Computing project 2014/2015

PHYSICS SIMULATION

Jacob Shirley

Contents

[1 Analysis 3](#_Toc413235325)

[1.1 Background to/identification of problem 3](#_Toc413235326)

[1.2 Description of the current system 3](#_Toc413235327)

[1.3 Identification of prospective users 6](#_Toc413235328)

[1.4 Identification of user needs and limitations 6](#_Toc413235329)

[1.5 Data sources and destinations 7](#_Toc413235330)

[1.6 Data Volumes 7](#_Toc413235331)

[1.7 Data flow diagrams 7](#_Toc413235332)

[1.8 Analysis Data Dictionary 9](#_Toc413235333)

[1.9 OO Model 9](#_Toc413235334)

[1.10 Project Particleives 10](#_Toc413235335)

[1.11 Appraisal of potential solutions 11](#_Toc413235336)

[1.12 Justification of chosen solution 12](#_Toc413235337)

[2 Design 0](#_Toc413235338)

[2.1 Overall System Design 0](#_Toc413235339)

[2.2 Modular Structure 0](#_Toc413235340)

[2.3 OO Model 2](#_Toc413235341)

[2.4 Class Structures 3](#_Toc413235342)

[2.5 Input validations 6](#_Toc413235343)

[2.6 File organisation and processing 6](#_Toc413235344)

[2.7 Storage media and format 6](#_Toc413235345)

[2.8 Algorithms 6](#_Toc413235346)

[2.9 User Interface Design 11](#_Toc413235347)

[2.10 Security and Integrity of Data 13](#_Toc413235348)

[2.11 System Security 13](#_Toc413235349)

[2.12 Test Strategy 13](#_Toc413235350)

[3 System Testing 14](#_Toc413235351)

[3.1 Test Plan – Typical Data 14](#_Toc413235352)

[3.2 Test Plan – Erroneous Data 14](#_Toc413235353)

[3.3 Test Plan – Boundary Data 14](#_Toc413235354)

[3.4 Test Plan – Volume Testing 14](#_Toc413235355)

[4 System Maintenance 15](#_Toc413235356)

[4.1 System Overview 15](#_Toc413235357)

[4.2 Algorithms 15](#_Toc413235358)

[4.3 Procedure and variable lists 15](#_Toc413235359)

[4.4 Annotated listings / screens 15](#_Toc413235360)

[4.5 Database Definitions 15](#_Toc413235361)

[4.6 Forms / screens 15](#_Toc413235362)

[5 User Manual 16](#_Toc413235363)

[5.1 Contents page 16](#_Toc413235364)

[5.2 Introduction 16](#_Toc413235365)

[5.3 System Requirements 16](#_Toc413235366)

[5.4 Installation 16](#_Toc413235367)

[5.5 Using the system 16](#_Toc413235368)

[5.6 Error Handling 16](#_Toc413235369)

[6 Evaluation 17](#_Toc413235370)

[6.1 Project Performance against Project Particleives 17](#_Toc413235371)

[6.2 Client/ User Feedback 17](#_Toc413235372)

[6.3 Project Extensions 17](#_Toc413235373)

[6.4 Appendices 17](#_Toc413235374)

[6.5 Interview Transcripts 17](#_Toc413235375)

[6.6 Summary of Questionnaires 17](#_Toc413235376)

[6.7 Original documents of system 17](#_Toc413235377)

[6.8 Program listing 17](#_Toc413235378)

[6.9 Test data 17](#_Toc413235379)

[7 Appendices 0](#_Toc413235380)

[7.1 Plan of the potential solutions 0](#_Toc413235381)

# Analysis

## Background to/identification of problem

My client is Alison Frost a physics teacher at Varndean Sixth Form College, which is located in Brighton. The college teaches A Levels, as well as GCSES and other qualifications. The college has 1450 students currently attending. The learning resources available to students at the college are mostly centred on textbooks and online videos. The transcript of the interview I conducted with Alison Frost to gather the requirements for this project, are given in appendix 7.1.

For many teachers it is a difficult task to help students understand some of the challenging topics in physics. Even with the resources listed above, students may find the homework given after each class too difficult to do and therefore find the lessons much harder. This is especially true in physics, where many concepts are abstract and require a lot of explanation. As a student at the college, I find that interactive solutions, such as physics/maths simulations, make learning much easier.

One that is fundamental topics in physics is the conservation of energy during the collision of particles. This can be demonstrated through elastic collisions. In practice, however, it is hard to demonstrate these concepts in an intuitive way with the current resources available to students at the college due to the limitations of the equipment. The equipment is also not easy to set up and often requires a lot of time/people. It is difficult for my client to find time to set up the equipment and then to show the practical example to the students. My client wants to speed up this process by making it virtual. She can then also set it as homework for the students.

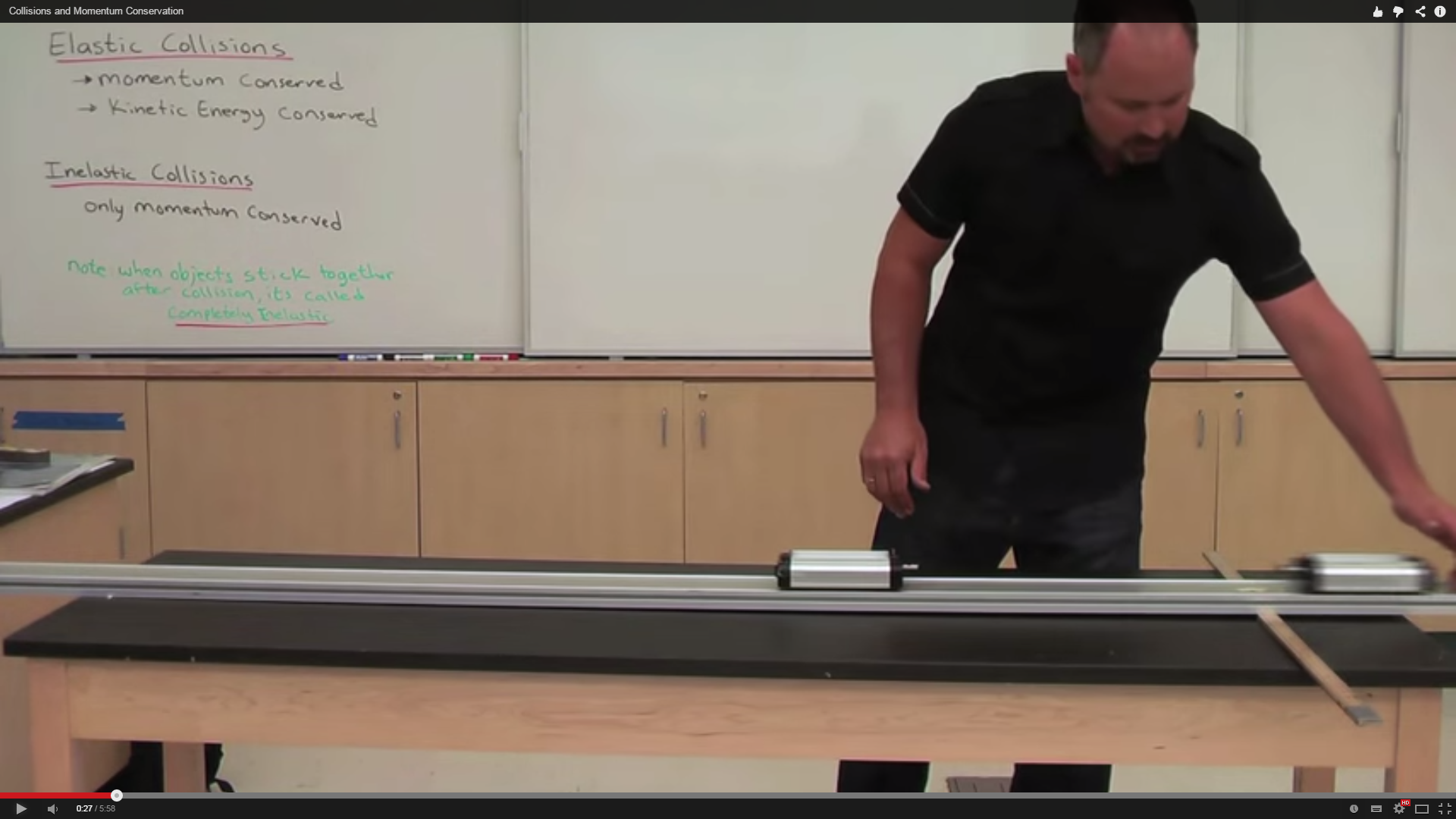
## Description of the current system

The current teaching of this topic is largely based on textbooks and videos which does not offer much opportunity to test these concepts interactively. Often students find it hard to visualise them, which has a negative effect on their understanding.

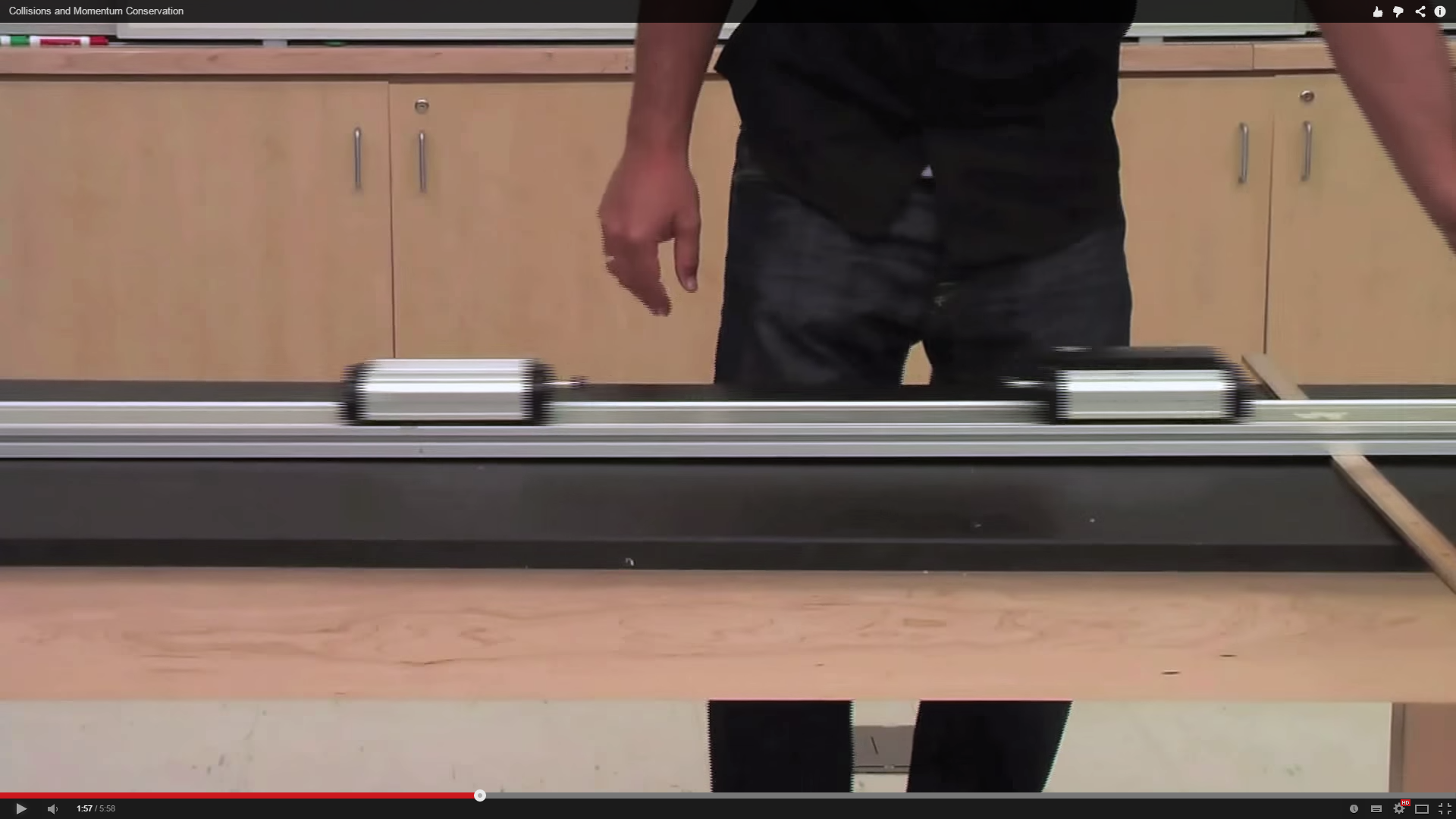
The teacher (Alison Frost) starts off by showing the students an example of an elastic collision with a real apparatus. This is an air-track (a track which blows air through holes in order to reduce the friction) on which two particles of specific masses are placed. They are then given different velocities. Their velocities is recorded by a sensor before collision and after collision.

An example of a practical experiment to explain this concept involves two “frictionless” carts, of unknown mass but equal mass, on wheels which move on a “frictionless” surface. The experiment is repeated numerous times, each time changing the different variables. It is done as follows:

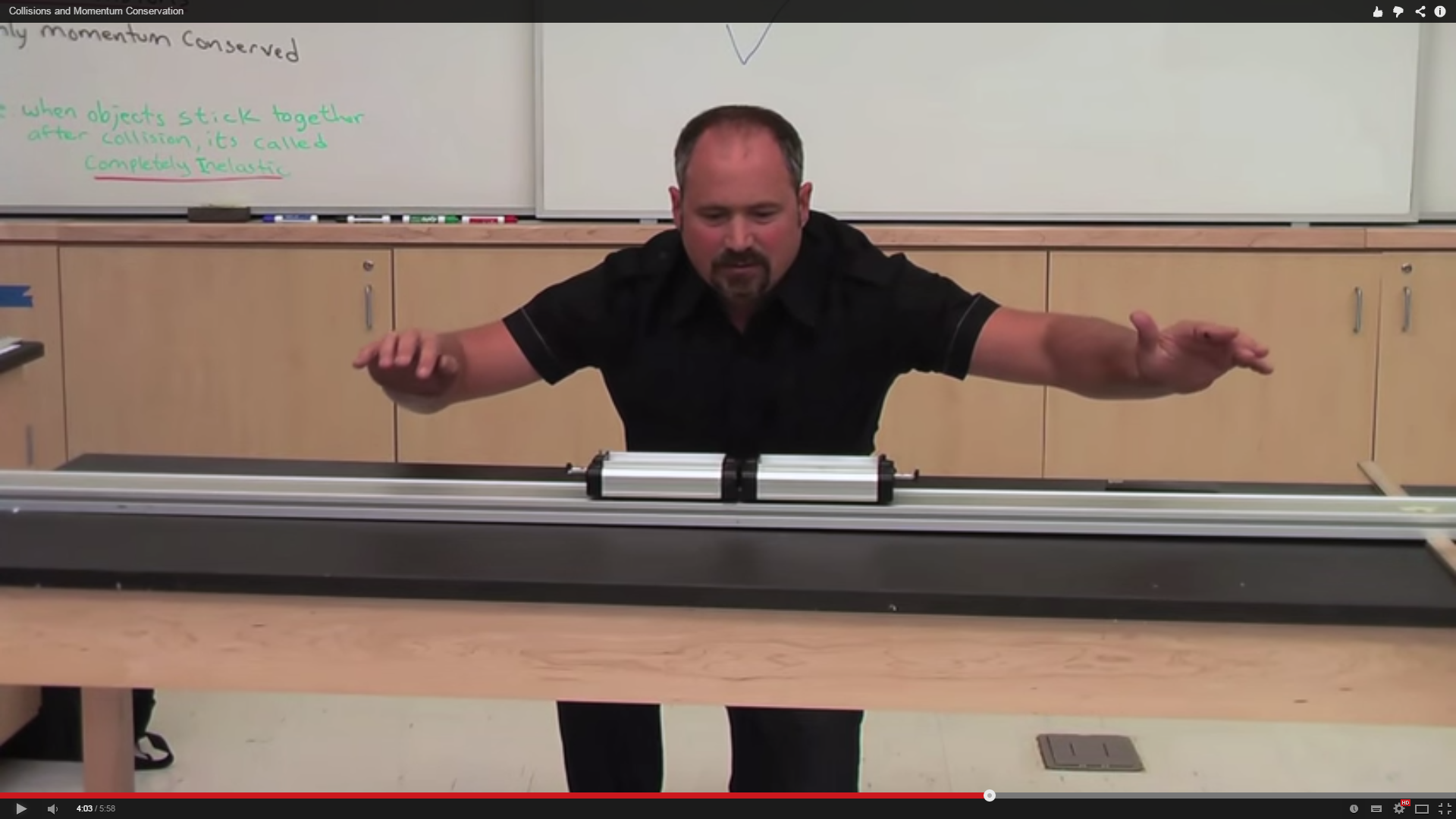
1. The first time the first cart is accelerated towards the second cart, which has no velocity. As expected, after the collision the second cart moves in the same direction as the first, which transfers most of its momentum to the second.



1. Next time, the first cart is accelerated towards the second cart, which has no velocity but an increased mass. This time after the collision the second cart moves in the same direction as the first, but the first cart moves out in the opposite direction. It “bounces” off the second, which has more mass.



1. Finally they are both accelerated towards each other. This time their momentum is equal but opposite, so when they collide, both of their momentums cancel out. This makes both of their velocities equal to zero (therefore they do not move).



This helps give students an intuitive look at the fundamentals of elastic collisions and the conservation of momentum.

The problems with practical experiments, however, are that:

* They take up teacher time which could be spent helping students.
* They cost money because of the price of the equipment.
* Students often don’t get a chance to use the equipment.
* They are often difficult to set up.

After the practical, the teacher proceeds to write on the board the main concepts that need to be learned, while the students also ask questions. This is usually two particles, each of different masses, before, during, and after collision. This may be followed by a short video on the topic. The students then take notes of everything that the teacher says/writes on the board, and attempt to answer the questions set. An example question would be: calculate the initial velocity of an particle before collision given that you know the resultant velocities and masses of both particles. The students are then expected to revise this for the next lesson and ultimately their exam.

## Identification of prospective users

There will be two types of users:

* Teachers – This type of user will be using this system to demonstrate elastic collisions to a class. They may want to project the application through a projector onto a screen. They will need to know fully how to use the system.
* Students – They will use the system as homework set by the teacher. It will probably be used in the same way as the teachers.

## Identification of user needs and limitations

Both teachers and students have the same needs.

Shared needs:

* A user-friendly interface.
* A way of changing the speed of the simulation.
* Interaction with the particles in the simulation.
* Information displayed about the system: energy in the system, direction of the particles, velocities, etc.

## Data sources and destinations

|  |  |  |
| --- | --- | --- |
| **Data** | **Source** | **Destination** |
| Variables: radius of the particles, mass of the particle, coefficient of restitution, colour of the particle, simulation speed, graph zoom, graph y offset. | Radius – on the add menu, when the mouse is scrolled the radius of the particle will change size depending on whether the mouse was scrolled up or down (up = bigger, down = smaller).  Mass of the particle – scroll-bar  Coefficient of restitution – scroll-bar  Colour of the particle – button to generate random colours  Simulation speed – scroll-bar  Graph zoom – buttons for zoom in and out  Graph y offset – button for y increase (move graph down) and y decrease (move graph up) | Stored as variables with their respective names in the system. |

## Data Volumes

Data may be stored for the simulation in the future in a database, but for now data will only be used temporarily while the program is running. Variables stored in a URL could be used by the program, but this would be a final addition, after the bulk of the program has been created.

The processing of this data will be done at runtime.

## Data flow diagrams

**Context diagram:**

Students

Physics concepts

**Data flow for current system:**

Notes

D1

Learned information

Students

Teachers

Sets up practical

Does Practical

1

Learned information

Information

Student questions

Information about elastic collisions

Practical finished

Complete questions

3

Does Practical

2

**Data flow diagram for proposed system:**

User

Enters data into system, which get stored in RAM

1

Data

Variables

RAM

D1

Data

Variables

Variables

Changes variables

2

Simulation runs, performing calculations using stored variables

3

Data displayed visually to the user

4

Particle positions, colours, and radii

## Analysis Data Dictionary

**Mass** – The quantity of matter in a body regardless of its volume or of any forces acting on it.

**Velocity** - A vector quantity that refers to "the rate at which an particle changes its position."

**Coefficient of restitution** – The coefficient of restitution (COR) of two colliding particles is typically a positive real number between 0.0 and 1.0 representing the ratio of speeds after and before an impact, taken along the line of the impact. Pairs of particles with COR = 1 collide elastically, while particles with COR < 1 collide inelastically. (Wikipedia)

**Elastic collision** – All the energy and momentum before the collision are conserved after the collision.

**Inelastic (plastic) collision** – Some energy is lost in a collision, but momentum is conserved.

**Particle** – An particle in the system that visually represents the concepts of elastic or inelastic collisions.

**Joule** – A derived unit of energy, work, or amount of heat.

**Kilojoule** – One thousand joules.

## OO Model

User interface:

Simulation

Widget

Graph

Particles:

Particle

PhysicsObject

Engine:

SimulationEngine

Math:

PositionVector

## Project Objectives

1. The system will make learning about elastic collisions and the laws of conservation of energy easier for students to understand.
2. The system will allow the visual simulation of collisions of up to 100 particles in two dimensions.
3. The system will accurately follow the physics principals of elastic and inelastic collisions.
4. The variable properties of particles in the simulation, such as mass, will be able to be modified by the user.
5. Particles can be added and removed from the simulation. A maximum of 100 particles can be added.
6. Start/pause/resume buttons will be added to control the simulation.
7. Step forward will be added to manually “update” the simulation. Step backward cannot be added as it would require too much data to be stored of previous updates.
8. Tracing particles functionality will be added to help the user find out how the particles in the simulation are behaving.
9. The system will display graphs showing the energy of all particles as the simulation progresses.
10. Simulation data will be shown. This includes:

* Displaying the direction of particles before and after collisions.
* The energy stored in a particle.
* The properties of the particle (velocity, mass, coefficient of restitution)

1. Simulation saving and loading will be added to restore a previous state of the simulation, i.e, just before two particles are about to collide, so that the exact collision can be repeated.

## Appraisal of potential solutions

**Online software:**

There is a software application which can help explain the concept of elastic collisions and energy conservation laws in existence: <http://www.hoomanr.com/Demos/Elastic2/> . This website offers a basic understanding of the above concepts in a simulation that runs in Adobe Flash Player. It features three particles, one red, blue and green. The simulation can be paused and displays the direction of the particles, their velocities and a scroll bar for their mass, each of which can be modified. It also offers written information below it. It’s easy to use, completely free, and may be good enough for an introduction to these topics. Disadvantages of this software include:

* There is not enough information displayed about the collisions happening in the simulation.
* It requires a plugin to run.
* More particles cannot be added.
* The user interface of this system is unintuitive: there is no way of knowing if you have selected a particle in order to change its properties.
* There’s no way of changing the volume without changing the mass, or vice versa.
* There is no energy graph.

**Off the shelf software:**

Simulation Library Vol. II is: *“Comprehensive software library of physical science simulations and labs designed to provide a highly-investigative learning environment for students at a variety of levels.”*

This software needs to be downloaded onto the user’s computer in order to be run. It is designed for use in a learning environment such as a college as it includes many other simulations that are part of many physics courses. It includes a number of simulations around the ‘conservation of energy’ concepts, as well as many topics beyond A-level physics. In the future this could be useful for teaching other parts of the course. The price of a single user license is: $229.00 up to $890.00 for an “Unlimited Site License”. It works out-of-the-box and requires little time to set up.

**Bespoke software (web-based):**

A system that would be designed to meet all the requirements of the customer. It could be designed in HTML and JavaScript as these are two of the most important web languages around at the moment which provide a powerful set of design possibilities and can be used to create intuitive web applications. They would both be used to manage the UI and JavaScript would control everything that has to be updated in real time such as the physics collisions involved. Using JavaScript is perfect for an application of this type due to the fact that it can be run on all of the most common web browsers and therefore not requiring a plugin to run. This means it can be run from virtually anywhere with an internet connection (most areas of the country have access to the internet nowadays) and could potentially be accessed from a number of devices if they support the required JavaScript functions.

A web-based bespoke software solution would be able to do everything that non-web-based solution could do:

* It will follow the principals of the conservation of energy and momentum
* It will display to a high degree of accuracy elastic and inelastic collisions
* It will work exactly the same as a normal desktop application, or any other web application, as well as being available from any computer with internet access.

**Bespoke software (desktop):**

A system that would be designed to meet all the requirements of the customer. It would be designed for the Windows operating system as most personal computers, at home or at college, run on it. This means that it could be built in almost any language. I would choose Java as this application requires a lot of visual effects and Java has a very good API for achieving this. It also happens to be supported on most of the popular operating systems. The code is also easy to understand and write, and it supports the OOP (Particle-Oriented Programming) model which makes extending the application, if written correctly, much easier. It has the advantage of not requiring an internet connection to run. The disadvantage of it is that Java requires a plugin to execute code, which could be a problem for systems with not much hard-drive space or systems that do not support it.

The user will be able to do everything listed in “Project Particleives”. It would cost a lot of money and also take some time to complete.

## Justification of chosen solution

I have chosen the web-based bespoke software solution for a number of reasons. Compared with the online software, it would be designed to be used in a more academic environment and therefore provide a lot more functionality than that system currently has. Other reasons outlining the disadvantages of the online software solution are:

* It requires a plugin to run
* does not allow user to modify many variables
* does not include a graph of energy changes over time
* does not display energy of each particle
* does not allow user to add particles
* does not allow saving of simulation (saving variables, masses, directions)

My main reason for not electing to go with the standard software solution is the high price due to it containing many other physics solutions that would not be used as part of the current teaching of the course. It could also only be used at the college as the license only applies to the premises of the buyer. This means that the students could not access the simulations from home and the teacher could therefore not set it as homework. Furthermore, as with any download, it can require a considerable time to download and run. With the web-based bespoke system this would not be a problem as it runs through a web browser.

In conclusion, I think that the web-based bespoke software offers the best possible solution. It may be expensive and take time, but the reward will be higher because the customer (the teacher) can request exactly what they want for their individual needs. In an academic environment, this is essential for helping students understand the difficult topics of physics.

# Design

Ball

Physics Object

Ball Environment

Physics

## Overall System Design

|  |  |
| --- | --- |
| **Input** | **Processes** |
| Particle variables: mass, velocity, coefficient of restitution, radius, speed of simulation | Take user input  Add/delete/modify particle  Update simulation  Check for collisions  Handle collision  Draw simulation  Update graph  Draw graph |
| **Output** | **Storage** |
| Display simulation on the screen  Display graph on the screen | No data is stored in this system |

Add/delete/modify particle

Take user input

Variables

Keyboard

Handle collisions

Check for collisions

Update Simulation

Update graph

Simulation

Screen

Graph

Screen

## Modular Structure

There will be two types of modular systems governing the overall simulation. First, there is one for each HTML panel. One will contain buttons for controlling particles, one will contain a graph, one will contain buttons and sliders for adding particles, and one will again contain buttons for showing collision information. The second modular system is built around the internal functions for the JavaScript collision system, and on abstracting the physics simulation away from the rendering system. The structure could look something like this:

Key: \* = reference to other module

**Simulation Widget**

JavaScript widgets

Simulation

Graph

\*Physics

**Physics**

Ball Environment

Physics Object

Ball

Update Position

Draw Ball

This is the structure of the index html page.

**Ecollision**

Index.html

HTML Widgets

Graph

Simulation controls

Debug controls

Ball controls

\*Simulation

## OO Model

**Physics**

Simulation Engine

Physics Object

Particle

Update Position

Draw Particle

**Widgets**

Graph

Simulation

**Math**

Position Vector

## Class Structures

**CLASS** PhysicsObject:

**FIELDS:**

**PUBLIC:**

**INT** x

**INT** y

**INT** Mass

**INT** XVelocity //velocity in the x direction

**INT** YVelocity //velocity in the y direction

**DisplayParticle** DisplayParticle

**PRIVATE:**

**METHODS:**

**PUBLIC:**

**PROC** Update() //update position etc

**PROC** Draw(x, y) //draw the particle to screen

**PRIVATE:**

**CLASS** Particle **EXTENDS** PhysicsObject:

**FIELDS:**

**PUBLIC:**

**INT** Radius

**STRING** Style //color, gradients, etc

**FLOAT** CoefficientOfRestitution

**PRIVATE:**

**BOOLEAN** IsSelected

**METHODS:**

**PUBLIC:**

**PRIVATE:**

**CLASS** SimulationEngine:

**FIELDS:**

**PUBLIC:**

**INT** Width

**INT** Height

**ARRAY OF Particle** Particles

**PRIVATE:**

**METHODS:**

**PUBLIC:**

**BOOLEAN** TestCollision(particle1, particle2, collision)

**BOOLEAN** HandleEdgeCollision(particle)

**BOOLEAN** HandleCollision(collision)

**PRIVATE:**

**CLASS** PositionVector:

**FIELDS:**

**PUBLIC:**

**INT** XComponent

**INT** YComponent

**PRIVATE:**

**METHODS:**

**PUBLIC:**

**INT** DocProduct(positionVector)

**INT** GetMagnitude()

**INT** GetMagnitudeSqr() //get the magnitude squared

**PROC** Rotate(angle)

**PRIVATE:**

**CLASS** Widget:

**FIELDS:**

**PUBLIC:**

**INT** Width

**INT** Height

**Stage** Stage

**PRIVATE:**

**METHODS:**

**PUBLIC:**

**PROC** Init()

**PROC** Destroy()

**PROC** Restart()

**PROC** Stop()

**PROC** Draw()

**PRIVATE:**

**CLASS** Simulation **EXTENDS** Widget:

**FIELDS:**

**PUBLIC:**

**ARRAY OF PhysicsParticle** Particles

**PRIVATE:**

**INT** GameRate

**INT** UpdateDuration

**METHODS:**

**PUBLIC:**

**Particle** AddParticle(x, y, style)

**PROC** DeleteParticle(index)

**PROC** UpdateSimulation()

**PROC** SetGameRate(newGameRate)

**FLOAT** CalculateEnergy()

**PRIVATE:**

**CLASS** Graph **EXTENDS** Widget:

**FIELDS:**

**PUBLIC:**

**FLOAT** ScaleX

**FLOAT** ScaleY

**PRIVATE:**

**FLOAT** OffsetX

**FLOAT** OffsetY

**ARRAY OF Point** Data

**METHODS:**

**PUBLIC:**

**PROC** AddData(dataX, dataY)

**PROC** ZoomIn()

**PROC** ZoomOut()

**PROC** Calibrate()

**PROC** AttachSimulation(simulation)

**PROC** DetachSimulation()

**PRIVATE:**

**CLASS** Collision **EXTENDS** Widget:

**FIELDS:**

**PUBLIC:**

**PhysicsParticle** Particle1

**PhysicsParticle** Particle2

**INT** Time

**PRIVATE:**

**METHODS:**

**PUBLIC:**

**PRIVATE:**

## Input validations

All sliders are limited between values so there can never be an invalid input from these sliders.

|  |  |  |
| --- | --- | --- |
| **Value** | **Limits** | **Type** |
| Velocity | The user may drag the mouse as far as the width of the simulation | Misc |
| Mass | 1 – 1000 kg | Slider |
| Coefficient Of Restitution | 0.0 – 1.0 | Slider |
| Radius | 5 - 30 | Mouse wheel |
| Simulation Speed | 0.0 – 2.0 | Slider |
| Graph zoom | The graph can be zoomed out by 5 times and zoomed in by 5 times | Buttons |

## File organisation and processing

## Storage media and format

## Algorithms

1. Start/Pause Simulation – starts the simulation

**IF** StartButtonPressed **THEN**

Started = **NOT** Started //Toggles the start variable

**ENDIF**

1. Restart Simulation – restarts the simulation

**IF** RestartButtonPressed **THEN**

**CALL** RestoreParticlesToSavedState

**ENDIF**

**PROCEDURE** RestoreParticlesToSavedState:

**BEGIN**

Particles = LastSavedParticles

**END PROCEDURE**

1. Step Back – reverts the simulation back by one tick (or update).

**IF** StepBackButtonPressed **THEN**

**IF NOT** Started **THEN**

**CALL** StepBack

**ENDIF**

**ENDIF**

**PROCEDURE** StepBack:

**BEGIN**

**CALL** UpdateSimulationWithOppositeValues

**END PROCEDURE**

1. Step Forward – moves the simulation forward by one tick (or update).

**IF** StepForwardButtonPressed **THEN**

**IF NOT** Started **THEN**

**CALL** StepForward

**ENDIF**

**ENDIF**

**PROCEDURE** StepForward:

**BEGIN**

**CALL** UpdateSimulation

**END PROCEDURE**

1. Trace paths of particles – traces the path of each particle on the screen

**IF** TracePathsButtonSelected **THEN**

TracePaths = **NOT** TracePaths //Toggles the trace paths variable

**ENDIF**

1. Add particle – add a particle to the stage

**IF** AddButtonPressed **THEN**

**CALL** AddParticle(randomX, randomY, radiusFieldValue, massFieldValue)

**ENDIF**

1. Delete particle – deletes a selected particle from the stage

**IF** DeleteButtonPressed **THEN**

**CALL** DeleteParticle(selectedParticleIndex)

**ENDIF**

1. Enable Gravity

**IF** EnableGravityButtonPressed **THEN**

Gravity = True

**ENDIF**

**FUNCTION** GetTimeOfCollision(Particle1, Particle2)

**BEGIN**

**STORE** PositionDifference = **NEW** PositionVector(Particle1.x-Particle2.x, Particle1.y-Particle2.y)

**STORE** VelocityDifference = **NEW** PositionVector(Particle1.XVelocity-Particle2.XVelocity, Particle1.YVelocity-Particle2.YVelocity)

//Do the quadratic formula to get time

**STORE** a = **CALL** VelocityDifference.GetMagnitudeSquared();

**STORE** b = **CALL** VelocityDifference.GetDotProduct(PositionDifference)

**STORE** c = **CALL** PositionDifference.GetMagnitudeSquared()- Square(Particle1.radius+Particle2.radius)

//Get the discriminant

**STORE** Discriminant = (b \* b) - (4 \* a \* c)

//If there are one or more solutions

**IF** (Discriminant >= 0) **THEN**

**//**Quadratic formula = (-b +- SquareRoot(Sqr(b)-4\*a\*c))/(2\*a)

//With quadratic equations you get two results usually

**STORE** Time1 = (-b - SquareRoot(Discriminant)) / (2 \* a)

**STORE** Time2 = (-b + SquareRoot(Discriminant)) / (2 \* a)

**END IF**

//Check which result is valid (must be between 0 and 1)

**IF** (Time1 > 0.0 **AND** Time1 <= 1.0) **THEN**

**RETURN** Time1

**ELSE IF** (Time2 > 0.0 **AND** Time2 <= 1.0) **THEN**

**RETURN** Time2

**ELSE**

**RETURN** 1.0

**END FUNCTION**

**FUNCTION** TestCollision(Particle1, Particle2, Collision):

**BEGIN**

**STORE** DiffX = Particle1.x-Particle2.x

**STORE** DiffY = Particle1.y-Particle2.y

**STORE** TotalRadius = Particle1.radius+Particle2.radius

**IF** (DiffX\*DiffX + DiffY\*DiffY) <= (TotalRadius\*TotalRadius) **THEN**

**STORE** TimeOfCollision = GetTimeOfCollision(Particle1, Particle2)

Collision.Time = TimeOfCollision

**RETURN TRUE**

**END IF**

**RETURN FALSE**

**END FUNCTION**

**PROCEDURE** HandleEdgeCollision(particle):

**BEGIN**

**IF** (Particle.x-Particle.radius < 0)

**THEN**

Particle.x = 0 + Particle.radius

Particle.xVel = Particle.xVel \* -1 //reversing the x velocity

**ELSE IF** (Particle.x+Particle.radius > SimulationWidth)

**THEN**

Particle.x = SimulationWidth-Particle.radius

Particle.xVel = Particle.xVel \* -1 //reversing the x velocity

**END IF**

**IF** (Particle.y-Particle.radius < 0)

**THEN**

Particle.y = 0 + Particle.radius

Particle.yVel = Particle.yVel \* -1 //reversing the y velocity

**ELSE IF** (Particle.y+Particle.radius > SimulationHeight)

**THEN**

Particle.y = SimulationHeight-Particle.radius

Particle.yVel = Particle.yVel \* -1 //reversing the y velocity

**END IF**

**END PROCEDURE**

**PROCEDURE** SeperateParticles(particle1, particle2, collision):

**BEGIN**

Particle1.x -= Particle1.XVelocity \* Collision.Time

Particle1.x -= Particle1.YVelocity \* Collision.Time

Particle2.x -= Particle2.XVelocity \* Collision.Time

Particle2.x -= Particle2.YVelocity \* Collision.Time

**END**

**PROCEDURE** HandleCollision(collision):

**BEGIN**

**STORE** SplitVelocityObj1 = SplitVelocities(collision.particle1, collision.particle2) //split velocities of particle 1 into perpendicular and parallel components

**STORE** SplitVelocityObj2 = SplitVelocities(collision.particle2, collision.particle1) //split velocities of particle 2 into perpendicular and parallel components

**CALL** CalculateNewVelocity(Particle1, Particle2, SplitVelocityObj1, SplitVelocityObj2)

**CALL** CalculateNewVelocity(Particle2, Particle1, SplitVelocityObj2, SplitVelocityObj1)

**CALL** SeperateParticles(collision)

**END PROCEDURE**

**PROCEDURE** CalculateNewVelocity(Particle1, Particle2, SplitVelocityObj1, SplitVelocityObj2):

**BEGIN**

**STORE** AngleBetweenParticles = GetAngleBetweenParticles(Particle1, Particle

**STORE** NewVelocity = ((SplitVelocityObj1.x \* (Particle1.mass - Particle2.mass)) + (2 \* Particle2.mass \* SplitVelocityObj2.x)) / (Particle1.mass + Particle2.mass);

**STORE** VelocityParallel= **NEW** PositionVector

VelocityParallel.x = NewVelocity \* Cosine(AngleBetweenParticles)

VelocityParallel.y = NewVelocity \* Sin(AngleBetweenParticles)

**STORE** VelocityPerpendicular= **NEW** PositionVector

VelocityPerpendicular.x = SplitVelocityObj1.y \* Cosine(AngleBetweenParticles)

VelocityPerpendicular.y = SplitVelocityObj1.y \* Sin(AngleBetweenParticles)

**STORE** ResultantVelocity = **NEW** PositionVector

ResultantVelocity.x = VelocityParallel.x + VelocityPerpendicular.x

ResultantVelocity.y = VelocityParallel.y - VelocityPerpendicular.y

**RETURN** ResultantVelocity

**END PROCEDURE**

**FUNCTION** AddParticle(x, y, radius, mass):

**BEGIN**

**STORE** Particle = New Particle

Particle.x = x

Particle.y = y

Particle.radius = radius

Particle.mass = mass

**CALL** AddParticleToParticlesArray(Particle)

**RETURN** Particle

**END FUNCTION**

**PROCEDURE** RemoveParticle(index):

**BEGIN**

**CALL** RemoveParticleFromParticlesArray(index)

**END PROCEDURE**

**PROCEDURE** UpdateSimulation:

**BEGIN**

**FOR** I = 0 **TO** Length Of PhysicsParticles **DO**

**CALL** Update(PhysicsParticles[I])

**CALL** HandleEdgeCollision(PhysicsParticles[I])

**END FOR**

**STORE** CollisionArray <- **NEW ARRAY OF** Collision

**FOR** I = 0 **TO** Length Of PhysicsParticles **DO**

**FOR** J = I+1 **TO** Length Of PhysicsParticles **DO**

**IF** TestCollision(PhysicsParticles[i], PhysicsParticles[j], Collision)

**THEN**

**STORE** Collision.Particle1 = PhysicsParticles[I]

**STORE** Collision.Particle2 = PhysicsParticles[J]

**END IF**

**CALL** AddCollisionToArray(Collision)

**END FOR**

**END FOR**

**CALL** SortCollisionsByTime(CollisionArray)

**FOR** I = 0 **TO** Length Of CollisionArray **DO**

**CALL** HandleCollision(CollisionArray[I])

**END FOR**

**END PROCEDURE**

**PROCEDURE** DrawParticles:

**BEGIN**

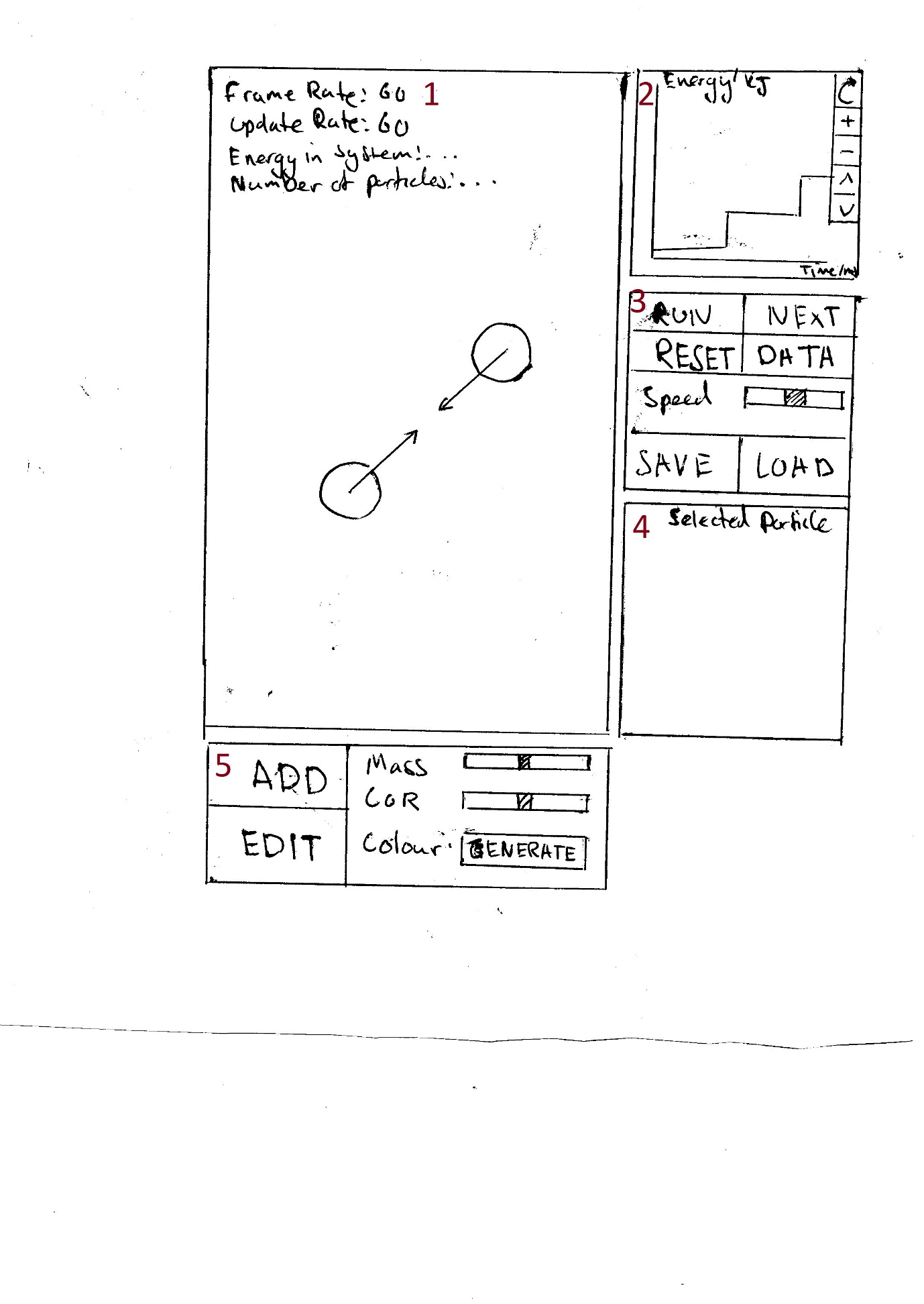
**FOR** I = 0 **TO** Length Of PhysicsParticles **DO**

**CALL** DrawParticle(PhysicsParticles[I]);

**END FOR**

**END PROCEDURE**

## User Interface Design



1. This is where the main simulation takes place; the particles in the simulation are rendered here inside an HTML canvas. When the DATA button is clicked, the text info in the top left will appear. The default is set to hidden, so it will only be toggled when the DATA button is clicked.
2. This is where the energy graph is displayed. From the top-to-bottom order of the buttons, they are:
   * Calibrate
   * Zoom in
   * Zoom out
   * Move up
   * Move down
3. This panel controls the simulation.
   * The RUN button turns into a PAUSE button when clicked, and reverts back when clicked again.
   * The NEXT button moves the simulation forward by one update.
   * The DATA button displays the text info in the top left of the simulation.
   * The Speed slider adjusts the speed of the simulation.
   * The SAVE button saves the current properties of the particles, like position and velocity.
   * The LOAD button loads the previous saved state.
4. This is the panel in which information about a selected particle, such as mass, will be shown.
5. This panel controls the adding, editing and removing of particles.
   * The ADD button makes the overlay appear and a particle can be placed, either in a snap-to or free place system.
   * The EDIT button makes the overlay appear so that particles can be modified, copied, or removed.
   * The Mass slider allows a particle’s mass property to be changed.
   * The CoR slider changes a particle’s coefficient of restitution property.
   * The Colour generator creates a random new colour for a particle.

## Security and Integrity of Data

There is no data stored by this system. This means that no data has to be kept securely.

## System Security

The security of the system will be determined by the place it is hosted. I will use paid hosting, costing roughly £5 a month.

## Test Strategy

The tests of the system will be carried out in order of priority of functionality – for instance, if adding a particle does not work, the rest of the system fails. The method I have elected for this is system testing – checking the core parts of the system. There will also be tests of some basic elastic collisions experiments used in real physics classrooms, for instance Newton’s Cradle.

If a test fails, I will go back to the code and rewrite the parts that are not working. Provided in each test will be a number of the test for reference, a short description of the actual test, the data used in the test (will be left blank if self-explanatory or none is used), and the expected result. The actual results of each test will be shown in a recorded video of all the tests. The types of test are typical, erroneous and boundary.

# System Testing

## Test Plan – Typical Data

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Number** | **Test** | **Test Data** | **Expected Result** |
| 1 | Adding a particle |  | A cross hair should appear so that the user can place the particle at a specific location |
| 2 | Delete a particle |  | The particle should be removed from the simulation |
| 3 | Selecting a particle |  | A red ring should be visible around a particle to signify that it has been selected and information about the particle should appear in the “Selected particle” panel |
| 4 | Speeding up and down the simulation |  | The simulation should speed up and down |
| 5 | Changing a particle’s mass after it has been added |  | The particles mass should change and affect its collision with other particles |
| 6 | Pausing and resuming the simulation |  | The simulation should pause and then resume |
| 7 | Zooming in and out of the graph |  | The graph will zoom in and out |
| 8 | Moving the graph up and down |  | The graph should move up and down |
| 9 | Calibrate the graph |  | The graph will adjust the y render position back to zero |
| 10 | Reset the simulation |  | All the particles should disappear and the graph should be cleared |
| 11 | Stepping the simulation forward by one update |  | The simulation should move forward by one tick |
| 12 | Save and load the simulation’s state |  | The simulation’s state will be saved and then loaded to the last saved state |
| 13 | Generating a particle colour |  | A random colour should be generated |
| 14 | Making a Newton’s Cradle |  | A teaching example in which all the momentum and energy are conserved. The results should match that of a real Newton’s Cradle (see here https://www.youtube.com/watch?v=JadO3RuOJGU) |
| 15 | Colliding two particles with the same mass and coefficient of restitution, and equal but opposite velocities |  | Both particles should stop completely |

## Test Plan – Erroneous Data

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Number** | **Test** | **Test Data** | **Expected Result** |
| 1 | Using Firefox instead of Chrome |  | The simulation should still work |
| 2 | Using Internet Explorer instead of Chrome |  | The simulation should still work but the html won’t be formatted properly |
| 3 | Using the system on a mobile (Nexus 5) | The Google Chrome web browser includes multiple mobile emulators, including Nexus 5. | The simulation should still work but might not have all the same functionality |

## Test Plan – Boundary Data

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Number** | **Test** | **Test Data** | **Expected Result** |
| 1 | Colliding a particle of maximum mass with particle of minimum mass | **Both particles:**  CoR = 1.0  Radius = 30  **Particle one:**  Velocity = 50 px/s  Mass = 1000 kg  **Particle two:**  Velocity = 0 px/s  Mass = 1kg | The particles should be still obey the laws of conservation of energy and momentum. After collision, both particles should be moving in the same direction. The second particle should also have |
| 2 | Colliding a particle of minimum with particle of minimum mass |  |  |
| 3 | Changing simulation speed to maximum value | Simulation speed = 2.0 | The simulation should still run the same but be twice as fast |
| 4 | Changing simulation speed to minimum value | Simulation speed = 0.0 | The simulation should stop |
| 5 | Adding more than 100 particles | 101 particles | There should be a message that pops up saying maximum number of particles added. |
| 6 | Placing a particle at the edges/corners of the system |  | The particles should move out of the corner/edge. |

# System Maintenance

## System Overview

This system is a physics collision simulation that provides teachers a resource to teach the concepts of elastic and inelastic collisions, and the principals of energy conservation. The system is web-based and written in mostly JavaScript. It is hosted by a web-hosting company which has FTP and SSH access.

The system is designed with ease of updates in mind. Everything is modular, with two main divisions: HTML panels and JavaScript widgets. The HTML panels are split off from the rest of the code and therefore exist independently from the other code. This means they can be updated without affecting the rest of the system. Inside two of these panels is located the simulation and overlay, and an energy graph.

The JavaScript code is also modular with the engine being separated from the rendering system so that it sits on top of the engine. All of the aforementioned JavaScript widgets are then combined in one class located in ecollision.js, an instance of which is passed on to the script which takes user input and deals with it appropriately.

Many individual settings can be changed by changing the eCollisionSettings constant. A list of its properties is provided in the table below:

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | **Default value** |
| simulationWidth | The internal width of the simulation | 1000 |
| simulationHeight | The internal height of the simulation | 1000 |
| simulationCanvas | The name of the canvas to draw the simulation to | “simulation-canvas” |
| graphCanvas | The name of the canvas to draw the graph to | "graph-canvas" |
| overlayCanvas | The name of the canvas to draw the overlay to | "overlay-canvas" |
| refreshRate | The number of redraws per second | 60 |
| updateRate | The number of updates per second | 60 |
| showVelocities | Used internally as a flag to display the velocities of the particles | False |
| enableInterpolation | Make the simulation look smooth at low update rates | True |
| maxTraceLength | The maximum number of saved positions for a trace of an particle (the length of its trace tail) | 30 |
| graphScaleX | The default graph scale on the x axis | 1/50 |
| graphScaleY | The default graph scale on the y axis | 5 |
| graphZoomFactor | The factor to zoom in or out by | 1.25 |
| graphMinZoomIndex | The maximum number of times you can zoom out | 5 |
| graphMaxZoomIndex | The maximum number of times you can zoom in | 5 |
| speedConst | The speed modifier of the simulation | 1.0 |
| maxParticles | The maximum number of particles allowed | 100 |
| minRadius | The minimum radius of a particle | 5 |
| maxRadius | The maximum radius of a particle | 30 |
| errorTime | The amount of time to display an error in milliseconds | 5000 |

## Algorthms

**FUNCTION** GetTimeOfCollision(Particle1, Particle2)

**BEGIN**

**STORE** PositionDifference = **NEW** PositionVector(Particle1.x-Particle2.x, Particle1.y-Particle2.y)

**STORE** VelocityDifference = **NEW** PositionVector(Particle1.XVelocity-Particle2.XVelocity, Particle1.YVelocity-Particle2.YVelocity)

//Do the quadratic formula to get time

**STORE** a = **CALL** VelocityDifference.GetMagnitudeSquared();

**STORE** b = **CALL** VelocityDifference.GetDotProduct(PositionDifference)

**STORE** c = **CALL** PositionDifference.GetMagnitudeSquared()- Square(Particle1.radius+Particle2.radius)

//Get the discriminant

**STORE** Discriminant = (b \* b) - (4 \* a \* c)

//If there are one or more solutions

**IF** (Discriminant >= 0) **THEN**

**//**Quadratic formula = (-b +- SquareRoot(Sqr(b)-4\*a\*c))/(2\*a)

//With quadratic equations you get two results usually

**STORE** Time1 = (-b - SquareRoot(Discriminant)) / (2 \* a)

**STORE** Time2 = (-b + SquareRoot(Discriminant)) / (2 \* a)

**END IF**

//Check which result is valid (must be between 0 and 1)

**IF** (Time1 > 0.0 **AND** Time1 <= 1.0) **THEN**

**RETURN** Time1

**ELSE IF** (Time2 > 0.0 **AND** Time2 <= 1.0) **THEN**

**RETURN** Time2

**ELSE**

**RETURN** 1.0

**END FUNCTION**

**FUNCTION** TestCollision(Particle1, Particle2, Collision):

**BEGIN**

**STORE** DiffX = Particle1.x-Particle2.x

**STORE** DiffY = Particle1.y-Particle2.y

**STORE** TotalRadius = Particle1.radius+Particle2.radius

**IF** (DiffX\*DiffX + DiffY\*DiffY) <= (TotalRadius\*TotalRadius) **THEN**

**STORE** TimeOfCollision = GetTimeOfCollision(Particle1, Particle2)

Collision.Time = TimeOfCollision

**RETURN TRUE**

**END IF**

**RETURN FALSE**

**END FUNCTION**

**PROCEDURE** HandleEdgeCollision(particle):

**BEGIN**

**IF** (Particle.x-Particle.radius < 0)

**THEN**

Particle.x = 0 + Particle.radius

Particle.xVel = Particle.xVel \* -1 //reversing the x velocity

**ELSE IF** (Particle.x+Particle.radius > SimulationWidth)

**THEN**

Particle.x = SimulationWidth-Particle.radius

Particle.xVel = Particle.xVel \* -1 //reversing the x velocity

**END IF**

**IF** (Particle.y-Particle.radius < 0)

**THEN**

Particle.y = 0 + Particle.radius

Particle.yVel = Particle.yVel \* -1 //reversing the y velocity

**ELSE IF** (Particle.y+Particle.radius > SimulationHeight)

**THEN**

Particle.y = SimulationHeight-Particle.radius

Particle.yVel = Particle.yVel \* -1 //reversing the y velocity

**END IF**

**END PROCEDURE**

**PROCEDURE** SeperateParticles(particle1, particle2, collision):

**BEGIN**

**END**

**PROCEDURE** HandleCollision(collision):

**BEGIN**

**STORE** SplitVelocityObj1 = SplitVelocities(collision.particle1, collision.particle2) //split velocities of particle 1 into perpendicular and parallel components

**STORE** SplitVelocityObj2 = SplitVelocities(collision.particle2, collision.particle1) //split velocities of particle 2 into perpendicular and parallel components

**CALL** CalculateNewVelocity(Particle1, Particle2, SplitVelocityObj1, SplitVelocityObj2)

**CALL** CalculateNewVelocity(Particle2, Particle1, SplitVelocityObj2, SplitVelocityObj1)

**CALL** SeperateParticles(collision)

**END PROCEDURE**

**PROCEDURE** CalculateNewVelocity(Particle1, Particle2, SplitVelocityObj1, SplitVelocityObj2):

**BEGIN**

**STORE** AngleBetweenParticles = GetAngleBetweenParticles(Particle1, Particle

**STORE** NewVelocity = ((SplitVelocityObj1.x \* (Particle1.mass - Particle2.mass)) + (2 \* Particle2.mass \* SplitVelocityObj2.x)) / (Particle1.mass + Particle2.mass);

**STORE** VelocityParallel= **NEW** PositionVector

VelocityParallel.x = NewVelocity \* Cosine(AngleBetweenParticles)

VelocityParallel.y = NewVelocity \* Sin(AngleBetweenParticles)

**STORE** VelocityPerpendicular= **NEW** PositionVector

VelocityPerpendicular.x = SplitVelocityObj1.y \* Cosine(AngleBetweenParticles)

VelocityPerpendicular.y = SplitVelocityObj1.y \* Sin(AngleBetweenParticles)

**STORE** ResultantVelocity = **NEW** PositionVector

ResultantVelocity.x = VelocityParallel.x + VelocityPerpendicular.x

ResultantVelocity.y = VelocityParallel.y - VelocityPerpendicular.y

**RETURN** ResultantVelocity

**END PROCEDURE**

**FUNCTION** AddParticle(x, y, radius, mass):

**BEGIN**

**STORE** Particle = New Particle

Particle.x = x

Particle.y = y

Particle.radius = radius

Particle.mass = mass

**CALL** AddParticleToParticlesArray(Particle)

**RETURN** Particle

**END FUNCTION**

**PROCEDURE** RemoveParticle(index):

**BEGIN**

**CALL** RemoveParticleFromParticlesArray(index)

**END PROCEDURE**

**PROCEDURE** UpdateSimulation:

**BEGIN**

**FOR** I = 0 **TO** Length Of PhysicsParticles **DO**

**CALL** Update(PhysicsParticles[I])

**CALL** HandleEdgeCollision(PhysicsParticles[I])

**END FOR**

**STORE** CollisionArray <- **NEW ARRAY OF** Collision

**FOR** I = 0 **TO** Length Of PhysicsParticles **DO**

**FOR** J = I+1 **TO** Length Of PhysicsParticles **DO**

**IF** TestCollision(PhysicsParticles[i], PhysicsParticles[j], Collision)

**THEN**

**STORE** Collision.Particle1 = PhysicsParticles[I]

**STORE** Collision.Particle2 = PhysicsParticles[J]

**END IF**

**CALL** AddCollisionToArray(Collision)

**END FOR**

**END FOR**

**CALL** SortCollisionsByTime(CollisionArray)

**FOR** I = 0 **TO** Length Of CollisionArray **DO**

**CALL** HandleCollision(CollisionArray[I])

**END FOR**

**END PROCEDURE**

## Procedure and variable lists

**Class:** SimulationEngine

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| setBounds | Public | width, height |  | Sets the width and height of the engine |
| reset | Public |  |  | Removes all particles |
| addParticle | Public | particle |  | Adds a particle |
| removeParticle | Public | index |  | Removes a particle |
| getParticle | Public | index | Particle | Returns the particle at the index |
| numOfParticles | Public |  | Integer | Returns the number of particles in the engine |
| edgeCollision | Public | rebound |  | Handles an edge collision. If rebound is set, it will make the particle rebound |
| collide | Public | particle1, particle2, collision | Boolean | Checks if two particles are colliding |
| splitVelocity | Private | particle1, particle2 | PVector | Splits the velocity into parallel and perpendicular (to the other particle) components |
| handleCollision | Public | collision |  | Handles the actual collision between two particles |
| seperateParticles | Public | collision, particle1, particle2 |  | Seperates two particles |
| update | Public |  |  | Updates the simulation |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| width | Public | Integer | The width of the simulation engine |
| height | Public | Integer | The height of the simulation engine |
| particles | Private | Array of Particle | The particles stored in the engine |

**Class:** Widget

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| init | Public |  |  | This is called when a widget is created |
| addEventListener | Public | event, handler |  | Adds an event listener to the stage of the widget |
| draw | Public | interpolation |  | This is called when the widget is about to be drawn |
| restart | Public |  |  | This is called when the widget needs to restart |
| stop | Public |  |  | This is called when the simulation is stopped |
| resume | Public |  |  | This is called whne the simulation is resumed |
| resize | Public | newWidth, newHeight |  | This is called whne the simulation is resized |
| show | Public |  |  | Hides the widget |
| hide | Public |  |  | Shows the widget |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| hidden | Public | Boolean | The hidden flag – true when hidden |
| canvasName | Public | String | The name of the canvas that is loaded |
| canvas | Public | Canvas | The jQuery canvas particle |
| width | Public | Integer | The width of the canvas |
| height | Public | Integer | The height of the canvas |
| stage | Public | Easeljs.Stage | The stage particle |

**Class:** Simulation

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| resize | Public | newWidth, newHeight |  | Resizes the simulation |
| addParticle | Public | x, y, mass, radius, style | Particle | Adds a particle to the simulation and returns it |
| removeParticle | Public | index |  | Removes a particle from the simulation |
| loadParticles | Public | toBeLoaded |  | Loads the particles “toBeLoaded” |
| saveParticles | Public | saved | Integer | Stores all the particles in the simulation in “saved” |
| removeSelected | Public | index |  | Removes the currently selected particle |
| getSelectedID | Public |  | Integer | Gets the current selected particle’s ID |
| restart | Public |  |  | Restarts the simulation (removes all the particles) |
| draw | Public |  |  | Draws the simulation |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| engine | Public | SimulationEngine | The current simulation engine |
| selected | Public | Integer | The index of the currently selected particle |

**Class:** Graph

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| init | Public |  |  | Initialises the graph |
| draw | Public | interpolation |  | Draws the graph |
| restart | Public |  |  | Restart (resets) the graph |
| calibrate | Public |  |  | Calibrates the graph |
| zoomIn | Public |  |  | Zooms in the graph |
| zoomOut | Public |  |  | Zooms out the graph |
| moveUp | Public |  |  | Moves the graph up |
| moveDown | Public |  |  | Moves the graph down |
| addData | Public | x, y |  | Adds data to the graph |
| updateData | Public |  |  | Updates the data array |
| getEnergy | Public |  | Integer | Gets the energy of the particles in the simulation |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| scaleX | Public | Boolean | The current scale of the graph on the x axis |
| scaleY | Public | String | The current scale of the graph on the y axis |
| x | Public | Canvas | The top left x position of the graph |
| y | Public | Integer | The top left y position of the graph |
| engine | Public | SimulationEngine | The current engine attached to the graph |
| graph | Private | createjs.Shape | The place that the graph draws its data to |
| offsetX | Private | Integer | The x offset of the graph |
| offsetY | Private | Integer | The y offset of the graph |
| userY | Private | Integer | The user’s offset on the y axis |
| data | Private | Array of Point | The data to be displated by the graph |
| start | Private | Integer | The start of the data on the graph. This is because the data is rotated so that is increased |
| maxLen | Private | Integer | The maximum length of the graph’s data |
| updated | Private | Boolean | The flag to tell if the graph’s data has been updated |
| currX | Private | Integer | The last x value added |
| currY | Private | Integer | The last y value added |
| zoomIndex | Private | Integer | The current zoom amount (index) |

**Class:** PhysicsParticle

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| capture | Public |  |  | Save the current x and y |
| update | Public | interpolation |  | Update the position of the particle |
| addEventHandler | Public | event, handler |  | Add an event to the display particle |
| getEnergy | Public |  | Integer | Get the kinetic energy of the particle |
| draw | Public | x, y |  | Draw the particle |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| x | Public | Integer | The current x value |
| y | Public | Integer | The current y value |
| lastX | Public | Integer | The last x value stored by capture |
| lastY | Public | Integer | The last y value stored by capture |
| xVel | Public | Integer | The velocity on the x axis |
| yVel | Public | Integer | The velocity on the y axis |
| mass | Public | Integer | The mass of the particle |
| displayObj | Public | Easeljs.Shape | The place where the particle draws to |

**Class:** Overlay

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| gcd | Private | a, b | Integer | Gets the greatest common divisor of two numbers |
| handleMouseWheel | Private | ev |  | Handles the mouse scroll event – makes the particle radius bigger or smaller |
| resize | Public | width, height |  | Resizes the overlay |
| init | Public |  |  | Initialises the overlay |
| handleMouseMove | Private | ev |  | Handles the mouse move event – moves the particle |
| handleClick | Private | ev |  | Handles the mouse click event |
| draw | Public | interpolation |  | Draws the overlay |
| reset | Private |  |  | Resets the overlay |
| beginAdd | Public | mass, cOR, style |  | Begins the add process |
| beginEdit | Public | x, y |  | Begins the edit process |
| end | Public |  |  | Ends the add or edit process |
| getCurrentParticle | Public |  | Particle | Gets the current particle being added |
| getMode | Public |  | Integer | Gets the mode – either add (0) or edit (1). |

**Constants:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Constant** | **Access** | **Value** | **Description** |
| INDEX\_PLACE | Private | 0 | The index number of when particles are being placed |
| INDEX\_VELOCITY | Private | 1 | The index number of when particles are being given velocities |
| INDEX\_MODIFY | Private | 2 | The index number of when particles are being modified |
| MODE\_ADD | Private | 0 | The mode number of when add mode |
| MODE\_EDIT | Private | 1 | The mode number of edit mode |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| index | Public | Integer | The current index, or stage, of the adding or editing process |
| mode | Public | Integer | The current mode |
| mouseX | Public | Integer | The mouse’s x position |
| mouseY | Public | Integer | The mouse’s y position |
| crossX | Public | Integer | The snap-to x position |
| crossY | Private | createjs.Shape | The snap-to y position |
| velocityLine | Private | Easeljs.Shape | The line that appears when a particle is given velocity |
| infoText | Private | Easeljs.Text | The information about the particle’s velocity |
| errorText | Private | Easeljs.Text | The error text box that pops up when an error occurs |
| errorTimer | Private | Integer | The timer for the error to appear |
| showError | Private | Boolean | True if an error should appear |
| modeText | Private | Easeljs.Text | The text box to show which mode is currently being used (Add or Edit) |
| freePlace | Private | Boolean | True if Shift is pressed – use free place mode |
| copyPlace | Private | Boolean | True if CTRL is pressed – copy selected particle |
| lastX | Private | Integer | The x position of the picked up particle in edit mode |
| lastY | Private | Integer | The y position of the picked up particle in edit mode |
| tempParticle | Private | Particle | The current particle being added (add mode) or selected (edit mode) |

**Class:** Particle

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| draw | Public | x, y |  | Draws the particle |
| select | Public |  |  | Sets the selected flag that makes “draw” show debug info |
| deselect | Public |  |  | Deselects the particle |
| update | Public |  |  | Updates the particle |
| copy | Public |  | Particle | Makes an exact copy of the particle |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| radius | Public | Integer | The radius of the particle |
| style | Public | String | The colour of the particle |
| cOR | Public | Double | The coefficient of restitution of the particle |
| selected | Public | Boolean | True if selected |
| pastPositions | Private | Array of Point | The past positions of the particle |

**Class:** PVector

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **Access** | **Parameters** | **Return value** | **Description** |
| getMagnitude | Public |  | Integer | Gets the magnitude of the vector |
| getMagnitudeNS | Public |  | Integer | Gets the squared magnitude of the vector |
| dotProduct | Public | vec | Integer | Gets the dot product between this vector and vec |
| getNormal | Public |  | PVector | Gets the normal to this vector |
| rotate | Public | angle |  | Rotates this vector by angle |

**Fields:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Access** | **Type** | **Description** |
| x | Public | Integer | The x component of the vector |
| y | Public | Integer | The y component of the vector |

## Annotated listings / screens

*//Collision class that stores which particles collided and a time constant of how much to seperate them by.*

function Collision() {

this.time = 0.0;

this.particle = null;

this.particle2 = null;

}

*//This detects if there is an edge collision.*

this.edgeCollision = function (particle, rebound) {

var cOR = particle.cOR;

*//If the particle is outside the width set, it will be placed back inside*

if (particle.x + particle.radius >= this.width) {

if (rebound) {

*//The particle may lose energy if it's coefficient of restitution is less than 1.*

particle.xVel \*= -cOR;

particle.yVel \*= cOR;

} else {

particle.x = this.width - particle.radius;

}

} else if (particle.x - particle.radius <= 0) {

if (rebound) {

particle.xVel \*= -cOR;

particle.yVel \*= cOR;

} else {

particle.x = particle.radius;

}

}

*//If the particle is outside the height set, it will be placed back inside*

if (particle.y + particle.radius >= this.height) {

if (rebound) {

particle.xVel \*= cOR;

particle.yVel \*= -cOR;

} else {

particle.y = this.height - particle.radius;

}

} else if (particle.y - particle.radius <= 0) {

if (rebound) {

particle.xVel \*= cOR;

particle.yVel \*= -cOR;

} else {

particle.y = particle.radius;

}

}

}

*//This functions checks for a collision and returns true or false if yes. It also calculates the required amount of time to seperate the particles.*

this.collide = function(particle, particle2, collision) {

*//Take the distances between the particles on the x and y axes*

var dX = particle2.x - particle.x;

var dY = particle2.y - particle.y;

*//Calculate the square of the distance*

var sqr = (dX \* dX) + (dY \* dY);

var r = particle2.radius + particle.radius;

*//Could sqrt to get the distance, but there's no need because the otherside would also have to be sqrted*

if (sqr < r \* r) {

*//Now to get the time constant between the last update and this update at which the particles would have collided perfectly*

*//Put into pvectors as we need to get the dot products*

var pDiff = new PVector(particle.x - particle2.x, particle.y - particle2.y);

var vDiff = new PVector(particle.xVel - particle2.xVel, particle.yVel - particle2.yVel);

*//The following can be derived thus:*

*// At the time of a perfect collision:*

*// let dx = particle2\_currentX - particle1\_currentX*

*// let dy = particle2\_currentY - particle1\_currentY*

*//*

*// let diffVelocityX = particle2\_velocityX-particle1\_velocityX*

*// let diffVelocityY = particle2\_velocityY-particle1\_velocityY*

*//*

*// let particle1\_xFinal = particle1\_currentX - (particle1\_velocityX \* time)*

*// let particle1\_yFinal = particle1\_currentY - (particle1\_velocityY \* time)*

*//*

*// let particle2\_xFinal = particle2\_currentX - (particle2\_velocityX \* time)*

*// let particle2\_yFinal = particle2\_currentY - (particle2\_velocityY \* time)*

*//*

*// We need to solve for time:*

*// let diffX = particle2\_xFinal-particle1\_xFinal*

*// let diffY = particle2\_yFinal-particle1\_yFinal*

*//*

*// Rearranging and subbing-in this gives:*

*// diffX = particle2\_currentX - (particle2\_velocityX \* time) - particle1\_currentX - (particle1\_velocityX \* time)*

*// = (particle2\_currentX - particle1\_currentX) - time\*(particle2\_velocityX-particle1\_velocityX)*

*// = dx - time\*diffVelocityX*

*//*

*// diffY = particle2\_currentY - (particle2\_velocityY \* time) - particle1\_currentY - (particle1\_velocityY \* time)*

*// = (particle2\_currentY - particle1\_currentY) - time\*(particle2\_velocityY-particle1\_velocityY)*

*// = dy - time\*diffVelocityY*

*//*

*// Now it is just like a collision check, as above, except this time we can solve for time:*

*// let sqr = sqr(diffX) + sqr(diffY)*

*// = sqr(dx - time\*diffVelocityX) + sqr(dy - time\*diffVelocityY)*

*//*

*// Now to expand the brackets:*

*// sqr = sqr(dx) - 2\*time\*diffVelocityX\*dx + sqr(time)\*sqr(diffVelocityX) + sqr(dy) - 2\*time\*diffVelocityY\*dy + sqr(time)\*sqr(diffVelocityY)*

*//*

*// We're trying to find time, and or a perfect collision, sqr must equal the sum of the radii squared*

*// So our quadratic equation is:*

*// let radiiSqred = sqr(particle1\_radius+particle2\_radius)*

*//*

*// sqr = a\*sqr(time) + b\*time + c-radiiSqred = 0*

*// a = sqr(diffVelocityX)+sqr(diffVelocityY) (NOTE: dotProduct as below)*

*// b = -2\*(dx\*diffVelocityX + diffVelocityY\*dy) (NOTE: dotProduct as below)*

*// c = sqr(dx)+sqr(dy) - radiiSqred*

*//*

*// We then use the quadratic formula (-b +- sqrt(b\*b - 4\*a\*c))/(2\*a) to calculate time*

var a = vDiff.dotProduct(vDiff);

var b = -2 \* vDiff.dotProduct(pDiff);

var c = (pDiff.dotProduct(pDiff)) - (r \* r);

var discr = (b \* b) - (4 \* a \* c);

var t = 0.0;

var t2 = 0.0;

if (discr >= 0) {

t = (-b - Math.sqrt(discr)) / (2 \* a);

t2 = (-b + Math.sqrt(discr)) / (2 \* a);

}

if (t > 0.0 && t <= 1.0)

collision.time = t;

else if (t2 > 0.0 && t2 <= 1.0)

collision.time = t2;

else

collision.time = 1.0;

return true;

}

return false;

}

*//This function splits particle1's velocity into parallel and perpendicular components.*

function splitVelocity(particle1, particle2) {

*//The overall process of visualising how this works is:*

*// 1. Imagine the collision happening such that particle1's velocity is rotated so that it is in one dimension, or the x-axis*

*// 2. Then calculate its parallel and perpendicular components when it collides.*

*// 3. Finally rotate these components back by the same amount.*

*//*

*//Store the current velocity in a vector structure*

var velocity = new PVector(particle1.xVel, particle1.yVel);

*//Default angle*

var a = Math.PI / 2;

*//Calculate the angle of the velocity*

if (particle1.xVel !== 0) {

a = Math.atan(particle1.yVel / particle1.xVel);

}

*//Calculate the magnitude as if it were on the x-axis only. This was originally part of my rotate function.*

*//See math/pector.js for similarities*

var magnitude = (particle1.xVel \* Math.cos(-a) - particle1.yVel \* Math.sin(-a)) \* particle1.cOR;

var dx = particle1.x - particle2.x;

var dy = particle1.y - particle2.y;

*//Calculate the position angle*

var ang = 0;

if (dx !== 0) {

ang = Math.atan(dy / dx);

} else {

ang = Math.atan(dy / (dx - 0.00001));

}

*//This is a simplification of multiple cosines and sines using trig identities. It is essentially doing stages 2 and 3 as stated above.*

velocity.x = magnitude \* (Math.cos(ang - a));

velocity.y = magnitude \* (Math.sin(ang - a));

return velocity;

}

*//This function actually handles the collision between two particles.*

this.handleCollision = function (collision) {

var particle = collision.particle;

var particle2 = collision.particle2;

*//Split the velocities into parallel and perpendicular components. See "splitVelocity" above.*

var thisVel = splitVelocity(particle, particle2);

var particleVel = splitVelocity(particle2, particle);

*//Finally do some real physics. This calculates the new velocities of the parallel components as if they were one-dimensional.*

var newV = ((thisVel.x \* (particle.mass - particle2.mass)) + (2 \* particle2.mass \* particleVel.x)) / (particle.mass + particle2.mass);

var newV2 = ((particleVel.x \* (particle2.mass - particle.mass)) + (2 \* particle.mass \* thisVel.x)) / (particle.mass + particle2.mass);

*//Calculate the angle between the particles*

var ang = Math.atan((particle.y - particle2.y) / (particle.x - particle2.x));

var cosA = Math.cos(ang);

var sinA = Math.sin(ang);

*//Then these new velocityies are split further so they fit the Cartesian coordinate system. They are then added to the remaining velocity from the perpendicular components*

var x1 = (newV \* cosA) + (thisVel.y \* sinA);

var y1 = (newV \* sinA) - (thisVel.y \* cosA);

var x2 = (newV2 \* cosA) + (particleVel.y \* sinA);

var y2 = (newV2 \* sinA) - (particleVel.y \* cosA);

*//Seperate the particles. See "seperateParticles" below.*

this.seperateParticles(collision, particle, particle2);

*//Finally give each particle their new velocities.*

particle.xVel = x1;

particle.yVel = y1;

particle2.xVel = x2;

particle2.yVel = y2;

}

*//This function seperates the particles after collision.*

this.seperateParticles = function(collision, particle, particle2) {

*//Add a small extra amount of time so that the particles can never get stuck on each other*

var t = collision.time + (0.001 \* collision.time);

if (t < 1.0) {

*//Pull both particles back by the perfect collision time. See "collide" function*

particle.x -= particle.xVel \* settings.speedConst \* t;

particle.y -= particle.yVel \* settings.speedConst \* t;

particle2.x -= particle2.xVel \* settings.speedConst \* t;

particle2.y -= particle2.yVel \* settings.speedConst \* t;

} else {

*//Failsafe method of seperating particles*

*//First calculate the overlap*

var dX = particle2.x - particle.x;

var dY = particle2.y - particle.y;

var sqr = (dX \* dX) + (dY \* dY);

var overlap = particle2.radius - Math.abs(Math.sqrt(sqr) - particle.radius) + 0.1;

var vel1 = new PVector(particle.xVel, particle.yVel).getMagnitudeNS()+0.0001;

var vel2 = new PVector(particle2.xVel, particle2.yVel).getMagnitudeNS()+0.0001;

*//Total velocity*

var vT = vel1 + vel2;

*//Work out the first propotion for movement*

var i = vel1 / vT;

ang = Math.atan2(particle.y - particle2.y, particle.x - particle2.x);

*//Move particle*

particle.x += overlap \* Math.cos(ang) \* i;

particle.y += overlap \* Math.sin(ang) \* i;

*//Work out other proportion for movement*

i = 1 - i;

particle2.x -= overlap \* Math.cos(ang) \* i;

particle2.y -= overlap \* Math.sin(ang) \* i;

}

}

*//This function causes the particles to update and react to each other. It is the heart of the system.*

this.update = function () {

*//Loop through the particles, make sure they are not overlapping with the edges, then update their position.*

for (var i = 0; i < particles.length; i++) {

var particle = particles[i];

this.edgeCollision(particle, true);

particle.update();

}

var colParticles = [];

*//Loop through the particles, check for collisions once between pairs of particles.*

*//If colliding, add them to a collision array*

for (var i = 0; i < particles.length; i++) {

var particle = particles[i];

for (var i2 = i + 1; i2 < particles.length; i2++) {

var particle2 = particles[i2];

var collision = new Collision();

if (this.collide(particle, particle2, collision)) {

collision.particle = particle;

collision.particle2 = particle2;

colParticles.push(collision);

}

}

}

*//Loop through the collision array and sort out which one happened first*

colParticles.sort(function (a, b) {

return a.time < b.time;

});

*//Handle the collisions stored in the collision array. See "handleCollision" above*

for (var i = 0; i < colParticles.length; i++) {

var collision = colParticles[i];

this.handleCollision(collision);

}

*//Finally check for an edge collision again but do not rebound the particle*

for (var i = 0; i < particles.length; i++) {

var particle = particles[i];

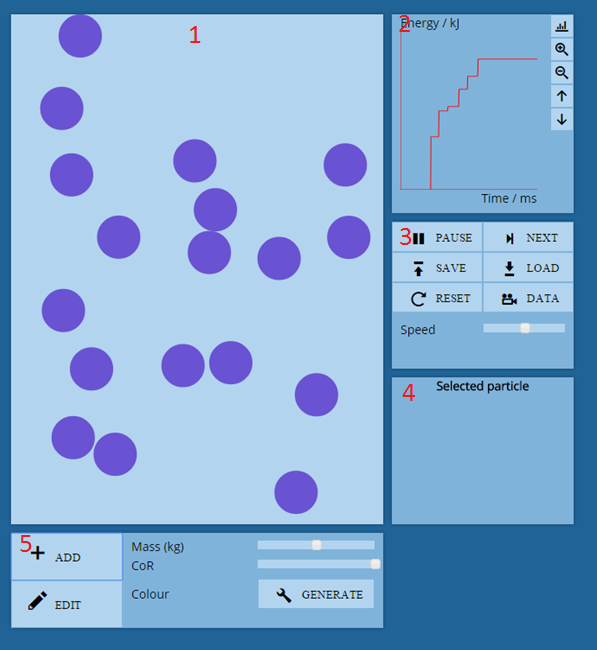
this.edgeCollision(particle, false);

}

}

## Database Definitions

## Forms / screens



1. This is where the main simulation takes place; the particles in the simulation are rendered here inside an HTML canvas. When the DATA button is clicked, the text info in the top left will appear. The default is set to hidden, so it will only be toggled when the DATA button is clicked.
2. This is where the energy graph is displayed. From the top-to-bottom order of the buttons, they are:
   * Calibrate
   * Zoom in
   * Zoom out
   * Move up
   * Move down
3. This panel controls the simulation.
   * The RUN button turns into a PAUSE button when clicked, and reverts back when clicked again.
   * The NEXT button moves the simulation forward by one update.
   * The DATA button displays the text info in the top left of the simulation.
   * The Speed slider adjusts the speed of the simulation.
   * The SAVE button saves the current properties of the particles, like position and velocity.
   * The LOAD button loads the previous saved state.
4. This is the panel in which information about a selected particle, such as mass, will be shown.
5. This panel controls the adding, editing and removing of particles.
   * The ADD button makes the overlay appear and a particle can be placed, either in a snap-to or free place system.
   * The EDIT button makes the overlay appear so that particles can be modified, copied, or removed.
   * The Mass slider allows a particle’s mass property to be changed.
   * The CoR slider changes a particle’s coefficient of restitution property.
   * The Colour generator creates a random new colour for a particle.

# User Manual

## Contents page

Table of Contents

Introduction1

System Requirements4

Installation1

Using the system4

Controlling the simulation5

Run/pause/step forward the simulation6

Saving/loading state 6

Resetting the simulation 6

Getting simulation data, e.g., the directions of the particles 6

Simulation speed 6

Managing particles5

Adding particles 6

Deleting particles 6

Copying particles 6

Setting particle values6

What is “CoR” 6

Graph5

Calibration 6

Zooming6

Moving graph6

Error Handling1

## Introduction

This system is a simulation designed to make learning the concepts of various types of collisions and energy conservation in physics much more interactive. Through the application of an easy to use interface and a powerful set of tools, students should get a firm grasp of the concepts for which this system was created.

This application provides an area which displays the collisions in progress, and a graph which shows the energy changes of each particle. Both of these systems can be controlled using different buttons and sliders.

## System Requirements

The system must be hosted on a site running the most recent version of Apache.

Minimum requirements:

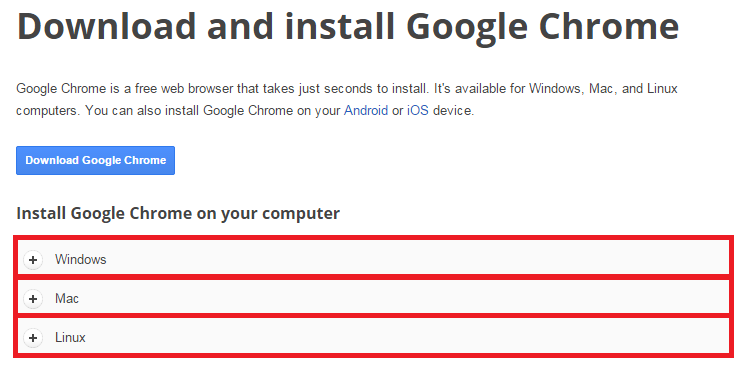
* 1.3 GHz Processor
* 512 MB RAM
* 350 MB Disk space
* Internet connection
* Google Chrome

Recommended requirements:

* 2.0 GHz Processor
* 1024 MB RAM
* 350 MB Disk space
* Internet connection
* Google Chrome

## Installation

To install this system, all that has to be done is to install Google Chrome. Follow this guide: <https://support.google.com/chrome/answer/95346?hl=en>

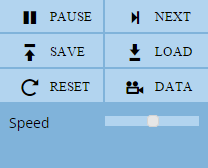


Select one of the red boxes which matches your system. A guide will come up on how to install Google Chrome.

## Using the system

1. **Controlling the simulation**

All of the controls for the simulation are located in this panel:



* 1. Run/pause/step forward the simulation

To pause the simulation, click PAUSE.

To resume the simulation, click RUN (the pause button will change to RUN).

TO step forward the simulation by one update, click NEXT.

* 1. Saving/loading state

To save the simulation’s current state, click SAVE.  
To load the last state of the simulation, click LOAD.

* 1. Resetting the simulation

To reset the simulation back to default, click RESET.

* 1. Getting simulation data, e.g., the directions of the particles

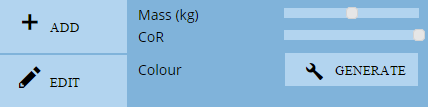
Click the DATA button to get information about the current simulation.

* 1. Simulation speed

Slide the Speed slider to change the speed of the simulation.

1. **Managing particles**

All of the controls for adding/removing particles are located in this panel:



* 1. Adding particles

Click ADD or press “a”. An overlay will appear in which you place a particle either in an alignment system (default) or in a free-place system (hold ctrl). To change the radius of a particle, scroll up to make bigger or down to make smaller. On the first click, the particle will be placed. To unplace the current particle, right click. After the first click, you can give the particle a velocity by moving the mouse a certain distance from the particle and clicking. This distance is the distance which a particle will move in one second.

The properties of the particle can be changed (see “Setting particle values” below).

* 1. Deleting particles

Click EDIT or press “e”. To delete a particle, right click the particle.

* 1. Copying particles

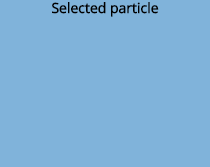
Click EDIT or press “e”. To copy a particle, hold SHIFT and click on the particle. This will copy it and the copy can be placed (see “Adding particles” above).

* 1. Setting particle values

A particle’s properties can be changed before being added and after. Before added, scroll up or down to change the radius, and adjust the sliders to change the mass and coefficient of restitution. To change the particle’s colour click GENERATE.

To change a particle’s properties after it has been added, select it by clicking on the particle. Then adjust the sliders.

* 1. Getting particle information

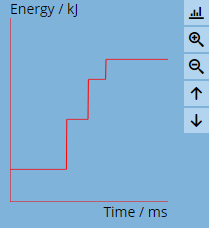
To get particle information, click on the particle. Information will appear in this panel: 

* 1. What is “CoR”

CoR stands for “Coefficient of Restitution”. This is a ratio of the amount of energy after the collision and before the collision (=Eafter/Ebefore).

1. **Graph**

All of the controls for the graph are located in this panel:

****

* 1. Calibration

To calibrate the graph (set the y offset to zero), press .

* 1. Zooming  
     To zoom the graph in, press .

To zoom the graph out, press .

* 1. Moving the graph

To move the graph up (y offset increases), press .

To move the graph down (y offset decreases), press .

## Error Handling

**Number of particles exceeds the maximum value set**

This occurs when the number of particles within the system is at its maximum. A maximum of 100 particles can be added to the system. If you get this error, you will have to delete some particles. See “Using the system” above on how to manage particles.

# Evaluation

The aim of this project was to make learning about collisions and energy conservation easier for students. This involved creating a virtual simulation to model these concepts realistically.

## Project Performance against Project Particleives

1. **The system will make learning about elastic collisions and the laws of conservation of energy easier to understand for students.**

**This will be analysed in the next section.**

1. **The system will allow the visual simulation of collisions of up to 100 particles in two dimensions.**

Collisions currently work in two dimensions to a high degree of accuracy. They still follow the same rules even when there are 100 particles.

1. **The system will correctly follow the physics principals of elastic and inelastic collisions.**

The particles keep the same amount of energy before and after an elastic collision (when the CoR = 1.0). The particles lose energy in an inelastic collision (in an inelastic collision).

1. **The variable properties of particles in the simulation, such as mass, will be able to be modified by the user.**

The properties velocity, mass, coefficient of restitution and radius, can be changed before a particle is added. The properties of the particles can also be modified once they’ve been added by clicking on the EDIT button.

1. **Particles can be added and removed from the simulation. A maximum of 100 particles can be added.**

Particles can be added with properties set by the user. Once the ADD button is clicked, an overlay appears while blurring the background out. It allows the user to place particles and change their radius by scrolling, as well as giving particles a velocity (direction and speed) in an intuitive way (by clicking and display a red line showing the velocity). Particles can be placed using an alignment based (snap-to) system so that particles can be placed in a perfectly straight line, and also a free-place system so that particles can be placed per pixel. Up to 100 particles can be added.

Particles can be deleted by clicking on EDIT. The same overlay appears and an particle can be deleted by right clicking on it.

1. **Start/pause/resume buttons will be added to control the simulation.**

The simulation can be paused and resumed using one button. It was not necessary to add a start, pause and resume button because pause and resume functionality could be achieved using only one button.

1. **Step forward will be added to manually “update” the simulation. Step backward cannot be added as it would require too much data to be stored of previous updates.**

A step forward button is implemented to move the simulation forward by one update.

1. **Tracing particles functionality will be added to help the user find out how the particles in the simulation are behaving.**

The trace of a particle’s path, up to a maximum of 50 saved positions, can be shown by selecting the particle. Once unselected or another particle is selected, its trace path is reset.

1. **The system will display graphs showing the energy of all particles as the simulation progresses.**

There is an energy graph implemented that updates as the simulation updates. The graph displays the combined energy changes of the system. It can be moved up and down (y offset changed), calibrated (the y offset reset to 0), and zoomed in and out (scaleX and scaleY increased and decreased). The speed at which the graph redraws is directly proportional to the refresh rate. On the x axis is time in milliseconds, on the y axis is energy in kilojoules.

1. **Simulation data will be shown. This includes:**

* **Displaying the direction of particles before and after collisions.**
* **The energy stored in a particle.**
* **The properties of the particle (velocity, mass, coefficient of restitution)**

The DATA button makes particles show their velocity (direction and speed). The energy stored in an individual particle can be shown by selecting the particle (clicking it). This also shows the other properties of the particle (velocity, mass, coefficient of restitution, radius, etc).

1. **Simulation saving and loading will be added to restore a previous state of the simulation, i.e, just before two particles are about to collide, so that the exact collision can be repeated.**

The particles of the simulation can be saved by pressing the SAVE button and can be loaded again by press LOAD button. Only one set of particles can be saved. One save can be loaded multiple times, however.

## Client/ User Feedback

## Project Extensions

Given more time here are a few other things I would like to add to the system:

* I would like to make it possible to save a whole configuration of the system so that it could be reloaded at a later date. This would mean that teachers could save specific simulations and get their students to load them for homework.
* I would like to add is an individual particle energy graph. This would add a way of seeing directly how a particle changes during collisions.
* I would like to add a proper way of selecting colours, perhaps via a colour wheel. This would also more user customisation to the simulation
* Finally I would like to add multiple users using one simulation via a server controlling the simulation (managing particles and sending them to users). This would add an interesting dynamic as classrooms could incorporate multiple students learning together.

# Appendices

## Interview Transcripts

**What is your name and what is your job?**

My name is Alison Frost and I’m a physics teacher at Varndean Sixth-Form College in Brighton.

**What is the current system for teaching the concepts of collision between particles and energy conservation, and what would be a typical question that would have to be solved?**

I usually use a practical experimental involving an air-track, which is basically a track with holes along it that it blows a light current of air through. This is to reduce friction. A car is then attached to the track and sent along the track with a block with black lines on it so that either velocity or deceleration can be recorded. The track has a sensor on it that is attached to the computer so it can calculate the speed of the cart as it moves through the sensor.

After that I write calculations and equations on the board, such as m1v1 = m2v2 [conservation of momentum] and drawing diagrams of the stages of collision of two particles (before collision, collision, after collision). I then explain how momentum is conserved in both types of collision [elastic and inelastic] but that energy is not conserved in inelastic collisions. My students then ask questions and start taking notes and I give them questions to do either for that lesson or the next lesson.

A typical question would be something like: calculate the initial velocity of an particle before collision given that you know the resultant velocities and masses of both particles.

**What is the problem with the current system?**

The main problems we have is that the air track is difficult to set up as it’s quite delicate, and we often have trouble fitting it into our teaching schedule as it can take quite some time to get working.

**How would an interactive simulation that explains visually these concepts be used in the physics course?**

I want it to be used in addition to my current methods of teaching. It’s a good way to supplement the teaching of these concepts and make them seem more real to the student. For this reason I would use it in class and also set it as homework. I want the students to be able to test out different kinds of collision like particles with larges masses and small masses. I don’t want it to seem too difficult for the students either.

**What kinds of tools would you expect to exist inside a collision and energy conservation simulation?**

I want a way of showing the collision, either elastic or inelastic, between two or more particles with different masses and a graph showing the energy changes that take place during it. I also want there to be a graph displaying the kinetic energy changes of the entire system, a way of showing an individual particle’s properties, like energy and direction, and perhaps a way of slowing down and speeding up the simulation.

**What information about the particles in the simulation (e.g. kinetic energy of a particle) is essential for the user to understand these concepts?**

The main information that has to be displayed is the velocities (magnitude and direction) and mass of each particle.

## Summary of Questionnaires

## Original documents of system

## Program listing

**ecollision/settings.js**

var eCollisionSettings = {

simulationWidth: 1000,

simulationHeight: 1000,

simulationCanvas: "simulation-canvas",

graphCanvas: "graph-canvas",

overlayCanvas: "overlay-canvas",

refreshRate: 60,

updateRate: 60,

showVelocities: false,

enableInterpolation: true,

maxTraceLength: 30,

graphScaleX: 1/50,

graphScaleY: 5,

graphZoomFactor: 1.25,

graphMinZoomIndex: 5,

graphMaxZoomIndex: 5,

speedConst: 1.0,

maxParticles: 100,

minRadius: 5,

maxRadius: 30,

errorTime: 5000

}

**ecollision/ecollision.js**

function ECollision(settings) {

this.settings = settings;

this.paused = false;

this.engine = new SimulationEngine(settings.simulationWidth, settings.simulationHeight, this.settings);

this.simulationUI = new Simulation(settings.simulationCanvas, this.engine, this.settings);

this.graphUI = new Graph(settings.graphCanvas, this.engine, 1/50, 5, this.settings);

this.overlayUI = new Overlay(settings.overlayCanvas, this.simulationUI, this.settings);

this.fps = 0;

var widgets = [this.simulationUI, this.graphUI, this.overlayUI];

var fpsCount = 0;

var fps = 0;

var fpsTime = 0;

var newTime = timeStamp = curTime = 0;

var updateRate = settings.updateRate;

var updateTime = 1000.0 / updateRate;

var refreshTime = 1000/settings.refreshRate;

var interpolation = 0.0;

var thread = -1;

var ecol = this; *//so that i can refer to this particle inside nested functions - javascript problem solved*

this.start = function() {

for (var i = 0; i < widgets.length; i++) {

widgets[i].init();

}

thread = setInterval(this.tick, 1000.0 / settings.refreshRate);

}

this.restart = function() {

for (var i = 0; i < widgets.length; i++) {

widgets[i].restart();

}

}

this.resume = function() {

this.paused = false;

for (var i = 0; i < widgets.length; i++) {

widgets[i].resume();

}

}

this.pause = function() {

this.paused = true;

for (var i = 0; i < widgets.length; i++) {

widgets[i].pause();

}

}

this.stop = function() {

if (thread != -1) {

clearInterval(thread);

thread = -1;

}

}

this.getUpdateRate = function() {

return updateRate;

}

this.getUpdateTime = function() {

return updateTime;

}

this.setUpdateRate = function(rate) {

updateRate = rate;

updateTime = 1000.0 / updateRate;

}

this.setSpeedConst = function(speedConst) {

this.engine.speedConst = speedConst;

}

this.onTick = function() {};

this.update = function() {

curTime += refreshTime;

if (newTime + updateTime < curTime) {

timeStamp = curTime;

if (settings.enableInterpolation) {

for (var i = 0; i < this.engine.numOfParticles(); i++) {

this.engine.getParticle(i).capture();

}

}

while (newTime + updateTime < curTime) {

this.engine.update();

newTime += updateTime;

}

}

interpolation = Math.min(1.0, (curTime - timeStamp) / updateTime);

}

this.tick = function() {

if (!ecol.paused) {

ecol.update();

}

var fpsCurTime = new Date().getTime();

fpsCount++;

if (fpsCurTime - fpsTime >= 1000) {

ecol.fps = fpsCount;

fpsCount = 0;

fpsTime = fpsCurTime;

}

for (var i = 0; i < widgets.length; i++) {

widgets[i].draw(interpolation);

}

ecol.onTick();

}

}

**ecollision/engine/simulation-engine.js**

function SimulationEngine(width, height, settings) {

this.width = width;

this.height = height;

var particles = [];

this.setBounds = function(width, height) {

this.width = width;

this.height = height;

}

this.reset = function() {

particles = [];

}

this.addParticle = function(particle) {

if (particles.length < settings.maxParticles) {

particle.index = particles.length;

particles.push(particle);

} else {

throw "ERROR: Number of particles exceeds the maximum value set.";

}

}

this.removeParticle = function(index) {

particles.splice(index, 1);

}

this.getParticle = function(index) {

return particles[index];

}

this.numOfParticles = function() {

return particles.length;

}

*//Collision class that stores which particles collided and a time constant of how much to seperate them by.*

function Collision() {

this.time = 0.0;

this.particle = null;

this.particle2 = null;

}

*//This detects if there is an edge collision.*

this.edgeCollision = function (particle, rebound) {

var cOR = particle.cOR;

*//If the particle is outside the width set, it will be placed back inside*

if (particle.x + particle.radius >= this.width) {

if (rebound) {

*//The particle may lose energy if it's coefficient of restitution is less than 1.*

particle.xVel \*= -cOR;

particle.yVel \*= cOR;

} else {

particle.x = this.width - particle.radius;

}

} else if (particle.x - particle.radius <= 0) {

if (rebound) {

particle.xVel \*= -cOR;

particle.yVel \*= cOR;

} else {

particle.x = particle.radius;

}

}

*//If the particle is outside the height set, it will be placed back inside*

if (particle.y + particle.radius >= this.height) {

if (rebound) {

particle.xVel \*= cOR;

particle.yVel \*= -cOR;

} else {

particle.y = this.height - particle.radius;

}

} else if (particle.y - particle.radius <= 0) {

if (rebound) {

particle.xVel \*= cOR;

particle.yVel \*= -cOR;

} else {

particle.y = particle.radius;

}

}

}

*//This functions checks for a collision and returns true or false if yes. It also calculates the required amount of time to seperate the particles.*

this.collide = function(particle, particle2, collision) {

*//Take the distances between the particles on the x and y axes*

var dX = particle2.x - particle.x;

var dY = particle2.y - particle.y;

*//Calculate the square of the distance*

var sqr = (dX \* dX) + (dY \* dY);

var r = particle2.radius + particle.radius;

*//Could sqrt to get the distance, but there's no need because the otherside would also have to be sqrted*

if (sqr < r \* r) {

*//Now to get the time constant between the last update and this update at which the particles would have collided perfectly*

*//Put into pvectors as we need to get the dot products*

var pDiff = new PVector(particle.x - particle2.x, particle.y - particle2.y);

var vDiff = new PVector(particle.xVel - particle2.xVel, particle.yVel - particle2.yVel);

*//The following can be derived thus:*

*// At the time of a perfect collision:*

*// let dx = particle2\_currentX - particle1\_currentX*

*// let dy = particle2\_currentY - particle1\_currentY*

*//*

*// let diffVelocityX = particle2\_velocityX-particle1\_velocityX*

*// let diffVelocityY = particle2\_velocityY-particle1\_velocityY*

*//*

*// let particle1\_xFinal = particle1\_currentX - (particle1\_velocityX \* time)*

*// let particle1\_yFinal = particle1\_currentY - (particle1\_velocityY \* time)*

*//*

*// let particle2\_xFinal = particle2\_currentX - (particle2\_velocityX \* time)*

*// let particle2\_yFinal = particle2\_currentY - (particle2\_velocityY \* time)*

*//*

*// We need to solve for time:*

*// let diffX = particle2\_xFinal-particle1\_xFinal*

*// let diffY = particle2\_yFinal-particle1\_yFinal*

*//*

*// Rearranging and subbing-in this gives:*

*// diffX = particle2\_currentX - (particle2\_velocityX \* time) - particle1\_currentX - (particle1\_velocityX \* time)*

*// = (particle2\_currentX - particle1\_currentX) - time\*(particle2\_velocityX-particle1\_velocityX)*

*// = dx - time\*diffVelocityX*

*//*

*// diffY = particle2\_currentY - (particle2\_velocityY \* time) - particle1\_currentY - (particle1\_velocityY \* time)*

*// = (particle2\_currentY - particle1\_currentY) - time\*(particle2\_velocityY-particle1\_velocityY)*

*// = dy - time\*diffVelocityY*

*//*

*// Now it is just like a collision check, as above, except this time we can solve for time:*

*// let sqr = sqr(diffX) + sqr(diffY)*

*// = sqr(dx - time\*diffVelocityX) + sqr(dy - time\*diffVelocityY)*

*//*

*// Now to expand the brackets:*

*// sqr = sqr(dx) - 2\*time\*diffVelocityX\*dx + sqr(time)\*sqr(diffVelocityX) + sqr(dy) - 2\*time\*diffVelocityY\*dy + sqr(time)\*sqr(diffVelocityY)*

*//*

*// We're trying to find time, and or a perfect collision, sqr must equal the sum of the radii squared*

*// So our quadratic equation is:*

*// let radiiSqred = sqr(particle1\_radius+particle2\_radius)*

*//*

*// sqr = a\*sqr(time) + b\*time + c-radiiSqred = 0*

*// a = sqr(diffVelocityX)+sqr(diffVelocityY) (NOTE: dotProduct as below)*

*// b = -2\*(dx\*diffVelocityX + diffVelocityY\*dy) (NOTE: dotProduct as below)*

*// c = sqr(dx)+sqr(dy) - radiiSqred*

*//*

*// We then use the quadratic formula (-b +- sqrt(b\*b - 4\*a\*c))/(2\*a) to calculate time*

var a = vDiff.dotProduct(vDiff);

var b = -2 \* vDiff.dotProduct(pDiff);

var c = (pDiff.dotProduct(pDiff)) - (r \* r);

var discr = (b \* b) - (4 \* a \* c);

var t = 0.0;

var t2 = 0.0;

if (discr >= 0) {

t = (-b - Math.sqrt(discr)) / (2 \* a);

t2 = (-b + Math.sqrt(discr)) / (2 \* a);

}

if (t > 0.0 && t <= 1.0)

collision.time = t;

else if (t2 > 0.0 && t2 <= 1.0)

collision.time = t2;

else

collision.time = 1.0;

return true;

}

return false;

}

*//This function splits particle1's velocity into parallel and perpendicular components.*

function splitVelocity(particle1, particle2) {

*//The overall process of visualising how this works is:*

*// 1. Imagine the collision happening such that particle1's velocity is rotated so that it is in one dimension, or the x-axis*

*// 2. Then calculate its parallel and perpendicular components when it collides.*

*// 3. Finally rotate these components back by the same amount.*

*//*

*//Store the current velocity in a vector structure*

var velocity = new PVector(particle1.xVel, particle1.yVel);

*//Default angle*

var a = Math.PI / 2;

*//Calculate the angle of the velocity*

if (particle1.xVel !== 0) {

a = Math.atan(particle1.yVel / particle1.xVel);

}

*//Calculate the magnitude as if it were on the x-axis only. This was originally part of my rotate function.*

*//See math/pector.js for similarities*

var magnitude = (particle1.xVel \* Math.cos(-a) - particle1.yVel \* Math.sin(-a)) \* particle1.cOR;

var dx = particle1.x - particle2.x;

var dy = particle1.y - particle2.y;

*//Calculate the position angle*

var ang = 0;

if (dx !== 0) {

ang = Math.atan(dy / dx);

} else {

ang = Math.atan(dy / (dx - 0.00001));

}

*//This is a simplification of multiple cosines and sines using trig identities. It is essentially doing stages 2 and 3 as stated above.*

velocity.x = magnitude \* (Math.cos(ang - a));

velocity.y = magnitude \* (Math.sin(ang - a));

return velocity;

}

*//This function actually handles the collision between two particles.*

this.handleCollision = function (collision) {

var particle = collision.particle;

var particle2 = collision.particle2;

*//Split the velocities into parallel and perpendicular components. See "splitVelocity" above.*

var thisVel = splitVelocity(particle, particle2);

var particleVel = splitVelocity(particle2, particle);

*//Finally do some real physics. This calculates the new velocities of the parallel components as if they were one-dimensional.*

var newV = ((thisVel.x \* (particle.mass - particle2.mass)) + (2 \* particle2.mass \* particleVel.x)) / (particle.mass + particle2.mass);

var newV2 = ((particleVel.x \* (particle2.mass - particle.mass)) + (2 \* particle.mass \* thisVel.x)) / (particle.mass + particle2.mass);

*//Calculate the angle between the particles*

var ang = Math.atan((particle.y - particle2.y) / (particle.x - particle2.x));

var cosA = Math.cos(ang);

var sinA = Math.sin(ang);

*//Then these new velocityies are split further so they fit the Cartesian coordinate system. They are then added to the remaining velocity from the perpendicular components*

var x1 = (newV \* cosA) + (thisVel.y \* sinA);

var y1 = (newV \* sinA) - (thisVel.y \* cosA);

var x2 = (newV2 \* cosA) + (particleVel.y \* sinA);

var y2 = (newV2 \* sinA) - (particleVel.y \* cosA);

*//Seperate the particles. See "seperateParticles" below.*

this.seperateParticles(collision, particle, particle2);

*//Finally give each particle their new velocities.*

particle.xVel = x1;

particle.yVel = y1;

particle2.xVel = x2;

particle2.yVel = y2;

}

*//This function seperates the particles after collision.*

this.seperateParticles = function(collision, particle, particle2) {

*//Add a small extra amount of time so that the particles can never get stuck on each other*

var t = collision.time + (0.001 \* collision.time);

if (t < 1.0) {

*//Pull both particles back by the perfect collision time. See "collide" function*

particle.x -= particle.xVel \* settings.speedConst \* t;

particle.y -= particle.yVel \* settings.speedConst \* t;

particle2.x -= particle2.xVel \* settings.speedConst \* t;

particle2.y -= particle2.yVel \* settings.speedConst \* t;

} else {

*//Failsafe method of seperating particles*

*//First calculate the overlap*

var dX = particle2.x - particle.x;

var dY = particle2.y - particle.y;

var sqr = (dX \* dX) + (dY \* dY);

var overlap = particle2.radius - Math.abs(Math.sqrt(sqr) - particle.radius) + 0.1;

var vel1 = new PVector(particle.xVel, particle.yVel).getMagnitudeNS()+0.0001;

var vel2 = new PVector(particle2.xVel, particle2.yVel).getMagnitudeNS()+0.0001;

*//Total velocity*

var vT = vel1 + vel2;

*//Work out the first propotion for movement*

var i = vel1 / vT;

ang = Math.atan2(particle.y - particle2.y, particle.x - particle2.x);

*//Move particle*

particle.x += overlap \* Math.cos(ang) \* i;

particle.y += overlap \* Math.sin(ang) \* i;

*//Work out other proportion for movement*

i = 1 - i;

particle2.x -= overlap \* Math.cos(ang) \* i;

particle2.y -= overlap \* Math.sin(ang) \* i;

}

}

*//This function causes the particles to update and react to each other. It is the heart of the system.*

this.update = function () {

*//Loop through the particles, make sure they are not overlapping with the edges, then update their position.*

for (var i = 0; i < particles.length; i++) {

var particle = particles[i];

this.edgeCollision(particle, true);

particle.update();

}

var colParticles = [];

*//Loop through the particles, check for collisions once between pairs of particles.*

*//If colliding, add them to a collision array*

for (var i = 0; i < particles.length; i++) {

var particle = particles[i];

for (var i2 = i + 1; i2 < particles.length; i2++) {

var particle2 = particles[i2];

var collision = new Collision();

if (this.collide(particle, particle2, collision)) {

collision.particle = particle;

collision.particle2 = particle2;

colParticles.push(collision);

}

}

}

*//Loop through the collision array and sort out which one happened first*

colParticles.sort(function (a, b) {

return a.time < b.time;

});

*//Handle the collisions stored in the collision array. See "handleCollision" above*

for (var i = 0; i < colParticles.length; i++) {

var collision = colParticles[i];

this.handleCollision(collision);

}

*//Finally check for an edge collision again but do not rebound the particle*

for (var i = 0; i < particles.length; i++) {

var particle = particles[i];

this.edgeCollision(particle, false);

}

}

}

**ecollision/ui/widget.js**

function Widget(canvasName) {

this.hidden = false;

this.canvasName = canvasName;

this.canvas = $("#"+canvasName);

this.width = this.canvas.width();

this.height = this.canvas.height();

this.stage = new createjs.Stage(canvasName);

this.canvas.attr("width", this.width);

this.canvas.attr("height", this.height);

this.init = function() {}

this.addEventListener = function(event, handler) {

this.stage.addEventListener(event, handler);

}

this.draw = function (interpolation) {}

this.restart = function () {}

this.stop = function() {}

this.resume = function() {

this.paused = false;

}

this.pause = function() {

this.paused = true;

}

this.resize = function(newWidth, newHeight) {}

this.show = function() {

this.hidden = false;

this.canvas.fadeIn(200);

}

this.hide = function() {

this.hidden = true;

this.canvas.fadeOut(200);

}

}

**ecollision/ui/simulation.js**

function Simulation(canvasName, engine, settings) {

Widget.call(this, canvasName);

this.engine = engine;

this.engine.width = this.width;

this.engine.height = this.height;

var selected = -1;

this.resize = function(newWidth, newHeight) {

this.engine.setBounds(newWidth, newHeight);

}

this.addParticle = function(x, y, mass, radius, style) {

var particle = new Particle(x, y, radius, style, settings);

particle.mass = mass;

var engine = this.engine;

particle.addEventHandler("click", function (ev) {

var p = engine.getParticle(selected);

if (selected != -1) {

p.deselect();

}

for (var i = 0; i < engine.numOfParticles(); i++) {

if (engine.getParticle(i).displayObj == ev.target) {

if (i != selected) {

selected = i;

engine.getParticle(i).selected = true;

} else {

selected = -1;

}

break;

}

}

});

this.stage.addChild(particle.displayObj);

this.engine.addParticle(particle);

return particle;

}

this.removeParticle = function(index) {

this.stage.removeChild(this.engine.getParticle(index).displayObj);

this.engine.removeParticle(index);

}

this.loadParticles = function(toBeLoaded) {

this.restart();

for (var i = 0; i < toBeLoaded.length; i++) {

var obj = toBeLoaded[i];

var particle = this.addParticle(obj.x, obj.y, obj.mass, obj.radius, obj.style);

particle.xVel = obj.xVel;

particle.yVel = obj.yVel;

particle.cOR = obj.cOR;

}

}

this.saveParticles = function(saved) {

for (var i = 0; i < this.engine.numOfParticles(); i++) {

var obj = this.engine.getParticle(i);

saved.push(obj.copy());

}

}

this.removeSelected = function() {

if (selected != -1) {

this.removeParticle(selected);

selected = -1;

}

}

this.getSelected = function() {

var sel = null;

if (selected != -1) {

sel = this.engine.getParticle(selected);

}

return sel;

}

this.getSelectedID = function() {

return selected;

}

this.restart = function () {

this.stage.removeAllChildren();

selected = -1;

this.engine.reset();

}

this.draw = function (interpolation) {

for (var i = 0; i < this.engine.numOfParticles(); i++) {

var obj = this.engine.getParticle(i);

var newX = obj.x;

var newY = obj.y;

if (settings.enableInterpolation) {

var diffX = obj.x - obj.lastX;

var diffY = obj.y - obj.lastY;

newX = obj.lastX + (interpolation \* diffX);

newY = obj.lastY + (interpolation \* diffY);

}

obj.draw(newX, newY);

}

this.stage.update();

}

}

Simulation.prototype = new Widget();

**ecollision/ui/graph.js**

function Graph(canvasName, engine, scaleX, scaleY, settings) {

Widget.call(this, canvasName);

this.scaleX = scaleX;

this.scaleY = scaleY;

this.x = 0;

this.y = 0;

this.engine = engine;

var graph = new createjs.Shape();

var offsetX = 0.0;

var offsetY = 0.0;

var userY = 0;

var data = [];

var start = 0;

var maxLen = 150;

var updated = false;

var currX = 0;

var currY = 0;

var zoomIndex = 0;

this.init = function() {

var xAxis = new createjs.Shape();

var yAxis = new createjs.Shape();

xAxis.graphics.beginStroke("red").moveTo(this.x, this.height).lineTo(this.width, this.height);

yAxis.graphics.beginStroke("red").moveTo(this.x, this.y).lineTo(this.x, this.height);

this.stage.addChild(xAxis);

this.stage.addChild(yAxis);

this.stage.addChild(graph);

this.updateData();

}

this.draw = function (interpolation) {

if (this.engine != null) {

var g = graph.graphics;

g.clear();

var length = data.length-1;

var total = 0;

for (var j = 0; j < length-1; j++) {

if (updated) {

updated = false;

return;

}

total += data[j].y;

*//calculate offsetted index for point at index j*

var i = (start+j)%length;

var i2 = (start+j+1)%length;

var x2 = (data[i].x\*this.scaleX)-offsetX;

var y2 = (data[i].y\*this.scaleY)+offsetY+userY;

*//if second x value is larger than width, move graph along*

if (x2 > this.width) {

offsetX += x2-this.width;

}

var x1 = (data[i].x\*this.scaleX)-offsetX;

var y1 = (data[i].y\*this.scaleY)+offsetY+userY;

var x3 = (data[i2].x\*this.scaleX)-offsetX;

var y3 = (data[i2].y\*this.scaleY)+offsetY+userY;

g.beginStroke("red").moveTo(this.x+x1, this.y+this.height-y1).lineTo(this.x+x3, this.y+this.height-y3);

}

if (!this.paused) {

currX += 1000/settings.updateRate;

currY = this.getEnergy();

this.addData(currX, currY);

}

var dataY = total/data.length;

var targetY = this.height/2;

offsetY = targetY-(dataY\*this.scaleY);

this.stage.update();

}

}

this.restart = function() {

data = [];

start = 0;

currX = currY = 0;

offsetX = offsetY = 0;

updated = true;

}

this.calibrate = function() {

userY = 0;

}

this.zoomIn = function() {

if (zoomIndex < settings.graphMaxZoomIndex) {

this.scaleX \*= settings.graphZoomFactor;

this.scaleY \*= settings.graphZoomFactor;

offsetX \*= this.scaleX;

offsetY \*= this.scaleY;

this.updateData();

zoomIndex++;

} else throw("ERROR: Maximum zoom reached");

}

this.zoomOut = function() {

if (zoomIndex > -settings.graphMinZoomIndex) {

this.scaleX /= settings.graphZoomFactor;

this.scaleY /= settings.graphZoomFactor;

offsetX \*= this.scaleX;

offsetY \*= this.scaleY;

this.updateData();

zoomIndex--;

} else throw("ERROR: Minimum zoom reached");

}

this.moveUp = function() {

userY -= 5;

}

this.moveDown = function() {

userY += 5;

}

this.addData = function(x, y) {

if (data.length > maxLen) {

var s = start;

start = (start + 1)%maxLen;

data[s] = new Point2D(x, y);

} else {

data.push(new Point2D(x, y));

}

}

this.updateData = function() {

var data2 = [];

maxLen = Math.round(this.width/((1000/settings.updateRate)\*this.scaleX))+5;

var aLen = data.length-1;

var diff = 0;

if (aLen > maxLen) {

diff = aLen-maxLen;

}

for (var j = diff; j < aLen; j++) {

var i = (start+j)%aLen;

data2.push(data[i]);

}

updated = true;

start = 0;

data = data2;

}

this.getEnergy = function() {

var energy = 0.0;

for (var i = 0; i < this.engine.numOfParticles(); i++) {

energy += this.engine.getParticle(i).getEnergy();

}

return Math.round(energy/1000);

}

}

Graph.prototype = new Widget();

**ecollision/ui/overlay.js**

function Overlay(canvasName, simulation, settings) {

Widget.call(this, canvasName);

this.hide();

var INDEX\_PLACE = 0;

var INDEX\_VELOCITY = 1;

var INDEX\_MODIFY = 2;

var MODE\_ADD = 0;

var MODE\_EDIT = 1;

var index = 0;

var mode = -1;

var mouseX = crossX = this.width/2;

var mouseY = crossY = this.height/2;

var velocityLine = new createjs.Shape();

var infoText = new createjs.Text("", "bold 15px Arial");

var errorText = new createjs.Text("", "bold 15px Arial", "red");

var errorTimer = 0;

var showError = false;

var modeText = new createjs.Text("", "bold 15px Arial");

modeText.x = (this.width/2)-40;

modeText.y = 10;

var freePlace = false;

var copyPlace = false;

var lastX = 0;

var lastY = 0;

var tempParticle = null;

function gcd(a, b) {

if ( ! b) {

return a;

}

return gcd(b, a % b);

}

var interval = gcd(this.width, this.height);

var overlay = this; *//so that i can refer to this particle inside nested functions - javascript problem solved*

function handleMouseWheel(ev) {

var d = ev.deltaY;

if (d < 0) {

if (tempParticle.radius > settings.minRadius) {

tempParticle.radius -= 1;

}

} else {

if (tempParticle.radius < settings.maxRadius) {

tempParticle.radius += 1;

}

}

}

this.canvas.mousewheel(handleMouseWheel);

this.resize = function(width, height) {

interval = gcd(this.width, this.height);

}

$(document).keydown(function(event) {

freePlace = event.ctrlKey;

copyPlace = event.shiftKey;

});

$(document).keyup(function(event) {

freePlace = false;

copyPlace = false;

});

this.canvas.bind('contextmenu', function(e){

return false;

});

this.init = function() {

this.stage.removeAllChildren();

mouseX = crossX = this.width/2;

mouseY = crossY = this.height/2;

this.stage.addChild(modeText);

}

function handleMouseMove(ev) {

mouseX = crossX = ev.stageX;

mouseY = crossY = ev.stageY;

if (!freePlace) {

var gridX = Math.round(mouseX/interval);

var gridY = Math.round(mouseY/interval);

crossX = gridX\*interval;

crossY = gridY\*interval;

}

switch (index) {

case INDEX\_PLACE:

velocityLine.x = crossX;

velocityLine.y = crossY;

if (tempParticle != null) {

tempParticle.x = crossX;

tempParticle.y = crossY;

}

infoText.x = crossX-50;

infoText.y = crossY-50;

break;

case INDEX\_VELOCITY:

var g = velocityLine.graphics;

var dx = crossX-velocityLine.x;

var dy = crossY-velocityLine.y;

infoText.x = velocityLine.x + (dx/2);

infoText.y = velocityLine.y + (dy/2);

infoText.text = Math.round(Math.sqrt(dx\*dx + dy\*dy)) + " px/s";

tempParticle.xVel = dx/settings.updateRate;

tempParticle.yVel = dy/settings.updateRate;

g.clear().beginStroke("red").setStrokeStyle(3).moveTo(0, 0).lineTo(dx, dy);

break;

case INDEX\_MODIFY:

for (var i = 0; i < simulation.engine.numOfParticles(); i++) {

var p = simulation.engine.getParticle(i);

var dx = p.x-mouseX;

var dy = p.y-mouseY;

if (dx\*dx + dy\*dy <= p.radius\*p.radius) {

p.select();

tempParticle = p;

} else {

p.deselect();

}

}

break;

}

}

this.stage.addEventListener("stagemousemove", handleMouseMove);

function handleClick(ev) {

if (ev.button == 2 && index != INDEX\_MODIFY) {

switch (index) {

case INDEX\_PLACE:

overlay.end();

reset();

break;

case INDEX\_VELOCITY:

reset();

break;

}

} else {

switch(index) {

case INDEX\_PLACE:

velocityLine.graphics.clear();

overlay.stage.addChild(velocityLine);

overlay.stage.addChild(infoText);

index = INDEX\_VELOCITY;

break;

case INDEX\_VELOCITY:

try {

var p = simulation.addParticle(tempParticle.x, tempParticle.y, tempParticle.mass, tempParticle.radius, tempParticle.style);

p.xVel = tempParticle.xVel;

p.yVel = tempParticle.yVel;

p.cOR = tempParticle.cOR;

overlay.stage.removeChild(velocityLine);

overlay.stage.removeChild(infoText);

tempParticle.xVel = tempParticle.yVel = 0;

if (mode == MODE\_EDIT && !copyPlace) {

index = INDEX\_MODIFY;

overlay.stage.removeChild(tempParticle.displayObj);

} else

index = INDEX\_PLACE;

} catch (e) {

errorText.text = e;

errorText.x = overlay.width-(errorText.getMeasuredWidth())-10;

errorText.y = overlay.height/2;

overlay.stage.addChild(errorText);

errorTimer = settings.errorTime;

showError = true;

}

break;

case INDEX\_MODIFY:

tempParticle.displayObj.dispatchEvent("click");

if (ev.button == 2) {

simulation.removeSelected();

} else {

var selected = tempParticle;

tempParticle = selected.copy();

lastX = selected.x;

lastY = selected.y;

overlay.stage.addChild(tempParticle.displayObj);

if (!copyPlace)

simulation.removeSelected();

index = INDEX\_PLACE;

}

break;

}

}

ev.stopPropagation();

}

this.canvas.mousedown(handleClick);

this.draw = function(interpolation) {

if (!this.hidden) {

if (tempParticle != null) {

tempParticle.draw(tempParticle.x, tempParticle.y);

}

if (showError) {

errorTimer -= 1000/settings.updateRate;

if (errorTimer <= 0) {

showError = false;

this.stage.removeChild(errorText);

}

}

this.stage.update();

}

}

function reset() {

if (mode == MODE\_EDIT) {

var p = simulation.addParticle(lastX, lastY, tempParticle.mass, tempParticle.radius, tempParticle.style);

p.xVel = tempParticle.xVel;

p.yVel = tempParticle.yVel;

overlay.removeChild(tempParticle.displayObj);

tempParticle = null;

index = INDEX\_MODIFY;

} else {

overlay.stage.removeChild(velocityLine);

index = INDEX\_PLACE;

}

}

this.beginAdd = function(mass, cOR, style) {

this.show();

this.init();

tempParticle = new Particle(crossX, crossY, 25, style, settings);

tempParticle.mass = mass;

tempParticle.cOR = cOR;

infoText.x = mouseX;

infoText.y = mouseY;

this.stage.addChild(tempParticle.displayObj);

modeText.text = "Mode: Add";

index = INDEX\_PLACE;

mode = MODE\_ADD;

}

this.beginEdit = function() {

this.show();

this.init();

modeText.text = "Mode: Edit";

index = INDEX\_MODIFY;

mode = MODE\_EDIT;

}

this.end = function() {

this.hide();

this.stage.removeAllChildren();

if (tempParticle != null) {

tempParticle = null;

}

mode = -1;

freePlace = false;

copyPlace = false;

}

this.getCurrentParticle = function() {

return tempParticle;

}

this.getMode = function() {

return mode;

}

}

Overlay.prototype = new Widget();

**ecollision/particles/physics-particle.js**

function PhysicsParticle(x, y, mass) {

this.x = x;

this.y = y;

this.lastX = this.x;

this.lastY = this.y;

this.xVel = 0.0;

this.yVel = 0.0;

this.mass = mass;

this.displayObj = new createjs.Shape();

this.displayObj.x = this.x;

this.displayObj.y = this.y;

*//private function*

this.capture = function () {

this.lastX = this.x;

this.lastY = this.y;

};

this.update = function () {

this.x += this.xVel\*speedConst;

this.y += this.yVel\*speedConst;

};

this.addEventHandler = function (event, handler) {

this.displayObj.addEventListener(event, handler);

}

this.getEnergy = function() {

return 0.5 \* this.mass \* ((this.xVel\*this.xVel) + (this.yVel\*this.yVel));

}

this.draw = function (x, y) {

this.displayObj.x = x;

this.displayObj.y = y;

}

}

**ecollision/particles/particle.js**

function Particle(x, y, radius, style, settings) {

PhysicsParticle.call(this, x, y, 10);

this.radius = radius;

this.style = style;

this.cOR = 1.0;

this.selected = false;

var pastPositions = [];

var curPos = 0;

this.draw = function (x, y) {

this.displayObj.x = x;

this.displayObj.y = y;

var graphics = this.displayObj.graphics;

graphics.clear();

if (this.selected) {

var len = pastPositions.length;

for (var i = 1; i < len; i++) {

var p = pastPositions[(i + curPos) % len];

var px = p.x-x;

var py = p.y-y;

var r\_a = i / len;

var col = "rgba(100, 100, 100, "+r\_a+")";

graphics.beginStroke(col).drawCircle(px, py, this.radius).endStroke();

}

graphics.beginStroke("red").setStrokeStyle(3).drawCircle(0, 0, this.radius).endStroke();

}

graphics.beginFill(this.style).drawCircle(0, 0, this.radius).endFill();

if (this.selected || settings.showVelocities) {

graphics.beginStroke("red").setStrokeStyle(3).moveTo(0, 0).lineTo(this.xVel\*settings.updateRate, this.yVel\*settings.updateRate).endStroke();

}

};

this.select = function() {

this.selected = true;

}

this.deselect = function() {

this.selected = false;

pastPositions = [];

}

this.update = function () {

this.x += this.xVel\*settings.speedConst;

this.y += this.yVel\*settings.speedConst;

var len = pastPositions.length;

if (this.selected) {

curPos++;

curPos %= settings.maxTraceLength;

if (len < settings.maxTraceLength) {

pastPositions.push(new Point2D(this.x, this.y));

} else {

pastPositions[curPos] = new Point2D(this.x, this.y);

}

}

};

this.copy = function() {

var p = new Particle(this.x, this.y, this.radius, this.style, settings);

p.index = this.index;

p.cOR = this.cOR;

p.mass = this.mass;

p.xVel = this.xVel;

p.yVel = this.yVel;

return p;

}

}

Particle.prototype = new PhysicsParticle();

**ecollision/math/p-vector.js**

function PVector(x, y) {

this.x = x;

this.y = y;

this.getMagnitude = function () {

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

}

this.getMagnitudeNS = function () {

return (this.x \* this.x) + (this.y \* this.y);

}

this.dotProduct = function (vec) {

return (this.x \* vec.x) + (this.y \* vec.y);

}

this.getNormal = function () {

return new PVector(-this.y, this.x);

}

this.rotate = function (angle) {

var x2 = this.x;

var y2 = this.y;

this.x = x2 \* Math.cos(angle) - y2 \* Math.sin(angle);

this.y = x2 \* Math.sin(angle) + y2 \* Math.cos(angle);

}

}

**ecollision/math/point-2d.js**

function Point2D(x, y) {

this.x = x;

this.y = y;

}

**script.js**

$.widget("custom.sliderEx", $.ui.slider, {

\_unit:"",

\_amount: null,

\_formatVal: function(val) {

if (val > 0.09 && val < 1) {

val = val.toPrecision(2);

}

return val+" "+this.\_unit;

},

\_slide: function () {

this.\_superApply(arguments);

this.\_amount.text(this.\_formatVal(this.options.value));

var pos = this.handle.position();

var width = this.\_amount.width()/2;

var newLeft = pos.left;

this.\_amount.css("left", newLeft+"px");

},

\_start: function() {

this.\_superApply(arguments);

var left = this.handle.css("left");

this.\_amount.css('visibility','visible').hide().fadeIn("fast").css("left", left);

},

\_stop: function() {

this.\_superApply(arguments);

this.\_amount.fadeOut("fast");

},

\_create: function() {

var min = parseFloat(this.element.attr("min"));

var max = parseFloat(this.element.attr("max"));

this.options.min = min;

this.options.max = max;

this.options.step = parseFloat(this.element.attr("step")) || 1.0;

this.options.value = parseFloat(this.element.attr("value")) || (min+max/2);

var unit = this.element.attr("unit");

this.\_unit = unit || "";

this.\_amount = $('<div class="slider-amount">'+this.\_formatVal(this.options.value)+'</div>');

this.element.append(this.\_amount).mousedown(function(event) {

event.stopPropagation();

});

this.\_super();

}

});

function getRandomColor() {

var letters = '0123456789ABCDEF'.split('');

var color = '#';

for (var i = 0; i < 6; i++) {

color += letters[Math.floor(Math.random() \* 16)];

}

return color;

}

function toDegrees(ang) {

var a = ((ang / Math.PI) \* 180)+90;

if (a < 0)

a += 360;

else if (a > 360) {

a -= 360;

}

return a;

}

function setCol(text, col) {

return ("" + text).fontcolor(col);

}

function setColGreen(text) {

return setCol(text, "green");

}

function dbgBool(bool) {

if (bool)

return setCol(""+bool, "green");

else

return setCol(""+bool, "red");

}

function log(s) {

console.log(s);

}

$("#slider-mass").sliderEx({

slide: function(event, ui) {

var cp = ecollision.overlayUI.getCurrentParticle() || ecollision.simulationUI.getSelected();

if (cp != null)

cp.mass = ui.value;

}

});

$("#slider-cor").sliderEx({

slide: function(event, ui) {

var cp = ecollision.overlayUI.getCurrentParticle() || ecollision.simulationUI.getSelected();

if (cp != null)

cp.cOR = ui.value;

}

});

function openAdd() {

var mode = ecollision.overlayUI.getMode();

if (mode == 0) {

ecollision.overlayUI.end();

} else {

var mass = $("#slider-mass").sliderEx("value");

var cOR = $("#slider-cor").sliderEx("value");

ecollision.overlayUI.beginAdd(mass, cOR, currentColor);

}

}

function openEdit() {

var mode = ecollision.overlayUI.getMode();

if (mode == 1) {

ecollision.overlayUI.end();

} else {

ecollision.overlayUI.beginEdit();

}

}

$(document).keypress(function(ev) {

if (ev.charCode == 97) {

openAdd();

} else if (ev.charCode == 101) {

openEdit();

}

});

$("#add-particle").click(function() {

openAdd();

});

$("#remove-particle").click(function() {

openEdit();

});

var currentColor = getRandomColor();

$("#generate-colour").click(function() {

var cp = ecollision.overlayUI.getCurrentParticle() || ecollision.simulationUI.getSelected();

if (cp != null) {

currentColor = getRandomColor();

cp.style = currentColor;

}

})

$("#calibrate").click(function() {

ecollision.graphUI.calibrate();

});

$("#zoom-in").click(function() {

ecollision.graphUI.zoomIn();

});

$("#zoom-out").click(function() {

ecollision.graphUI.zoomOut();

});

$("#move-up").click(function() {

ecollision.graphUI.moveUp();

});

$("#move-down").click(function() {

ecollision.graphUI.moveDown();

});

$("#btn-sim-data").click(function() {

eCollisionSettings.showVelocities = !eCollisionSettings.showVelocities;

});

$("#btn-run-pause").click(function() {

if (ecollision.paused)

ecollision.resume();

else

ecollision.pause();

changeRunPauseBtn();

});

function changeRunPauseBtn() {

if (!ecollision.paused) {

$("#btn-run-pause").removeClass('icon-playback-play').addClass('icon-pause').text("PAUSE");

} else {

$("#btn-run-pause").removeClass('icon-pause').addClass('icon-playback-play').text("RUN");

}

}

$("#btn-next").click(function() {

ecollision.update();

});

var savedState = [];

$("#btn-save").click(function() {

savedState = [];

ecollision.simulationUI.saveParticles(savedState);

});

$("#btn-load").click(function() {

ecollision.simulationUI.loadParticles(savedState);

});

$("#btn-reset").click(function() {

ecollision.restart();

});

$("#sim-speed-slider").sliderEx({

slide: function(event, ui) {

eCollisionSettings.speedConst = parseFloat(ui.value);

}

});

var ecollision = new ECollision(eCollisionSettings);

var fpsDiv = $("#fps-div");

var particleInfo = $("#particle-info-box");

ecollision.onTick = function() {

var fpsCurTime = new Date().getTime();

if (eCollisionSettings.showVelocities) {

var fps = "";

if (ecollision.fps < 24) {

fps = setCol(ecollision.fps, "red");

} else {

fps = setCol(ecollision.fps, "green");

}

debugStr = "Frame rate: " + fps + " Hz" +

"<br /> Update rate: " + setColGreen(ecollision.getUpdateRate()) + " Hz" +

"<br /> Energy in system: " + setColGreen(ecollision.graphUI.getEnergy()) + " kJ" +

"<br /> Number of particles: " + setColGreen(ecollision.engine.numOfParticles());

fpsDiv.html(debugStr);

} else fpsDiv.html("");

var selected = ecollision.simulationUI.getSelected();

if (selected != null) {

var str = "<b>XVel:</b> " + Math.round(selected.xVel\*eCollisionSettings.updateRate) + " px/s" +

"<br /> <b>YVel:</b> " + Math.round(selected.yVel\*eCollisionSettings.updateRate) + " px/s" +

"<br /> <b>Direction:</b> " + Math.round(toDegrees(Math.atan2(selected.yVel, selected.xVel))) + " degrees" +

"<br /> <b>Mass:</b> " + selected.mass + " kg" +

"<br /> <b>CoR:</b> " + selected.cOR +

"<br /> <b>Radius:</b> " + selected.radius + " px"

"<br /> <b>Energy:</b> " + Math.round(selected.getEnergy()) + " J";

particleInfo.html(str);

} else {

particleInfo.html("");

}

}

ecollision.start();

**style.css**

**body**, **head** {

background-color: #206396;

font-size: 14px;

font-family:Open sans, sans-serif;

color: black;

}

**ul**, **ol** {

list-style: none;

}

**li** {

#border-radius: 2px;

background-color: #B99DD1;

}

\* {

margin: 0;

padding: 0;

}

*/\*/*

*/\* canvas*

*/\*/*

#widget {

#background-color:#9DCFD1;

border-radius: 0px;

}

#canvas-wrapper {

width:100%;

position:relative;

background-color: #B2D4EF;

#background-color:#212121;

}

#simulation-canvas {

width:100%;

height:100%;

position:relative;

}

#overlay-canvas {

z-index:5;

position:absolute;

top:0;

width:100%;

height:100%;

background-color:rgba(178, 212, 239, 0.3);

display:none;

}

#simulation-area {

#background-color:#091D3D;

display: block;

padding: 7.5em 0 5.5em;

}

#graph-canvas {

width:100%;

height:100%;

}

*/\*/*

*/\* gridster*

*/\*/*

.gridster {

#width: 100px;

#margin: 0 auto;

background-color:#206396;

display: block;

padding: 7.5em 0 5.5em;

}

.gridster .gs-w {

background:#80B3DA;

#background:#646464;

cursor: pointer;

-webkit-box-shadow: 0 0 5px rgba(0,0,0,0.3);

box-shadow: 0 0 5px rgba(0,0,0,0.3);

}

.gridster .player {

-webkit-box-shadow: 3px 3px 5px rgba(0,0,0,0.3);

box-shadow: 3px 3px 5px rgba(0,0,0,0.3);

}

.gridster .preview-holder {

border: none!important;

border-radius: 0!important;

background: rgba(255,255,255,.2)!important;

}

*/\*buttons\*/*

*/\*http://paletton.com/#uid=13x0u0kiCFn8GVde7NVmtwSqXtg\*/*

#control-panel {

width:100%;

#height:60px;

position:relative;

display:inline-table;

}

#add-del-panel {

width:40%;

height:100%;

position:relative;

}

**button**{

position:relative;

border:0;

background-color:#B2D4EF;

border: solid #80B3DA 1px;

}

**button**:active {

background-color:#3977A7;

}

.control-button:before{

position:absolute;

left:0;

top:0;

width:50px;

height:100%;

#line-height:2;

padding:10px 0px 10px 5px;

font-size:130%;

text-align:center;

}

.config-button {

font-size:110%;

text-align:center;

padding:5px 5px 5px 5px;

}

.control-button {

width:50%;

margin-right:0px;

#margin-bottom:2px;

text-align:left;

padding:10px 20px 10px 50px;

width:50%;

float:left;

}

.control-button.max{

width:100%;

padding:10px 20px 10px 70px;

}

.control-button.no-icon {

width:100%;

padding:10px 20px 10px 20px;

text-align:center;

}

.add-del-button:before {

font-size:190%;

}

.add-del-button{

width:100%;

height:50%;

margin-bottom:0;

}

*/\* slider \*/*

.particle-slider-wrapper {

margin-top:3px;

margin-right:10px;

padding-top:3px;

float:left;

width:100%;

}

.particle-slider {

float:right;

width:45%;

margin-right:10px;

}

.slider-label {

float:left;

padding:0 10px 0 10px;

word-wrap: break-word;

margin-top:-3px;

}

.slider-amount {

position:absolute;

background:#3977A7;

visibility:hidden;

border-radius:5px;

z-index:5;

margin-left:-20px;

padding: 5px 10px 5px 10px;

top:-32px;

font-size:80%;

font-weight:150%;

width:auto;

height:auto;

}

.ui-slider-horizontal {

position:relative!important;

background:#B2D4EF!important;

border:0!important;

border-radius:0!important;

height: 10px!important;

padding:0!important;

}

.ui-slider-handle {

width:10px!important;

height:10px!important;

top:-1px!important;

border-radius:4px!important;

margin-left:-5px!important;

}

*/\* colour pickers \*/*

.colour-picker {

width:100%;

height:20px;

float:right;

margin-right:10px;

}

.minicolors-theme-default .minicolors-swatch {

top: 5px!important;

left: 5px;

width: 10px!important;

height: 10px!important;

}

.minicolors-theme-default.minicolors-position-right .minicolors-swatch {

left: auto;

right: 5px;

}

**index.html**

**<html>**

**<head>**

**<title>**ECollision**</title>**

*<!-- Load Open Sans font -->*

**<link** href='http://fonts.googleapis.com/css?family=Open+Sans' rel='stylesheet' type='text/css'**>**

*<!-- Load jQuery library -->*

**<script** src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"**></script>**

*<!-- Load jQuery UI library -->*

**<link** rel="stylesheet" href="https://ajax.googleapis.com/ajax/libs/jqueryui/1.11.3/themes/smoothness/jquery-ui.css"**>**

**<script** src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.11.3/jquery-ui.min.js"**></script>**

*<!-- Load jQuery gridster plugin -->*

**<link** rel="stylesheet" type="text/css" href="lib/jquery.gridster.min.css"**>**

**<script** src="lib/jquery.gridster.min.js"**></script>**

*<!-- Load jQuery mousewheel plugin -->*

**<script** src="lib/jquery-mousewheel-3.1.12/jquery.mousewheel.js"**></script>**

*<!-- Load EaselJS graphics library -->*

**<script** src="lib/easeljs-0.7.1.min.js"**></script>**

*<!-- Load IcoMoon font stylesheet -->*

**<link** rel="stylesheet" type="text/css" href="fonts/icomoon/style.css"**>**

*<!-- Load my stylesheet -->*

**<link** rel="stylesheet" type="text/css" href="style.css"**>**

**</head>**

**<body>**

**<div** class="gridster"**>**

**<ul></ul>**

**</div>**

**<script>**

var BUILD\_SCRIPT = "bin/build.js"

var gridster;

$(function(){

function resizeWidgets(e, ui, $widget) {

}

gridster = $(".gridster > ul").gridster({

widget\_margins: [5, 5],

widget\_base\_dimensions: [100, 50],

max\_cols: 6,

max\_rows: 6,

resize: {

enabled:true,

resize: resizeWidgets,

stop: resizeWidgets

}

}).data('gridster');

var panels = [ ['simulation', 4, 10],

['graph', 2, 4],

['sim-controls', 2, 3],

['particle-info', 2, 3],

['particle-controls', 4, 2]

];

var numLoadedPanels = 0;

$.each(panels, function(i, panel){

var p = panel[0];

panel[0] = '<li id="'+p+'"></li>';

gridster.add\_widget.apply(gridster, panel);

console.log("Loading: "+p);

$("#"+p).load("panels/"+p+".html", function() {

console.log("Successfully loaded: "+p);

numLoadedPanels++;

if (numLoadedPanels == panels.length) {

var element = document.createElement("script");

element.src = BUILD\_SCRIPT;

document.body.appendChild(element);

}

});

});

});

**</script>**

**</body>**

**</html>**

**panels/graph.html**

**<div** id="graph" style=" overflow:auto;"**>**

**<div** id="buttons" style="width:13%; float:right;"**>**

**<button** id="calibrate" class="icon-bars config-button" style="margin:0;"**></button>**

**<button** id="zoom-in" class="icon-zoomin config-button" style="margin:0;"**></button>**

**<button** id="zoom-out" class="icon-zoomout config-button" style="margin:0;"**></button>**

**<button** id="move-up" class="icon-arrow-up config-button" style="margin:0;"**></button>**

**<button** id="move-down" class="icon-arrow-down config-button" style="margin:0;"**></button>**

**</div>**

**<div** id="graph-wrapper" style="width:75%; height:100%; left: 10px; position: relative; float:left;"**>**

**<div>**Energy / kJ**</div>**

**<canvas** id="graph-canvas" style="width:100%; height:80%;"**>**

**</canvas>**

**<div** style="text-align:right;"**>**Time / ms**</div>**

**</div>**

**</div>**

**panels/particle-controls.html**

**<div** id="config-particles" style="height:100%;"**>**

**<div** id="add-del-panel" style="width: 30%; float:left;"**>**

**<button** id="add-particle" class="icon-plus add-del-button control-button"**>**ADD**</button>**

**<button** id="remove-particle" class="icon-pencil add-del-button control-button"**>**EDIT**</button>**

**</div>**

**<div** id="sliders" style="margin-top:3px; width: 70%; float:right;"**>**

**<div** class="particle-slider-wrapper"**>**

**<div** class="slider-label"**>**Mass (kg)**</div>**

**<div** id="slider-mass" class="particle-slider" min=1 max=1000**></div>**

**</div>**

**<div** class="particle-slider-wrapper"**>**

**<div** class="slider-label"**>**CoR**</div>**

**<div** id="slider-cor" class="particle-slider" min=0.01 max=1.00 step=0.01 value=1.0**></div>**

**</div>**

**<div** class="particle-slider-wrapper"**>**

**<div** class="slider-label" style="margin-top:8px;"**>**Colour**</div>**

**<button** id="generate-colour" class="icon-wrench control-button" style="float:right;margin-right:10px;width:45%;"**>**GENERATE**</button>**

**</div>**

**</div>**

**</div>**

**panels/particle-info.html**

**<center><p><b>**Selected particle**</b></p></center>**

**<div** id="particle-info-box"**></div>**

**panels/sim-controls.html**

**<div** id="control-panel"**>**

**<button** id="btn-run-pause" class="icon-pause control-button"**>**PAUSE**</button>**

**<button** id="btn-next" class="icon-next control-button"**>**NEXT**</button>**

**<button** id="btn-save" class="icon-move-up control-button"**>**SAVE**</button>**

**<button** id="btn-load" class="icon-move-down control-button"**>**LOAD**</button>**

**<button** id="btn-reset" class="icon-spinner control-button"**>**RESET**</button>**

**<button** id="btn-sim-data" class="icon-camera control-button"**>**DATA**</button>**

**<div** class="particle-slider-wrapper" style="margin-top:10px;"**>**

**<div** class="slider-label"**>**Speed**</div>**

**<div** id="sim-speed-slider" class="particle-slider" min=0.0 max=2.0 step=0.02 value=1.0**></div>**

**</div>**

**</div>**

**panels/simulation.html**

**<div** id="fps-div" style="z-index:10; position:absolute;"**></div>**

**<div** id="canvas-wrapper"**>**

**<canvas** id="simulation-canvas"**></canvas>**

**<canvas** id="overlay-canvas"**></canvas>**

**</div>**

## Test data

|  |  |  |  |
| --- | --- | --- | --- |
|  | Solution | Advantages | Disadvantages |
| Online software | * The user accesses a website and the demo runs in a browser. * <http://www.hoomanr.com/Demos/Elastic2/> | * No cost to access * Easy to use * The site includes an explanation of the physics behind the simulation | * Requires a plugin to run * Does not allow user to modify many variables * Does not include a graph of energy changes over time * Does not display energy of each particle * Does not allow user to add particles * Does not allow saving of simulation (saving variables, masses, directions) |
| Standard software | * The user downloads the software onto their computer * https://www.physicscurriculum.com/exploration\_of\_physical\_science.htm | * Works out-of-the-box * TODO: test the advantages of this program | * Expensive: a single user license costs $229.00 * Users would have to be at college to run the software * Must be run from the desktop * Customer service may not be good. * Includes unnecessary software |
| Bespoke solution | A custom designed solution | * The user can run the software from a web browser. This means they can run it from anywhere and on most devices. * Much more customizable; the customer can specify exactly what they want. * Bugs can be fixed more easily as the creators can be contacted. | * Expensive. * Takes time to make. |