable is set with the “xstart” and “ystart” values as the X and Y components of the vector respectively, storing the starting point coordinates. The canvas starting point is set to (0,0) by default as this will be later updated by the “universalToCanvasBlocks” function. These canvas coordinates are used to store where the block is on the screen and HTML canvas while the other starting point is the position of the block in the simulation. The length will set the “xlength” and “ylength” values as the X and Y components respectively.

The only variables to be checked are the length variable values. These are checked to ensure that the values entered are greater than zero. This is for the same reason as the friction zone, where the block must have length in both directions in order for the ball to have interaction with the block, as the interactions are dependent on the length. Therefore the block must have length in both directions. The values are set to be positive as the block has the same standard as the friction zone where the starting point will always be the top left corner, as this will make it easier to remember when using and the top left is how the object is considered to start in collision/interaction detection code is used. If either value is less than or equal to zero then the value will be set to one as the minimum.

Variables that are set as the value entered without checking are the “xstart”, “ystart”, “outline” and “fill” variables. This is because the start values will be checked later in the initiate function, and there is no way to ensure the outline and fill entries are colours in JavaScript. The block is then added to the “BlockArray” allowing for it to be called as previously explained.

The next class is the “Canvas” class, which also only contains a constructor within its code. It also has variables of the object set to vector objects. These variables are the “universalStart”, “canvasStart” and “length” variables. The universal start vector is set with the “xstart” and “ystart” variables, the canvas start vector with (0,0) by default and the length vector with “xlength” and “ylength”. These are all set with vectors as each has X and Y components. The canvas start vector is set with (0,0) by default as this will be later updated by the “universalToCanvasBorder” function.

The constructor will also ensure that the lengths are above zero, as again this is to keep the top left corner as the beginning point for rectangle objects as standard, and also because the canvas is meant to hold objects, therefore it can not have a length of zero for it to be of any use. The friction is also set to one if it is above one for the same reason as the friction zone friction value.

The ”xstart” and “ystart” variables are set to the vector without being checked as these will be later adjusted in the initiate function if there is an issue. The “outline” and “fill” will be set without adjustment for the same reason as previous colour variables. The canvas is then added to the “CanvasArray” allowing for it to be called as previously explained.

The next, and final, class is the teleporter class, which contains the constructor and the “drawTeleporter” function. The constructor sets several variables as vectors, the “start”, “canvasStart”, “length”, “teleportPoint”, “canvasTeleportPoint”, “circleCentre” and “canvasCentre”. Each was stored as a vector as each has X and Y components. The start is set with the “xstart” and “ystart” variables, the canvasStart is set with (0,0) by default, the length is set with the “xlength” and “ylength” variables, the teleport point is set with the “xteleport” and “yteleport” variables, the canvasTeleportPoint is set with (0,0) by default, the circle centre is set with the “xstart” plus half of the “xlength” variable and “ystart” plus half of the “ylength” variable and the canvas centre is set with (0,0) by default. The canvas variables are all set with (0,0) by default as these variables are all updated by the “universalToCanvasTeleporters” function. The centre circle is set with half of the length added to the starting points as this is where the centre of the square is for the circle to be drawn from.

The constructor will ensure that the lengths are greater than zero for the same reason as the previous rectangles. The radius will be set to the shorter side divided by 5, as ensured by an if statement that checks which side is longer, so that the circle will fit inside the rectangle.

The variables that are set with the values that are entered are the X and Y components of the starting point and teleport point, the “outline”, “fill” and “circle”. The starting point components are left unchanged as these will be adjusted in the initiate function if necessary. The outline, fill and circle colours will not be changed for the same reason as the colour variables in other class constructors. The teleporter will then be added to the “TeleporterArray” allowing for it to be called as previously explained.

The “drawTeleporter” function is then used to draw both the rectangle and then the circle within it, given as a unique drawing function due to having both sections required and no outline on the circle. The function will call the “drawRectangle” function before then using the “CanvasRenderingContext2D” variable “ctx” to draw the circle using the .arc method to draw a circle with the centre points from the canvas centre vector, radius calculated and the points of zero as the starting point and having two times pi as the end point, drawing a full circle. This draws a full circle as the end angle will be the same point as the start as the unit is in radians. The .fillStyle and .fill methods are then used to fill in the colour of the circle with the colour entered in the constructor to “circle”, with .fillStyle setting the colour of the fill and .fill doing the actual filling of the circle.

There are only functions after the “Teleporter” class. The first function is the aforementioned “drawRectangle” function. This is used to take in a rectangular object and draw it on the canvas. This is done with the “ctx” variable, where it will begin the path and use the .rect method to take in the starting points of the object in the canvas coordinates, and the lengths of the object to draw it out. The .strokeStyle and .stroke method are used to set the outline colour of the rectangle and then draw this outline respectively. The .fillStyle and .fill methods are used in the same manner as the “drawTeleporter” function, albeit for a rectangle instead and with the “fill” variable of the object. The canvas coordinates are used as it is on the canvas that the objects are drawn and so it must be the canvas coordinates that are used rather than the universal coordinates as these will decide where on the screen the object will appear.

The next function is the “drawCircle” function which functions similarly to the “drawTeleporter” function after the “drawRectangle” call is made, as it also uses the .arc method with the canvas coordinates of the object. The only addition is the use of .strokeStyle to set an outline for the circular object with the “outline” variable setting the colour.

The next function is the “checkCollision” function, which takes in two balls. This function starts by calculating the distance between the two balls in the X and Y planes by taking the X and Y coordinates of ball two from ball one. These values are stored as “distance\_x” and “distance\_y”. The magnitude of these is then calculated for a new variable called “distance\_magnitude”, and this is calculated with Equation 1. It is then checked if the sum of the two balls’ radii is greater than the magnitude of the distance, as this will determine if the balls are overlapping or not, because if the two radii are greater than the distance magnitude then the balls must be overlapping as the magnitude is the total distance between the two balls. If this is the case then the difference between the distance magnitude and the sum of the two balls’ radii will be found by taking the distance magnitude from the sum:

Equation : difference in radii sum and distance magnitude

Where r1 is the radius of ball one, r2 is the radius of ball two and d is the distance magnitude.

This difference is stored as “diff” which is then used to calculate how much each ball should move, stored in X and Y directions as “move\_x” and “move\_y”, and each being calculated with:

Equation : movement of balls during penetration amount

Where dx/y is the distance in the X or Y direction, d is the distance magnitude and diff is the difference previously calculated in Equation 4.

This movement is calculated this way as this movement is the amount each ball must be moved in each direction in order to remove all penetration and just have the balls touching instead. This is calculated this way as this will calculate the unit vector in that direction, and then the difference is divided by two as each ball will only move half of the penetration distance to move equal amounts, and these are then multiplied to find the distance in that direction to be moved.

There are then if statements to check if the balls are stationary or not. If both balls aren’t stationary then the move variables calculated will be added to ball one and taken from ball two. These are taken from ball 2 as the variables are calculated from the angle of ball one so ball two must move in the opposite direction, hence taking the move variables away rather than adding them. If either ball is stationary then the move variable multiplied by two will be added or taken away (depending on which ball is being moved) as the stationary ball should not move at all, therefore the moving ball must move the entire distance of the penetration away from the stationary ball.

This function was originally part of the “ballVectors” function, however this was split into two functions, as each has one particular purpose even if they both deal with the same situation (a collision between two balls). They were also split because there was an issue with the balls not detecting collisions properly and just sliding off of each other, so this function was created to hold this code and it was called twice, once at the start of each main loop and once at the end to check if the ball was still penetrating with other balls after the resolution in case this was the cause of the improper collision resolution. In the end the issue was because the vector calculation was only happening when the balls were just touching, which the balls weren’t exactly being moved to, still penetrating slightly, causing the balls to then not deflect as the new vectors were not calculated. This was eventually found and the collision check function being left on its own is a remnant of that issue and the search for it. In the end it was decided that it would be left this way due to each function having one specific purpose anyway.

The next function is the “ballVectors” function which also takes in two balls. Due to originally having the code from the “checkCollision” function in this function, there is a repeat of code for the distance magnitude calculation and the check to see if the total of the radii of the balls is greater than or equal to the distance magnitude before performing the calculations necessary for the resolution of the collision. The reasons for this were explained in the above paragraph. After the if statement to check if the balls were colliding, the ball will go through a series of calculation that were taken from a document online, in source [7]. The first calculation is for the normal vector of the collision, i.e. the vector that runs perpendicular to the point where the collision has occurred between the balls:

Equation : normal vector of collision calculation

Where n is the normal vector, x is the X point of the centre of each ball and y is the Y point of the centre of each ball.

This was programmed in the same way the equation is written by taking the centre points of ball one from the centre points of ball two and storing these values in a new vector object for the normal. This is found by dividing both components of the vector by the vectors magnitude. This normal vector is then used in order to find the unit vector of the normal, i.e. the vector that has a magnitude of one and runs in the same direction as the normal vector. This is shown in the equation:

Equation : unit vector calculation

This unit normal vector is calculated by creating a new vector object for the storage of its values and setting each component with:

let unitNormalVector = new Vector (normalVector.x\_vector / normalVector.magnitude() , normalVector.y\_vector / normalVector.magnitude());

Code Snippet : code used for unit normal vector

This shows that the magnitude function of the vector class is used in order to calculate the magnitude without needing to type it out again. The unit tangent vector is calculated next, which is the unit vector in the direction of the tangent along the point of collision. This is equal to:

Equation : calculation for unit tangent vector

Where unit t is the unit tangent vector, uny is the Y component of the unit normal vector and unx is the X component of the unit normal vector. [7] Again the unit tangent vector is stored as a vector object.

Next to be calculated is ball one’s initial velocity vector projected in the direction of the normal, which is calculated by taking the dot product of ball one’s initial velocity with the unit normal vector. This is done with the .dotproduct function in the vector class, using Equation 2. This is repeated to find ball one’s velocity vector projected in the direction of the tangent, by taking the dot product with the unit tangent vector instead. This is repeated for ball two’s velocity in the normal and tangent vector directions.

The tangential direction of the velocity will not change post collision as there is no force in the tangential direction from the collision. [7] Therefore the final tangential velocity will be equal to the initial. The normal direction of the velocity will change and the calculation for the final velocity in the normal direction will be calculated with:

Equation : calculation for final velocity in normal direction for ball 1

Where vn,1,f is the final velocity of ball one in the normal direction, vn,1,I is the initial velocity of ball one in the normal direction, m1 is the mass of ball one, m2 is the mass of ball two and vn,2,i­ is the initial velocity of ball two in the normal direction. This calculation is the same for ball two but all values with a subscript of one must be replaced with two, and vice versa (i.e. now taking mass one from mass two, multiplying by initial normal velocity of ball two, and adding product of mass one and initial normal velocity of ball one for the top line). This is the section where mass can not be zero for both balls as this will cause a division by zero, which will cause an arithmetic error, hence the need for correction.

This calculation is done as stated in the equation in the code, and set to variables to hold the values. These values calculated are then used to find the new velocity vector in the normal direction for balls one and two by multiplying these values by the X and Y components for the unit vector in the normal direction for each ball. This calculates the new velocity in each direction for each ball. The new tangent velocity for each ball is then calculated by multiplying the unit tangent vector components of both balls by the velocity in the tangent direction of each ball. Both types of velocity are stored in vector objects as the velocities have X and Y components.

The final velocity of each component of each ball is then calculated by adding the normal and tangent velocity of each component of each ball together (i.e. the X component of the velocity in the normal direction of ball one is added to the X component of the velocity in the normal direction of ball one to give the final X velocity of ball one). These values are applied straight to each ball’s velocity variable so that the value is changed as soon as the new value is calculated. The value is only changed if the ball is not stationary, if the ball is not stationary then the value will stay the same (i.e. the values will be zero).

These calculations are entirely within an if statement to ensure that the balls are touching/penetrating, or else the calculations won’t happen since it is unnecessary if the balls are not in contact.

After this function is the “addAcceleration” function. This function takes in one ball. This function will first check that the acceleration vector’s X component is greater than 0.01 or less than -0.01, as these are values set to ignore negligible values, however since the acceleration can only be set by the keyboard arrows at this level of development, this is not currently applicable in the code unless the user sets the acceleration scalar to below this value. Within this is another if statement that will check if the X component of the acceleration vector is greater than zero. This is designed to check if the value is positive as the next if statement is dependent on this. The next if statement checks if the ball’s velocity is less than the ball’s max velocity. If it is then the X acceleration value will be added to the velocity’s X component. It must be checked that the ball’s velocity is less than the maximum before adding to it so that the maximum velocity is not exceeded. It is important that it is checked if the acceleration value is negative or positive as, if a ball is going right at the maximum velocity (i.e. with a positive value) then the user must be able to reduce the ball’s speed in that direction by holding the key for the opposite direction, in this case left. Therefore there must be a check to see which direction the acceleration is being applied in, so that the user isn’t locked from changing the velocity with the keys when the maximum velocity is reached in one direction. This was originally what happened when the maximum velocity values were checked in the same if statement without discerning whether the value being added was positive or negative (right or left for the X component). After the if statement checking if the X component is positive ends, there is another if statement to check if it is negative for the reason listed above, carrying the same check to ensure the ball stays below the maximum velocity in the other direction. This is all repeated in the Y direction, including the minimum value check afterwards.

After this is the “addVelocity” function, which takes in one ball again. This function’s code will just add the velocity vector values to the ball’s centre values. This is done in order to imitate movement by having the ball’s coordinates changed to add the components of the velocity vector, having the ball change position in small jumps, with the movement often small enough to show smooth movement on screen to the human eye. The value changed is the ball’s universal coordinate which will be changed to the canvas coordinates later for drawing purposes.

After this is the “checkFriction” function which takes in one ball and one friction zone. The first code in the function is an if statement to check if the friction check of the ball entered is false. This is done to check that the ball hasn’t already had friction applied to it this loop. If it has then the code ahead does not need to be checked as the friction can only be applied once, therefore the code ahead will not be applicable. This saves time by not requiring the ball’s location be checked with further friction zones. If the check is false then there is another if statement, this time checking the ball is on the friction zone. This is detected by checking if the ball’s centre’s X component is greater than the friction zone’s start’s X component and less than the friction zone’s start’s X component plus the friction zone’s length’s X component. This is done as by being between these values it ensures that the ball is within the two sides of the friction zone. This is the same for the Y components of these vectors, also ensuring that the ball is within the top and bottom of the friction zone. If all of these are true then the ball will be counted as on top of the friction zone, and so the friction value of the friction zone will be applied to the ball’s velocity and this is applied with Equation 3 for both components of the velocity vector. The friction check of the ball is then set to true, meaning friction can not be applied again in this loop to this ball.

After the friction check is the “checkEdges” function, which takes in one ball and the canvas (object not HTML). This function will check if the ball’s centre’s X component minus the radius is less than the start of the canvas’s X component or greater than the start’s X component plus the length’s X component. It is the centre minus or plus the radius that is checked as this is where the point furthest to that side of the ball will be, so the centre minus the radius checks the leftmost point and plus the radius gives the rightmost point. The leftmost point is checked against the starting point of the canvas’s X component as this is the left side of the canvas, so if the ball’s centre minus the radius is less than this value then that means a part of the ball is going outside of the canvas which is not allowed. This is the same for the right side, which is given by the start plus the length in the X components.

Writing the code and building the system, challenges while doing so (reasoning behind coding itself, why it is written the way it is) plus think it’s also like recounting while writing it??

go into detail about the choices in equations and maths and all here, saying how each bit is written for its purpose, how it works, sources if necessary.

the nitty gritty crap

use this bit for player control method.

This code was adapted from source [7], updating the code for a newer update of JavaScript where keyCode is put as “ArrowRight”, “ArrowLeft”, “ArrowUp” and “ArrowDown”, rather than the numbers that were used in the source.

Also quote that one article thing you have bookmarked for the ball vectors method bit

# Chapter 5: Testing

The testing of the physics engine was largely performed while still designing it. These tests were all to do with the functionality of the physics engine. The full test case tables are available in the appendix.

The method of choosing areas of testing was through reading through the code, and looking at each function and designing a test case for these functions. If there were conditional statements or different possibilities for how the code would run depending on the values entered, then each possibility became a test case in order to ensure all possibilities for how the code would run were working. This ensured that the testing would be thorough. It was also important to check that each area worked together, so it was important first to test each area individually, hence the test cases.

The demos were in a way test cases for the functions of the physics engine working together, ensuring that they also worked in collaboration and not just individually. These demos show that they do work in unison, which is important as the functions need to be able to work in collaboration to ensure a more accurate emulation of physics than if the functions only worked individually.

The parts of the project most likely to have bugs in it are the parts that cover any form of interaction between objects or have human input in them. The interactions between objects will have higher chances of issue as this is a part that is unable to be controlled by a person as it happens and that is entirely dependent on the code working it out, so these parts have to be coded well and tested well to find and eliminate bugs. Parts with human input are also likely to have bugs as it is possible for a human to make mistakes that a computer might not, such as entering information incorrectly. This means that any correction for potential human error must be tested well to ensure that the correction takes place to avoid further error from the incorrect input. It is because of these areas having a higher likelihood for bugs that these are the areas most covered in the test cases.

Some of the most important, and some of the first, testing to be done for each object class (ball, block, canvas, friction zone, magnet and teleporter) was to ensure all objects were drawn correctly when displayed on the canvas. These were drawn through the draw square, draw circle and draw teleporter functions, so by creating these objects to test their constructors, these functions were also tested. These tests showed that all of the objects were drawn out as specified in terms of position, dimensions and colours, meaning that the constructors and drawing functions all worked, as the information had been correctly passed from the programming interface to the physics engine to produce the drawing of each object on the browser. This test also showed that the import into the separate file from the engine was successful as, by having the objects drawn from the file where the objects were created, it showed that the import of the initiate function and object classes was successful. It was important to test that these functions worked as without them the physics engine would not work. The user of the piece of code the programmer had made would not be able to see the objects created without them working, meaning that they would not be able to do anything with what the programmer had made. This made it very important to test that these functions worked.

Table : examples of test cases for object creation and drawing minus code section, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | Test draw circle function | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | Ball should be drawn on canvas to given specifications, proving the draw circle function has been called and worked correctly | Ball drawn on canvas to specifications | Pass |
| 8 | Test block constructor | 1. Import objects and initiate function into JavaScript file 2. Create block object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | Block will be drawn out on canvas to given specifications | Block drawn to specifications | Pass |

Another set of tests relating to the constructors for each class was ensuring that if a property of the objects was entered incorrectly, then it was correctly fixed. The ball class constructor has five different Boolean properties, magnetic, stationary, player, ghost and friction check, each of which has if statements to fix any wrong data type entries, so these had to be tested. In order to ensure these worked, each had to be tested by creating a ball for each type of property with that property entered incorrectly, and adding a line of code of “console.log(“*property*:“+ball\_1.*property*);” for each Boolean property into the physics engine. By then loading the file in the browser and opening the console section of the developer’s tools of the browser, it showed that the property had changed from the wrong data entry to “false”. With this result it showed that the correction had worked.

Table : example of test case for boolean correction code in ball constructor, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 49 | Test that ball’s stationary property can not be anything but true or false, set to false as default | 1. Import objects and initiate function into JavaScript file 2. Create ball with stationary entry as anything but true or false 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s stationary Boolean 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | Stationary shall be changed to false | Stationary changed to false | Pass |

There were also test cases to ensure the if statements that ensure numbers entered into the ball, block, canvas, friction zone, magnet and teleporter constructors are within certain ranges all work. These also had to be tested in a similar manner to the Boolean checks, where an object was created with a purposefully incorrect value and checked to see if the value was corrected properly. The values that had corrections to be tested were the radius of non-ghost balls, the mass of balls, the radius of magnets, the lengths of blocks, the lengths and friction values of canvases, the lengths and friction values of friction zones and lengths of teleporters. Each class had a new object created for each property that their constructor could correct with that property entered outside of the required range (e.g. a ball with a radius of -1 and no other problems). This was done in order to test each correction individually, and it was seen that each correction took place, whether that was through seeing that on screen through the drawing of the object, for properties such as radius and length, or through using “console.log(*object.property*)” to see what the property was changed to. It was important for code like this and the Boolean tests above to be tested as, if they failed when the programmer incorrectly entered properties, it would have caused the code to not work as required. One example with the code was if the programmer entered a mass of 0 for a ball. This would have caused great issues if the ball then collided with another ball, as part of the calculation to resolve the collision would have divided by the mass of the ball, and a division by 0 would have caused an arithmetic exception. Thus it was important to test that these pieces of code worked so the physics engine could run without issue.

Table : example of test case for numerical correction in magnet and ball constructors, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 | Test that magnets have a radius of at least 1 if number less than 1 is entered | 1. Import objects and initiate function into JavaScript file 2. Create magnet with radius of 0 or less 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | Magnet shall be created with radius of 1 | Magnet created with radius of 1 | Pass |
| 47 | Test that ball’s magnetism can not be negative | 1. Import objects and initiate function into JavaScript file 2. Create ball with magnetic property and magnetism less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s magnetism 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | Ball’s magnetism shall be set to 0 | Ball’s magnetism set to 0 | Pass |

Another section to be tested was the correction of the positioning of the objects if they were created outside of the boundaries of the canvas. This was tested by creating objects of each type (other than canvas) outside of the boundaries of the canvas on purpose and then loading the file in a browser to observe what happened. It was seen that all objects that were purposefully created outside of the canvas would be corrected to be within the canvas. It was also required test the change of the teleporter’s teleport coordinates. It was tested in the same way, creating a teleporter with teleport coordinates outside of the canvas, where it was then corrected to be within the canvas. These tests were important to cover as objects being created or moved outside of the canvas would have caused great issues, as much of the code was built around everything being within the canvas, and so chunks of code working incorrectly, due to this not being the case, would have caused glitches with the code, such as balls not being able to move, so it was important that this code worked correctly, which is why it was tested.

Table : examples of test cases for positional correction for blocks and friction zones, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 35 | Test that blocks created with coordinates that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create block with top left and length that would cause block to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | Block shall be moved to have top left corners at top left corner of canvas | Block moved to have starting point of (canvas starting point x, canvas starting point y) | Pass |
| 36 | Test that friction zones created with coordinates that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create friction zone with top left and length that would cause friction zone to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | Friction zone shall be moved to have top left corners at top left corner of canvas | Friction zone moved to have starting point of (canvas starting point x, canvas starting point y) | Pass |

Another area that needed testing was the different properties of the balls and ensuring they worked properly. The magnetic property needed testing in several different ways. One was the magnetic property on two balls and their interactions, with matching and opposing poles, with different masses and when one was stationary. These were tested by creating balls with the described properties in sets and checking that the balls acted as expected in each test case.

Table : example of test cases for magnetic balls, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 24 | Test ball magnetic property with other balls affected by mass | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create other magnetic ball object with south pole and different mass as first 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | Heavier ball shall move near other magnetic ball and be attracted to it slower than the lighter ball is attracted to it | Heavier ball moved near other magnetic ball and was attracted to it slower than the other ball was attracted to it | Pass |

It was also important to test the interaction between a magnetic ball and a magnet, with matching and opposing balls, how balls with different mass interacted with the magnet and a stationary magnetic ball with a magnet. These were tested in a similar manner as the magnetic balls scenarios where a magnetic ball and a magnet were created with the different properties in sets required to test each case.

Table : example of test case for magnetic balls and magnets, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 57 | Test that stationary balls do not move in magnetic interactions with magnets | 1. Import objects and initiate function into JavaScript file 2. Create magnetic stationary ball 3. Create magnet 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | Stationary ball won’t be moved toward magnet | Stationary ball didn’t move toward magnet | Pass |

The ball’s stationary property also had to be tested to make sure it did not move when in a collision with another ball. This was tested by having a non-stationary ball collide with the stationary ball.

Table : example of test case for stationary ball

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 58 | Test that stationary balls do not move when in collision | 1. Import objects and initiate function into JavaScript file 2. Create stationary ball 3. Create ball 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Move other ball into stationary ball | Stationary ball won’t be moved after collision with other ball | Stationary ball didn’t move after collision with other ball | Pass |

The player property also had to be tested, as a lot of the coding of the physics engine was made to be dependent on the player property. The player ball being controllable had to be tested, as well as the player ball remaining at the centre of the canvas by having every other object move around it. These were tested by having a player ball created and attempting to control it using the arrow keys, as this tested both issues at once by seeing if the ball moved and seeing if it remained at the centre of the screen.

Table : example of test case for player ball, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 14 | Test screen following player controlled ball | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser 6. Press right key | Ball shall be drawn and move to right, border shall disappear and canvas shall move round to left to show ball moving to the right while staying centre of the screen | Ball drawn and moved to right, border disappearing off screen and canvas moving to left to show ball moving while ball stayed in centre of screen | Pass |

The ghost property was also necessary to test to ensure that it worked. This was tested by creating a ball with the ghost property and then moving it into one of each of the other types of object and another ball. This was to test if the ball with the ghost property interacted with the other objects or not.

Table : example of test case for ghost ball, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 15 | Test ball ghost property works with blocks | 1. Import objects and initiate function into JavaScript file 2. Create ghost ball object 3. Create block object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until into ) block | Ball shall be drawn and move right, before going inside block with no collision | Ball drawn and moved to right, moving through the block | Pass |

It was important to test these properties of the ball due to the importance of the properties to the function of the physics engine, as without the magnetic ball property working, no magnetic effect would have worked as the magnets only affected magnetic balls, since magnet class objects do not move and so they are unaffected by other magnet class objects. This meant that balls with the magnetic property only affected balls that also had the magnetic property. Without the stationary property working, it is impossible to have balls stay still without constant correction of its position. Without the player property it is not possible for a ball to be controlled or for the camera to move, which in turn can cause the animation to break as the drawing and positioning of everything in each frame is dependent on the position of the player ball. Without the ghost property it is impossible for a programmer to create a simulation where they can watch without affecting anything since the user will be controlling a ball that may interact with other objects, thus the ghost property is needed in order to watch without affecting the environment. These reasons show that it was important to test these properties worked correctly.

It was also necessary for the interactions of a ball with other balls (outside of properties already tested) and objects other than magnets to be tested. These interactions to be tested were the collision between two moving balls, the collision between a ball and the edge of the canvas, the collision between a ball and a block, the detection of a ball on a friction zone the application of friction to a ball, the detection of a ball on a teleporter and the teleportation of it and the application of friction to a ball from the canvas. Each was tested by creating a ball controlled by the player and moving it into the other object to ensure that the interaction that was meant to take place, took place.

The ball’s interaction with another ball was tested by moving the player ball into the other ball to ensure that the balls both move post collision and that their resultant velocities are appropriate to the collision that occurred. The ball’s interaction with the edge of the canvas was tested by moving the ball into the edge of the canvas and ensuring that the ball’s velocity was changed to show a deflection off the edge based on the ball’s initial velocity. The ball’s interaction with the block was tested by moving the ball into the block and ensuring that the ball’s velocity was appropriately changed to show a deflection off the block based on the ball’s initial velocity. The ball’s interactions with the friction zone’s and the canvas’s friction values were tested by moving the player ball onto the friction zone to ensure the friction is applied when the ball enters the area of the friction zone. This friction change was checked by using the “console.log()” code for the friction of the canvas and the friction zone and then checking that the friction value applied to the ball changed, in order to test that both the friction zone’s and the canvas’s friction values were being applied correctly. The ball’s interaction with the teleporter was tested by moving the ball onto the teleporter and ensuring that the ball was teleported when it entered this area, and that it was teleported to the correct area.

These functions were tested this way as each object’s function is how the object interacts with a ball. Without these functions working properly, the engine would not work at all, as the key features of the physics engine are all about emulating the interaction between these objects in a manner that reflects reality (for those that are possible in reality, i.e. not the teleporter). This meant that it was very important for these functions to be tested.

Table : example of test cases for ball’s interactions with other objects, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 28 | Test ball collisions occur with block edges | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create friction zone 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until hitting block | Ball shall rebound off block edges | Ball rebounded off block edge | Pass |
| 29 | Test ball affected by friction zone friction | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create friction zone object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right arrow key to build speed 8. Release arrow key on friction zone | Ball shall be slowed down by friction factor of friction zone and not canvas when on friction zone | Ball slowed down by friction factor of friction zone and not canvas when on friction zone | Pass |

Another set of functions that were important to test were the default canvas and player ball constructors when they were required. The default canvas constructor was tested by not putting a canvas object in a file when creating a simulation, then by putting two in, in order to test both possible ways it could be required. This gave the result that the default canvas was created, and in the case of multiple being created both created were deleted, proving the tests were passed. This was also the same for the default player ball constructor, where two test cases were done, one with no player ball and another with two. Again these tests were successful, where a default player ball was created, removing the two as player balls in the case where two were created, showing the tests were passed. These were important test cases to check as these default objects are required in order to work. A canvas is required for all other objects to be displayed within, and the physics engine is currently not capable of working with multiple canvases, meaning no canvas or multiple canvases will both cause the simulation or game to break. A player ball is also required as it will be the focal point for how other objects are displayed (relative to it), therefore a player ball is required, however this camera effect can only work with one player ball, as all other object’s positions are taken relative to the player ball, therefore if there are multiple player balls they must be removed. This meant that a default canvas and player ball were incredibly important to have in order to stop the programmer’s simulation breaking when no or multiple of these objects are created.

Table : example of test cases for default creation with no canvas/player ball or multiple of each, full table available in appendix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 | Test default camera ball creation | 1. Import objects and initiate function into JavaScript file 2. Call initiate function 3. Call JavaScript in HTML file 4. Open HTML file in browser | Default camera ball from initiate function should be created (will be 0 pixels in radius, but controllable by arrow keys) | Camera ball created | Pass |
| 32 | Test multiple canvases can’t be created | 1. Import objects and initiate function into JavaScript file 2. Create multiple canvas objects 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | Canvases shall be removed from canvas array and default canvas shall be used | Canvases removed and default canvas used | Pass |

# Chapter 6: Evaluation and Conclusion

Obvious stuff like improvements to go here, say what was done well too?

# Appendix

## Tables of figures

[Code Snippet 1: code used for import of important functions and objects into JavaScript of programmer 6](#_Toc113147478)

[Code Snippet 2: example of code used to create new canvas object 6](#_Toc113147479)

[Code Snippet 3: example of code used to create new ball object 6](#_Toc113147480)

[Code Snippet 4: example of code used to create new block object 7](#_Toc113147481)

[Code Snippet 5: example of code used to create new FrictionZone object 7](#_Toc113147482)

[Code Snippet 6: example of code used to create new Magnet object 7](#_Toc113147483)

[Code Snippet 7: example of code used to create new Teleporter object 7](#_Toc113147484)

[Code Snippet 8: setting of x and y vectors of vector object in constructor 17](#_Toc113147485)

[Code Snippet 9: generic code for setting variable with value entered into constructor 18](#_Toc113147486)

[Code Snippet 10: code for setting centre of ball using vector object 18](#_Toc113147487)

[Table 1: examples of test cases for object creation and drawing minus code section, full table available in appendix 22](#_Toc113147466)

[Table 2: example of test case for boolean correction code in ball constructor, full table available in appendix 23](#_Toc113147467)

[Table 3: example of test case for numerical correction in magnet and ball constructors, full table available in appendix 24](#_Toc113147468)

[Table 4: examples of test cases for positional correction for blocks and friction zones, full table available in appendix 24](#_Toc113147469)

[Table 5: example of test cases for magnetic balls, full table available in appendix 25](#_Toc113147470)

[Table 6: example of test case for magnetic balls and magnets, full table available in appendix 25](#_Toc113147471)

[Table 7: example of test case for stationary ball 25](#_Toc113147472)

[Table 8: example of test case for player ball, full table available in appendix 25](#_Toc113147473)

[Table 9: example of test case for ghost ball, full table available in appendix 26](#_Toc113147474)

[Table 10: example of test cases for ball’s interactions with other objects, full table available in appendix 27](#_Toc113147475)

[Table 11: example of test cases for default creation with no canvas/player ball or multiple of each, full table available in appendix 27](#_Toc113147476)

[Table 12: Test cases C](#_Toc113147477)

[Equation 1: calculation for magnitude of a vector 17](#_Toc113147463)

[Equation 2: calculation for dot product of two vectors 18](#_Toc113147464)

[Equation 3: velocity calculation in friction zone per loop 20](#_Toc113147465)

[Code block 1: example code for creation HH](#_Toc113147458)

[Code block 2: index.html LL](#_Toc113147459)

[Code block 3: engineConfig.js LL](#_Toc113147460)

[Code block 4: physics.js LL](#_Toc113147461)

[Code block 5: objectCreation.js PPP](#_Toc113147462)

[Figure 1: example of UI with one of each type of object created 8](#_Toc113147447)

[Figure 2: Picture of ball object DD](#_Toc113147448)

[Figure 3: Picture of block object (lighter grey rectangles) DD](#_Toc113147449)

[Figure 4: Canvas object edges, green being outside and darker grey being inside of canvas EE](#_Toc113147450)

[Figure 5: Two friction zones pictured, the purple one has a positive friction value, slowing balls down, and the orange has a negative value, speeding balls up EE](#_Toc113147451)

[Figure 6: a magnet (white circle) pulling a ball (red circle) towards its centre, dragging the ball into the path of a teleporter (gold rectangle) FF](#_Toc113147452)

[Figure 7: teleporter represented by a circle in a square GG](#_Toc113147453)

[Figure 8: Player ball about to knock yellow ball into teleporter to have it trapped in "pen" KK](#_Toc113147454)

[Figure 9: yellow ball now trapped with magnetic turquoise balls after being knocked into teleporter KK](#_Toc113147455)

## References

# Bibliography

|  |  |
| --- | --- |
| [1] | MDN Contributors, “Window.requestAnimationFrame(),” MDN web docs, 11 August 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. [Accessed 24 August 2022]. |
| [2] | “JavaScript Where To,” [Online]. Available: https://www.w3schools.com/js/js\_whereto.asp. |
| [3] | N. DQ, “Magnitude of a vector definition,” Math Insight, [Online]. Available: https://mathinsight.org/definition/magnitude\_vector. [Accessed 3 September 2022]. |
| [4] | D. Nykamp, “The dot product,” Math Insight, [Online]. Available: https://mathinsight.org/dot\_product. [Accessed 3 September 2022]. |
| [5] | MDN contributors, “Window.requestAnimationFrame(),” mozilla, 11 August 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame. [Accessed 4 September 2022]. |
| [6] | “CanvasRenderingContext2D.arc(),” Mozilla, 22 July 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/arc. [Accessed 3 September 2022]. |
| [7] | S. Wieczorak, “How to use arrow keys to move object smoothly in canvas,” stack overflow, 23 May 2017. [Online]. Available: https://stackoverflow.com/questions/39806858/how-to-use-arrow-keys-to-move-object-smoothly-in-canvas. [Accessed 2 September 2022]. |
| [8] | D. Nykamp, “Unit vector definition,” Math Insight, [Online]. Available: https://mathinsight.org/definition/unit\_vector. [Accessed 3 September 2022]. |

## Test Cases

Table : Test cases

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Case No | Test Case Description | Test Steps | Test Data | Expected Result | Actual Result | Pass/Fail |
| 1 | Test import objects from physics.js into JavaScript file of programmer | 1. Import objects and initiate function into JavaScript file 2. Create one of each object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas = new Canvas(0, 0, 1500,3000, 0.01, "black", "orange");  let Friction1 = new FrictionZone(500, 350, 100, 100, "black", "purple", 0.05);  let block1 = new Block(600, 300, 300, 300, "Black", "Silver");  let magnet1 = new Magnet (550, 20, 20, 20, "North", "black", "white");  let Teleporter1 = new Teleporter (500, 600, 150, 150, 100, 100, "black", "gold", "silver");  let Ball1 = new Ball(110, 450, 20, 100, 5, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);  initiate();  Launch file in browser | Objects should be drawn on the canvas, proving the objects and functions have been correctly imported. | Objects drawn on canvas | Pass |
| 2 | Test draw circle function | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas = new Canvas(0, 0, 1500,3000, 0.01, "black", "orange");  let Ball1 = new Ball(110, 450, 20, 100, 5, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);  initiate();  Launch file in browser | Ball should be drawn on canvas to given specifications, proving the draw circle function has been called and worked correctly | Ball drawn on canvas to specifications | Pass |
| 3 | Test draw square function | 1. Import objects and initiate function into JavaScript file 2. Create block object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas = new Canvas(0, 0, 1500,3000, 0.01, "black", "orange");  let block1 = new Block(600, 300, 300, 300, "Black", "Silver");  initiate();  Launch file in browser | Block should be drawn on canvas to given specifications, proving draw square function has been called and worked correctly | Block drawn on canvas to specifications | Pass |
| 4 | Test default canvas creation | 1. Import objects and initiate function into JavaScript file 2. Call initiate function 3. Call JavaScript in HTML file 4. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  initiate();  Launch file in browser | Default canvas from initiate function should be created | Canvas created | Pass |
| 5 | Test default camera ball creation | 1. Import objects and initiate function into JavaScript file 2. Call initiate function 3. Call JavaScript in HTML file 4. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  initiate();  Launch file in browser | Default camera ball from initiate function should be created (will be 0 pixels in radius, but controllable by arrow keys) | Camera ball created | Pass |
| 6 | Test canvas constructor | 1. Import objects and initiate function into JavaScript file 2. Create canvas object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas = new Canvas(0, 0, 1500,3000, 0.01, "black", "orange");  initiate();  Launch file in browser | Canvas will be drawn out to given specifications | Canvas drawn to specifications | Pass |
| 7 | Test ball constructor | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 5, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);  initiate();  Launch file in browser | Ball will be drawn out on canvas to given specifications | Ball drawn to specifications | Pass |
| 8 | Test block constructor | 1. Import objects and initiate function into JavaScript file 2. Create block object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let block1 = new Block(600, 300, 300, 300, "Black", "Silver");  initiate();  Launch file in browser | Block will be drawn out on canvas to given specifications | Block drawn to specifications | Pass |
| 9 | Test friction zone constructor | 1. Import objects and initiate function into JavaScript file 2. Create friction zone object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Friction1 = new FrictionZone(500, 350, 100, 100, "black", "purple", 0.05);  initiate();  Launch file in browser | Friction zone will be drawn out on canvas to given specifications | Friction zone drawn to specifications | Pass |
| 10 | Test magnet constructor | 1. Import objects and initiate function into JavaScript file 2. Create magnet object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let magnet1 = new Magnet (550, 20, 20, 20, "North", "black", "white");  initiate();  Launch file in browser | Magnet will be drawn out on canvas to given specifications | Magnet drawn to specifications | Pass |
| 11 | Test teleporter constructor | 1. Import objects and initiate function into JavaScript file 2. Create teleporter object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Teleporter1 = new Teleporter (500, 600, 150, 150, 100, 100, "black", "gold", "silver");  initiate();  Launch file in browser | Teleporter will be drawn out on canvas to given specifications | Teleporter drawn to specifications | Pass |
| 12 | Test ball movement | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 5, 0, 5, 0, true, "South", 2, false, false, false, "black", "red", false);  initiate();  Launch file in browser | Ball shall be drawn out and move to right | Ball drawn and moved to right | Pass |
| 13 | Test player controlled ball movement | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser 6. Press right key | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  Launch file in browser  Press right arrow key | Ball shall be drawn and move to right | Ball drawn and moved to right | Pass |
| 14 | Test screen following player controlled ball | 1. Import objects and initiate function into JavaScript file 2. Create ball object 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser 6. Press right key | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  Launch file in browser  Press right arrow key | Ball shall be drawn and move to right, border shall disappear and canvas shall move round to left to show ball moving to the right while staying centre of the screen | Ball drawn and moved to right, border disappearing off screen and canvas moving to left to show ball moving while ball stayed in centre of screen | Pass |
| 15 | Test ball ghost property works with blocks | 1. Import objects and initiate function into JavaScript file 2. Create ghost ball object 3. Create block object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until into ) block | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, true, "black", "red", false);  let block1 = new Block(600, 300, 300, 300, "Black", "Silver");  initiate();  Launch file in browser  Press right arrow key | Ball shall be drawn and move right, before going inside block with no collision | Ball drawn and moved to right, moving through the block | Pass |
| 16 | Test ball ghost property works with friction zones | 1. Import objects and initiate function into JavaScript file 2. Create ghost ball object 3. Create friction zone object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until into friction zone | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, true, "black", "red", false);  let Friction1 = new FrictionZone(500, 400, 100, 100, "black", "purple", 0.05);  initiate();  Launch file in browser  Press right arrow key | Ball shall move over friction zone without effect | Ball moved over friction zone without effect | Pass |
| 17 | Test ball ghost property works with balls | 1. Import objects and initiate function into JavaScript file 2. Create ghost ball object 3. Create other ball object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until into other ball | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, true, "black", "red", false);  let Ball2 = new Ball(500, 450, 20, 100, -5, 0, 5, 1, true, "South", 3, false, false, false, "black", "yellow", false);  initiate();  Launch file in browser  Press right arrow key | Ball shall move through ball without effect | Ball moved through ball without effect | Pass |
| 18 | Test ball ghost property works with magnets | 1. Import objects and initiate function into JavaScript file 2. Create ghost magnetic ball object 3. Create magnet object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until into magnet | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, true, "black", "red", false);  let magnet1 = new Magnet (550, 450, 20, 20, "North", "black", "white");  initiate();  Launch file in browser  Press right arrow key | Ball shall move near magnet without effect | Ball moved near magnet without effect | Pass |
| 19 | Test ball ghost property works with teleporters | 1. Import objects and initiate function into JavaScript file 2. Create ghost ball object 3. Create teleporter object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until into teleporter | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, true, "black", "red", false);  let Teleporter1 = new Teleporter (400, 400, 150, 150, 100, 100, "black", "gold", "silver");  initiate();  Launch file in browser  Press right arrow key | Ball shall move over teleporter without effect | Ball moved over teleporter without effect | Pass |
| 20 | Test ball magnetic property works with other magnetic balls with opposite poles | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create other magnetic ball object with south pole and same mass as first 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Ball2 = new Ball(300, 450, 20, 100, 0, 0, 5, 0, true, "North", 2, false, false, false, "black", "red", false);  initiate();  Launch file in browser  Press right arrow key | Ball shall move near other magnetic ball and be attracted to it | Ball moved near other magnetic ball and was attracted to it | Pass |
| 21 | Test ball magnetic property works with magnets with opposite poles | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create magnet object with south pole 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let magnet1 = new Magnet (550, 450, 20, 20, "North", "black", "white");  initiate();  Launch file in browser  Press right arrow key | Ball shall move near magnet and be attracted to it | Ball moved near magnet and was attracted to it | Pass |
| 22 | Test ball magnetic property works with other magnetic balls with matching poles | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create other magnetic ball object with north pole and same mass as first 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Ball2 = new Ball(300, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, false, false, "black", "red", false);  initiate();  Launch file in browser  Press right arrow key | Ball shall move near other magnetic ball and be repulsed by it | Ball moved near other magnetic ball and was repulsed by it | Pass |
| 23 | Test ball magnetic property works with magnets with matching poles | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create magnet object with north pole 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let magnet1 = new Magnet (550, 450, 20, 20, "South", "black", "white");  initiate();  Launch file in browser  Press right arrow key | Ball shall move near magnet and be repulsed by it | Ball moved near magnet and was repulsed by it | Pass |
| 24 | Test ball magnetic property with other balls affected by mass | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create other magnetic ball object with south pole and different mass as first 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Ball2 = new Ball(300, 450, 20, 50, 0, 0, 5, 0, true, "North", 2, false, false, false, "black", "red", false);  initiate();  Launch file in browser  Press right arrow key | Heavier ball shall move near other magnetic ball and be attracted to it slower than the lighter ball is attracted to it | Heavier ball moved near other magnetic ball and was attracted to it slower than the other ball was attracted to it | Pass |
| 25 | Test ball magnetic property with magnets affected by mass | 1. Import objects and initiate function into JavaScript file 2. Create magnetic ball object with north pole 3. Create magnet object with north pole 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Repeat 1-7 with ball with different mass | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let magnet1 = new Magnet (550, 450, 20, 20, "North", "black", "white");  initiate();  Launch file in browser  Press right arrow key  Repeat above but replace  Ball1 with:  let Ball1 = new Ball(110, 450, 20, 50, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false); | Heavier ball shall move to the magnet slower than the lighter ball | Heavier ball moved to the magnet slower than the lighter magnet | Pass |
| 26 | Test ball movement affected by canvas friction | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create canvas object with friction value 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right arrow key to build speed 8. Release arrow key when on friction zone | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let defaultCanvas2 = new Canvas(1500, 3000, 4000,3000, 0.05, "black", "orange");  initiate();  Launch file in browser  Press right arrow key  Release right arrow key | Ball shall be slowed down by friction while on canvas (i.e. not on friction zone) | Ball slowed down while on canvas and not friction zone | Pass |
| 27 | Test ball collisions occur with canvas edges | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser 6. Press left key until hitting canvas edge | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  Launch file in browser  Press left arrow key  Release left arrow key as the canvas edge is reached | Ball shall rebound off canvas edges | Ball rebounded off canvas edges | Pass |
| 28 | Test ball collisions occur with block edges | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create friction zone 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right key until hitting block | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let block1 = new Block(600, 300, 300, 300, "Black", "Silver");  initiate();  Launch file in browser  Press left arrow key  Release left arrow key as the block edge is reached | Ball shall rebound off block edges | Ball rebounded off block edge | Pass |
| 29 | Test ball affected by friction zone friction | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create friction zone object 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right arrow key to build speed 8. Release arrow key on friction zone | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Friction1 = new FrictionZone(500, 400, 100, 100, "black", "purple", 0.05);  initiate();  Launch file in browser  Press right arrow key  Release right arrow key when on friction zone | Ball shall be slowed down by friction factor of friction zone and not canvas when on friction zone | Ball slowed down by friction factor of friction zone and not canvas when on friction zone | Pass |
| 30 | Test ball affected by teleporters | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create teleporter 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Press right arrow key until hitting teleporter | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Teleporter1 = new Teleporter (400, 400, 150, 150, 100, 100, "black", "gold", "silver");  initiate();  Launch file in browser  Press right arrow key | Ball shall be moved to teleport point of teleporter when centre of ball is on area of teleporter | Ball moved to teleport point of teleporter when centre of ball is on area of teleporter | Pass |
| 31 | Test multiple player balls can’t be created | 1. Import objects and initiate function into JavaScript file 2. Create multiple player ball 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Ball2 = new Ball(210, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  Launch file in browser | Player balls shall be made non player and camera ball will instead be created | Player balls made non player and camera ball created | Pass |
| 32 | Test multiple canvases can’t be created | 1. Import objects and initiate function into JavaScript file 2. Create multiple canvas objects 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas1 = new Canvas(0, 0, 1500,3000, 0.05, "black", "orange");  let defaultCanvas2 = new Canvas(1500, 3000, 4000,3000, 0.05, "black", "orange");  initiate();  Launch file in browser | Canvases shall be removed from canvas array and default canvas shall be used | Canvases removed and default canvas used | Pass |
| 33 | Test that balls created with a radius that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create ball with centre and radius that would cause ball to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(0, 0, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  Launch file in browser | Ball shall be moved to have centre of (radius, radius) so edges just touched canvas edges | Ball moved to have centre of (radius, radius) | Pass |
| 34 | Test that magnets created with a radius that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create magnet with centre and radius that would cause magnet to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let magnet1 = new Magnet (0, 0, 20, 20, "South", "black", "white");  initiate();  Launch file in browser | Magnet shall be moved to have centre of (radius, radius) so edges just touched canvas edges | Magnet moved to have centre of (radius, radius) | Pass |
| 35 | Test that blocks created with coordinates that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create block with top left and length that would cause block to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let block1 = new Block(-600, -300, 300, 300, "Black", "Silver");  initiate();  Launch file in browser | Block shall be moved to have top left corners at top left corner of canvas | Block moved to have starting point of (canvas starting point x, canvas starting point y) | Pass |
| 36 | Test that friction zones created with coordinates that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create friction zone with top left and length that would cause friction zone to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Friction1 = new FrictionZone(-500, -400, 100, 100, "black", "purple", 0.05);  initiate();  Launch file in browser | Friction zone shall be moved to have top left corners at top left corner of canvas | Friction zone moved to have starting point of (canvas starting point x, canvas starting point y) | Pass |
| 37 | Test that teleporters created with coordinates that would cause them to exit the canvas will be corrected to be in the top left corner just touching the edges | 1. Import objects and initiate function into JavaScript file 2. Create teleporter with top left and length that would cause teleporter to extend out of canvas 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Teleporter1 = new Teleporter (-400, -400, 150, 150, 100, 100, "black", "gold", "silver");  initiate();  Launch file in browser | Teleporter shall be moved to have top left corners at top left corner of canvas | Teleporter moved to have starting point of (canvas starting point x, canvas starting point y) | Pass |
| 38 | Test that balls have a radius of at least 1 if number less than 1 is entered unless a ghost ball | 1. Import objects and initiate function into JavaScript file 2. Create ball with radius of 0 or less that is not a ghost ball 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 0, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  Launch file in browser | Ball shall be created with radius of 1 | Ball created with radius of 1 | Pass |
| 39 | Test that magnets have a radius of at least 1 if number less than 1 is entered | 1. Import objects and initiate function into JavaScript file 2. Create magnet with radius of 0 or less 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let magnet1 = new Magnet (550, 450, 0, 20, "South", "black", "white");  initiate();  Launch file in browser | Magnet shall be created with radius of 1 | Magnet created with radius of 1 | Pass |
| 40 | Test that blocks have a length of at least 1 in each direction if created with lengths of less than 1 | 1. Import objects and initiate function into JavaScript file 2. Create block with length in x or y of less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let block1 = new Block(600, 300, 0, 0, "Black", "Silver");  initiate();  Launch file in browser | Block shall be created with length of 1 | Block created with length of 1 | Pass |
| 41 | Test that friction zones have a length of at least 1 in each direction if created with lengths of less than 1 | 1. Import objects and initiate function into JavaScript file 2. Create friction zone with length in x or y of less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Friction1 = new FrictionZone(500, 400, 0, 0, "black", "purple", 0.05);  initiate();  Launch file in browser | Friction zone shall be created with length of 1 | Friction zone created with length of 1 | Pass |
| 42 | Test that teleporters have a length of at least 1 in each direction if created with lengths of less than 1 | 1. Import objects and initiate function into JavaScript file 2. Create teleporter with length in x or y of less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Teleporter1 = new Teleporter (400, 400, 0, 0, 100, 100, "black", "gold", "silver");  initiate();  Launch file in browser | Teleporter shall be created with length of 1 | Teleporter created with length of 1 | Pass |
| 43 | Test that the canvas will have a length of at least 1 in each direction if created with lengths of less than 1 | 1. Import objects and initiate function into JavaScript file 2. Create canvas with length in x or y of less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas2 = new Canvas(1500, 3000, 0,0, 0.01, "black", "orange");  initiate();  Launch file in browser | Canvas shall be created with length of 1 | Canvas created with length of 1 | Pass |
| 44 | Test that teleporters will not move outside of canvas | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create teleporter with teleport coordinates outside of canvas coordinates 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Move player ball onto teleporter | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Teleporter1 = new Teleporter (400, 400, 100, 100, -100, -100, "black", "gold", "silver");  initiate();  Launch file in browser | Teleport coordinates shall be moved to centre of canvas | Teleport coordinates moved to centre of canvas | Pass |
| 45 | Test that mass can not be 0 or negative | 1. Import objects and initiate function into JavaScript file 2. Create ball with mass of 0 or less 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s mass 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  initiate();  console.log(ball\_1.mass); below line 1053  Launch file in browser  Press f12 | Mass shall be changed to 0.00001 | Mass changed to 0.00001 | Pass |
| 46 | Test that ball’s magnetic property can not be anything but true or false, set to false as default | 1. Import objects and initiate function into JavaScript file 2. Create ball with magnetic entry as anything but true or false 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s magnetic Boolean 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, “blue”, "South", 2, false, true, false, "black", "red", false);  initiate();  console.log(ball\_1.magnetic); below line 1053  Launch file in browser  Press f12 | Magnetic shall be changed to false | Magnetic changed to false | Pass |
| 47 | Test that ball’s magnetism can not be negative | 1. Import objects and initiate function into JavaScript file 2. Create ball with magnetic property and magnetism less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s magnetism 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", -2, false, true, false, "black", "red", false);  initiate();  console.log(ball\_1.magnetism); below line 1053  Launch file in browser  Press f12 | Ball’s magnetism shall be set to 0 | Ball’s magnetism set to 0 | Pass |
| 48 | Test that ball’s acceleration scalar can not be negative | 1. Import objects and initiate function into JavaScript file 2. Create ball with acceleration scalar less than 0 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s acceleration scalar 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, -1, true, "South", 2, false, true, false, "black", "red", false);  initiate();  console.log  (ball\_1.accelerationscalar);  below line 1053  Launch file in browser  Press f12 | Ball’s acceleration scalar set to 0 | Ball’s acceleration scalar set to 0 | Pass |
| 49 | Test that ball’s stationary property can not be anything but true or false, set to false as default | 1. Import objects and initiate function into JavaScript file 2. Create ball with stationary entry as anything but true or false 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s stationary Boolean 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, “blue”, true, false, "black", "red", false);  initiate();  console.log(ball\_1.stationary);  below line 1053  Launch file in browser  Press f12 | Stationary shall be changed to false | Stationary changed to false | Pass |
| 50 | Test that ball’s player property can not be anything but true or false, set to false as default | 1. Import objects and initiate function into JavaScript file 2. Create ball with player entry as anything but true or false 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s player Boolean 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, false, “blue”, false, "black", "red", false);  initiate();  console.log(ball\_1.player);  below line 1053  Launch file in browser  Press f12 | Player shall be changed to false | Player changed to false | Pass |
| 51 | Test that ball’s ghost property can not be anything but true or false, set to false as default | 1. Import objects and initiate function into JavaScript file 2. Create ball with ghost entry as anything but true or false 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check ball’s ghost Boolean 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, false, true, “blue”, "black", "red", false);  initiate();  console.log(ball\_1.ghost);  below line 1053  Launch file in browser  Press f12 | Ghost shall be changed to false | Ghost changed to false | Pass |
| 52 | Test that friction only applied once, from friction zone or from canvas | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Call initiate function 4. Call JavaScript in HTML file 5. Add console.log to check what friction values are applied 6. Open HTML file in browser 7. Press F12 to open developer’s tools and go to console 8. Move over canvas then friction zone | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas2 = new Canvas(0, 0, 1500, 3000, 0.01, "black", "orange");  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Friction1 = new FrictionZone(500, 400, 100, 100, "black", "purple", 0.05);  initiate();  console.log(square\_1.friction);  below line 453  console.log  (CanvasArray[0].friction) below line 876  Launch file in browser  Press f12  Press right key until on friction zone | Friction value applied shall change when ball moves from canvas to friction zone | Friction value applied changed when ball moved from canvas to friction zone | Pass |
| 53 | Test that friction zones can not have a friction value greater than 1 | 1. Import objects and initiate function into JavaScript file 2. Create friction zone with friction greater than 1 3. Create player ball 4. Call initiate function 5. Call JavaScript in HTML file   Add console.log to check friction of zone under ball   1. Open HTML file in browser 2. Press F12 to open developer’s tools and go to console 3. Move ball over friction zone | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Friction1 = new FrictionZone(500, 400, 100, 100, "black", "purple", 1.4);  initiate();  console.log(square\_1.friction);  below line 453  Launch file in browser  Press f12  Press right key until on friction zone | Friction value shall be changed to 1 | Friction value changed to 1 | Pass |
| 54 | Test that magnets can not have a negative magnetism | 1. Import objects and initiate function into JavaScript file 2. Create magnet with magnetism less than 0 3. Create player ball 4. Call initiate function 5. Call JavaScript in HTML file 6. Add console.log to check magnetism of magnets 7. Open HTML file in browser 8. Press F12 to open developer’s tools and go to console 9. Move ball near magnet | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(10, 10, 10, 0, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let magnet1 = new Magnet (550, 450, 20, -20, "South", "black", "white");  initiate();  console.log  (magnet\_1.magnetism);  below line 855  Launch file in browser  Press f12  Press right key until on friction zone | Magnet’s magnetism value shall be changed to 0 | Magnet’s magnetism value changed to 0 | Pass |
| 55 | Test that canvas can not have a friction greater than 1 | 1. Import objects and initiate function into JavaScript file 2. Create friction zone with friction greater than 1 3. Create player ball 4. Call initiate function 5. Call JavaScript in HTML file   Add console.log to check friction of zone under ball   1. Open HTML file in browser 2. Press F12 to open developer’s tools and go to console 3. Move ball over friction zone | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let defaultCanvas2 = new Canvas(0, 0, 1500, 3000, 0.01, "black", "orange");  initiate();  console.log  (CanvasArray[0].friction);  below line 1014  Launch file in browser  Press f12  Press right key until on friction zone | Friction value shall be changed to 1 | Friction value changed to 1 | Pass |
| 56 | Test that stationary balls do not move in magnetic interactions with other balls | 1. Import objects and initiate function into JavaScript file 2. Create magnetic stationary ball 3. Create magnetic ball 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);  let Ball2 = new Ball(200, 450, 20, 100, 0, 0, 5, 0, true, "North", 2, false, false, false, "black", "red", false);  initiate();  Launch file in browser | Stationary ball won’t be moved toward other ball | Stationary ball didn’t move toward other ball | Pass |
| 57 | Test that stationary balls do not move in magnetic interactions with magnets | 1. Import objects and initiate function into JavaScript file 2. Create magnetic stationary ball 3. Create magnet 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);  let magnet1 = new Magnet (200, 450, 20, 20, "South", "black", "white");  initiate();  Launch file in browser | Stationary ball won’t be moved toward magnet | Stationary ball didn’t move toward magnet | Pass |
| 58 | Test that stationary balls do not move when in collision | 1. Import objects and initiate function into JavaScript file 2. Create stationary ball 3. Create ball 4. Call initiate function 5. Call JavaScript in HTML file 6. Open HTML file in browser 7. Move other ball into stationary ball | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(110, 450, 20, 100, 0, 0, 5, 0, false, "n/a", 0, true, false, false, "black", "red", false);  let Ball2 = new Ball(200, 450, 20, 100, 0, 0, 5, 0, false, "n/a", 0, false, false, false, "black", "red", false);  initiate();  Launch file in browser  Press left arrow key | Stationary ball won’t be moved after collision with other ball | Stationary ball didn’t move after collision with other ball | Pass |
| 59 | Test balls aren’t drawn when off screen | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create ball 4. Call initiate function 5. Call JavaScript in HTML file 6. Add console.log to check if object was drawn 7. Open HTML file in browser 8. Press F12 to open developer’s tools and go to console 9. Move ball down until other ball is off screen | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(200, 250, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Ball2 = new Ball(110, 250, 20, 100, 0, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);  initiate();  console.log(“ball drawn”) below line 743  Launch file in browser  Press f12  Press down arrow key until ball 2 is off screen | Console shall not display message to say ball drawn when it is off screen | Console did not display message to say ball drawn when it was off screen | Pass |
| 60 | Test blocks aren’t drawn when off screen | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create block 4. Call initiate function 5. Call JavaScript in HTML file 6. Add console.log to check if object was drawn 7. Open HTML file in browser 8. Press F12 to open developer’s tools and go to console 9. Move ball down until block is off screen | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(200, 250, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let block1 = new Block(100, 250, 100, 100, "Black", "Silver");  initiate();  console.log(“block drawn”) below line 778  Launch file in browser  Press f12  Press down arrow key until block is off screen | Console shall not display message to say block drawn when it is off screen | Console did not display message to say block drawn when it was off screen | Pass |
| 61 | Test friction zones aren’t drawn when off screen | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create friction zone 4. Call initiate function 5. Call JavaScript in HTML file 6. Add console.log to check if object was drawn 7. Open HTML file in browser 8. Press F12 to open developer’s tools and go to console 9. Move ball down until friction zone is off screen | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(200, 250, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Friction2 = new FrictionZone(110, 250, 100, 100, "black", "purple", 0.05);  initiate();  console.log(“friction zone drawn”) below line 795  Launch file in browser  Press f12  Press down arrow key until friction zone is off screen | Console shall not display message to say friction zone drawn when it is off screen | Console did not display message to say friction zone drawn when it was off screen | Pass |
| 62 | Test magnets aren’t drawn when off screen | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create magnet 4. Call initiate function 5. Call JavaScript in HTML file 6. Add console.log to check if object was drawn 7. Open HTML file in browser 8. Press F12 to open developer’s tools and go to console 9. Move ball down until magnet is off screen | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(200, 250, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let magnet1 = new Magnet (110, 250, 20, 20, "South", "black", "white");  initiate();  console.log(“magnet drawn”) below line 761  Launch file in browser  Press f12  Press down arrow key until magnet is off screen | Console shall not display message to say magnet drawn when it is off screen | Console did not display message to say magnet drawn when it was off screen | Pass |
| 63 | Test teleporters aren’t drawn when off screen | 1. Import objects and initiate function into JavaScript file 2. Create player ball 3. Create teleporter 4. Call initiate function 5. Call JavaScript in HTML file 6. Add console.log to check if object was drawn 7. Open HTML file in browser 8. Press F12 to open developer’s tools and go to console 9. Move ball down until teleporter is off screen | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Ball1 = new Ball(200, 250, 20, 100, 0, 0, 5, 0, true, "South", 2, false, true, false, "black", "red", false);  let Teleporter1 = new Teleporter (110, 250, 100, 100, 100, 100, "black", "gold", "silver");  initiate();  console.log(“teleporter drawn”) below line 778  Launch file in browser  Press f12  Press down arrow key until teleporter is off screen | Console shall not display message to say teleporter drawn when it is off screen | Console did not display message to say teleporter drawn when it was off screen | Pass |
| 64 | Test draw teleporters function | 1. Import objects and initiate function into JavaScript file 2. Create teleporter 3. Call initiate function 4. Call JavaScript in HTML file 5. Open HTML file in browser | import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';  let Teleporter1 = new Teleporter (110, 250, 100, 100, 100, 100, "black", "gold", "silver");  initiate();  Launch file in browser | Teleporter shall be drawn to specifications given in constructor | Teleporter drawn to specifications given in constructor | Pass |

## User documentation

**Generic 2D Physics Engine in JavaScript**

User DocumentationIntroduction

This physics engine is designed for the use in simple 2D games in which the player may control a ball in order to interact with the game, or simply watch, in which case they will be able to control another ball, this time invisible and that does not interact with the objects of the world, allowing them to view without causing any change.

The design, properties and placement of the objects are entirely up to the user, allowing for a world to be designed to the user’s desire. There are several types of object that may be used, some that will be “flat” against the background, meaning the object may be passed over by the balls, or they may not be, meaning that the ball will stop or deflect off of these objects when it strikes them. Different objects will have different effects on the balls when passed over as well.

In this documentation, a description of each object and all of its properties will be listed as well as a list of requirements when using the physics engine.

Welcome to the usage of this physics engine, and enjoy!

### Types of object

#### Ball

The ball is the only object that can be controlled by the player and has many properties. These properties that must be specified when the object of the ball is created are (in order of entry when creating the object):

* The X and Y coordinates for the centre of the ball for its creation (numbers)
* The radius of the ball (if set to 0 or less, then the radius will be set to 1 unless the ball to be created has the ghost Boolean as true) (number)
* The mass of the ball (if set to 0 or less, then the mass will be set to 0.00001 as a mass greater than 0 was required for calculations to work) (number)
* The X and Y values for the initial velocity of the ball (numbers)
* The maximum velocity the ball can attain from movement controlled by the player (irrelevant if non player controlled ball, value must be entered) (number)
* The acceleration scalar value that will be used when holding down the arrow keys for the ball, a higher value results in a higher acceleration meaning the ball will achieve its maximum velocity faster (if set below 0 then number multiplied by -1 to make it positive) (number)
* Whether the ball is magnetic or not (Boolean)
* The pole of the magnetism (North or South in string)
* The value of the magnetism (higher number results in a stronger connection for that ball with other magnetic objects) (if set below 0 then value set to 0) (number)
* Whether the ball is stationary or not, the ball will not be able to be moved at all if this is set (automatically set to false if anything but true or false entered) (Boolean)
* Whether the ball is controlled by the player or not (only one ball may be controlled at once, if multiple are set then they will be unset automatically and a camera ball will be created which does not interact with the environment. This will also be the case if no ball is set true for player) (automatically set to false if anything but true or false entered) (Boolean)
* Whether the ball will interact with the environment or not, this setting will cause a ball to not interact with any other objects other than the edges of the canvas (automatically set to false if anything but true or false entered) (Boolean)
* The colour of the outline of the ball (string listing a colour)
* The colour of the fill of the ball (string listing a colour)
* Whether the ball has been checked against friction each loop (will automatically set itself to false at start of every loop and at creation in constructor) (Boolean)

Balls can interact with other balls via collisions and magnetism if applicable, blocks by collisions, friction zones by having friction applied to them, the canvas by colliding with the edges and having the friction of the zone applied to them, magnets by magnetism if applicable, and be moved by teleporters if the ball enters their area. The magnetism calculation between two balls is dependent on the masses of the balls, their magnetism values and the distance between their centres. The magnetic velocities will only apply when above a certain value.

Shape, square

Description automatically generated

Figure : Picture of ball object

#### Block

The block object has one purpose and that is as a barrier to moving balls. Blocks are rectangles. Blocks also have several properties to them (in order of entry when creating the object):

* The X and Y coordinates for the top left corner of the block for its creation (numbers)
* The lengths in X and Y directions for the block from the top left corner to its bottom right corner (numbers)
* The colour of the outline of the block (string listing a colour)
* The colour of the fill of the block (string listing a colour)

Blocks can only interact with balls by deflecting them when there is a collision.

Chart

Description automatically generated with medium confidence

Figure : Picture of block object (lighter grey rectangles)

#### Canvas

The canvas object is actually the “world” that the rest of the objects will exist within. It is a rectangle. It has several properties (in order of entry when creating the object):

* The X and Y coordinates for the top left corner of the canvas for its creation (numbers)
* The lengths in X and Y directions for the canvas from the top left corner to its bottom right corner (numbers)
* A friction value that will be applied to moving balls every loop, affecting the velocity of the ball (negative values will cause an increase of speed, acting as a boost to the ball) (number)
* The colour of the outline of the canvas (string listing a colour)
* The colour of the fill of the canvas (string listing a colour)

Its sides will act as the edges of this “world” and so balls will potentially collide with them and will deflect off of them. There can also be a friction value applied to the canvas to avoid the creation of many friction zones. The friction value of the canvas will not apply to a ball when it is on a friction zone.

Shape

Description automatically generated with medium confidence

Figure : Canvas object edges, green being outside and darker grey being inside of canvas

#### Friction Zone

The friction zone object is a rectangle and acts to apply friction to moving balls. It has several properties (in order of entry when creating the object):

* The X and Y coordinates for the top left corner of the canvas for its creation (numbers)
* The lengths in X and Y directions for the canvas from the top left corner to its bottom right corner (numbers)
* The colour of the outline of the canvas (string listing a colour)
* The colour of the fill of the canvas (string listing a colour)
* A friction value that will be applied to moving balls every loop, affecting the velocity of the ball (negative values will cause an increase of speed, acting as a boost to the ball) (number)

When the centre of a ball (chosen to give a 2D representation of a sphere) crosses into the inside of a friction zone, a friction value will be applied and the ball’s velocity will change as appropriate, slowing down or speeding up the ball depending on the friction value given on creation of the object.

Shape, square

Description automatically generated

Figure : Two friction zones pictured, the purple one has a positive friction value, slowing balls down, and the orange has a negative value, speeding balls up

#### Magnet

The magnet object is used as a “flat” (i.e. the ball will not collide with it) version of a magnet, and is circular in shape. It has several properties (in order of entry when creating the object):

* The X and Y coordinates for the centre of the magnet for its creation (numbers)
* The radius of the magnet (if set to 0 or less, then the radius will be set to 1) (number)
* The value of the magnetism (higher number results in a stronger connection for that ball with other magnetic objects) (if set below 0 then value set to 0) (number)
* The pole of the magnetism (North or South in string)
* The colour of the outline of the magnet (string listing a colour)
* The colour of the fill of the magnet (string listing a colour)

The magnet will only act on balls that are magnetic, and magnets will not affect each other as magnets are stationary objects. The magnet acts on the ball by calculating a magnetic velocity between the centres of the ball and the magnet and if the velocity calculated is above a certain value then the velocity will be added to the velocity of the ball, beginning to attract the ball to or repel the ball from the centre of the magnet. The calculation for a magnet depends on the mass of the ball, the magnetism values of the ball and magnet, and the distance between the ball’s and the magnet’s centres.

Chart, square

Description automatically generated

Figure : a magnet (white circle) pulling a ball (red circle) towards its centre, dragging the ball into the path of a teleporter (gold rectangle)

#### Teleporter

The teleporter is “flat” so no collision will take place between it and the balls. The teleporter is represented by a rectangle with a circle in its middle. It has several features (in order of entry when creating the object):

* The X and Y coordinates for the top left corner of the teleporter for its creation (numbers)
* The lengths in X and Y directions for the teleporter from the top left corner to its bottom right corner (numbers)
* The X and Y coordinates for the point the ball should be teleported to (numbers)
* The colour of the outline of the teleporter (string listing a colour)
* The colour of the fill of the teleporter (string listing a colour)
* The colour of the circle in the centre of the teleporter (string listing a colour)

The teleporter will move any ball (that does not have the ghost Boolean as true) to the specified point when the centre of the ball crosses into its area.

Shape

Description automatically generated

Figure : teleporter represented by a circle in a square

### Using the physics engine

#### Requirements

When using the physics engine to create any sort of game or simulation, it is very important to note:

* The “initiate” function and “Canvas” object must be imported from ‘./physics.js’, anything else is optional, but must also be imported if it is to be used
* The “initiate” function must be used at the end of the JavaScript file containing the code created in order to have the physics engine take effect
* There must always be a canvas object created and there can only be one created per simulation
* When creating an object other than the canvas, these objects must be within the boundaries of the canvas in their entirety
* There cannot be zero or more than one player ball, only one may exist. If there isn’t one or more than one, a camera ball will be created and any player balls will be removed as player balls and made into regular balls
* When creating an object of any type, ALL properties of its constructor must be filled in

#### Steps to use

This physics engine has only a few easy steps:

1. Import the initiate function and the objects required from ‘./physics.js’
2. Create the canvas object to required specifications
3. Create the other objects that are desired for the game/simulation
4. Call the initiate function

#### Example code

When these rules are obeyed the physics engine will work with the objects that are created. After this it is up to the user to create the objects as they desire and create the games and simulations with them. Here is an example code for a game where a player can control a ball to round up other balls:

Code block : example code for creation

import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';

*let* defaultCanvas = new Canvas(0, 0, 1200,1200, 0.01, "black", "orange");

*let* Friction1 = new FrictionZone(500, 300, 200, 200, "black", "purple", 0.1);

*let* block1 = new Block(400, 0, 100, 400, "Black", "Silver");

*let* block2 = new Block(700, 0, 100, 400, "Black", "Silver");

*let* block3 = new Block(500, 500, 200, 100, "Black", "Silver");

*let* Teleporter1 = new Teleporter (525, 600, 150, 150, 600, 150, "black", "gold", "silver");

*let* Ball1 = new Ball(110, 450, 20, 10, 0, 0, 5, 0.1, false, "/", 0, false, true, false, "black", "red", false);

*let* Ball2 = new Ball(600, 50, 30, 10, 0, 0, 0, 0, true, "North", 3, true, false, false, "black", "turquoise", false);

*let* Ball3 = new Ball(600, 250, 30, 10, 0, 0, 0, 0, true, "North", 3, true, false, false, "black", "turquoise", false);

*let* Ball4 = new Ball(1000, 450, 20, 10, -5, -3, 5, 1, true, "South", 2, false, false, false, "black", "yellow", false);

*let* Ball5 = new Ball(300, 700, 20, 15, 5, 2, 5, 1, true, "South", 2, false, false, false, "black", "yellow", false);

*let* Ball6 = new Ball(900, 950, 20, 5, -3, 2, 5, 1, true, "South", 2, false, false, false, "black", "yellow", false);

*let* Ball7 = new Ball(1000, 250, 20, 30, 2, -4, 5, 1, true, "South", 2, false, false, false, "black", "yellow", false);

*let* Ball8 = new Ball(100, 150, 20, 20, 0, 0, 5, 1, true, "South", 2, false, false, false, "black", "yellow", false);

initiate();

Here the canvas is created, as required. It has a width and height of 1200 pixels. It will have a friction value of 0.01, meaning that the friction value will be multiplied by 0.99 (1-0.01) every loop. It will have a black outline and an orange fill.

Next to be created is a friction zone, starting at (500, 300) and with a height and width of 200 pixels. It has a friction value of 0.1, providing a higher friction value than the canvas, meaning it will slow balls much faster. It has a black outline and a purple fill.

After this three blocks are created. Block1 starts at (400, 0) and block2 at (700, 0), meaning they will be against the top edge of the canvas. They then both extend 100 pixels to the right and 400 pixels down from this point. They both have a black outline and silver fill. Block3 starts at (500, 500), and is 200 pixels wide and 100 pixels in height, also with a black outline and silver fill.

Next to be created is a teleporter, with a start point of (525, 600) and has a height and width of 150 pixels. This teleporter will move balls that move over it to (600, 150), putting them between the blocks and friction zones. It has a black outline, gold fill, and silver circle in its centre.

After this the balls are created. Ball1 is the player ball meaning that it is the one to be controlled by the player with the arrow keys. It has a starting centre of (110, 450), radius of 20 pixels, mass of 10, starting velocity of (0, 0), maximum velocity of 5, an acceleration scalar of 5, false for magnetic, a value of “/” for the pole since it is not magnetic (this will not matter as the pole won’t be considered due to a lack of magnetic property), a value of 0 for magnetism (again this will not matter as the magnetism won’t be considered due to a lack of magnetic property), a value of false for stationary so it can move, a value of true for player so that it is a player ball, a value of false for ghost so that it will interact with other objects, a black outline, red fill and a value of false for the friction check.

The next two balls (Ball2 and Ball3) have several important differences from Ball1. The balls are in different positions, have different radii, have different velocities etc. However what is most important to note for these balls compared to Ball1 is that they have a value of true for their magnetic property, “North” for their pole property and a value of 3 for their magnetism. This means that these balls will have an effect on other magnetic balls (so Ball1 is not included here.) They also have a value of true for their stationary value, meaning that they will not affect each other even though they have the same pole, meaning they would otherwise repel each other. They also both have a value of false for the player property. They each have a black outline and turquoise fill.

The other five balls (Ball4, Ball5, Ball6, Ball7 and Ball8) all have several important differences from the other balls. They all have different positions too, different velocities in X and Y directions and each has the same radius as Ball1. Each ball is magnetic like Ball2 and Ball3, however they all have the same pole property value of “South.” They also have a value of false for the stationary property. This means that they can be moved about, so they will repel each other due to all sharing the same magnetic pole. They will also be attracted to Ball2 and Ball3 due to having opposite poles to these two balls. They can also be moved by Ball1 colliding with them.

With all of these objects in place, it is then possible for the player of this game to control Ball1 and use it to knock the yellow balls (Ball4-8) into the teleporter, teleporting them into the “pen” created by the blocks and friction zone, where they will be trapped by Ball2 and Ball3. If the player accidentally moves into the teleporter with their ball, it is possible to leave by moving over the friction zone, as the player ball will not be stuck by the magnets due to the lack of magnetic property on it.

Shape, square

Description automatically generated

Figure : Player ball about to knock yellow ball into teleporter to have it trapped in "pen"

Shape, bubble chart, square

Description automatically generated

Figure : yellow ball now trapped with magnetic turquoise balls after being knocked into teleporter

## Code blocks

Code block : index.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <style>

        #canvas{

*border*: 1px solid black;

*display*: block;

*margin*: 0 auto;

*background*: green;

        }

    </style>

    <title>Home!</title>

</head>

<body>

    <a href="ballroundup.html"> Ball Round Up game!</a>

    <a href="obstaclecourse.html"> Obstacle Course!</a>

    <canvas id="canvas" tabindex = 0 width="1500" height="900"></canvas>

    <script type = "module" src="objectCreation.js"></script>

</body>

</html>

Code block : engineConfig.js

*const* canvas = document.getElementById('canvas');

*const* ctx = canvas.getContext('2d');

*var* width = canvas.width;

*var* height = canvas.height;

export { canvas, ctx, width, height };

Code block : physics.js

//importing variables from engineconfig to do with canvas properties

import { canvas, ctx, width, height } from './engineConfig.js';

//creation of arrays to hold objects of each class

*const* BallArray = [];

*const* PlayerBallArray = [];

*const* FrictionSquareArray = [];

*const* MagnetArray = [];

*const* BlockArray = [];

*const* CanvasArray = [];

*const* TeleportArray = [];

//booleans used for ball movement for players

*let* moveLeft = false;

*let* moveRight = false;

*let* moveDown = false;

*let* moveUp = false;

//vector class used for storing x and y directions and coordinates for variables in classes and objects

*class* Vector{

    //constructor for vector class

*constructor*(*x\_vector*, *y\_vector*){

        this.x\_vector = *x\_vector*;

        this.y\_vector = *y\_vector*;

    }

    //method to get magnitude of variable

    magnitude(){

        return Math.sqrt(this.x\_vector\*\*2 + this.y\_vector\*\*2);

    }

    //method to get unit vector of vector

    unit(){

        //if the magnitude will be 0, sets unit vector as 0 as division by 0 will cause problems

        if (this.magnitude() === 0){

            return new Vector (0,0);

        } else {

            return new Vector (this.x\_vector/this.magnitude(), this.y\_vector/this.magnitude());

        }

    }

    //method to get dot product of vector

    dotproduct(*vector\_2*){

        return ((this.x\_vector\**vector\_2*.x\_vector)+(this.y\_vector\**vector\_2*.y\_vector));

    }

}

//ball class used for creation of ball objects

*class* Ball{

    //constructor used to construct ball object, with corrections for any potential incorrect entries in construction of ball

*constructor*(*xpoint*, *ypoint*, *radius*, *mass*, *velocityx*, *velocityy*, *maxvelocity*, *accelerationScalar*, *magnetic*, *pole*, *magnetism*,

*stationary*, *player*, *ghost*, *outline*, *fill*, *frictionCheck*){

        //setting centre as vector so that both x and y positions can be held

        this.centre = new Vector(*xpoint*, *ypoint*);

        //sets radius as 1 so ball is visible if the radius entred is less than or equal to 0 unless the ball is a ghost property

        if (*radius*<=0 && *ghost* == false){

            this.radius = 1;

        } else {

            this.radius = *radius*;

        }

        //if statement to catch if mass is less than or equal to 0, negative values are impossible

        //and a value of 0 will cause problems with collision resolution

        if (*mass*<=0){

            this.mass = 0.00001;

        } else{

            this.mass = *mass*;

        }

        //if statement to ensure magnetic is true or false

        if (*magnetic* !== false && *magnetic* !== true){

*magnetic* = false;

        }

        this.magnetic = *magnetic*;

        //if statement to ensure magnetism isn't below 0

        if (*magnetism*<0){

*magnetism* = 0;

        }

        //sets pole and magnetism only if the magnetic value is true

        if (this.magnetic == true){

            this.pole = *pole*;

            this.magnetism = *magnetism*;

        } else {

            this.pole = "N/A";

            this.magnetism = 0;

        }

        //sets velocity as vector object so it can contain x and y components

        this.velocity = new Vector(velocityx,velocityy);

        this.maxvelocity = maxvelocity;

        //sets acceleration scalar to positive of value entered if negative value entered

        if (accelerationScalar <0){

            accelerationScalar \*= -1;

        }

        this.accelerationScalar = accelerationScalar;

        //acceleration vector set to 0

        this.accelerationVector = new Vector(0,0);

        //if statement to ensure stationary is true or false

        if (stationary!== true && stationary!== false){

            stationary = false;

        }

        this.stationary = stationary;

        //if statement to ensure player is true or false

        if (player!== true && player!== false){

            player = false;

        }

        this.player = player;

        this.outline = outline;

        this.fill = fill;

        //if statement to correctly set camera for the ball when player and adds to player array

        if (this.player == true){

            this.canvasCentre = new Vector (width/2,height/2);

            PlayerBallArray.push(this);

        } else{

            this.canvasCentre = new Vector (0,0);

        }

        //if statement to ensure ghost is true or false

        if (ghost!== true && ghost!== false){

            ghost = false;

        }

        this.ghost = ghost;

        //sets frictioncheck to false if not

        if (frictionCheck !== false){

            frictionCheck = false;

        }

        this.frictionCheck = frictionCheck

        //adds object to ball array

        BallArray.push(this);

    }

}

//friction zone class for creation of friction zone object

*class* FrictionZone{

    //constructor used to create friction zone object with potential corrections for properties

*constructor*(*upperxpoint*, *upperypoint*, *xlength*, *ylength*, *outline*, *fill*, *friction*){

        //sets upper point x and y coordinates to vector object

        this.upperpoint = new Vector (upperxpoint, upperypoint);

        //sets drawing coordinates to 0,0 as default

        this.canvasStart = new Vector (0,0);

        //if statement to set length to at least 1 for each direction

        if (xlength<=0){

            xlength = 1;

        }

        if (ylength<=0){

            ylength = 1;

        }

        //sets length values to vector for x and y components

        this.length = new Vector (xlength, ylength);

        this.outline = outline;

        this.fill = fill;

        //if statement so that friction value can not be greater than 1

        if (friction>1){

            friction = 1;

        }

        this.friction = friction;

        //adds friction zone to array

        FrictionSquareArray.push(this);

    }

}

//magnet class for creation of magnet objects

*class* Magnet{

    //constructor used to create magnet objects with potential corrections if required for properties

*constructor*(*xpoint*, *ypoint*, *radius*, *magnetism*, *pole*, *outline*, *fill*){

        //sets centre values in vector as there are x and y components

        this.centre = new Vector (xpoint, ypoint);

        //sets canvas centre as 0,0 for default as will be corrected based on player/camera ball

        this.canvasCentre = new Vector(0,0);

        //if statement to stop radius being 0 or less

        if (radius<=0){

            this.radius = 1;

        } else {

            this.radius = radius;

        }

        this.radius = radius;

        //if statement to stop magnetism being less than 0

        if (magnetism<0){

            magnetism = 0;

        }

        this.magnetism = magnetism;

        this.pole = pole;

        this.outline = outline;

        this.fill = fill;

        //adds magnet object to magnet array

        MagnetArray.push(this);

    }

}

//block class for creation of block object

*class* Block {

    //constructor used to create block objects with potential corrections

*constructor*(*xstart*, *ystart*, *xlength*, *ylength*, *outline*, *fill*){

        //vector object used to hold starting points for x and y coordinates

        this.start = new Vector (xstart, ystart);

        //canvas start set as 0,0 by default before being changed to centre round player/camera

        this.canvasStart = new Vector (0,0);

        //if statement to stop length being less than 0 for x and y

        if (xlength<=0){

            xlength = 1;

        }

        if (ylength<=0){

            ylength = 1;

        }

        //length values stored in vector due to x and y coordinates

        this.length = new Vector (xlength, ylength);

        this.outline = outline;

        this.fill = fill;

        //block added to block array

        BlockArray.push(this);

    }

}

//canvas class used to construct canvas objects

*class* Canvas {

    //constructor used to create canvas object with potential corrections

*constructor*(*xstart*, *ystart*, *xlength*, *ylength*, *friction*, *outline*, *fill*){

        //canvas start set as vector with x and y coordinates

        this.universalStart = new Vector(xstart, ystart);

        //canvas start set as 0,0 by default before being changed to centre round player/camera

        this.canvasStart = new Vector(0,0);

        //if statement so length of canvas is not 0 or less

        if (xlength<=0){

            xlength = 1;

        }

        if (ylength<=0){

            ylength = 1;

        }

        //length set as vector due to having x and y coordinates

        this.length = new Vector(xlength, ylength);

        if (friction>1){

            friction = 1;

        }

        this.friction = friction;

        this.outline = outline;

        this.fill = fill;

        //canvas added to array for access

        CanvasArray.push(this);

    }

}

//teleporter class to construct teleporter objects

*class* Teleporter {

    //constructor method to create teleporter object with potential corrections

*constructor*(*xstart*, *ystart*, *xlength*, *ylength*, *xteleport*, *yteleport*, *outline*, *fill*, *circle*){

        //start point set as vector to hold x and y coordinates

        this.start = new Vector (xstart, ystart);

        //canvas start set as 0,0 by default before being changed to centre round player/camera

        this.canvasStart = new Vector (0,0);

        //ensure length of at least 1 for both directions

        if (xlength<=0){

            xlength = 1;

        }

        if (ylength<=0){

            ylength = 1;

        }

        //length held as vector as it has x and y component

        this.length = new Vector(xlength, ylength);

        //teleport point held as vector with x and y component

        this.teleportPoint = new Vector (xteleport, yteleport);

        //canvas point set as 0,0 by default before being corrected for player/camera

        this.canvasTeleportPoint = new Vector(0,0);

        this.outline = outline;

        this.fill = fill;

        this.circle = circle;

        //circle centre set as vector with x and y components

        this.circleCentre = new Vector (this.start.x\_vector+(xlength/2),this.start.y\_vector+(ylength/2));

        //canvas centre set as 0,0 by default to be corrected based on player/camera

        this.canvasCentre = new Vector (0,0);

        //if statement so that radius will be pased on shorter component of rectangle of teleporter

        if (xlength>ylength){

            this.radius = ylength/5;

        } else {

            this.radius = xlength/5;

        }

        //teleporter added to teleporter array

        TeleportArray.push(this);

    }

    //draw teleporter method added as it uses both circle and rectangle drawing

    drawTeleporter(){

        drawRectangle(this);

        ctx.beginPath();

        ctx.arc(this.canvasCentre.x\_vector, this.canvasCentre.y\_vector, this.radius, 0, 2\*Math.PI);

        ctx.fillStyle = this.circle;

        ctx.fill();

        ctx.closePath();

    }

}

//draw rectangle function takes in rectangle objects and displays them on canvas

*function* drawRectangle (*object\_1*){

    ctx.beginPath();

    ctx.rect(object\_1.canvasStart.x\_vector, object\_1.canvasStart.y\_vector, object\_1.length.x\_vector, object\_1.length.y\_vector);

    ctx.strokestyle = object\_1.outline;

    ctx.stroke();

    ctx.fillStyle = object\_1.fill;

    ctx.fill();

    ctx.closePath();

}

//draw circle function takes in circle objects and displays them on canvas

*function* drawCircle (*object\_1*){

    ctx.beginPath();

    ctx.arc(object\_1.canvasCentre.x\_vector, object\_1.canvasCentre.y\_vector, object\_1.radius, 0, 2\*Math.PI);

    ctx.strokeStyle = object\_1.outline;

    ctx.stroke();

    ctx.fillStyle = object\_1.fill;

    ctx.fill();

    ctx.closePath();

}

//check collision function will take in two balls and check for penetration and if it exists then it will separate the balls

*function* checkCollision (*ball\_1*, *ball\_2*){

    //distance between two centres in x and y directions

*let* distance\_x = ball\_1.centre.x\_vector-ball\_2.centre.x\_vector;

*let* distance\_y = ball\_1.centre.y\_vector-ball\_2.centre.y\_vector;

    //magnitude of distance to give a scalar number distance between two centres

*let* distance\_magnitude = Math.sqrt(distance\_x\*\*2+distance\_y\*\*2);

    if (ball\_1.radius+ball\_2.radius>distance\_magnitude){

        //overlap between balls (if 0 then no overlap)

*let* diff = ball\_1.radius+ball\_2.radius - distance\_magnitude;

        //calculating amount of distance each ball must be moved to make them no longer overlap but still touch

        //calculated by taking the total distance in each plane and dividing it by the total distance magnitude and multiplying

        //it by the overlap divided by two so that it takes the fact they are each moving half the distance into account

        //each is then added to ball 1 and subtracted from ball 2 in order to ensure each ball moves an equal amount

        //from the other, producing no overlap in the end

*let* move\_x = (distance\_x/distance\_magnitude)\*(diff/2);

*let* move\_y = (distance\_y/distance\_magnitude)\*(diff/2);

        //if statement to check that the balls only move if they aren't stationary

        if (ball\_1.stationary == false && ball\_2.stationary == false){

            ball\_1.centre.x\_vector += move\_x;

            ball\_1.centre.y\_vector += move\_y;

            ball\_2.centre.x\_vector -= move\_x;

*ball\_2*.centre.y\_vector -= move\_y;

        } else if (*ball\_1*.stationary == true && *ball\_2*.stationary == false) {

*ball\_2*.centre.x\_vector -= 2\*move\_x;

*ball\_2*.centre.y\_vector -= 2\*move\_y;

        } else {

*ball\_1*.centre.x\_vector += 2\*move\_x;

*ball\_1*.centre.y\_vector += 2\*move\_y;

        }

    }

}

//function to resolve collisions between two balls

*function* ballVectors(*ball\_1*, *ball\_2*){

    //distance between two centres in x and y directions

*let* distance\_x = *ball\_1*.centre.x\_vector-*ball\_2*.centre.x\_vector;

*let* distance\_y = *ball\_1*.centre.y\_vector-*ball\_2*.centre.y\_vector;

    //magnitude of distance to give a scalar number distance between two centres

*let* distance\_magnitude = Math.sqrt(distance\_x\*\*2+distance\_y\*\*2);

    if (*ball\_1*.radius+*ball\_2*.radius>=distance\_magnitude){

        //normal vector of collision found from taking centre of circle 1 from centre of circle 2

*let* normalVector = new Vector(*ball\_2*.centre.x\_vector - *ball\_1*.centre.x\_vector,

*ball\_2*.centre.y\_vector - *ball\_1*.centre.y\_vector);

        //unit vector of normal found from dividing both components of vector by magnitude

*let* unitNormalVector = new Vector(normalVector.x\_vector/normalVector.magnitude(),

        normalVector.y\_vector/normalVector.magnitude());

        //get unit tangent vector of collision

*let* unitTangentVector = new Vector(unitNormalVector.y\_vector\*-1, unitNormalVector.x\_vector);

        //get initial ball 1 normal direction vector by taking dot product of ball's velocity and that of the unit normal vector

*let* initialBall1Normal = *ball\_1*.velocity.dotproduct(unitNormalVector);

        //get initial ball 1 tangent direction vector by taking dot product of ball's velocity and that of unit tangent vector

*let* ball1Tangent = *ball\_1*.velocity.dotproduct(unitTangentVector);

*let* initialBall2Normal = *ball\_2*.velocity.dotproduct(unitNormalVector);

*let* ball2Tangent = *ball\_2*.velocity.dotproduct(unitTangentVector);

        //final ball 1 normal velocity taken from formula

*let* finalBall1Normal = ((initialBall1Normal\*(*ball\_1*.mass-*ball\_2*.mass))+(2\**ball\_2*.mass\*initialBall2Normal))/

        (*ball\_1*.mass+*ball\_2*.mass);

*let* finalBall2Normal = ((initialBall2Normal\*(*ball\_2*.mass-*ball\_1*.mass))+(2\**ball\_1*.mass\*initialBall1Normal))/

        (*ball\_1*.mass+*ball\_2*.mass);

        //calculate new ball normal velocity vectors by multiplying each vector of unit normal vector by

        //the final ball normal velocity

*let* newBall1NormalVectorVelocity = new Vector (unitNormalVector.x\_vector\*finalBall1Normal,

        unitNormalVector.y\_vector\*finalBall1Normal);

*let* newBall2NormalVectorVelocity= new Vector (unitNormalVector.x\_vector\*finalBall2Normal,

        unitNormalVector.y\_vector\*finalBall2Normal);

        //calculate new ball tangent velocity vectors by multiplying each vector of unit tangent vector by

        //the final ball tangent velocity

*let* newBall1TangentVectorVelocity = new Vector (unitTangentVector.x\_vector\*ball1Tangent,

        unitTangentVector.y\_vector\*ball1Tangent);

*let* newBall2TangentVectorVelocity = new Vector (unitTangentVector.x\_vector\*ball2Tangent,

        unitTangentVector.y\_vector\*ball2Tangent);

        //calculate each ball's velocities by adding that of the normal vector and tangent vector velocity

        if (*ball\_1*.stationary == false){

*ball\_1*.velocity.x\_vector = newBall1NormalVectorVelocity.x\_vector+newBall1TangentVectorVelocity.x\_vector;

*ball\_1*.velocity.y\_vector = newBall1NormalVectorVelocity.y\_vector+newBall1TangentVectorVelocity.y\_vector;

        }

        if (*ball\_2*.stationary == false){

*ball\_2*.velocity.x\_vector = newBall2NormalVectorVelocity.x\_vector+newBall2TangentVectorVelocity.x\_vector;

*ball\_2*.velocity.y\_vector = newBall2NormalVectorVelocity.y\_vector+newBall2TangentVectorVelocity.y\_vector;

        }

    };

}

//function to add acceleration to the velocity

*function* addAcceleration(*ball\_1*){

    //if statement to check acceleration vector is enough to be added and not negligible

    if (*ball\_1*.accelerationVector.x\_vector>0.01 || *ball\_1*.accelerationVector.x\_vector<-0.01){

        //if statement so that acceleration is added when the velocity is below max velocity

        if (*ball\_1*.accelerationVector.x\_vector>0){

            if (*ball\_1*.velocity.x\_vector<*ball\_1*.maxvelocity){

*ball\_1*.velocity.x\_vector += *ball\_1*.accelerationVector.x\_vector;

            }

        }

        if (*ball\_1*.accelerationVector.x\_vector<0){

            if (*ball\_1*.velocity.x\_vector>-1\**ball\_1*.maxvelocity){

*ball\_1*.velocity.x\_vector += *ball\_1*.accelerationVector.x\_vector;

            }

        }

    }

    //if statement to check acceleration vector is enough to be added and not negligible

    if (*ball\_1*.accelerationVector.y\_vector>0.01 || *ball\_1*.accelerationVector.y\_vector<-0.01){

        //if statement so that acceleration is added when the velocity is below max velocity

        if (*ball\_1*.accelerationVector.y\_vector>0){

            if (*ball\_1*.velocity.y\_vector<*ball\_1*.maxvelocity){

*ball\_1*.velocity.y\_vector += *ball\_1*.accelerationVector.y\_vector;

            }

        }

        if (*ball\_1*.accelerationVector.y\_vector<0){

            if (*ball\_1*.velocity.y\_vector>-1\**ball\_1*.maxvelocity){

*ball\_1*.velocity.y\_vector += *ball\_1*.accelerationVector.y\_vector;

            }

        }

    }

}

//function to add velocity to the placement of the ball

*function* addVelocity(*ball\_1*){

    //velocities added to position of ball to move it

*ball\_1*.centre.x\_vector += *ball\_1*.velocity.x\_vector;

*ball\_1*.centre.y\_vector += *ball\_1*.velocity.y\_vector;

}

//function to check for friction zones

*function* checkFriction(*ball\_1*, *square\_1*){

    //only checks friction zone if the friction check is false

    if (*ball\_1*.frictionCheck == false){

        //if statement that the friction is only checked when the velocity is high enough so that the friction check

        //is not checked unnecessarily to help performance

        if ((*ball\_1*.velocity.x\_vector>0.005||*ball\_1*.velocity.x\_vector<-0.005)||

        (*ball\_1*.velocity.y\_vector>0.005||*ball\_1*.velocity.y\_vector<-0.005)){

            //checks that the centre of the ball (the point that would be touching the surface of the friction zone if ball

            //is considered a 2d representation of a sphere) is within the coordinates of the zone and applies friction of

            //zone to the ball

            if ((*ball\_1*.centre.x\_vector>=*square\_1*.upperpoint.x\_vector&&*ball\_1*.centre.x\_vector<=

                (*square\_1*.upperpoint.x\_vector+*square\_1*.length.x\_vector)) &&

                (*ball\_1*.centre.y\_vector>=*square\_1*.upperpoint.y\_vector&&*ball\_1*.centre.y\_vector<=

                (*square\_1*.upperpoint.y\_vector+*square\_1*.length.y\_vector))){

                    //reduces velocity with friction factor

*ball\_1*.velocity.x\_vector \*= 1 - *square\_1*.friction;

*ball\_1*.velocity.y\_vector \*= 1 - *square\_1*.friction;

                    //sets friction check as true for this loop for this ball as ball has already had friction calculations applied

*ball\_1*.frictionCheck = true;

            }

        }

    }

}

//function to check if the ball is against the edges

*function* checkEdges(*ball\_1*, *canvas\_1*){

    //checks that ball isn't past border of screen and reverses vector dependent on side it hits

    //also brings ball back to edge if it were to cross the border due to the small jumps the ball

    //actually makes to simulate movement

    if (((*ball\_1*.centre.x\_vector-*ball\_1*.radius) <= *canvas\_1*.universalStart.x\_vector ||

*ball\_1*.centre.x\_vector >= (*canvas\_1*.universalStart.x\_vector + *canvas\_1*.length.x\_vector-*ball\_1*.radius))&&

     ((*ball\_1*.centre.y\_vector-*ball\_1*.radius)>=(*canvas\_1*.universalStart.y\_vector-(*canvas\_1*.length.y\_vector/20)) &&

     (*ball\_1*.centre.y\_vector+*ball\_1*.radius)<=(*canvas\_1*.universalStart.y\_vector+*canvas\_1*.length.y\_vector+(*canvas\_1*.length.y\_vector/20)))){

        if (*ball\_1*.centre.x\_vector<=*ball\_1*.radius){

*ball\_1*.centre.x\_vector = *canvas\_1*.universalStart.x\_vector+*ball\_1*.radius;

        } else if (*ball\_1*.centre.x\_vector >= (*canvas\_1*.length.x\_vector-*ball\_1*.radius)){

*ball\_1*.centre.x\_vector = *canvas\_1*.length.x\_vector - *ball\_1*.radius;

        }

*ball\_1*.velocity.x\_vector = -1 \* *ball\_1*.velocity.x\_vector;

    }

    //checks that ball isn't past border of screen and reverses vector dependent on side it hits

    //also brings ball back to edge if it were to cross the border due to the small jumps the ball

    //actually makes to simulate movement

    if (((ball\_1.centre.y\_vector-ball\_1.radius) <= canvas\_1.universalStart.y\_vector ||

    ball\_1.centre.y\_vector >= (canvas\_1.universalStart.y\_vector + canvas\_1.length.y\_vector-ball\_1.radius))&&

    ((ball\_1.centre.x\_vector-ball\_1.radius)>=(canvas\_1.universalStart.x\_vector-(canvas\_1.length.x\_vector/20)) &&

    (ball\_1.centre.x\_vector+ball\_1.radius)<=(canvas\_1.universalStart.x\_vector+canvas\_1.length.x\_vector+(canvas\_1.length.x\_vector/20)))){

        if (ball\_1.centre.y\_vector<=ball\_1.radius){

            ball\_1.centre.y\_vector = ball\_1.radius;

        } else if (ball\_1.centre.y\_vector >= (canvas\_1.length.y\_vector-ball\_1.radius)){

            ball\_1.centre.y\_vector = canvas\_1.length.y\_vector - ball\_1.radius;

        }

        ball\_1.velocity.y\_vector = ball\_1.velocity.y\_vector \* -1;

    }

}

//function for checking magnetism in objects

*function* magnetismCalc(*object\_1*, *object\_2*){

    //distances calculated in x and y as centres from objects

*let* distance\_x = object\_2.centre.x\_vector-object\_1.centre.x\_vector;

*let* distance\_y = object\_2.centre.y\_vector-object\_1.centre.y\_vector;

    //magnitude of distance to give a scalar number distance between two centres

*let* distance\_magnitude = Math.sqrt(distance\_x\*\*2+distance\_y\*\*2);

    //magnetic inverse square given as 1/distance^2 to have distance between magnetic objects

    //affect value of magnetic velocities

*let* magnetInverseSquare = 1/(distance\_magnitude\*\*2);

    if (object\_1.pole === object\_2.pole){

        //multiplies value by -1 so that balls repel if poles are the same

        magnetInverseSquare \*= -1;

    }

    //magnetic velocity is a vector to hold x and y components

*let* magneticVelocity = new Vector(distance\_x\*magnetInverseSquare\*object\_1.magnetism\*object\_2.magnetism,

        distance\_y\*magnetInverseSquare\*object\_1.magnetism\*object\_2.magnetism);

    //magnetic velocity returned to other methods for use in calculations between 2 balls or ball and magnet

    return magneticVelocity;

}

//function to check magnetism against balls currently moving

*function* checkMagnets(*ball\_1*, *magnet\_1*){

    //magnetic velocity calculated by method

*let* magneticVelocity = magnetismCalc(ball\_1, magnet\_1);

    //mass of ball taken into consideration for calculation of velocity

    magneticVelocity.x\_vector \*= 1/ball\_1.mass;

    magneticVelocity.y\_vector \*= 1/ball\_1.mass;

    //magnetic velocity only added if value not negligible

    if ((magneticVelocity.x\_vector>0.03 || magneticVelocity.x\_vector<-0.03)&&ball\_1.stationary == false){

        ball\_1.velocity.x\_vector += magneticVelocity.x\_vector;

    }

    if ((magneticVelocity.y\_vector>0.03 || magneticVelocity.y\_vector<-0.03)&&ball\_1.stationary == false){

        ball\_1.velocity.y\_vector += magneticVelocity.y\_vector;

    }

}

//function to check two magnetic balls against each other

*function* checkMagneticBalls(*ball\_1*, *ball\_2*){

    //magnetic velocity calculated from method

*let* magneticVelocity = magnetismCalc(ball\_1, ball\_2);

    //magnetic velocity for balls calculated taking into consideration the mass of both balls along

    //with the other magnetic velocity calculated

*let* magneticVelocity\_1 = new Vector (magneticVelocity.x\_vector\*(ball\_2.mass/(ball\_1.mass+ball\_2.mass)),

    magneticVelocity.y\_vector\*(ball\_2.mass/(ball\_1.mass+ball\_2.mass)));

*let* magneticVelocity\_2 = new Vector (magneticVelocity.x\_vector\*(ball\_1.mass/(ball\_1.mass+ball\_2.mass)),

    magneticVelocity.y\_vector\*(ball\_2.mass/(ball\_1.mass+ball\_2.mass)));

    //if magnetic velocity in x not negligible then it is added to velocities of balls

    if (magneticVelocity.x\_vector>0.03 || magneticVelocity.x\_vector<-0.03){

        //second check to make sure magnetic velocity in x for ball 1 specifically is not negligible

        if (magneticVelocity\_1.x\_vector>0.03 || magneticVelocity\_1.x\_vector<-0.03){

            //adds the velocity with both balls' mass considered only if both aren't stationary,

            //otherwise adds original magnetic velocity without mass consideration if ball 1 isn't

            //stationary

            if (*ball\_2*.stationary == false && *ball\_1*.stationary== false){

*ball\_1*.velocity.x\_vector += magneticVelocity\_1.x\_vector;

            } else if (*ball\_1*.stationary == false) {

*ball\_1*.velocity.x\_vector += magneticVelocity.x\_vector;

            }

        }

        //second check to make sure magnetic velocity in x for ball 2 specifically is not negligible

        if (magneticVelocity\_2.x\_vector>0.03 || magneticVelocity\_2.x\_vector<-0.03){

            //adds the velocity with both balls' mass considered only if both aren't stationary,

            //otherwise adds original magnetic velocity without mass consideration if ball 2 isn't

            //stationary

            if (*ball\_1*.stationary == false && *ball\_2*.stationary == false){

*ball\_2*.velocity.x\_vector -= magneticVelocity\_2.x\_vector;

            } else if (*ball\_2*.stationary == false) {

*ball\_2*.velocity.x\_vector -= magneticVelocity.x\_vector;

            }

        }

    }

    //if magnetic velocity in y not negligible then it is added to velocities of balls

    if (magneticVelocity.y\_vector>0.03||magneticVelocity.y\_vector<-0.03){

        //second check to make sure magnetic velocity in y for ball 1 specifically is not negligible

        if (magneticVelocity\_1.y\_vector>0.03 || magneticVelocity\_1.y\_vector<-0.03){

            //adds the velocity with both balls' mass considered only if both aren't stationary,

            //otherwise adds original magnetic velocity without mass consideration if ball 1 isn't

            //stationary

            if (*ball\_2*.stationary == false && *ball\_1*.stationary == false){

*ball\_1*.velocity.y\_vector += magneticVelocity\_1.y\_vector;

            } else if (*ball\_1*.stationary == false) {

*ball\_1*.velocity.y\_vector += magneticVelocity.y\_vector;

            }

        }

        //second check to make sure magnetic velocity in y for ball 2 specifically is not negligible

        if (magneticVelocity\_2.y\_vector>0.03 || magneticVelocity\_2.y\_vector<-0.03){

            //adds the velocity with both balls' mass considered only if both aren't stationary,

            //otherwise adds original magnetic velocity without mass consideration if ball 2 isn't

            //stationary

            if (*ball\_1*.stationary == false && *ball\_2*.stationary == false){

*ball\_2*.velocity.y\_vector -= magneticVelocity\_2.y\_vector;

            } else if (*ball\_2*.stationary == false) {

*ball\_2*.velocity.y\_vector -= magneticVelocity.y\_vector;

            }

        }

    }

}

//function to check if ball collides with a block and resolve the collision if so

*function* checkBlocks(*ball\_1*, *block\_1*){

    //checks that the ball is within the limits of the block and then checks which side is being hit

    //with the velocity in the x direction needing to be positive to strike the left side, and also

    //hitting within a range of the left side for the purpose of ensuring that it does not consider

    //the ball to have hit the left and the top or the left and the bottom when the ball hits either

    //the top or the bottom of the block while travelling right, as this would result in the ball reversing

    //both x and y velocities

    if ((((*ball\_1*.centre.x\_vector+*ball\_1*.radius)>=*block\_1*.start.x\_vector)

    &&(*ball\_1*.centre.x\_vector+*ball\_1*.radius)<=(*block\_1*.start.x\_vector+(*block\_1*.length.x\_vector/10)))

    && *ball\_1*.velocity.x\_vector>0

    &&(((*ball\_1*.centre.y\_vector+*ball\_1*.radius)>=*block\_1*.start.y\_vector)

    &&(*ball\_1*.centre.y\_vector-*ball\_1*.radius)<=(*block\_1*.start.y\_vector+*block\_1*.length.y\_vector))){

*ball\_1*.centre.x\_vector = *block\_1*.start.x\_vector-*ball\_1*.radius;

*ball\_1*.velocity.x\_vector \*= -1;

    }

    //checks that the ball is within the limits of the block and then checks which side is being hit

    //with the velocity in the x direction needing to be negative to strike the right side, and also

    //hitting within a range of the right side for the purpose of ensuring that it does not consider

    //the ball to have hit the right and the top or the right and the bottom when the ball hits either

    //the top or the bottom of the block while travelling left, as this would result in the ball reversing

    //both x and y velocities

    else if ((((*ball\_1*.centre.x\_vector-*ball\_1*.radius)>=(*block\_1*.start.x\_vector+(*block\_1*.length.x\_vector\*(9/10))))

    &&(*ball\_1*.centre.x\_vector-*ball\_1*.radius)<=(*block\_1*.start.x\_vector+*block\_1*.length.x\_vector))

    && *ball\_1*.velocity.x\_vector<0

    &&(((*ball\_1*.centre.y\_vector+*ball\_1*.radius)>=*block\_1*.start.y\_vector)

    &&(*ball\_1*.centre.y\_vector-*ball\_1*.radius)<=(*block\_1*.start.y\_vector+*block\_1*.length.y\_vector))){

*ball\_1*.centre.x\_vector = *block\_1*.start.x\_vector+*block\_1*.length.x\_vector+*ball\_1*.radius;

*ball\_1*.velocity.x\_vector \*= -1;

    }

    //checks that the ball is within the limits of the block and then checks which side is being hit

    //with the velocity in the y direction needing to be positive to strike the top side, and also

    //hitting within a range of the top side for the purpose of ensuring that it does not consider

    //the ball to have hit the right and the top or the left and the top when the ball hits either

    //the left or the right of the block while travelling down, as this would result in the ball reversing

    //both x and y velocities

    else if ((((*ball\_1*.centre.y\_vector+*ball\_1*.radius)>=*block\_1*.start.y\_vector)

    &&(*ball\_1*.centre.y\_vector+*ball\_1*.radius)<=(*block\_1*.start.y\_vector+(*block\_1*.length.y\_vector/10)))

    && *ball\_1*.velocity.y\_vector>0

    &&(((*ball\_1*.centre.x\_vector+*ball\_1*.radius)>=*block\_1*.start.x\_vector)

    &&(*ball\_1*.centre.x\_vector-*ball\_1*.radius)<=(*block\_1*.start.x\_vector+*block\_1*.length.x\_vector))){

*ball\_1*.centre.y\_vector = *block\_1*.start.y\_vector-*ball\_1*.radius;

*ball\_1*.velocity.y\_vector \*= -1;

    }

    //checks that the ball is within the limits of the block and then checks which side is being hit

    //with the velocity in the y direction needing to be negative to strike the bottom side, and also

    //hitting within a range of the bottom side for the purpose of ensuring that it does not consider

    //the ball to have hit the right and the bottom or the left and the bottom when the ball hits either

    //the left or the right of the block while travelling up, as this would result in the ball reversing

    //both x and y velocities

    else if ((((*ball\_1*.centre.y\_vector-*ball\_1*.radius)>=(*block\_1*.start.y\_vector+(*block\_1*.length.y\_vector\*(9/10))))

    &&(*ball\_1*.centre.y\_vector-*ball\_1*.radius)<=(*block\_1*.start.y\_vector+*block\_1*.length.y\_vector))

    && *ball\_1*.velocity.y\_vector<0

    &&(((*ball\_1*.centre.x\_vector+*ball\_1*.radius)>=*block\_1*.start.x\_vector)

    &&(*ball\_1*.centre.x\_vector-*ball\_1*.radius)<(*block\_1*.start.x\_vector+*block\_1*.length.x\_vector))){

*ball\_1*.centre.y\_vector = *block\_1*.start.y\_vector+*block\_1*.length.y\_vector+*ball\_1*.radius;

*ball\_1*.velocity.y\_vector \*= -1;

    }

}

//function to check for teleport zones

*function* checkTeleport(*ball\_1*, *teleport\_1*){

    //checks that the centre of the sphere (the point that would be touching the surface of the teleport zone if ball

    //is considered a 2d representation of a sphere) is within the coordinates of the zone and teleports the ball to the specified area

    //if so

    if ((*ball\_1*.centre.x\_vector>=*teleport\_1*.start.x\_vector&&*ball\_1*.centre.x\_vector<=

        (*teleport\_1*.start.x\_vector+*teleport\_1*.length.x\_vector)) &&

        (*ball\_1*.centre.y\_vector>=*teleport\_1*.start.y\_vector&&*ball\_1*.centre.y\_vector<=

        (*teleport\_1*.start.y\_vector+*teleport\_1*.length.y\_vector))){

*ball\_1*.centre.x\_vector = *teleport\_1*.teleportPoint.x\_vector

*ball\_1*.centre.y\_vector = *teleport\_1*.teleportPoint.y\_vector

    }

}

//function to check when the player presses keys and take action when they do

*function* playerControl(*ball\_1*){

    //event listener waits for the player to press a key

    canvas.addEventListener('keydown', *function*(*key*){

        //checks for the four arrow keys and if they are pressed then the boolean associated is set to true

        //which will be used in a check for when the acceleration should be added to the velocity

        if(*key*.code === "ArrowRight"){

            moveRight = true;

        }

        if(*key*.code === "ArrowLeft"){

            moveLeft = true;

        }

        if(*key*.code === "ArrowDown"){

            moveDown = true;

        }

        if(*key*.code === "ArrowUp"){

            moveUp = true;

        }

    });

    //if statement so that the event listener for keys being released is only done when one key is currently pressed

    //to help performance as less code will need to be performed each loop when not checking for a key to be released

    //when none are pressed

    if (moveRight == true || moveLeft == true || moveDown == true || moveUp == true){

        //event listener to wait for keys to be released so booleans for each key can be set back to false

        canvas.addEventListener('keyup', *function*(*key*){

            if(*key*.code === "ArrowRight"){

                moveRight = false;

            }

            if(*key*.code === "ArrowLeft"){

                moveLeft = false;

            }

            if(*key*.code === "ArrowDown"){

                moveDown = false;

            }

            if(*key*.code === "ArrowUp"){

                moveUp = false;

            }

        });

    }

    //only checks through each boolean if one is true to save some performance

    if (moveRight == true || moveLeft == true || moveDown == true || moveUp == true){

        //if booleans are true then acceleration vectors will be added to in appropriate direction

        //if the direction would be negative in terms of the canvas then the scalar is multiplied by -1

        if(moveRight == true){

*ball\_1*.accelerationVector.x\_vector = *ball\_1*.accelerationScalar;

        }

        if(moveLeft == true){

*ball\_1*.accelerationVector.x\_vector = -*ball\_1*.accelerationScalar;

        }

        if(moveDown == true){

*ball\_1*.accelerationVector.y\_vector = *ball\_1*.accelerationScalar;

        }

        if(moveUp == true){

*ball\_1*.accelerationVector.y\_vector = -*ball\_1*.accelerationScalar;

        }

    }

    //if neither arrows in x directions are pressed then then acceleration is set back to 0 for x

    if(moveRight == false && moveLeft == false){

*ball\_1*.accelerationVector.x\_vector = 0;

    }

    //if neither arrows in y directions are pressed then then acceleration is set back to 0 for y

    if(moveDown == false && moveUp == false){

*ball\_1*.accelerationVector.y\_vector = 0;

    }

}

//function that will take the distance of each ball from the player/camera ball and move it appropriately in the

//canvas coordinates

*function* universalToCanvasBalls(*ball\_1*, *ball\_2*){

    //distance between ball centres taken in actual coordinates

*let* distance = new Vector(*ball\_1*.centre.x\_vector-*ball\_2*.centre.x\_vector, *ball\_1*.centre.y\_vector-*ball\_2*.centre.y\_vector);

    //canvas point is new vector that will get the coordinates for where ball 2 will be placed compared to ball 1

*let* canvasPoint = new Vector(*ball\_1*.canvasCentre.x\_vector-distance.x\_vector, *ball\_1*.canvasCentre.y\_vector-distance.y\_vector);

    //ball 2's points now set as canvas points calculated as above

*ball\_2*.canvasCentre.x\_vector = canvasPoint.x\_vector;

*ball\_2*.canvasCentre.y\_vector = canvasPoint.y\_vector;

    //if statement to ensure ball is only drawn if it could appear on screen

    if ((((*ball\_2*.canvasCentre.x\_vector+*ball\_2*.radius)>0)&&((*ball\_2*.canvasCentre.x\_vector-*ball\_2*.radius)<width))

    &&(((*ball\_2*.canvasCentre.y\_vector+*ball\_2*.radius)>0)&&((*ball\_2*.canvasCentre.y\_vector-*ball\_2*.radius)<height))){

        drawCircle(*ball\_2*)

    }

}

//function that will take the distance of each magnet from the player/camera ball and move it appropriately in the

//canvas coordinates

*function* universalToCanvasMagnets(*ball\_1*, *magnet\_1*){

    //distance between player centre and magnet centre taken in actual coordinates

*let* distance = new Vector(*ball\_1*.centre.x\_vector-*magnet\_1*.centre.x\_vector, *ball\_1*.centre.y\_vector-*magnet\_1*.centre.y\_vector);

    //canvas point is new vector that will get the coordinates for where the magnet will be placed compared to the player ball

*let* canvasPoint = new Vector(*ball\_1*.canvasCentre.x\_vector-distance.x\_vector, *ball\_1*.canvasCentre.y\_vector-distance.y\_vector);

    //magnet's points now set as canvas points calculated as above

*magnet\_1*.canvasCentre.x\_vector = canvasPoint.x\_vector;

*magnet\_1*.canvasCentre.y\_vector = canvasPoint.y\_vector;

    //if statement to ensure magnet is only drawn if it could appear on screen

    if ((((*magnet\_1*.canvasCentre.x\_vector+*magnet\_1*.radius)>0)&&((*magnet\_1*.canvasCentre.x\_vector-*magnet\_1*.radius)<width))

    &&(((*magnet\_1*.canvasCentre.y\_vector+*magnet\_1*.radius)>0)&&((*magnet\_1*.canvasCentre.y\_vector-*magnet\_1*.radius)<height))){

        drawCircle(*magnet\_1*);

    }

}

//function that will take the distance of each block from the player/camera ball and move it appropriately in the

//canvas coordinates

*function* universalToCanvasBlocks(*ball\_1*, *block\_1*){

    //distance between player centre and block top left corner taken in actual coordinates

*let* distance = new Vector(*ball\_1*.centre.x\_vector-*block\_1*.start.x\_vector, *ball\_1*.centre.y\_vector-*block\_1*.start.y\_vector);

    //canvas point is new vector that will get the coordinates for where block top left corner will be placed compared to the player

*let* canvasPoint = new Vector(*ball\_1*.canvasCentre.x\_vector-distance.x\_vector, *ball\_1*.canvasCentre.y\_vector-distance.y\_vector);

    //block's points now set as canvas points calculated as above

*block\_1*.canvasStart.x\_vector = canvasPoint.x\_vector;

*block\_1*.canvasStart.y\_vector = canvasPoint.y\_vector;

    //if statement to ensure block is only drawn if it could appear on screen

    if ((((*block\_1*.canvasStart.x\_vector+*block\_1*.length.x\_vector)>0)&&((*block\_1*.canvasStart.x\_vector)<width))

    &&(((*block\_1*.canvasStart.y\_vector+*block\_1*.length.y\_vector)>0)&&((*block\_1*.canvasStart.y\_vector)<height))){

        drawRectangle(*block\_1*);

    }

}

//function that will take the distance of each friction zone from the player/camera ball and move it appropriately in the

//canvas coordinates

*function* universalToCanvasFrictionZones(*ball\_1*, *square\_1*){

    //distance between player centre and friction zone top left corner taken in actual coordinates

*let* distance = new Vector(*ball\_1*.centre.x\_vector-*square\_1*.upperpoint.x\_vector, *ball\_1*.centre.y\_vector-*square\_1*.upperpoint.y\_vector);

    //canvas point is new vector that will get the coordinates for the friction zone top left corner will be placed compared to the player

*let* canvasPoint = new Vector(*ball\_1*.canvasCentre.x\_vector-distance.x\_vector, *ball\_1*.canvasCentre.y\_vector-distance.y\_vector);

    //friction zone's points now set as canvas points calculated as above

*square\_1*.canvasStart.x\_vector = canvasPoint.x\_vector;

*square\_1*.canvasStart.y\_vector = canvasPoint.y\_vector;

    //if statement to ensure friction zone is only drawn if it could appear on screen

    if ((((*square\_1*.canvasStart.x\_vector+*square\_1*.length.x\_vector)>0)&&((*square\_1*.canvasStart.x\_vector)<width))

    &&(((*square\_1*.canvasStart.y\_vector+*square\_1*.length.y\_vector)>0)&&((*square\_1*.canvasStart.y\_vector)<height))){

        drawRectangle(*square\_1*);

    }

}

//function that will take the distance of the edges of the canvas from the player/camera ball and move it appropriately in the

//canvas coordinates

*function* universalToCanvasBorder(*ball\_1*, *canvas\_1*){

    //distance between player centre and canvas top left corner taken in actual coordinates

*let* distanceUpper = new Vector(*ball\_1*.centre.x\_vector-*canvas\_1*.universalStart.x\_vector, *ball\_1*.centre.y\_vector-*canvas\_1*.universalStart.y\_vector);

    //canvas point is new vector that will get the coordinates for the canvas top left corner will be placed compared to the player

*let* canvasPointUpper = new Vector(*ball\_1*.canvasCentre.x\_vector-distanceUpper.x\_vector,

*ball\_1*.canvasCentre.y\_vector-distanceUpper.y\_vector);

    //canvas's points now set as canvas points calculated as above

*canvas\_1*.canvasStart.x\_vector = canvasPointUpper.x\_vector;

*canvas\_1*.canvasStart.y\_vector = canvasPointUpper.y\_vector;

    //canvas always drawn as it is always on screen

    drawRectangle(*canvas\_1*);

}

//function that will take the distance of each teleporter from the player/camera ball and move it appropriately in the

//canvas coordinates

*function* universalToCanvasTeleporters(*ball\_1*, *teleport\_1*){

    //distance between player centre and teleporter top left corner taken in actual coordinates

*let* distance = new Vector(*ball\_1*.centre.x\_vector-*teleport\_1*.start.x\_vector, *ball\_1*.centre.y\_vector-*teleport\_1*.start.y\_vector);

    //canvas point is new vector that will get the coordinates for the teleporter top left corner will be placed compared to the player

*let* canvasPoint = new Vector(*ball\_1*.canvasCentre.x\_vector-distance.x\_vector, *ball\_1*.canvasCentre.y\_vector-distance.y\_vector);

    //teleporter's points now set as canvas points calculated as above

*teleport\_1*.canvasStart.x\_vector = canvasPoint.x\_vector;

*teleport\_1*.canvasStart.y\_vector = canvasPoint.y\_vector;

    //distance between player centre and teleporter circle centre taken in actual coordinates

*let* distanceCentre = new Vector(*ball\_1*.centre.x\_vector-*teleport\_1*.circleCentre.x\_vector,

*ball\_1*.centre.y\_vector-*teleport\_1*.circleCentre.y\_vector);

    //canvas point is new vector that will get the coordinates for the teleporter circle centre will be placed compared to the player

*let* canvasPointCentre = new Vector(*ball\_1*.canvasCentre.x\_vector-distanceCentre.x\_vector,

*ball\_1*.canvasCentre.y\_vector-distanceCentre.y\_vector);

    //teleporter's points now set as canvas points calculated as above

*teleport\_1*.canvasCentre.x\_vector = canvasPointCentre.x\_vector;

*teleport\_1*.canvasCentre.y\_vector = canvasPointCentre.y\_vector;

    //if statement to ensure teleporter is only drawn if it could appear on screen

    if ((((*teleport\_1*.canvasStart.x\_vector+*teleport\_1*.length.x\_vector)>0)&&((*teleport\_1*.canvasStart.x\_vector)<width))

    &&(((*teleport\_1*.canvasStart.y\_vector+*teleport\_1*.length.y\_vector)>0)&&((*teleport\_1*.canvasStart.y\_vector)<height))){

*teleport\_1*.drawTeleporter();

    }

}

//group of functions to avoid code repetition

*function* functionGroup(*ball\_1*){

    //edges checked for every ball

    checkEdges(*ball\_1*, CanvasArray[0]);

    //if statement to ensure ghost balls aren't checked against other objects so they can move freely

    if (*ball\_1*.ghost == false){

        //ball checked against each teleporter

        TeleportArray.forEach(*teleport\_1* *=>*{

            checkTeleport(*ball\_1*, *teleport\_1*);

        })

        //sets friction check false so friction will apply, whether by canvas or friction zone depending on ball location

*ball\_1*.frictionCheck = false;

        //checks ball against each friction zone to check if ball is on any

        FrictionSquareArray.forEach(*square\_1* *=>* {

            checkFriction(*ball\_1*, *square\_1*);

        });

        //if statement to ensure ball isn't checked against magnets unless magnetic

        if (*ball\_1*.magnetic == true){

            //ball checked against each magnet

            MagnetArray.forEach(*magnet\_1* *=>* {

                checkMagnets(*ball\_1*, *magnet\_1*);

            });

        }

        //ball checked against each block

        BlockArray.forEach(*block\_1* *=>* {

            checkBlocks(*ball\_1*, *block\_1*);

        });

    }

    //checks ball has any acceleration before checking to add it to velocity, means only player ball is checked in current build

    //saves time with checking each ball in this method

    if (*ball\_1*.accelerationVector.x\_vector !== 0 || *ball\_1*.accelerationVector.y\_vector !== 0){

        addAcceleration(*ball\_1*);

    }

    //checks ball is not checked for friction already before considering friction from canvas (also ensure canvas does not affect ghost)

    if (*ball\_1*.frictionCheck == false && *ball\_1*.ghost == false){

*ball\_1*.velocity.x\_vector \*= 1-CanvasArray[0].friction;

*ball\_1*.velocity.y\_vector \*= 1-CanvasArray[0].friction;

*ball\_1*.frictionCheck = true;

    }

    //added so camera is easier to control, will slow down but only if ball is ghost and player controlled

    if (*ball\_1*.ghost == true && *ball\_1*.player == true){

*ball\_1*.velocity.x\_vector \*= 1-0.025;

*ball\_1*.velocity.y\_vector \*= 1-0.025;

    }

    //only adds velocity if it isn't negligible

    if ((*ball\_1*.velocity.x\_vector > 0.01 || *ball\_1*.velocity.x\_vector<-0.01)

    || (*ball\_1*.velocity.y\_vector > 0.01 || *ball\_1*.velocity.y\_vector<-0.01)) {

        addVelocity(*ball\_1*);

    }

}

//method for initial launch of the physics engine to ensure things are created correctly

*function* initiate(){

    //if statement to check that there is a canvas and if not then one is created.

    if (CanvasArray.length == 0 || CanvasArray.length >= 2){

        //if there is more than 1 then canvases are removed so they won't be acknowledged

        if (CanvasArray.length>1){

            CanvasArray.splice(0, CanvasArray.length);

        }

*let* defaultCanvas = new Canvas(0,0,1500,900, 0, "black", "turquoise");

    }

    //checks that each ball will appear within limits of canvas

    BallArray.forEach((*ball\_1*) *=>* {

        if (*ball\_1*.centre.x\_vector-*ball\_1*.radius<CanvasArray[0].universalStart.x\_vector || *ball\_1*.centre.x\_vector+*ball\_1*.radius

            > (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)){

*ball\_1*.centre.x\_vector = CanvasArray[0].universalStart.x\_vector+*ball\_1*.radius;

            }

        if (*ball\_1*.centre.y\_vector-*ball\_1*.radius<CanvasArray[0].universalStart.y\_vector || *ball\_1*.centre.y\_vector+*ball\_1*.radius

            > (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)){

*ball\_1*.centre.y\_vector = CanvasArray[0].universalStart.y\_vector+*ball\_1*.radius;

            }

    });

    //checks that each magnet will appear within limits of canvas

    MagnetArray.forEach((*magnet\_1*) *=>* {

        if (*magnet\_1*.centre.x\_vector-*magnet\_1*.radius<CanvasArray[0].universalStart.x\_vector || *magnet\_1*.centre.x\_vector+*magnet\_1*.radius

            > (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)){

*magnet\_1*.centre.x\_vector = CanvasArray[0].universalStart.x\_vector+*magnet\_1*.radius;

            }

        if (*magnet\_1*.centre.y\_vector-*magnet\_1*.radius<CanvasArray[0].universalStart.y\_vector || *magnet\_1*.centre.y\_vector+*magnet\_1*.radius

            > (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)){

*magnet\_1*.centre.y\_vector = CanvasArray[0].universalStart.y\_vector+*magnet\_1*.radius;

            }

    })

    //checks that each block will appear within limits of canvas

    BlockArray.forEach((*block\_1*) *=>* {

        if (*block\_1*.start.x\_vector<CanvasArray[0].universalStart.x\_vector ||

*block\_1*.start.x\_vector+*block\_1*.length.x\_vector

            > (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)){

                //ensures that if block's length is greater than canvas's it will be reduced to fit

                if (*block\_1*.length.x\_vector>CanvasArray[0].length.x\_vector){

*block\_1*.length.x\_vector = CanvasArray[0].length.x\_vector;

                }

*block\_1*.start.x\_vector = CanvasArray[0].universalStart.x\_vector;

            }

        if (*block\_1*.start.y\_vector<CanvasArray[0].universalStart.y\_vector ||

*block\_1*.start.y\_vector+*block\_1*.length.y\_vector

            > (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)){

                //ensures that if block's height is greater than canvas's it will be reduced to fit

                if (*block\_1*.length.y\_vector>CanvasArray[0].length.y\_vector){

*block\_1*.length.y\_vector = CanvasArray[0].length.y\_vector;

                }

                block\_1.start.y\_vector = CanvasArray[0].universalStart.y\_vector;

            }

    })

    //checks that each friction zone will appear within limits of canvas

    FrictionSquareArray.forEach((*friction\_1*) *=>* {

        if (friction\_1.upperpoint.x\_vector<CanvasArray[0].universalStart.x\_vector ||

            friction\_1.upperpoint.x\_vector+friction\_1.length.x\_vector

            > (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)){

                //ensures that if friction zone's length is greater than canvas's it will be reduced to fit

                if (friction\_1.length.x\_vector>CanvasArray[0].length.x\_vector){

                    friction\_1.length.x\_vector = CanvasArray[0].length.x\_vector;

                }

                friction\_1.upperpoint.x\_vector = CanvasArray[0].universalStart.x\_vector;

            }

        if (friction\_1.upperpoint.y\_vector<CanvasArray[0].universalStart.y\_vector ||

            friction\_1.upperpoint.y\_vector+friction\_1.length.y\_vector

            > (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)){

                //ensures that if friction zone's height is greater than canvas's it will be reduced to fit

                if (friction\_1.length.y\_vector>CanvasArray[0].length.y\_vector){

                    friction\_1.length.y\_vector = CanvasArray[0].length.y\_vector;

                }

                friction\_1.upperpoint.y\_vector = CanvasArray[0].universalStart.y\_vector;

            }

    })

    //checks that each teleporter will appear within limits of canvas and teleport within limits

    TeleportArray.forEach((*teleporter\_1*)*=>* {

        if (teleporter\_1.start.x\_vector<CanvasArray[0].universalStart.x\_vector ||

            teleporter\_1.start.x\_vector+teleporter\_1.length.x\_vector

            > (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)){

                //ensures that if teleporter's length is greater than canvas's it will be reduced to fit

                if (teleporter\_1.length.x\_vector>CanvasArray[0].length.x\_vector){

                    teleporter\_1.length.x\_vector = CanvasArray[0].length.x\_vector;

                } else{

                    teleporter\_1.start.x\_vector = CanvasArray[0].universalStart.x\_vector;

                }

            }

        if (teleporter\_1.start.y\_vector<CanvasArray[0].universalStart.y\_vector ||

            teleporter\_1.start.y\_vector+teleporter\_1.length.y\_vector

            > (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)){

                //ensures that if teleporter's height is greater than canvas's it will be reduced to fit

                if (teleporter\_1.length.y\_vector>CanvasArray[0].length.y\_vector){

                    teleporter\_1.length.y\_vector = CanvasArray[0].length.y\_vector;

                } else{

                    teleporter\_1.start.y\_vector = CanvasArray[0].universalStart.y\_vector;

                }

            }

        //ensures that if teleporter would move objects outside of canvas then teleport point is moved to centre of canvas for x or y

        if (teleporter\_1.teleportPoint.x\_vector<=CanvasArray[0].universalStart.x\_vector || teleporter\_1.teleportPoint.x\_vector

            >= (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)){

                teleporter\_1.teleportPoint.x\_vector = (CanvasArray[0].universalStart.x\_vector+CanvasArray[0].length.x\_vector)/2;

            }

        if (teleporter\_1.teleportPoint.y\_vector<=CanvasArray[0].universalStart.y\_vector || teleporter\_1.teleportPoint.y\_vector

            >= (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)){

                teleporter\_1.teleportPoint.y\_vector = (CanvasArray[0].universalStart.y\_vector+CanvasArray[0].length.y\_vector)/2;

            }

    });

    //if statement to check if there is more than one player ball or 0

    if (PlayerBallArray.length == 0 || PlayerBallArray.length >=2){

        //if there is more than 1 then balls are set as not player and they are removed from the player ball array

        if (PlayerBallArray.length>1){

            PlayerBallArray.forEach((*ball\_1*) *=>* {

                ball\_1.player = false;

            });

            PlayerBallArray.splice(0, PlayerBallArray.length);

        }

        //default ball created to act as camera to watch game/simulation

*let* PlayerCamera = new Ball(100, 100, 0, 0.00001, 0, 0, 5, 0.1, false, "/", 0, false, true, true, CanvasArray[0].fill, CanvasArray[0].fill);

    }

    //code goes into main loop that will repeat

    requestAnimationFrame(mainLoop);

}

//main set of code that will call the other methods

*function* mainLoop() {

    //ensures that the full areas are cleared so that there are not repeat frames on top of each other, even off screen, as

    //this can affect performance

    if (CanvasArray[0].length.x\_vector<=width&&CanvasArray[0].length.y\_vector<=height){

        ctx.clearRect(CanvasArray[0].universalStart.x\_vector, CanvasArray[0].universalStart.y\_vector,

            width, height);

    } else if (CanvasArray[0].length.x\_vector<=width&&CanvasArray[0].length.y\_vector>height){

        ctx.clearRect(CanvasArray[0].universalStart.x\_vector, CanvasArray[0].universalStart.y\_vector,

            width, CanvasArray[0].length.y\_vector);

    } else if (CanvasArray[0].length.x\_vector>width&&CanvasArray[0].length.y\_vector<=height){

        ctx.clearRect(CanvasArray[0].universalStart.x\_vector, CanvasArray[0].universalStart.y\_vector,

            CanvasArray[0].length.x\_vector, height);

    } else {

        ctx.clearRect(CanvasArray[0].universalStart.x\_vector, CanvasArray[0].universalStart.y\_vector,

            CanvasArray[0].length.x\_vector, CanvasArray[0].length.y\_vector);

    }

    //checks position of each object against the position of the player ball and then draws them if on screen (see universal

    //to canvas methods)

    universalToCanvasBorder(PlayerBallArray[0], CanvasArray[0]);

    FrictionSquareArray.forEach((*square\_1*) *=>* {

        universalToCanvasFrictionZones(PlayerBallArray[0], square\_1);

    });

    TeleportArray.forEach((*teleport\_1*) *=>* {

        universalToCanvasTeleporters(PlayerBallArray[0], *teleport\_1*);

    })

    MagnetArray.forEach((*magnet\_1*) *=>* {

        universalToCanvasMagnets(PlayerBallArray[0], *magnet\_1*);

    });

    BlockArray.forEach((*block\_1*) *=>* {

        universalToCanvasBlocks(PlayerBallArray[0], *block\_1*);

    });

    BallArray.forEach((*ball\_2*) *=>* {

        if (*ball\_2*.player == false){

            universalToCanvasBalls(PlayerBallArray[0], *ball\_2*);

        }

    });

    //draws player ball

    drawCircle(PlayerBallArray[0]);

    //cycles through each ball for their movement and comparisons to other balls

    BallArray.forEach((*ball\_1*, *index*) *=>* {

        //if statement to ensure only player balls affected by keyboard control

        if (*ball\_1*.player == true){

            playerControl(*ball\_1*);

        }

        //if statement so that last ball isn't chosen as this affects logic of for loop

        if (*index* !== BallArray.length-1){

            //for loop to cycle through and compare each ball to the balls further on in the list, ensuring each ball

            //only checked against each other once

            for(*let* num = *index*+1; num<BallArray.length; num++){

                //if statement to check neither ball is a ghost before comparing them in methods

                if (*ball\_1*.ghost == false && BallArray[num].ghost == false){

                    //methods used to check balls against each other

                    checkCollision(*ball\_1*, BallArray[num]);

                    ballVectors(*ball\_1*, BallArray[num]);

                    //if statement to ensure both balls are magnetic before calling method

                    if (*ball\_1*.magnetic == true && BallArray[num].magnetic == true){

                        checkMagneticBalls(*ball\_1*, BallArray[num]);

                    }

                }

            }

            //checks ball isn't stationary before checking movement based methods on them

            if (*ball\_1*.stationary == false){

                functionGroup(*ball\_1*);

            }

        }

        //used for checking the last ball against other objects (other than balls since it has been compared to the rest already)

        else {

            //if statement to check if the last ball is player controlled

            if (*ball\_1*.player == true){

                playerControl(*ball\_1*);

            }

            //if statement to check if last ball is stationary or not to go through with the function group method

            if (*ball\_1*.stationary == false){

                functionGroup(*ball\_1*);

            }

        }

    });

    //request animation frame will keep requesting this loop to ensure main loop keeps occurring

    requestAnimationFrame(mainLoop);

}

//method and classes exported for the programmer to use in another javascript file

export { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter};

Code block : objectCreation.js

import { initiate, FrictionZone, Ball, Magnet, Block, Canvas, Teleporter} from './physics.js';

*let* defaultCanvas = new Canvas(0, 0, 1500,3000, 0.01, "black", "orange");

*let* Friction1 = new FrictionZone(500, 350, 100, 100, "black", "purple", 0.05);

*let* Friction2 = new FrictionZone(1000, 550, 100, 100, "black", "purple", 0.05);

*let* block1 = new Block(600, 300, 300, 300, "Black", "Silver");

*let* magnet1 = new Magnet (550, 20, 20, 20, "North", "black", "white");

*let* magnet2 = new Magnet (950, 880, 20, 20, "South", "black", "white");

*let* Teleporter1 = new Teleporter (500, 600, 150, 150, 100, 100, "black", "gold", "silver");

*let* Ball1 = new Ball(110, 450, 20, 100, 5, 0, 5, 0, true, "South", 2, true, false, false, "black", "red", false);

*let* Ball2 = new Ball(500, 450, 20, 100, -5, 0, 5, 1, true, "South", 3, false, false, false, "black", "yellow", false);

*let* Ball3 = new Ball(100, 700, 30, 50, 2, -1, 5, 0.1, true, "North", 3, false, false, false, "black", "pink", false);

*let* Ball4 = new Ball(780, 100, 20, 50, 0, -4, 5, 1, false, "/", 0, false, false, false, "black", "black", false);

*let* Ball5 = new Ball(900, 100, 20, 50, -2, 4, 5, 1, false, "/", 0, false, false, false, "black", "black", false);

*let* Ball6 = new Ball(840,160, 20, 50, -2, 2, 5, 1, false, "/", 0, false, false, false, "black", "red", false);

*let* Ball7 = new Ball(660,740, 20, 50, 2, -2, 5, 1, false, "/", 0, false, false, false, "black", "red", false);

*let* Ball8 = new Ball(660,160, 20, 50, 2, 2, 5, 1, false, "/", 0, false, false, false, "black", "red", false);

*let* Ball9 = new Ball(1000,740, 20, 50, 7, 0, 5, 0.1, false, "/", 0, false, false, false, "black", "red", false);

*let* Ball10 = new Ball(550,840, 20, 50, 2, -2, 5, 1, false, "/", 0, false, false, false, "black", "black", false);

initiate();