Computing project 2014/2015

[Document subtitle]

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 Objectives 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 Objectives 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 objects. 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.

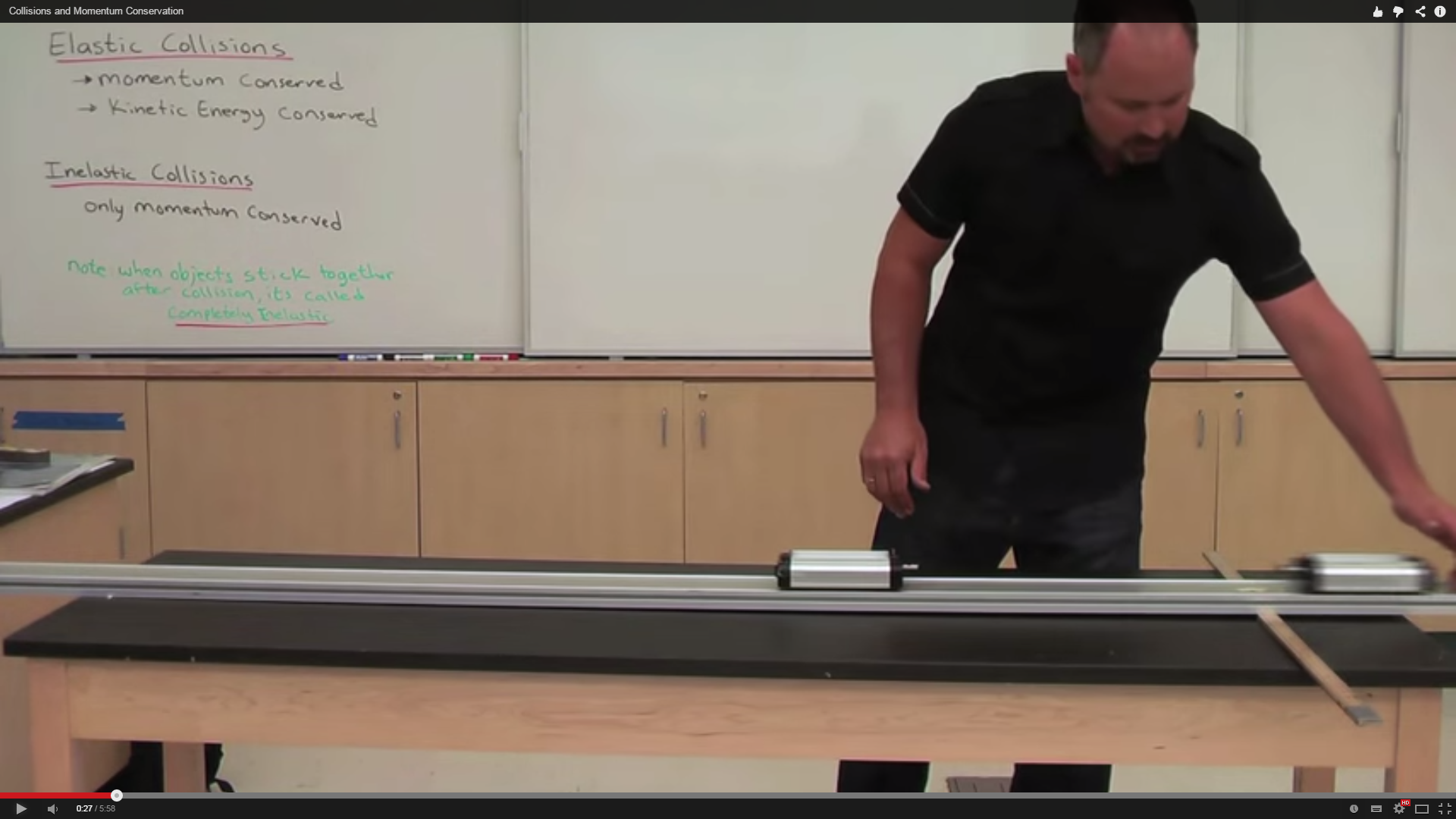
## Description of the current system

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.

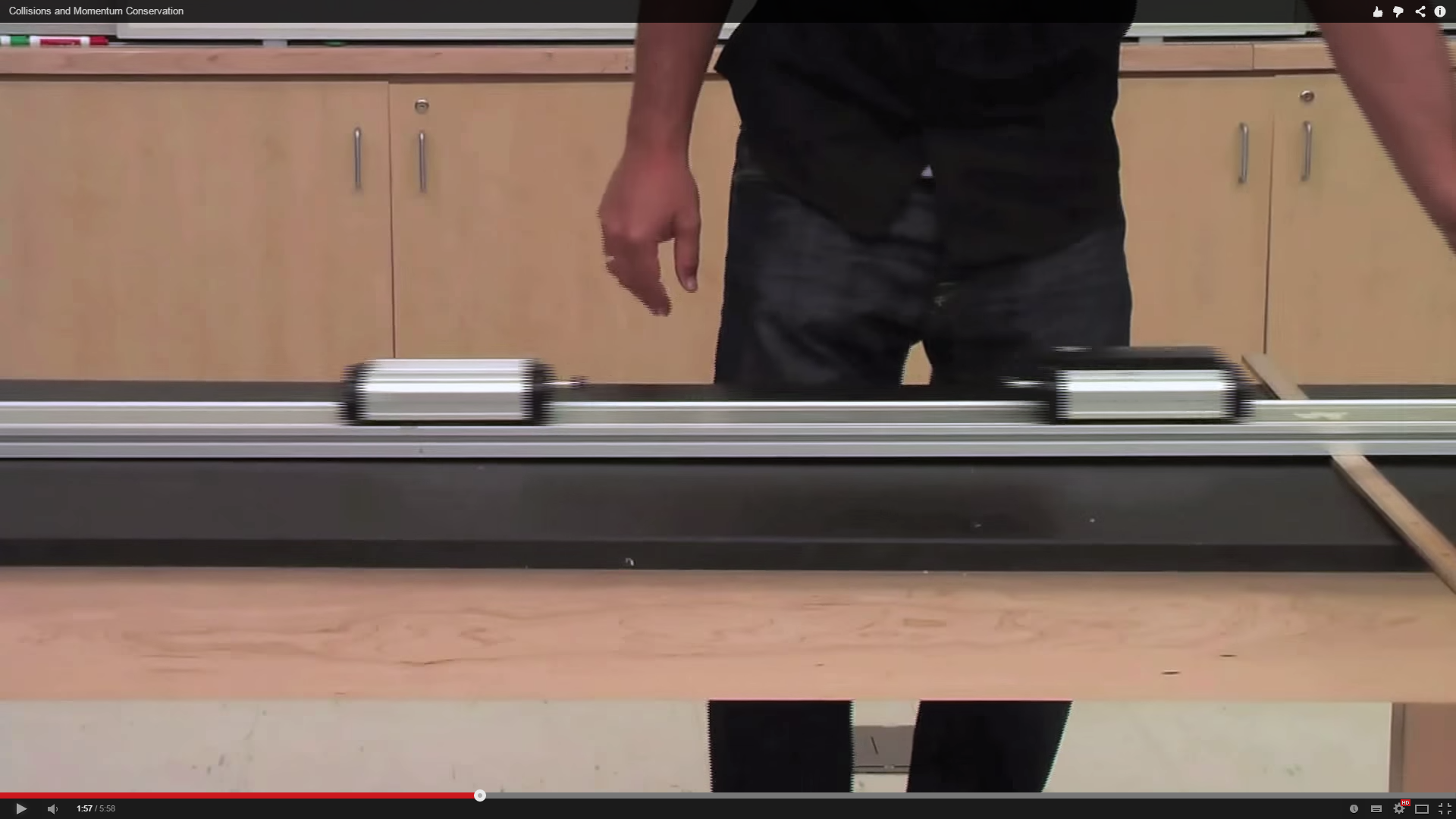
The teacher (Alison Frost) starts off by showing the students an example of an elastic collision with a real apparatus. This is an air-track (a track which blows air through holes in order to reduce the friction) on which two objects of specific masses are placed. They are then given different velocities.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

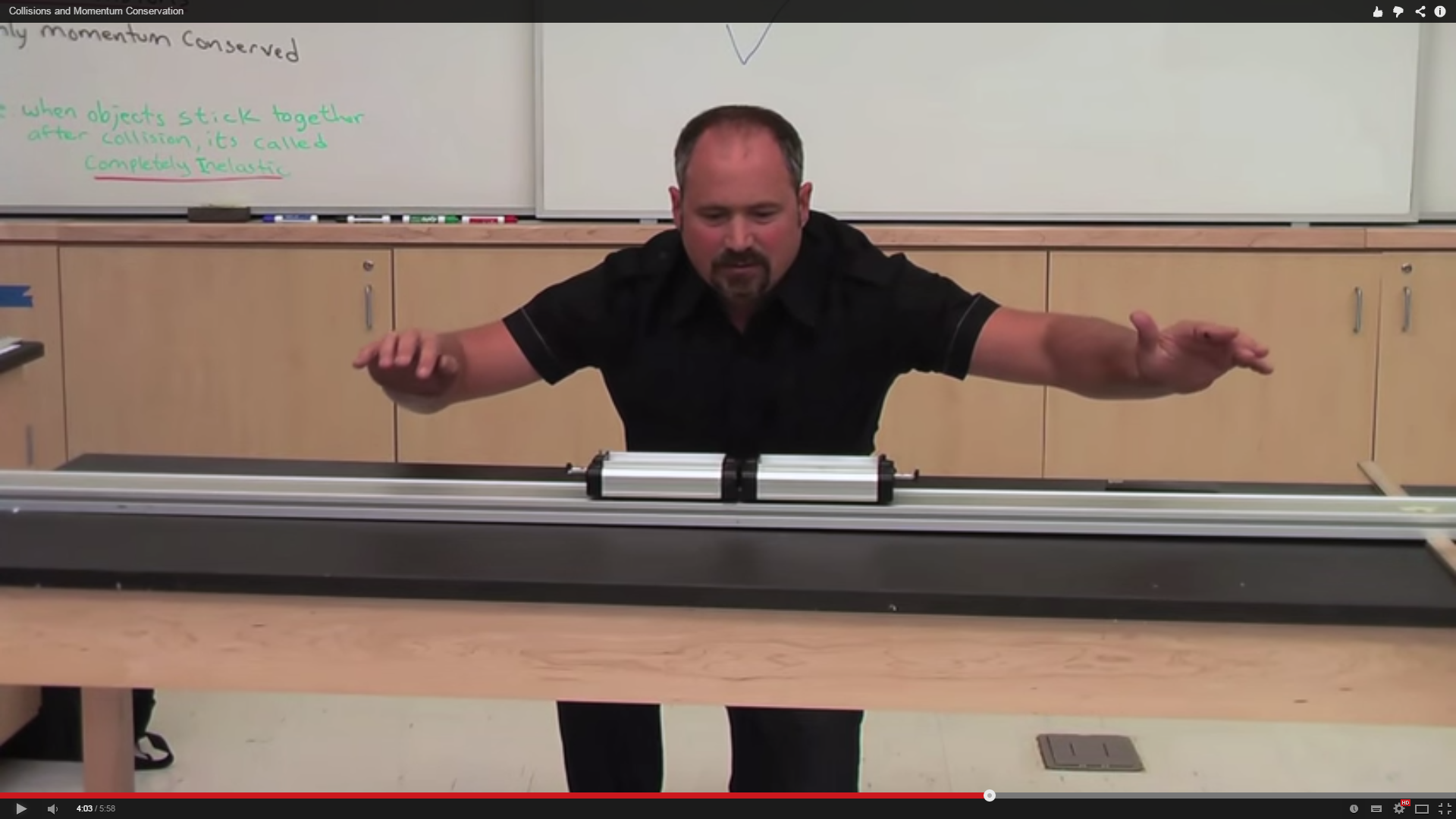
1. The first time the first cart is accelerated towards the second cart, which has no velocity. As expected, after the collision the second cart moves in the same direction as the first, which transfers most of its momentum to the second.



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. This is usually two balls, 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 question set. An example question would be: calculate the initial velocity of an object before collision given that you know the resultant velocities and masses of both objects. 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 objects 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:**

**Data flow for current system:**

Students

Teachers

Teacher explains concepts related to elastic collisions

Students learn concepts by taking notes/revising

Students get tested on what they have learned

Results

**Data flow diagram for proposed system:**

User

Enters variables into the system

D1

Stored in RAM

The user can change variables and add extra physics objects during simulation

Simulation runs, performing calculations using the variables from the user

Data displayed to user

## 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 object changes its position."

**Coefficient of restitution** – The coefficient of restitution (COR) of two colliding objects 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 objects with COR = 1 collide elastically, while objects 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 object 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

Main classes:

Class: Widget

Class: Simulation

Class: Graph

Class: PhysicsObject

Class: Particle

Other classes:

Class: Collision

Class: PVector

Class: Engine

## 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 objects in the simulation, such as mass, will be able to be modified by the user.
5. Objects can be added and removed from the simulation. A maximum of 100 objects can be added.
6. Start/pause/resume buttons will be added to control the simulation.
7. Step forward, step back will be added to manually “update” the simulation, either forwards or backwards in time.
8. Tracing objects functionality will be added to help the user find out how the balls 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 objects before and after collisions.
* The energy stored in a ball.
* The properties of the ball (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 balls 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 balls, one red, blue and green. The simulation can be paused and displays the direction of the balls, 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 balls cannot be added.
* The user interface of this system is unintuitive: there is no way of knowing if you have selected a ball 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 (Object-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 Objectives”. 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 object
* does not allow user to add balls
* 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** |
| Variables: mass, velocity, coefficient of restitution, radius, speed of simulation, enable gravity/debug | Take user input  Add/delete/modify ball  Check for collision between balls  Update ball velocities  Enable gravity/debug  Update simulation  Reverse simulation  Draw simulation |
| **Output** | **Storage** |
| Display simulation on the screen | No data is stored in this system |

## 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 objects, one will contain a graph, one will contain buttons and sliders for adding balls, 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

Draw Ball

Update Position

This is the structure of the index html page.

**Ecollision**

Main page

HTML Widgets

Graph

Simulation controls

Debug controls

Ball controls

\*Simulation

## OO Model

**Physics**

Simulation Engine

Physics Object

Particle

Draw Particle

Update Position

**Widgets**

Overlay

Simulation

Graph

**Maths**

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

**DisplayObject** DisplayObject

**PRIVATE:**

**METHODS:**

**PUBLIC:**

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

**PROC** Draw(x, y) //draw the object 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(object)

**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 PhysicsObject** Objects

**PRIVATE:**

**INT** GameRate

**INT** UpdateDuration

**METHODS:**

**PUBLIC:**

**Ball** 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:**

**PhysicsObject** Object1

**PhysicsObject** Object2

**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 | Slider |
| 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** RestoreObjectsToSavedState

**ENDIF**

**PROCEDURE** RestoreObjectsToSavedState:

**BEGIN**

Objects = LastSavedObjects

**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 objects – traces the path of each object on the screen

**IF** TracePathsButtonSelected **THEN**

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

**ENDIF**

1. Add ball – add a ball to the stage

**IF** AddButtonPressed **THEN**

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

**ENDIF**

1. Delete object – deletes a selected ball from the stage

**IF** DeleteButtonPressed **THEN**

**CALL** DeleteBall(selectedBallIndex)

**ENDIF**

1. Enable Gravity

**IF** EnableGravityButtonPressed **THEN**

Gravity = True

**ENDIF**

**FUNCTION** GetTimeOfCollision(Object1, Object2)

**BEGIN**

**STORE** PositionDifference = **NEW** PositionVector(Object1.x-Object2.x, Object1.y-Object2.y)

**STORE** VelocityDifference = **NEW** PositionVector(Object1.XVelocity-Object2.XVelocity, Object1.YVelocity-Object2.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(Object1.radius+Object2.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(Object1, Object2, Collision):

**BEGIN**

**STORE** DiffX = Object1.x-Object2.x

**STORE** DiffY = Object1.y-Object2.y

**STORE** TotalRadius = Object1.radius+Object2.radius

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

**STORE** TimeOfCollision = GetTimeOfCollision(Object1, Object2)

Collision.Time = TimeOfCollision

**RETURN TRUE**

**END IF**

**RETURN FALSE**

**END FUNCTION**

**PROCEDURE** HandleEdgeCollision(ball):

**BEGIN**

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

**THEN**

Ball.x = 0 + Ball.radius

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

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

**THEN**

Ball.x = SimulationWidth-Ball.radius

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

**END IF**

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

**THEN**

Ball.y = 0 + Ball.radius

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

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

**THEN**

Ball.y = SimulationHeight-Ball.radius

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

**END IF**

**END PROCEDURE**

**PROCEDURE** SeperateObjects(object1, object2, collision):

**BEGIN**

**END**

**PROCEDURE** HandleCollision(collision):

**BEGIN**

**STORE** SplitVelocityObj1 = SplitVelocities(collision.object1, collision.object2) //split velocities of object 1 into perpendicular and parallel components

**STORE** SplitVelocityObj2 = SplitVelocities(collision.object2, collision.object1) //split velocities of object 2 into perpendicular and parallel components

**CALL** CalculateNewVelocity(Object1, Object2, SplitVelocityObj1, SplitVelocityObj2)

**CALL** CalculateNewVelocity(Object2, Object1, SplitVelocityObj2, SplitVelocityObj1)

**CALL** SeperateObjects(collision)

**END PROCEDURE**

**PROCEDURE** CalculateNewVelocity(Object1, Object2, SplitVelocityObj1, SplitVelocityObj2):

**BEGIN**

**STORE** AngleBetweenObjects = GetAngleBetweenObjects(Object1, Object

**STORE** NewVelocity = ((SplitVelocityObj1.x \* (Object1.mass - Object2.mass)) + (2 \* Object2.mass \* SplitVelocityObj2.x)) / (Object1.mass + Object2.mass);

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

VelocityParallel.x = NewVelocity \* Cosine(AngleBetweenObjects)

VelocityParallel.y = NewVelocity \* Sin(AngleBetweenObjects)

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

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

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

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

ResultantVelocity.x = VelocityParallel.x + VelocityPerpendicular.x

ResultantVelocity.y = VelocityParallel.y - VelocityPerpendicular.y

**RETURN** ResultantVelocity

**END PROCEDURE**

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

**BEGIN**

**STORE** Ball = New Ball

Ball.x = x

Ball.y = y

Ball.radius = radius

Ball.mass = mass

**CALL** AddObjectToObjectsArray(Ball)

**RETURN** Ball

**END FUNCTION**

**PROCEDURE** RemoveBall(index):

**BEGIN**

**CALL** RemoveObjectFromObjectsArray(index)

**END PROCEDURE**

**PROCEDURE** UpdateSimulation:

**BEGIN**

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

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

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

**END FOR**

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

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

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

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

**THEN**

**STORE** Collision.Object1 = PhysicsObjects[I]

**STORE** Collision.Object2 = PhysicsObjects[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** DrawObjects:

**BEGIN**

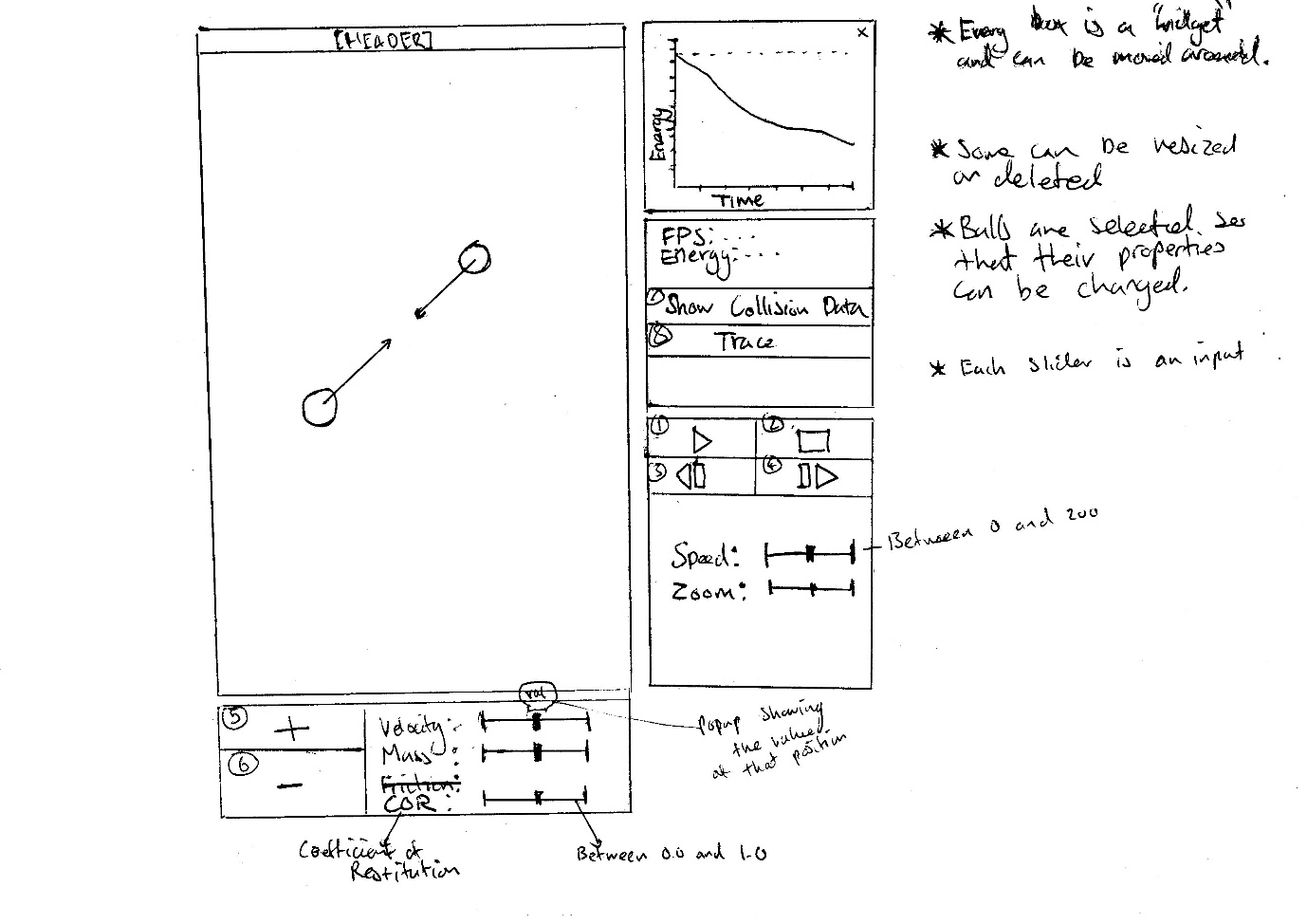
**FOR** I = 0 **TO** Length Of PhysicsObjects **DO**

**CALL** DrawObject(PhysicsObjects[I]);

**END FOR**

**END PROCEDURE**

## User Interface Design



1. Start/Pause simulation. When pressed its icon will move to the next state (i.e. from ‘started’ to ‘paused’).
2. Stop simulation. When pressed its icon will change into a reset icon. When this is pressed, the simulation returns back to the last saved state.
3. Step backward. ONLY WHEN PAUSED. This will move the simulation back by one update.
4. Step forward. ONLY WHEN PAUSED. This will move the simulation forward by one update.
5. Adds a ball to the stage. The simulation pane will blur out the background and an object. The crosshair can follow the mouse. When clicked, a ball will appear at the position of the cursor and its variables set to the sliders (velocity, mass and COR (Coefficient of Restitution)).
6. Removes a ball from the stage. The ball must be selected for this to happen (the ball is highlighted by a ‘selection colour’ so that the user knows which ball they have selected.)
7. This toggles the collision data that is showing (direction of movement,
8. This slider increases the velocity of the currently selected ball.
9. This slider increases the mass of the currently selected ball.
10. This slider increases the Coefficient of Restitution. The Coefficient of Restitution changes the amount of energy that is lost during a collision.
11. This button saves the current state of the simulation.

## 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

# System Testing

## Test Plan – Typical Data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Test** | **Test Data** | **Expected Result** | **Actual 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** | **Actual 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** | **Actual 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.

## Algorthms

**FUNCTION** GetTimeOfCollision(Object1, Object2)

**BEGIN**

**STORE** PositionDifference = **NEW** PositionVector(Object1.x-Object2.x, Object1.y-Object2.y)

**STORE** VelocityDifference = **NEW** PositionVector(Object1.XVelocity-Object2.XVelocity, Object1.YVelocity-Object2.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(Object1.radius+Object2.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(Object1, Object2, Collision):

**BEGIN**

**STORE** DiffX = Object1.x-Object2.x

**STORE** DiffY = Object1.y-Object2.y

**STORE** TotalRadius = Object1.radius+Object2.radius

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

**STORE** TimeOfCollision = GetTimeOfCollision(Object1, Object2)

Collision.Time = TimeOfCollision

**RETURN TRUE**

**END IF**

**RETURN FALSE**

**END FUNCTION**

**PROCEDURE** HandleEdgeCollision(ball):

**BEGIN**

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

**THEN**

Ball.x = 0 + Ball.radius

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

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

**THEN**

Ball.x = SimulationWidth-Ball.radius

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

**END IF**

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

**THEN**

Ball.y = 0 + Ball.radius

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

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

**THEN**

Ball.y = SimulationHeight-Ball.radius

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

**END IF**

**END PROCEDURE**

**PROCEDURE** SeperateObjects(object1, object2, collision):

**BEGIN**

**END**

**PROCEDURE** HandleCollision(collision):

**BEGIN**

**STORE** SplitVelocityObj1 = SplitVelocities(collision.object1, collision.object2) //split velocities of object 1 into perpendicular and parallel components

**STORE** SplitVelocityObj2 = SplitVelocities(collision.object2, collision.object1) //split velocities of object 2 into perpendicular and parallel components

**CALL** CalculateNewVelocity(Object1, Object2, SplitVelocityObj1, SplitVelocityObj2)

**CALL** CalculateNewVelocity(Object2, Object1, SplitVelocityObj2, SplitVelocityObj1)

**CALL** SeperateObjects(collision)

**END PROCEDURE**

**PROCEDURE** CalculateNewVelocity(Object1, Object2, SplitVelocityObj1, SplitVelocityObj2):

**BEGIN**

**STORE** AngleBetweenObjects = GetAngleBetweenObjects(Object1, Object

**STORE** NewVelocity = ((SplitVelocityObj1.x \* (Object1.mass - Object2.mass)) + (2 \* Object2.mass \* SplitVelocityObj2.x)) / (Object1.mass + Object2.mass);

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

VelocityParallel.x = NewVelocity \* Cosine(AngleBetweenObjects)

VelocityParallel.y = NewVelocity \* Sin(AngleBetweenObjects)

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

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

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

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

ResultantVelocity.x = VelocityParallel.x + VelocityPerpendicular.x

ResultantVelocity.y = VelocityParallel.y - VelocityPerpendicular.y

**RETURN** ResultantVelocity

**END PROCEDURE**

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

**BEGIN**

**STORE** Ball = New Ball

Ball.x = x

Ball.y = y

Ball.radius = radius

Ball.mass = mass

**CALL** AddObjectToObjectsArray(Ball)

**RETURN** Ball

**END FUNCTION**

**PROCEDURE** RemoveBall(index):

**BEGIN**

**CALL** RemoveObjectFromObjectsArray(index)

**END PROCEDURE**

**PROCEDURE** UpdateSimulation:

**BEGIN**

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

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

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

**END FOR**

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

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

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

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

**THEN**

**STORE** Collision.Object1 = PhysicsObjects[I]

**STORE** Collision.Object2 = PhysicsObjects[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

## Annotated listings / screens

//Collision class that stores which objects 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 "seperateObjects" below.

***this*.**seperateObjects**(**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*.**seperateObjects **=** ***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*** colObjects **=** **[];**

//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**;**

colObjects**.**push**(**collision**);**

**}**

**}**

**}**

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

colObjects**.**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 **<** colObjects**.**length**;** i**++)** **{**

***var*** collision **=** colObjects**[**i**];**

***this*.**handleCollision**(**collision**);**

**}**

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

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

***var*** obj **=** objects**[**i**];**

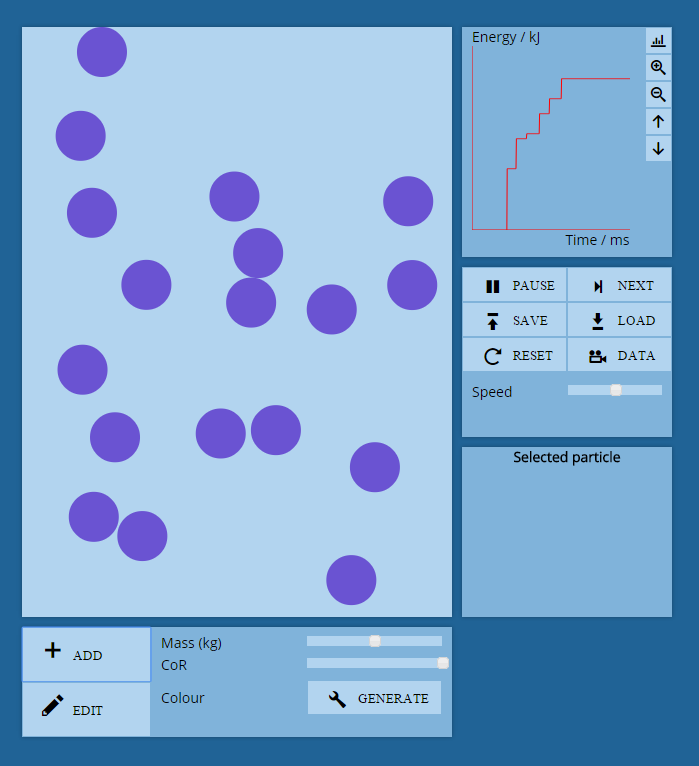
***this*.**edgeCollision**(**obj**,** ***false*);**

**}**

**}**

## Database Definitions

## Forms / screens



# 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 object. 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:

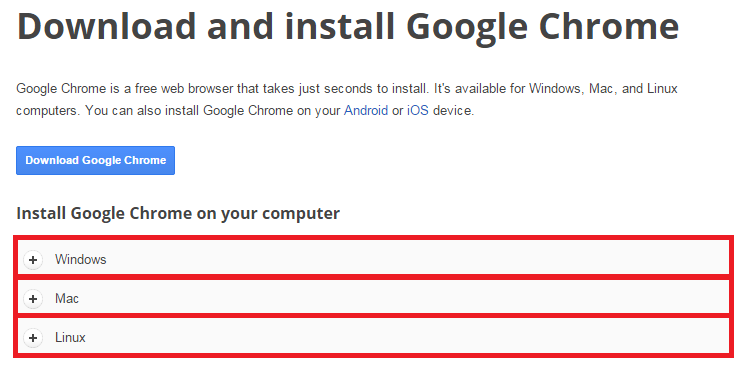
* 1.3 GHz Processor
* 512 MB RAM
* 350 MB Disk space
* Internet connection
* Google Chrome

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>

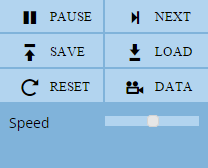


Select one of the red boxes which matches your system. A guide will come up on how to install Google Chrome.

## 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

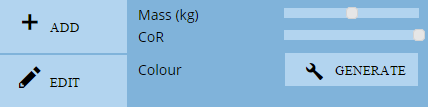
Click the DATA button to get information about the current simulation.

* 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

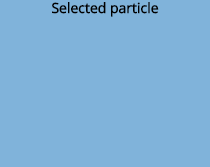
Click EDIT or press “e”. To copy a particle, hold SHIFT and click on the particle. This will copy it and the copy can be placed (see “Adding particles” above).

* 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

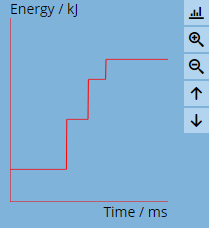
To get particle information, click on the particle. Information will appear in this panel: 

* 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

# 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 Objectives

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 objects 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 ball is added. The properties of the objects can also be modified once they’ve been added by clicking on the EDIT button.

1. **Objects can be added and removed from the simulation. A maximum of 100 objects can be added.**

Objects 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, step back will be added to manually “update” the simulation, either forwards or backwards in time.**

A step forward button is implemented to move the simulation forward by one update. However, a step back button was not possible due to the amount of data that would be required to store the simulations previous updates.

1. **Tracing objects functionality will be added to help the user find out how the balls 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 objects before and after collisions.**
* **The energy stored in a ball.**
* **The properties of the ball (velocity, mass, coefficient of restitution)**

The direction of the particles can be displayed to show the velocity of the

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

## Client/ User Feedback

## Project Extensions

# 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 balls (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.

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

**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 objects 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 objects 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 ball.

## Summary of Questionnaires

## Original documents of system

## Program listing

## 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 object * Does not allow user to add balls * 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. |