**ANALYSIS**

**The Problem Area:**

The problem area that I am interested in is in the development of a physics and collision engine, where the end user is able to test how particles of different mass, size and shapes behave and collide in different situations such as with different velocities, acceleration, densities and under the effect of different values of gravity. The reason why I chose this problem area is that I want to study Computer Science at University and one of my favourite areas of Computer Science is computer graphics. This paired with the fact that I enjoy mechanics makes this project perfect for someone like me.

There is a long list of simulators and physics engines online and in video games that have modelled physics and collisions realistically and have done so with knowledge of degree level maths and computer science. Since I have not yet begun learning advanced mechanics yet, it would be better to not include collisions between polygons other than quadrilaterals, as detecting these can get quite complex for someone of my ability. It is also a better idea to ignore rotation in collisions as this requires knowledge in advanced moments, which will allow me to demonstrate my programming ability without being hindered by my limited knowledge in mathematics and physics.

Because each particle will be affected by gravity and will be subject to a force, I know that Newton’s second law of motion, F=ma, will be a key equation in my project. Furthermore, I will most likely have a list of particles, in which each particle will be compared with every other particle to check for collisions, thus I can already predict that my program will have algorithms of at least O(n2) complexity. This paired with the fact that my program will be using object oriented programming, to keep the engine as modular as possible, I know that my program will be complex enough for an A-level project.

**End-user questionnaire:**

After a discussion with my end user, Dr Dominic Keogh PhD PGCE FRAS MSc, I was glad that he had given me a program to code that allows me to demonstrate my knowledge and passion for the subject. Since Dr Keogh is my physics teacher, it was very easy to get in touch with him to discuss the progress of the project with him after lessons or through emails when were not able to meet physically. After a thorough discussion with him about the program, I gave him a questionnaire to complete, which would allow me to set realistic and measurable set of specific objectives according to his needs..

1. What kind of functionality would you like the Collision simulator to have?  
   **The ability to take particles and show what is happening to their collision in 2 dimensions in different scenarios. The ability to see the speeds of the particles before and after and compare their speeds. I would like it to be clear so that I can show it to students easily. Also, I can understand that polygons other than rectangles might require algorithms that are too complex for someone of your current ability, so I want you to only include quadrilaterals and circles.**

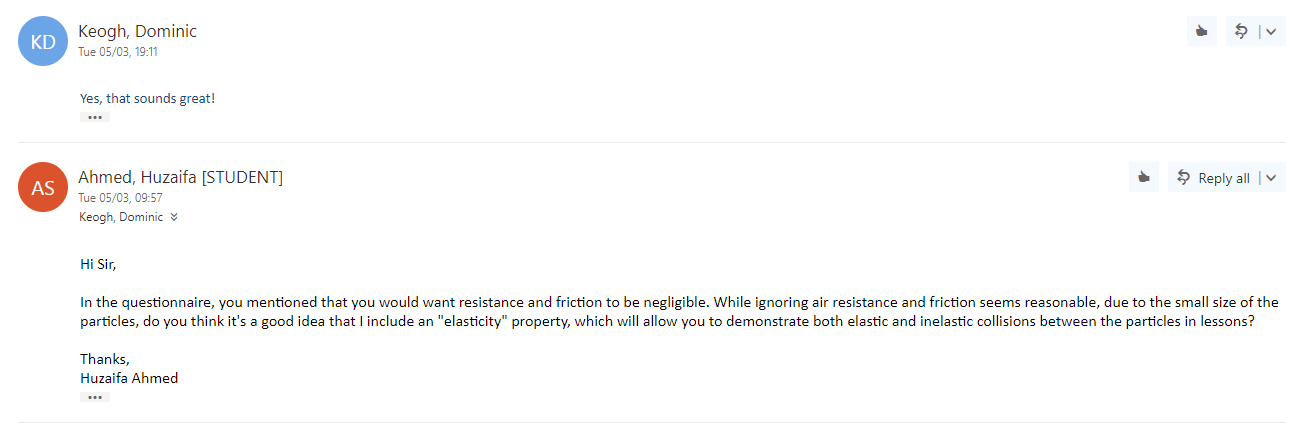
**I also know that you are interested in creating games and all sorts of other simulations, so it is okay if you make a program that acts more like a multi-purpose engine rather than just for my needs.**

1. Should I include rotation (in terms of collisions)  
   **This feature would be interesting to see and use but I suspect difficult to implement due to needing to include several aspects of moments which would go beyond standard collision effects. In terms of demonstrating the effects in a classroom, it would go beyond any use in schools where these effects are studied separately. So, I am happy for you to model the objects without rotation as that would be too complicated for an A-level project. However, this means that since quadrilaterals that collide with other particles won’t rotate, the simulation can appear inaccurate, so I am also happy for you to include quadrilaterals as static particles, without rotation, so they can act as walls/ obstacles.**
2. What assumptions should be made for the model?  
   ***I would expect objects to be modelled as point particles with other forces such as air resistance and friction to be negligible as these would make it more difficult to see what is going on in the collision.***
3. How many and what kind of options for the objects would you like?   
   **Customise both size and mass separately, hence density would be changed indirectly, automatically.**
4. Would you like keyboard shortcuts?  
   **I do not personally find these useful in software such as this. I don’t need these. I’d prefer the project to work from mouse clicks as it means fewer things to worry about.**
5. Would you like a slider that changes the value of g and the speed of the simulation?  
   **Both of these features would be useful for demonstrating the effect of different values of gravity.**
6. Would you like an option to pause the simulation?  
   **Yes, this would be good for showing what is happening at certain points**
7. Would you want to place additional objects while the simulation is running?  
   **Yes, adding additional particles during the simulation would certainly allow me to better demonstrate collisions.**

**It would also be a good idea to allow me to remove particles.**

1. Would you want to be able to drag around objects while the simulation is running?  
   **This sounds like a useful feature as it would allow me to alter the position of objects which will be very helpful in aiding students’ understanding.**
2. Would you like to be able to apply a force on the objects while the simulation is running?  
   **This would also be interesting and useful**

After exchanging further emails based upon this questionnaire, both I and Dr Keogh agreed that it would be a better idea to include an “elasticity” variable for each particle which will allow the end user to determine themselves if they want elastic or inelastic collisions.



**Questionnaire Analysis and Research**

From my research, I found that including rotation in quadrilateral collision would indeed be very difficult for someone with my mathematical knowledge. While collision detection in Axis aligned bounding boxes (AABBs) is very straightforward, collision detection with oriented bounding boxes require knowledge of the separating axis theorem and resolving these collisions need an advanced understanding in matrices and vector maths.

In terms of the User-Interface for the program, as this program’s main purpose is to be used in the classroom to demonstrate collisions to student’s, It should have a simple appearance and should only consist of controls that allow the end user to input the values for mass and dimensions of particles, the speed of the simulation, the value of acceleration due to gravity and the value of the coefficient of restitution. Furthermore, the UI should also contain buttons to pause and start the simulation, and while Dr Keogh Didn’t explicitly ask for the ability to remove particles or to reset the simulation, these features would improve the usability of the program so buttons to reset the simulation and to delete particles would be needed as well.

Furthermore, I found that what I intended on doing was assigning a Coefficient of restitution (CoR) to the collisions rather than defining a particle’s elasticity. This is because the CoR is the ratio of the final to initial velocities between two objects after they collide whereas elasticity is an object’s ability to resist deformation. However, since the coefficient of restitution is an experimental value, it relies on the ratio of final to initial velocities and the materials involved, it can’t be determined in my program as the final velocities relies on the coefficient themselves, which means that I will have to set my own coefficient of restitution for collisions.

Moreover, I realise that the main purpose of the program is to create an understanding in students of how elastic and inelastic collisions work and about the concept of conservation of momentum rather than using the program for scientific experiments. Therefore, I can assume that while the accuracy of physics is significant, it is not the most important aspect of the program. As long as the program is able to give students an idea of what happens during a collision, the program's main objective will be accomplished. That being said, I will still try to model the physics as accurately as possible, given my own understanding of the mathematics involved. Furthermore, to better aid a student’s understanding, a tutorial will be needed as well, so if students want to try the simulator themselves, they know how to use it.

**Modelling assumptions:**

* Dr Keogh specified in my questionnaire that he objects to modelled as point particles. This means that we can assume that the centre of mass of the particle is uniform and its mass always acts downwards from its centre.
* Because rotation is not required, all particles will be contained within axis-aligned bounding boxes.
* As specified by my end user, air resistance and friction will be negligible, hence all collisions will be elastic if the collisions and all particles will be modelled as smooth. However, after further dialogue, he is happy for me to include the coefficient of restitution, which will let him decide whether the collisions are elastic or inelastic; this still excludes air resistance and friction.
* All particles will be modelled as rigid bodies.

**Choosing the Programming Language**

Quora and Dreamincode are two websites used to research programming languages [1 and 2]

* **Visual Basic:** I Have been learning Visual Basic since year 10 so it can be argued that due to my familiarity with the language, this is a good option. Visual Studio as an IDE is very friendly which allows for the creation of classes and subroutines to be relatively fast and easy and its GUI allows me to create a UI with ease. Furthermore, Visual Basic is not a true Object-Oriented language although it claims to be therefore it becomes overly complex and difficult to manage on very large programs.
* **C#** is also a good option not only because I have had some experience with it before, mainly through Unity, but also because it is very similar to VB in a lot of ways. It combines the functionality of C and C++ languages with visual basic. Furthermore, it is Objected Oriented, so It can model complex real-world problem efficiently. C# has also become a very popular language within the industry, so it would be a very important language to learn for the future.
* **Java** is also a suitable language to be used for this project. It is very robust and simple to use, compile and debug. Java is also an object-oriented language, so it Is well suited for a modular program such as a physics engine. However, I am not very familiar with creating a GUI in Java and the syntax is also different to C# and VB, meaning I will have to learn and get used to the syntax used in Java, which would mean I get less time to create my project.
* **C++** is an extremely fastand efficient programming language that works well for GUI programming. However, it is mainly suitable for low-level programming and platform-specific applications. It is also a very complex language to learn and more complex programs, such as a physics engine, can be even harder to code and debug.
* While **Python** is a general-purpose object-oriented programming language, it is more appropriate for databased/ artificial intelligence applications. Python is also a very beginner friendly language; however, its syntax is wildly different to that of C# and VB, so it would also, similar to C++, require extra time to learn the language, taking away development time.

Therefore, after my research on different programming languages, I have decided to stick to **Visual Basic**. This is because while I have been learning VB for over 3 years, I have yet to fully understand and master the language. For example, drawing and taking complete advantage of the Object-Oriented Programming paradigm, by including encapsulation, polymorphism, Inheritance and composition in a complex program, such as a physics engine, is one of the things I have yet to attempt. Mastering Visual Basic now, in my opinion will allow me to better understand languages such as java and C++, which I will inevitably study at university. The main reason why I did not choose C# is because while I am familiar with the language, I do not fully understand it as well as Visual basic, which means that creating a program that relies on mathematics, including areas that I have yet to study, will be even harder with C# and might lead to a more incomplete/ lesser functional program.

**Choosing the Graphical Library**

One of the things I have never coded before is graphics, therefore, after some research to find out how graphics are drawn, I found that VB.net has a class-based API called **GDI+** (Graphic User Interface +)[3], which is an easy and effective way of creating graphics. After looking up the difference between GDI+ implementation between C# and VB.net [4], the two programming languages I am familiar with, I found that neither has any significant advantages or disadvantages over the other. **Processing** and **OpenGL** are two other graphics libraries that I could use. However, while I have had some experience in Processing, mainly following Daniel Schiffman’s tutorials on his YouTube channel: The Coding Train, the reason why I do not want to use Processing is that it uses Java, which I only have limited knowledge of. Similarly, while OpenGL is unarguably a better and powerful graphics library, I would much rather create something simpler yet complete rather than create something complex but incomplete (due to my lack of knowledge of OpenGL).

**Objectives broken down into measurable success criteria:**

|  |  |  |
| --- | --- | --- |
| Main Objective | Explanation and Analysis | Success Criteria |
| 1. Make a physics engine using particles. | A physics simulator with collisions can generally be split into two categories: physics engine and collision engine. The physics engine should be responsible for managing physics such as applying forces and updating positions and vectors. | |  | | --- | | 1. Particles should be able to move in any direction at a velocity. | | 1. Non-fixed particles should be subject to gravity unless they are static objects. | |
| 1. Make a collision engine using particles. | The collision engine would be responsible for collision detection and collision resolution. The collision resolution part can be further split into static resolution, pushing 2 colliding objects apart, and dynamic resolution, which determines the resultant forces and directions. | |  | | --- | | 1. Particles should not be able to intersect other particles unless they are both fixed, if they do, they both should be displaced accordingly so that they do not intersect anymore. | | 1. If a particle is “launched” towards another particle and they collide, a force should be applied on both particles in appropriate directions. The momentum in the collision should also be conserved. | | 1. Elastic and inelastic collisions will rely on a coefficient of restitution (CoR) variable, meaning a CoR of 1 will result in no energy loss of the particular particle and CoR of 0 will mean the particle will stop after any collision. | | 1. Non-fixed particles should stay within the bounds of the drawing area, if one attempts to leave, it shall bounce of/ displace accordingly. | |
| 1. Be able to spawn particles anywhere, with user-specified position, radius and mass and to be able to delete said particles. | Since my end user has asked me to create a program with the ability to take particles and show what happens during their collisions, to show how the particles interact, the end user needs to be able draw particles exactly where they want them to be. Furthermore, I want to avoid adding particles by specifying their positions in textboxes, instead, I think being able to spawn particles at the Mouse locations will be better and easier for the end user. | |  | | --- | | 1. The user is able to spawn a particle at the x and y position of their cursor. | | 1. The user is able to specify the radius or width and height and mass of the particle before spawning it in. | | 1. The user should be able to delete any specified particle. | |
| 1. Be able to change the speed of the simulation, as well as pause, and reset the simulation | My end user wants to clearly see and demonstrate what happens during a collision, so he has asked me that the program should allow him to pause the simulation as well as change its speed, so make it slower. This task should be straightforward as all I would need to do is change the timer tick interval and add a pause and stop timer button. | |  | | --- | | 1. If the simulation is running, it should be able to pause, and vice versa. | | 1. The end user should be able to control the speed of the simulation. I.e. make it slower through a slider. | | 1. The user can completely reset the simulation. (All particles should be removed and the values for the coefficient of restitution, speed of the simulation and acceleration due to gravity should be set to default, which are 1, 1x slower and 9.81 respectively.) | |
| 1. Be able to change the value of acceleration due to gravity and the Coefficient of restitution of collisions while the simulation is running. | This is another feature that allows the end user to test interactions between particles in different environments. If the end user wanted to model the collisions on the moon instead of the Earth, they could use 1.6 m/s2 instead of 9.8 m/s2 | |  | | --- | | 1. The end user can change the value of acceleration due to gravity with a slider. The minimum value should be 0, so they can simulate collisions in space. | | 1. The end user can change the value of the CoR with a slider. The minimum value should be 0 to demonstrate complete loss of energy and the maximum should be 1 to demonstrate complete conservation of energy. | |
| 1. Be able to manually alter the position of particles during a simulation. | The end user specified that being able to alter the positions of particles will allow him to better aid a student’s understanding, so this is an important objective. Instead of entering data into text boxes to alter particles positions, I want to use a system similar to spawning particles, as stated above, by using mouse coordinates. | |  | | --- | | 1. The user is able to pick up a particle, drag it around and drop it wherever their mouse cursor is. | |
| 1. Be able to apply a force to particles while the simulation is running | As my end-user specified in my questionnaire, he should be able to apply a force on a particle. Similar to placing particles, applying of a force shall be done through the mouse method as this is more convenient than entering numbers in text boxes. | |  | | --- | | 1. The user should be able to apply a force to particles with their mouse. | | 1. Newton’s second law of motion: F=ma should be obeyed when applying a force, so the same size of force on a heavier particle should apply a smaller acceleration than a lighter particle. | |
| 1. The speeds of particles before and after collisions should be clearly visible. | My end user has explicitly stated that the speeds of the particles should be clearly visible for his students. This can be simply achieved by creating velocity arrows for each particle to represent their speed and direction visually and then have the information display. Achieving this wouldn’t be tough as a listview in VB. | |  | | --- | | 1. A listview should include information about each particle such as the velocities of the particles, their positions, mass, and density | | 1. The listview should automatically add items when a new particle is created | | 1. Anytime a property of a particle is updated, such as its position, the corresponding value in the listview should update as well. | | 1. Each particle should have a arrow drawn on it to represent its velocity. | |

While it can be argued that my entire project is a physics engine, strictly speaking the physics engine would only consist of classes that control the physics and not the entire project itself. Creating the form and buttons and other sorts of inputs to meet the demands of the end user, in my opinion, are not part of the physics engine but they are an example of the engine being used to create something, they are an application of the engine.

**The critical path:**

As I have never had any experience with using GDI+, drawing particles would be one of the most important parts of my program. This is not only because I would need extra time to learn how to draw particles and refresh the screen but also because drawing particles is a key component of the program, without particles, there is no physics engine.

In terms of the critical path, below is a list of potential aspects of my program ranked in decreasing order of importance.

1. **Create a Vector class**

Although aspects #2-4 are very important, they cannot actually be achieved if I do not have the appropriate structure: vectors, for storing information like positions and velocities. However, the important part is to just create a very simple vector class at the start, methods for carrying out vector maths can be added later on as needed.

1. **Drawing Particles.**

As explained above, this is the most important part as without drawing particles; I cannot do any of the other objectives (apart from creating the UI)

1. **Applying realistic gravity to particles.**

This is ranked below drawing particles as I shouldn’t be applying gravity to particles if they are not visible on the screen because nothing would happen. This is above collision detection because it just as important for physics engine as collision detection but can be done a little quicker.

1. **Collision detection and resolution between particles.**

It can be argued that collision detection is one of the most important areas of a physics simulation. However, it is ranked third because particles can’t collide if they are not visible. This would probably be one of the most time consuming sections to code as well due to the fact that I would need to learn some collision detection and resolution logic and develop my own algorithms. This is only ranked underneath applying gravity and drawing particles because those areas are equally important and could arguably be achieved in less time.

1. **Being able manipulate particles (applying a force and dragging around).**

While applying a force to particles and dragging them around are important objectives specified by my end user, they do not contribute as much to the over functionality of a physics engine as much as the areas ranked higher.

1. **Being able to change the values of CoR, Gravity and speed of the simulation.**

The reason why this is ranked lower than #4 is because while it is just as important for the end user, being able to change these values can be simply done through sliders and this task will not take a lot of time.

1. **Having an organised UI**

An organised UI is very important for any simulation program. However, in terms of functionality and complexity of the other tasks, this is ranked lower because not only is creating a UI not time consuming as the other tasks, but also it does not contribute to the functioning of the program.

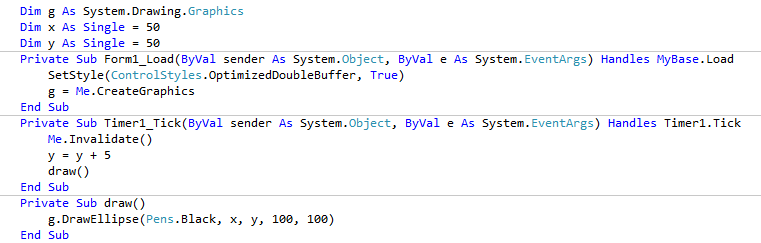
**Ancillary aspects:**

1. Pausing and resuming the simulation
2. Resetting the simulation
3. Displaying information of particles in tables.
4. Creating a tutorial.

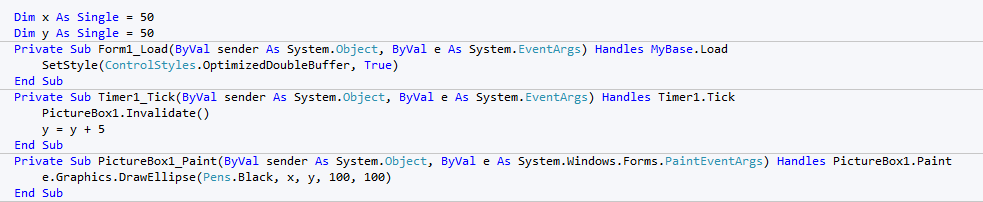
**Prototyping the critical path**

My main objective for this prototype was to simple work out how to draw particles, as this is something I have never done in VB.net before. During prototyping I found that despite setting the form’s double buffering to true, creating a graphics object and using that created a lot of unnecessary stutter when invalidating the form. The only fix I found to this was to create a picture box and use the picture box’s paint event to draw on it and invalidate only the picture box.

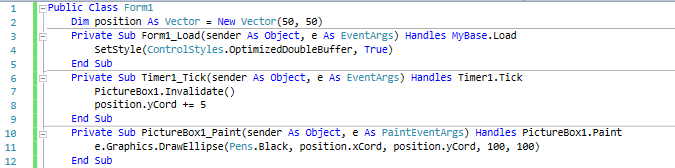
Once I had drawing particles sorted, I implemented a very basic form of gravity and circle to circle collisions as these were fairly easy to figure out. I also created a vector class which I show in the 3rd and 4th screen shot underneath.



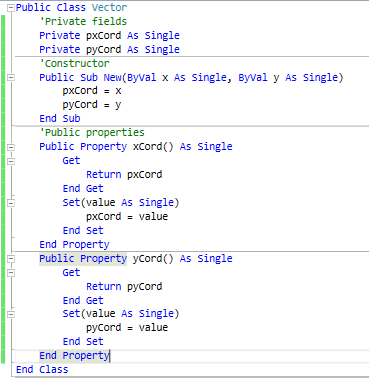
Method with Flicker



Flicker Free Method

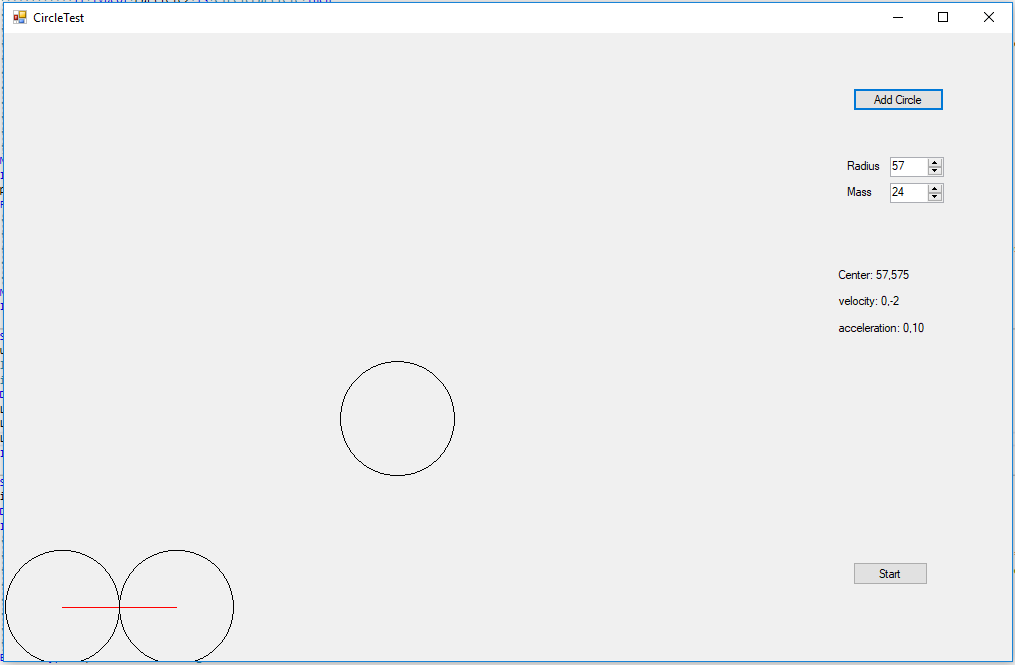


Flicker Free Method with a vector class



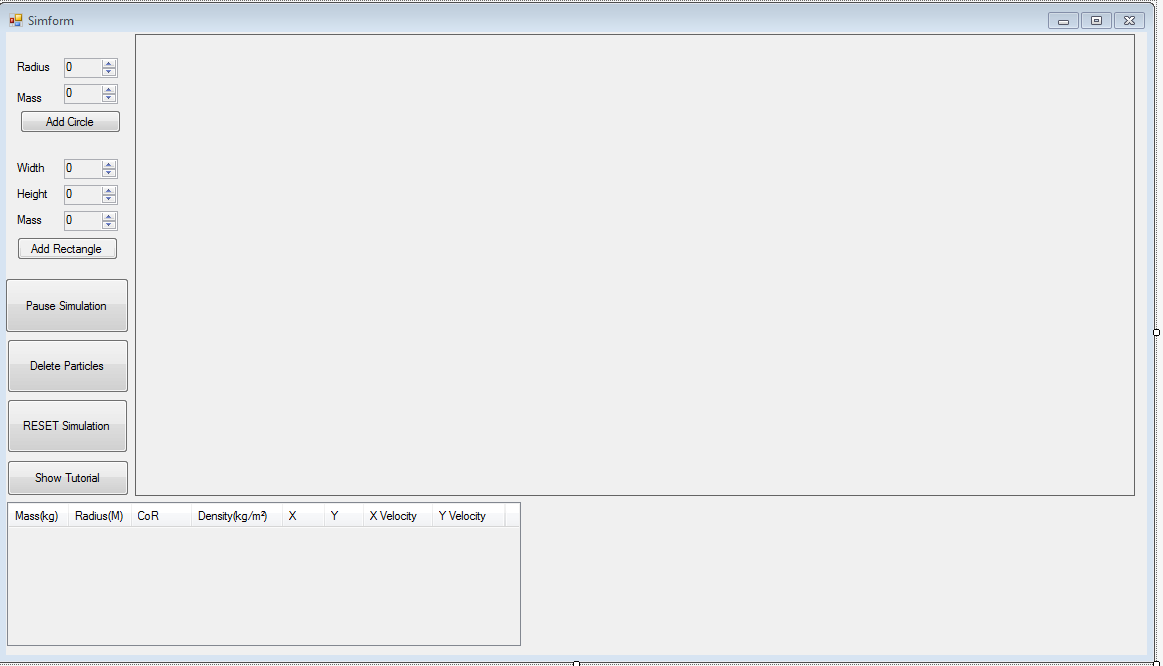
Vector Class

The fact that I must use the picture box’s paint event to draw my particles already suggests to me that I would need some public global variables because the paint event cannot be directly called and can’t have any additional arguments passed into it. However, having a couple of global variables is not a huge issue as the memory they will occupy will not be that significant and it will not make the programmer harder to debug.



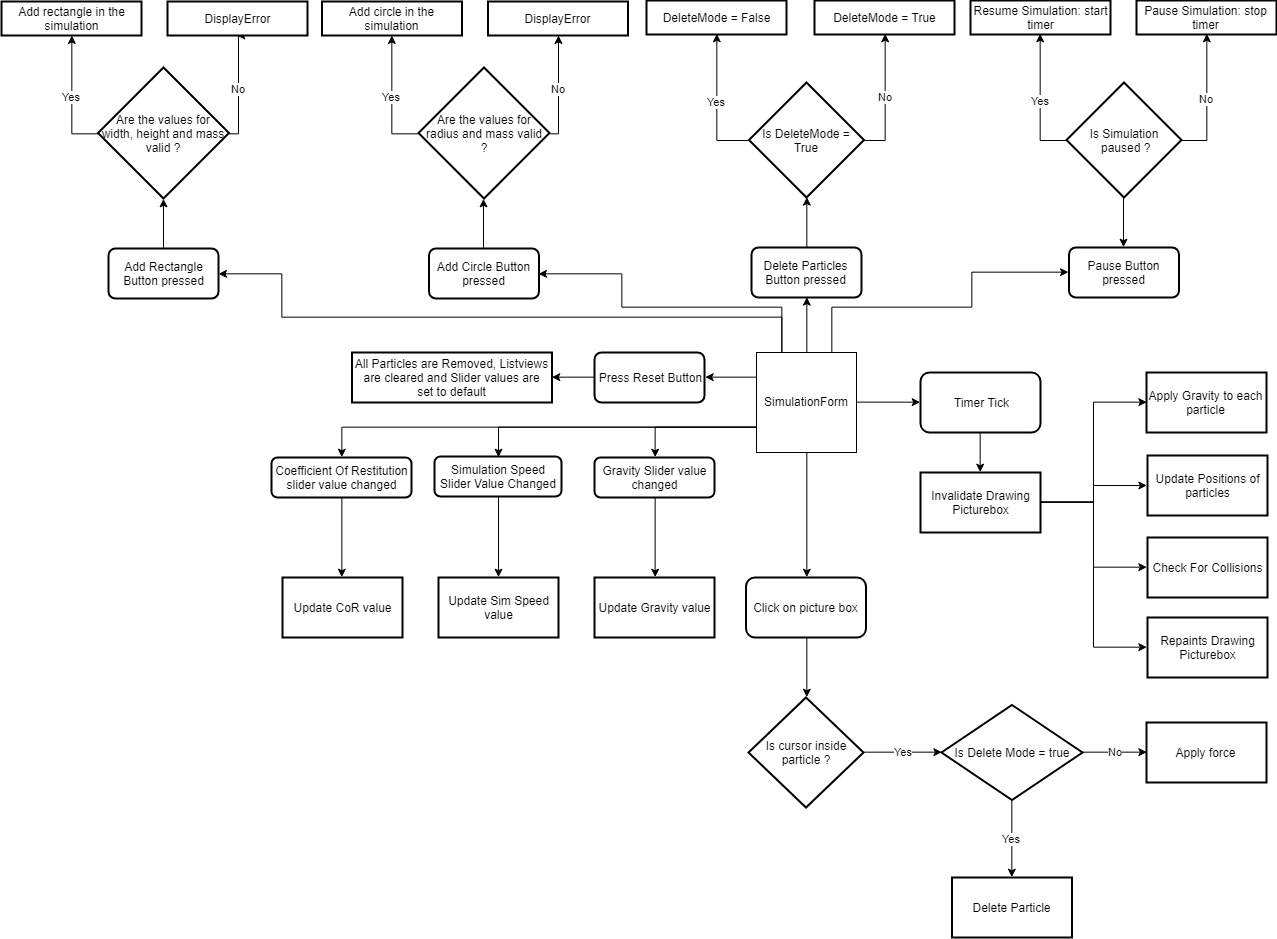
Prototype in action, the two particles at bottom left are colliding while the particle in the middle is falling.

However, as visible from the screenshot of the prototype above, the user interface is neither aesthetically pleasing nor organised, so the last stage of prototyping was to create an organised UI which can act as a foundation for the design stage, where I will design it properly.



Simulation UI prototype

While I was very pleased with my initial design, I knew I still had to add a way to specify the speed of the simulation, the value of acceleration due to gravity and the coefficient of restitution. The easiest and most logical method to adjust these values during runtime seems to be using Visual Basic’s trackbars. However, these were not a priority for a prototype and could be simply implemented later in the development stage as I left plenty of space for them.

**System Flowchart:**

Drawn with Draw.io

The system flowchart above is just a starting point and is likely to change during coding. However, it still provides me with a very basic, yet helpful structure to follow, and improve while I code the engine. The overall program, as visible from the flowchart, relies on the simulator form and the timer tick event. Each tick of the timer, a set of subroutines will run, which would apply gravity to the applicable particles, update their positions and velocities, check for collisions and draw the particles themselves. Everything else would be handled by controls such as buttons and sliders, which when clicked or changed, would do the appropriate action, such as pausing the simulation or changing the value of the CoR.

From my initial prototype,It is reasonable to assume that I would need a seperate class for rectangle particles and circle particles as they are fundamentally different objects. However, there are many attributes that both particles would share, such as positions, mass, colours, velocities and gravity hence, I would need a parent class that both these classes can inherit from. Therefore the list of possible classes that I would need for the program are:

* **CircleParticle**

Will contain information exclusive to the circle particles such as radius and the circle’s paint event.

* **RectangleParticle**

Will contain information exclusive to the rectangle particles such as width and height and the rectangle’s own paint event.

* **Abstract Shape**  
  Will contain information that is shared between both circle and rectangle classes, such as velocity, acceleration, force, mass and position. This class will be abstract as it will ever be instantiated since its sole purpose is to be inherited by sub classes. In VB.net abstract classes have to be declared as MustInherit.
* **Vector**

As explained previously, this class will be a form of data structure. The vector class will have x and y coordinate properties and it will contain mathematical methods such as finding the magnitude, normalising and rotating vectors.

Aside from the classes mentioned above, since I want to create a physics engine rather than single purpose program for my end-user, it should be able to have multiple applications. This can be done by not calculating the physics and collisions in the main Simulatorform as each new application would have a new and different form, so a better choice would be to have a separate class for physics and collissions, which means the engine can be used to create a lot of different programs without the need of recoding the physics. This can be achieved by creating two additional classes:

* **Simulator**

As the name suggests, this class could be responsible for the entire simulation. In a sense this would be the main class that would be called every timer tick. The simulator class will manage all the physics, it will apply and update velocities to particle and it will call the collider class, explained below, to resolve collisions.

* **Collider**The collider class, as the name suggests, will handle all the collisions in the program.

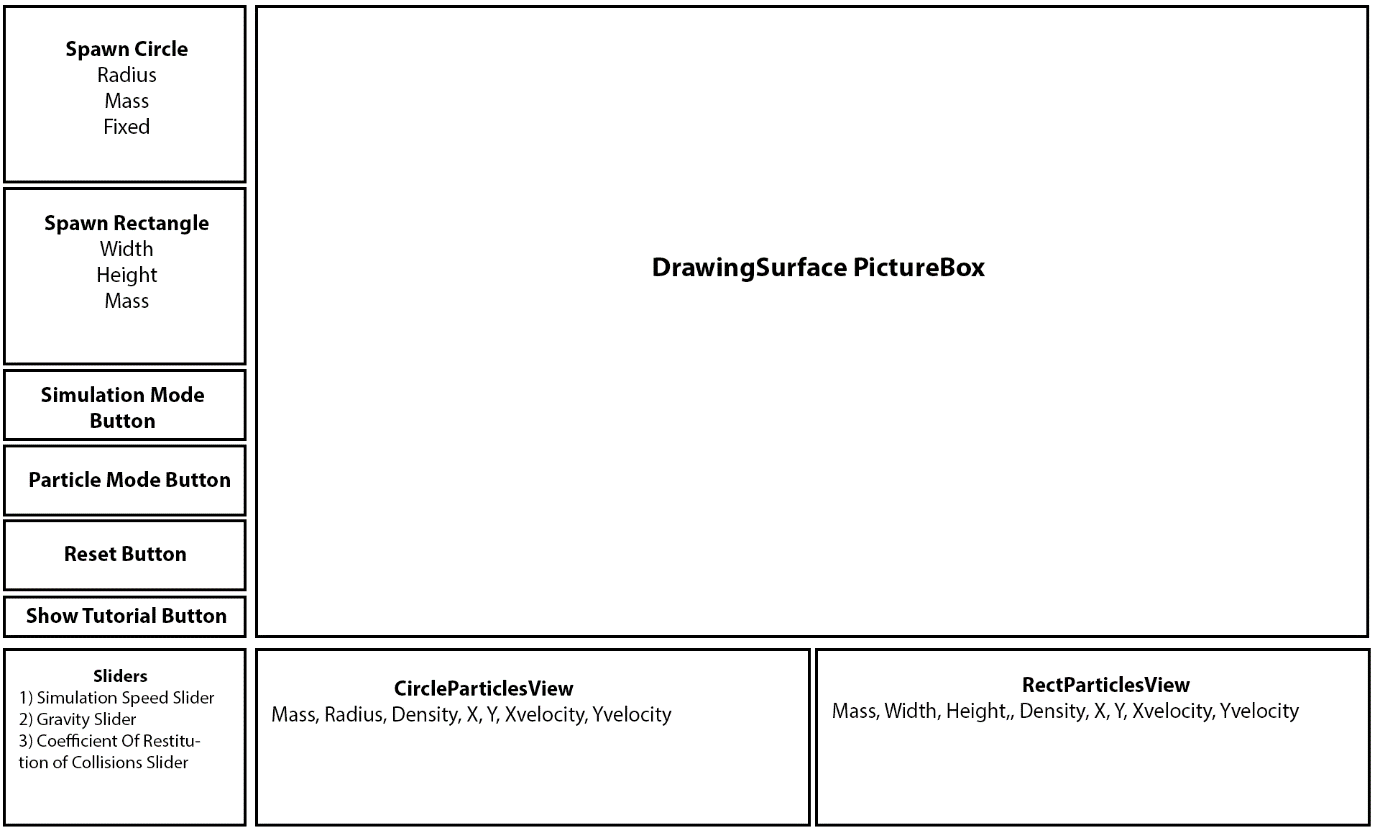
One advantage of creating classes for these separate things is that it keeps the code modular. If in the future, or even during implementation, I needed to make a certain change to a certain aspect of the program, I could just go to the corresponding class and make the changes there without directly affecting the rest of the program. For example, if I wanted to improve the method used for collision detection, I could simply go to the collider class and make improve the algorithms.

Overall, I think this project will be really fun to complete. While there are some challenges ahead of me such as learning how to program collision detection resolution, these would inevitably make me a better program and I look forward to seeing the complete program in action in the end.

**Design**

**Form Design**

**Simulation Form:**



Since the main part of the program is being able to see the simulation and interact with it, the largest component of my simulation form will be the drawing surface itself. The drawing surface will be a picture box because it is an efficient method to paint and repaint objects, as found in the prototype. Furthermore, instead of having a single table to represent all the particles, I decided to create two separate tables for circles and rectangles, which I believe will not only improve the overall experience of the end-user as they will be able to easily differentiate between different particles but will also allow the user to spawn more particles overall.

This is because during implementation, I found out that updating listviews in VB.net is quite glitchy. The two methods I tried were removing all the particles and adding them all again after each timer tick or looping through all the items in the listviews and updating each item. The latter method was not viable, and it created too much lag after a couple of particles were added whereas the former method only created a slight flicker and stopped me from scrolling down, meaning I could only spawn a certain number of particles. After researching the internet[5], I found several methods to “fix” the issue. However, none worked.

Therefore, I arrived on the conclusion that there is not much that I can do to avoid this flicker with VB.net as one of my friend’s tried this same method using C# and had no flicker.

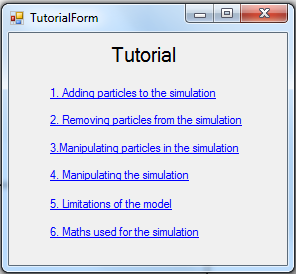
As specified in my objectives, I needed to have buttons to make changes to the simulation, such as pausing and resetting because my end user doesn’t want keyboard shortcuts, hence the Simulation Mode button will change the state of the simulation from paused to running, and vice versa, the particle Mode Button will change the state of the simulation to “delete mode”, so clicking on any particle will delete it, which makes it really easy for the end user to delete any specific particle they want.

The main disadvantage of changing the state of the simulation is that I will have to use global variables for this, as variables can’t be passed into button and mouse click events. However, using an additional global variable is a better and more efficient option than the alternative method, which is to assign a “removable” property to a particle and loop through each particle turning removable to either true or false, as it is much slower to do a loop.

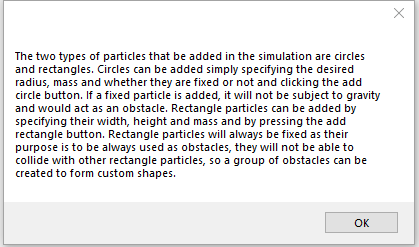
Most of the data for the program will be stored as variables and not constants because my end user wants to be able to model collisions in different scenarios and environments, hence the more variables there are, the more control the end user has.

**Tutorial Form:**

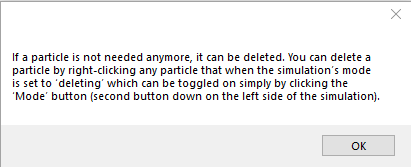
The Tutorial form will consist of 6 LinkLabels representing 6 tutorials. Tutorials numbers 1-5 will be displayed through the use of messageboxes as these will be text-based tutorials without the need for images. However, tutorial number 6 will have its own form created as this will explain the physics equations used in the formula and hence will need to contain images and hyperlinks, which a messagebox cannot do in VB.net.



1. **Adding particles:**

“The two types of particles that be added in the simulation are circles and rectangles. Circles can be added simply specifying the desired radius, mass and whether they are fixed or not and clicking the add circle button. If a fixed particle is added, it will not be subject to gravity and would act as an obstacle. Rectangle particles can be added by specifying their width, height and mass and by pressing the add rectangle button. Rectangle particles will always be fixed as their purpose is to be always used as obstacles, they will not be able to collide with other rectangle particles, so a group of obstacles can be created to form custom shapes.”

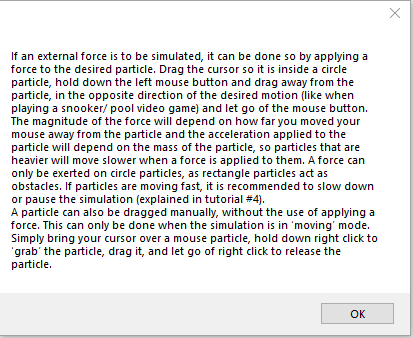
1. **Removing particles:**

“If a particle is not needed anymore, it can be deleted. You can delete a particle by right-clicking any particle that when the simulation’s mode is set to ‘deleting’ which can be toggled on simply by clicking the ‘Mode’ button (second button down on the left side of the simulation).”

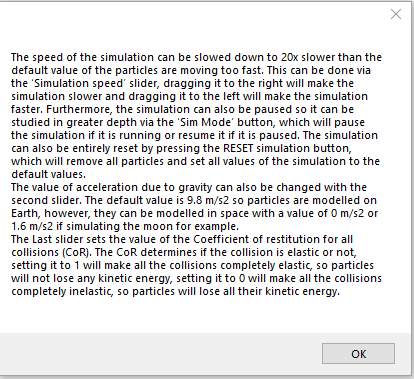
1. **Manipulating particles:**

“If an external force is to be simulated, it can be done so by applying a force to the desired particle. Drag the cursor so it is inside a circle particle, hold down the left mouse button and drag away from the particle, in the opposite direction of the desired motion (like when playing a snooker/ pool video game) and let go of the mouse button. The magnitude of the force will depend on how far you moved your mouse away from the particle and the acceleration applied to the particle will depend on the mass of the particle, so particles that are heavier will move slower when a force is applied to them. A force can only be exerted on circle particles, as rectangle particles act as obstacles. If particles are moving fast, it is recommended to slow down or pause the simulation (explained in tutorial #4).

A particle can also be dragged manually, without the use of applying a force. This can only be done when the simulation is in ‘moving’ mode. Simply bring your cursor over a mouse particle, hold down right click to ‘grab’ the particle, drag it, and let go of right click to release the particle.”



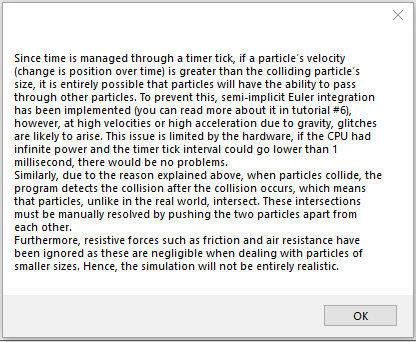
1. **Manipulating the simulation:**

 “The speed of the simulation can be slowed down to 20x slower than the default value of the particles are moving too fast. This can be done via the ‘Simulation speed’ slider, dragging it to the right will make the simulation slower and dragging it to the left will make the simulation faster. Furthermore, the simulation can also be paused so it can be studied in greater depth via the ‘Sim Mode’ button, which will pause the simulation if it is running or resume it if it is paused. The simulation can also be entirely reset by pressing the RESET simulation button, which will remove all particles and set all values of the simulation to the default values. "

The value of acceleration due to gravity can also be changed with the second slider. The default value is 9.8 m/s2 so particles are modelled on Earth, however, they can be modelled in space with a value of 0 m/s2 or 1.6 m/s2 if simulating the moon for example.

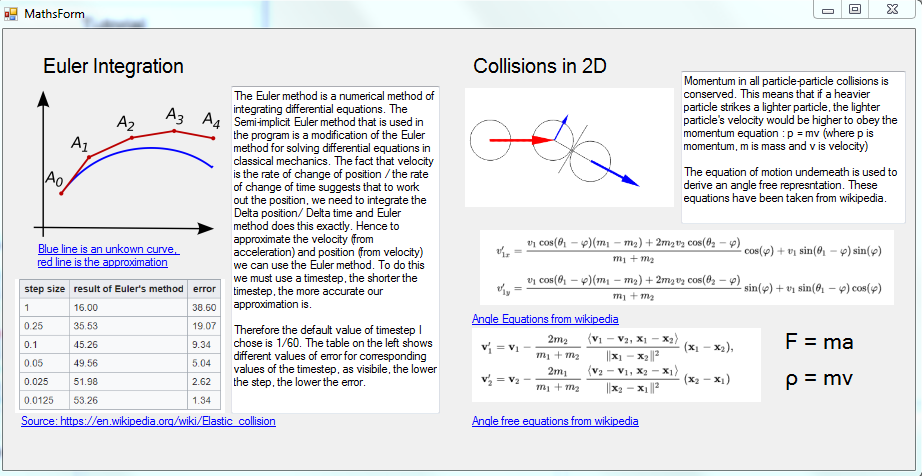
The Last slider sets the value of the Coefficient of restitution for all collisions (CoR). The CoR determines if the collision is elastic or not, setting it to 1 will make all the collisions completely elastic, so particles will not lose any kinetic energy, setting it to 0 will make all the collisions completely inelastic, so particles will lose all their kinetic energy.”

1. **Limitations of the model:**

 “Since time is managed through a timer tick, if a particle’s velocity (change is position over time) is greater than the colliding particle’s size, it is entirely possible that particles will have the ability to pass through other particles. To prevent this, semi-implicit Euler integration has been implemented (you can read more about it in tutorial #6), however, at high velocities or high acceleration due to gravity, glitches are likely to arise. This issue is limited by the hardware, if the CPU had infinite power and the timer tick interval could go lower than 1 millisecond, there would be no problems.

Similarly, due to the reason explained above, when particles collide, the program detects the collision after the collision occurs, which means that particles, unlike in the real world, intersect. These intersections must be manually resolved by pushing the two particles apart from each other.

Furthermore, resistive forces such as friction and air resistance have been ignored as these are negligible when dealing with particles of smaller sizes. Hence, the simulation will not be entirely realistic.”

1. **Maths used for the simulation:**

**Algorithm Design**

It can be argued that the two most important, yet complicated areas of the program are collision detection and resolution as the program will simply not be functional without these features. Hence, the algorithms to achieve these take high priority. Compared to collision resolution, collision detection is much easier to implement as it involves using basic Pythagoras and if statements.

If I was programming PONG, to resolve a collision, since the ball is moving at a constant velocity, once it has collided with a paddle, the X component of velocity is simply multiplied by -1. This same logic cannot be used for my physics engine as particles will inevitably lose kinetic energy, and hence travel at smaller speeds, if the coefficient of restitution is less than 1, which means that once the speed of the particle is small enough, it will start to sink into the other object, as the velocity being inverted will be negligible. Furthermore, if a particle is moving too fast, by the time the program detects a collision, there is a high likely hood that this particle has already penetrated another particle – it will be inside another particle. For this reason, the collision resolution will be split into two parts: **static resolution**, which displaces the colliding particles so there is no longer an intersection and **dynamic resolution**, where the final velocities of the particles are determined.

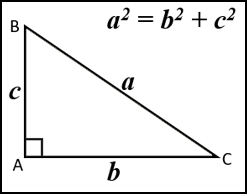
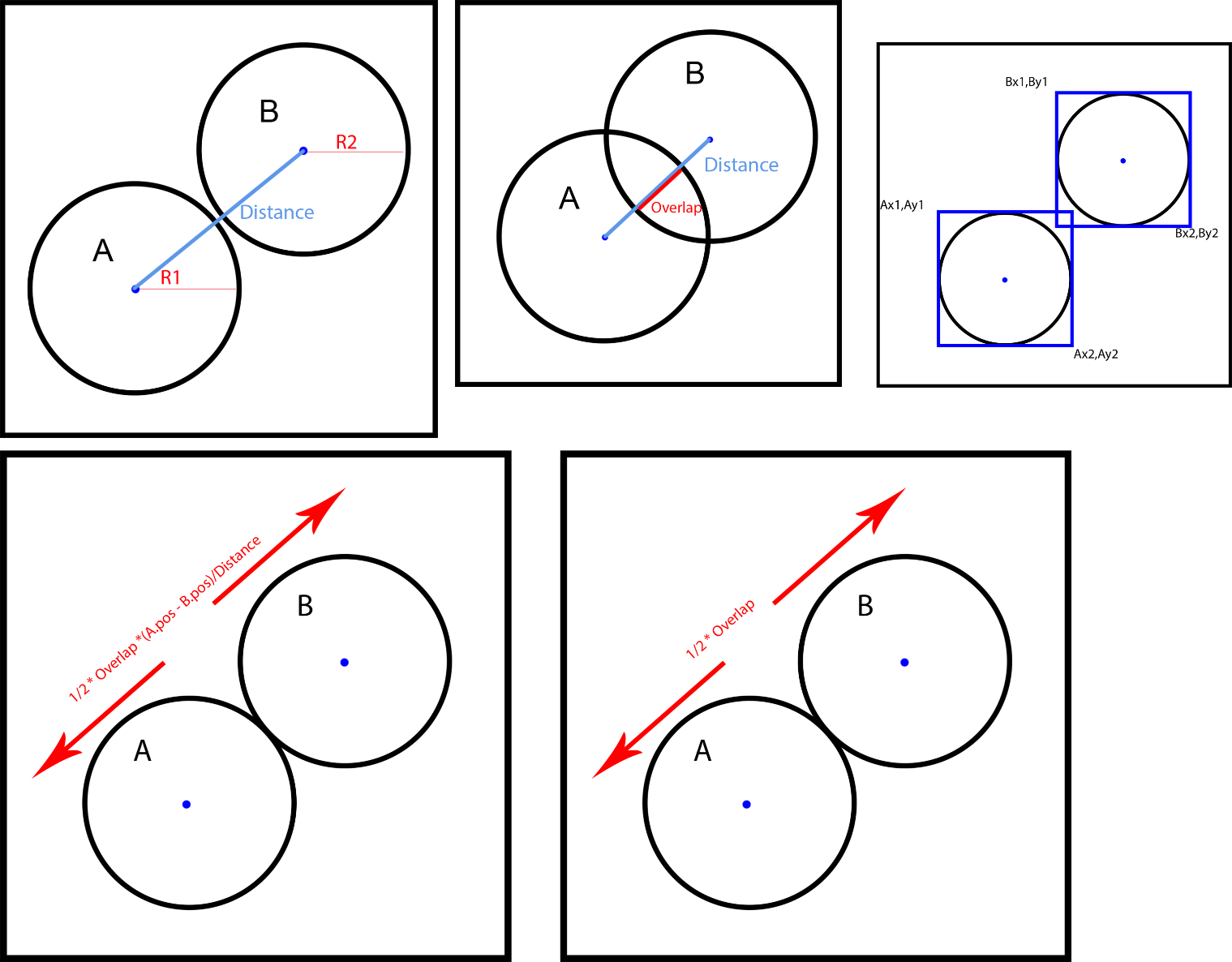
**Circle against Circle**

For collisions involving two circles, all I need to do is get the distance between each circle’s centre point and compare it with the sum of the radii. If the distance is less than the sum of radii, then there is an intersection, hence we resolve the intersection, by moving the two circles apart. The Distance between each centre can be found using Pythagoras Theorem by finding the difference between the Circles’ X and Y points and then square rooting the sum of the squares of said differences.

*Pythagoras’ Theorem*

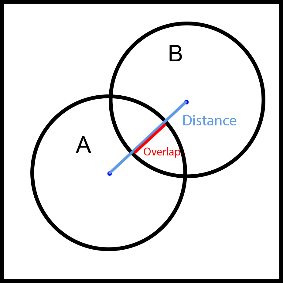
*Comparing distances between circles.*

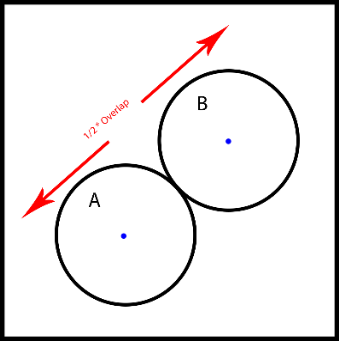
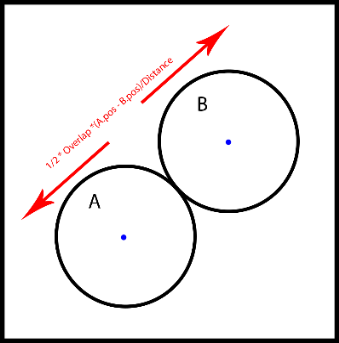
*Pseudo code to detect a collision between two circles*



*Distance = Math.Sqrt((A.x - B.x)2+(A. y – B .y)2)  
RadiiSum = R1+R2  
If Distance < RadiiSum then  
Resolve ()  
End if*

**Static Resolution:**

Once a distance has been obtained, the Overlap between A and B can be found be Distance – RadiiSum. My original idea was to create the Overlap as a vector and move apart from each circle by half the penetration depth. However, this resulted in a rather “glitchy” resolution as B would well inside A before the program moved them apart, which would result in an inaccurate simulation as particles should not be seen intersecting. While researching how to do the dynamic resolution for circle to circle collisions, I stumbled upon a video on YouTube [6] in which the programmer: *OneLoneCoder/ javidx9* used a method very similar to mine for static resolution, however, he normalised the overlap vector, producing a much smoother and accurate result.

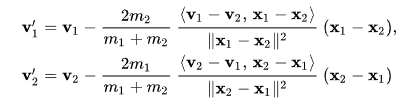


Left: My method

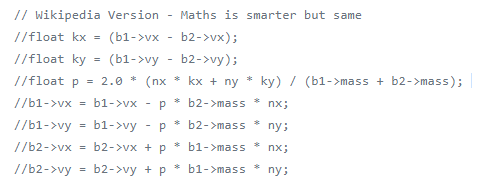
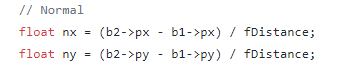
Right: javidx9’s Method

**Dynamic Resolution**

As mentioned above, I found a method for the dynamic resolution in a YouTube video. I decided to use this method as deriving my own equations would not only be very difficult for me, as I do not understand the mathematics involved and it would be pointless because the equations I need already exist. In fact, the YouTuber Javidx9 himself equations from a page about elastic collisions on Wikipedia. As I neither understand the mathematics nor the derivation for these formulas, I will not try to explain them.



[7] Equations on Wikipedia: https://en.wikipedia.org/wiki/Elastic\_collision



[8] OneLoneCoder's Github: https://github.com/OneLoneCoder/videos/blob/master/OneLoneCoder\_Balls1.cpp

I decided to use his Wikipedia Version instead of the equations he derived himself as these are simpler, shorter and uses ‘smarter’ maths. Hence, using the reasoning above, the algorithm can be written as:

I used DistanceSquared here to do a collision check as it is argued by many programmers that Math.Sqrt is an inherently slow and inefficient method as it uses an iterative method, such as the Newton-Raphson method that we learn in A-Level Mathematics. Upon doing some research on the internet, the opinion on this topic is evenly split. Therefore, just to be on the safe side, I would only compute the square root IF a collision had occurred, making it so that the function is not used every frame.

*DistanceSquared = (A.centre.x - B..centre.x)2+(A..centre.y – B .centre.y)2  
RadiiSum = R1+R2****If*** *DistanceSquared < RadiiSum2* ***then****Distance = Math.Sqrt((A.centre.x - B..centre.x)2+(A..centre.y – B .centre.y)2)**Overlap = (Distance - RadiiSum) \* 0.5****//Static Resolution****A.position -= ((A. centre – B. centre) / Distance) \* Overlap  
B.position += ((A. centre – B. centre) / Distance) \* Overlap****//Dynamic Resolution****Normal as vector = getNormal(A.centre, B centre)  
k as vector = A.velocity – B.velocity  
p = 2 \* ((normal.x \* k.x) + (normal.y \* k.y)) / (A.mass + B.mass)  
A.velocity.x -= p \* B.Mass \* normal.x \* CoefficientOfRestitution  
A.velocity.y -= p \* B.Mass \* normal.y \* CoefficientOfRestitution*

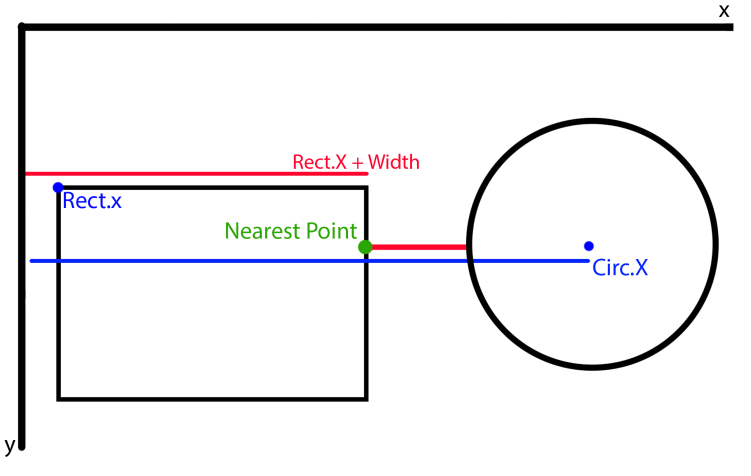
*B.velocity.x += p \* A.Mass \* normal.x \* CoefficientOfRestitution*

*B.velocity.y += p \* A.Mass \* normal.y \* CoefficientOfRestitution****End if***

**Circle against Rectangle**  
My original strategy to resolve circle to rectangle collisions was to model circles as rectangles and then resolve a rectangle to rectangle collisions. I wanted to do this as it simplified the collision detection and resolution, removing much of the mathematics that would have been used otherwise. However, the problem with this method is that if I model a circle as a rectangle during the collision, some collisions might occur at corners that really shouldn’t have, leading to inaccurate results. Furthermore, this method would mostly consist of if statements, one for each corner of the rectangle, which will lead to an inefficient algorithm.

However, after spending some time, mainly on paper, I figured out a method to detect a collision between a circle and a rectangle. The basic principle of this method is to use a point to circle collision method, the point being the closest point on the edges of the rectangle to the circle.

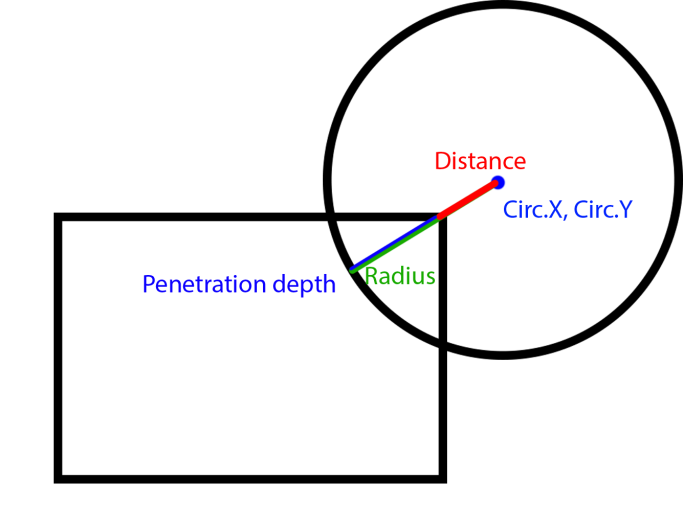
Diagram showing the nearest point



As visible on the diagram to the left, getting the smaller value between the Circle’s X position and the rectangle’s X position plus will yield us the x position of the nearest point on the rectangle. However, in case the circle was to the left of the rectangle, I need to compare this point with the rectangle’s X position and get the biggest value, which will still be the correct nearest point. As Rect.X+ Width < Circ.X, it will be our value we compare with Rect.X, in which case Rect.X+Width > Rect.X, therefore the X value for nearest point is Rect.X + Width. This same logic can be applied to find the Y value of the nearest point. Note that the origin of this graph starts from the top left; this is because VB.Net treats the top left point to be the origin, where going downwards increases the Y value and going to the right increases the X value. Using the logic described above, I arrived at the following pseudo code:

Pseudo code for calculating the Nearest Point

*ClosestX = Math.Max(Rect.X, Math.Min(CircleX, RectX + RectWidth))  
ClosestY = Math.Max(Rect.Y, Math.Min(CircleY, RectY + RectHeight))*

After obtaining the closest point, I can simply find the difference between the closest point and the circle’s centre and create a distance vector. Using Pythagoras’ Theorem on this vector will calculate the actual distance between the nearest point and the centre of the circle which can be compared with the radius of the circle to determine if a collision has occurred or not. After some research on the internet, I found that this is, in fact, a standard method for detecting collisions between circles and rectangles, which means this algorithm Can be used in the future if I decided to include rotation.

Calculating the penetration depth

**Static Resolution:**

The static resolution is very simple. As I already have the distance between the closest point and the circle’s centre, a Penetration Depth, much like the circle to circle method, can be easily obtained by subtracting this distance from the circle’s radius as seen from the diagram to the left. Using the same logic as the static response for circle to circle static resolution, normalising the distance as a vector would give the vector a magnitude of 1, but would specify the direction the ball needs to be displaced so multiplying the normalised distance to the penetration depth would give a vector that can just be added to the circle’s position. As pseudocode, the static resolution will be something like:

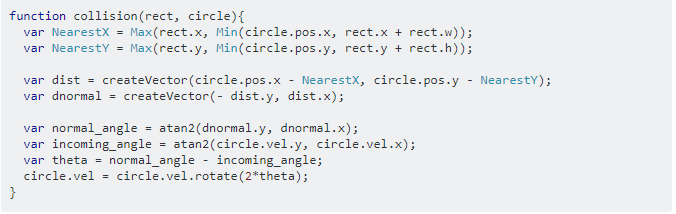
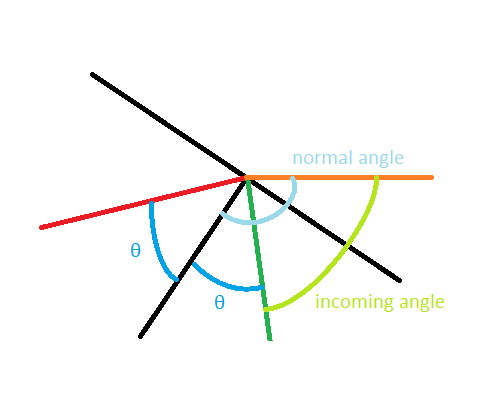
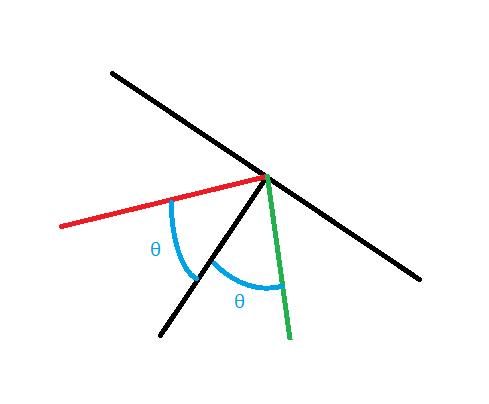
Pseudocode for displacing the circle (static resolution)

*Distance as vector = (Circ.X –ClosestX, Circ.Y – ClosestY)  
PenetrationDepth = Circ.Radius – Distance.Magnitude  
DisplacementVector as vector = Distance.normalise \* PenetrationDepth  
Circ.position + = DisplacementVector*

**Dynamic Resolution:**

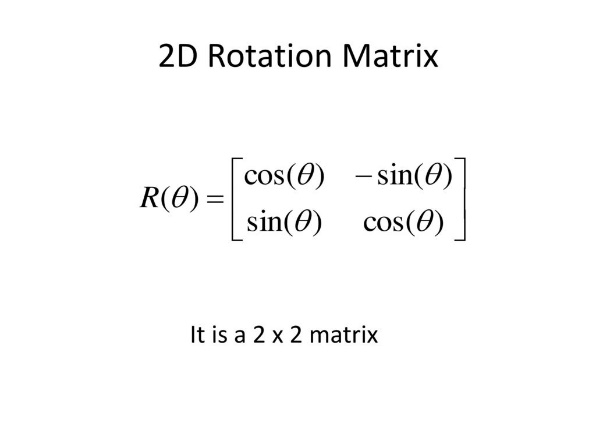
While the dynamic resolution can be done simply by determining where on the rectangle the collision happened and inverting the velocities accordingly, for example, a collision on the top or bottom means multiplying the Y-velocity by the minus coefficient of restitution, this method would inefficient as it would contain a lot of If-statements. Instead, I found a more mathematical and efficient method on stack overflow [7]. While I do not completely understand the mathematics involved here, as it's beyond the specification for A-Level Mathematics and Physics (I most likely would not have been able to derive this), I do have some understanding about how it works. For the purpose of documentation and to further my own understanding, I will try my best to explain the solution here:

If a circle collides at a plane, such as the side of a rectangle, the angle of incidence will be the same as the angle of reflection, this is simple enough. We call this angle θ  
If we find the normal to the distance vector, calculated above, using the atan2 function on this normal will calculate an angle from the normal to the perpendicular of the plane, called the normal angle. The atan2 function can also be used on the circle’s velocity to work out the incoming angle, subtracting the incoming angle from the normal angle will give us θ. Now that the difficult part is done, rotating the circle’s velocity by 2 \* θ, well give us the final trajectory of the ball, as seen from the diagram.



[7] = https://stackoverflow.com/questions/45370692/circle-rectangle-collision-response

As VB.net does not have a vector class, thus it does not have a rotate function, I will have to rotate the velocity manually. This is very simple as we can just multiply the velocity by the rotation matrix, where θ will be equal 2θ.

This is the rotation matrix for 2 dimensions. Hence, the new velocity will equal the old velocity \* rotation matrix, which can also be written as the following pseudocode:

Pseudocode rotating the original velocity by 2θ

*Circ.velocity = (velocity.x \* Cos(2θ* *) – velocity.y\*Sin(2θ), velocity.x\*Sin(2θ) + velocity.y\*Cos(2θ))*

Hence, using the reasoning above, the algorithm can be written as:

*ClosestX = Math.Max(Rect.X, Math.Min(Circle.centre.X, Rect.X + RectWidth))  
ClosestY = Math.Max(Rect.Y, Math.Min(Circle.centre.Y, Rect.Y + RectHeight))  
Distance as Vector = new Vector (Circle.centre.X –ClosestX, Circle.centre.Y – ClosestY)*

***If*** *Magnitude (Distance) < Circle.radius* ***then*** *CollisionDepth = Circle.radius – Magnitude (Distance)  
DisplacementVector as Vector = new Vector (Normalise (Distance) \* CollisionDepth)****//Static Resolution*** *Circle. Position += DisplacementVector****//Dynamic Resolution*** *NormalOfDistance as Vector = new Vector(-distance.y, Distance.x)  
NormalAngle = Math.Atan2(NormalOfDistance.y, NormalOfDistance.x)  
Incoming Angle = Math.Atan2(Circle.velocity.y, Circle.velocity.x)  
θ = NormalAngle – IncomingAngle  
Circle.velocity = (Circle.velocity.x \* Cos(2θ* *) – Circle.velocity.y\*Sin(2θ), Circle.velocity.x\*Sin(2θ) + Circle.velocity.y\*Cos(2θ)) \*CoR****End if***

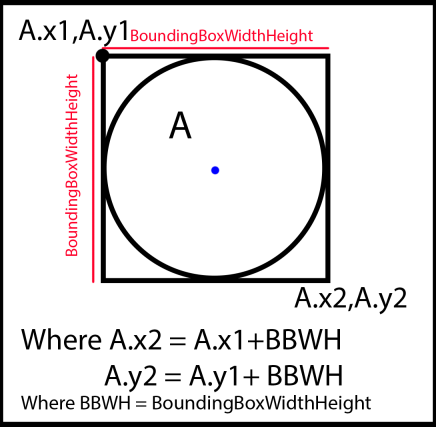
However, during implementation, it was found that simply multiplying the Circ.velocity at the end by the CoR would produce an inaccurate result when pushing a circle particle while it is resting atop a rectangle. This is because when the circle particle is resting on top of a rectangle particle, it is always colliding, hence its velocity is always being multiplied by the CoR, any CoR that is less than 1 will mean that a the force being applied to the circle particle would have negligible effect. To solve this simple if statement could be used like this:

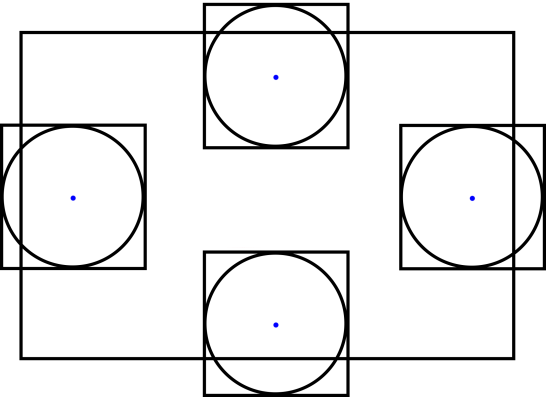
*Circle.velocity = (Circle.velocity.x \* Cos(2θ* *) – Circle.velocity.y\*Sin(2θ), Circle.velocity.x\*Sin(2θ) + Circle.velocity.y\*Cos(2θ))****if*** *Circle.X < Rect.X or Circle..X + Circle.Radius > Rect..X + Rect.Width* ***Then*** *Circle.velocity.X \*= CoR  
Else  
Circle.velocity.X \*= CoR****End if***

This is statement checks whether the circle is colliding with the horizontal or vertical planes of the rectangle, if it is colliding with the vertical plane, then only the horizontal velocity of the circle should be affected and vice versa. This ensures a smooth “bounce” but in the case of a circle resting on top of a rectangle, since the circle is colliding with the rectangle’s horizontal plane, only its vertical velocity gets multiplied by the CoR, hence it can be pushed across the rectangle.

**Boundary Collisions:**

This is the last type of collision detection algorithm required, and arguably the easiest. The boundary detection algorithm will only apply to circle particles; as rectangles will act as potential walls/ obstacles, there will be no need to include them in. However, rectangles can be added to the boundary collision algorithm as it will apply the same logic and will only need a couple of if statements.

To develop a relatively easy, yet accurate algorithm for boundary detection, the circles will be modelled as rectangles. This is because VB.net already treats circles as rectangles in the first place, as the coordinates specified when drawing a circle are not of the circle’s centre, but instead of the top left point as shown in the diagram to the left. Where A.x1 and A.y1 are the coordinates specified when drawing a circle. Furthermore, modelling this collision as a rectangle to rectangle collision will not decrease the accuracy of the simulation as using the circle to point collision method, explained above, will theoretically result in the same collisions

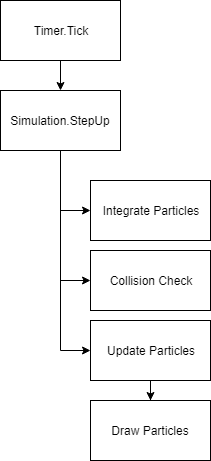
As there are 4 sides of the picturebox (drawing surface) where a collision might occur, there will be four if statement checks to determine a collision, one for each side of the picturebox.

Hence if a point on the leftmost side of the circle’s bounding box goes beyond the picture box’s leftmost point a collision has occurred. Applying this logic to all the sides of the picturebox will yield us with an algorithm such as:

I have included the static and dynamic resolutions in the algorithm to the left as the basic principle of these have already been explained in great detail previously. However, a collision depth is not needed in this case as we already know the positions of the edges of the picturebox so setting the position of the circle so that it is not colliding anymore will work. Furthermore, if a collision happens with the sides, then only the x component of velocity is inverted, multiplying this by the Coefficient of Restitution will give the actual final velocity; the same logic is applied to the y velocities.

*If Circle.x < picturebox.x  
//collision  
Circle.x = picturebox.x  
Circle.Velocity.x \*= - CoefficientOfRestitution  
elsseif circle.x +circle.BoundingBoxWidthHeight > Picturebox.x + picturebox.width  
//collision  
Circle.x = picturebox.x + picturebox.width – Circle.BoundingBoxWidthHeight  
Circle.Velocity.x \*= - CoefficientOfRestitution  
elsseif circle.y < Picturebox.y  
//collision  
Circle.y = picturebox.y  
Circle.Velocity.y \*= - CoefficientOfRestitution  
elsseif circle.y +circle.BoundingBoxWidthHeight > Picturebox.y + picturebox.height  
//collision  
Circle.y = picturebox.y + picturebox.height – circle.BoundingBoxWidthHeight  
Circle.Velocity.y \*= -CoefficientOfRestitution  
end if*

**Simulation Loop:**

As will any simulation or game, there is always the main loop which updates everything during the game or simulation. While formally there will not be an actual loop statement in my program, a set of procedures will be run every tick of the timer. These will consist of a Draw subroutine, which will handle all the drawing in the simulation, a collision check subroutine an integration subroutine and an update simulation. The integration subroutine will be used to implement a numerical method to find out the velocities and positions in a realistic matter, which obeys the SUVAT equations of motion.

An article by Glenn Fiedler on gafferongames.com [13] explains that a semi-implicit Euler integrator is the integrator used by most commercial game physics engines. While the purpose of my physics engine is to be used in the classroom and not for a video game specifically, it will still provide with realistic results, hence why I will use a semi-implicit Euler integrator instead of other methods such as Verlet integration or the RK4 method.

Diagram of the intended simulation loop

The semi-implicit Euler can be simply implemented when updating the velocities and positions of the particles. Instead of updating them the normal way:

*Circ.velocity = Circ.velocity + Circ.acceleration  
Circ.position = Circ.position + circ.velocity*

Conventional way of updating velocity and position

The acceleration and velocity will be multiplied by a timestep

*Circ.velocity = Circ.velocity + Circ.acceleration \* timestep  
Circ.position = Circ.position + circ.velocity \* timestep*

Implementing semi-implicit Euler

Another advantage of implementing integration in my program is that it allows me to meet one of my objectives rather efficiently: being able to change the speed of the simulation. Since the timestep mentioned above will be set by myself (depending on which values create a good motion), the timestep can also be changed through the use of a slider. Decreasing the value of the timestep would mean the velocities and positions would change at a smaller rate, slowing down the simulation. This is more efficient than increasing the interval for the timer tick as while that will slow down the simulation, the motion will not be smooth and would stutter unreasonably (at really high values of the interval).

**The final algorithm I ended up for the simulation loop is**:  
**primary purpose**: step up the simulation by updating all aspects of the simulation

***Sub*** *StepUp****For*** *i = o to particles.count -1  
Integrate(particles(i)))  
CollisionCheck(particles(i))  
particles(i).update  
collider.BoundaryDetection(particles(i))****Next  
End Sub***

The algorithm loops through each particle and integrates its velocity and position, performs collision checks and updates the particle. There is an additional collision detection check, this time only the boundary detection test, as a bug was discovered during coding where particles would sometimes just move out the boundaries when they shouldn’t be, this solves that issue.

**The integrate Sub**: (logic of semi-implicit Euler explained above)  
**primary purpose**: apply semi-implicit Euler integration to particles

***Sub*** *Integrate (abstractshape)****if*** *tempparticle.BeingDragged = false and tempparticle.fixed = false* ***then*** *Circ.velocity = Circ.velocity + Circ.acceleration \* GetDT  
Circ.position = Circ.position + circ.velocity \* GetDT****End if******End Sub***

The if statement is used above to ensure that only particles that have physics enabled on them, such as particles that are not fixed and are not being manually moved, are integrated because particles that are being moved or are fixed would have velocities = 0 and integrating this is just a waste of resources.

**The GetDT function  
primary purpose**: Get a value of the timestep to be used for the integration

***Function*** *GetDT****if*** *simform.startBtn.text = “Sim mode: Paused”* ***then  
return*** *0****Elseif*** *simspeedslider.value <> 60* ***then******return*** *(1/ simspeedslider.value) \* 10****Else******return*** *1/6****End function***

A value of 0 is returned when the simulation is paused as it means that the velocities of the particles will be multiplied by 0, essentially making it so particles do not obey physics, yet can be dragged around manually. A default value of 1/6 is used as it was determined through experimentation and it seemed to provide the best and most realistic default speed.

**The collision check sub:  
primary purpose**: Call the appropriate methods in the collider class with appropriate arguments

***Sub*** *CollisionCheck abstractshape)  
Collider.BoundaryDetection(tempparticle)****For*** *j = 0 to particles.count – 1****if*** *temparticle* ***isnot*** *particles(j)  
Collider.DetermineCollision(temparticle, particles(j))****End if******Next******End Sub***

As mentioned above, boundary detection must happen twice to avoid the bug where particles move out of the boundaries. Since this sub contains a loop and is called in a loop (StepUp), the overall algorithm is O(n2) as it is a loop inside a loop. The reason why it has been split into two different subroutines is make sure that subroutines are single purpose: the purpose of the StepUp sub is to update the entire simulation, by calling different subs and the purpose of CollisionCheck is to call the appropriate methods in the collider class. Furthermore, the if statement being used here is very important as it make sure that the same particle is not being compared by itself, which would cause a lot of bugs.

**The DetermineCollision sub:  
primary purpose:** Pass appropriate particles to appropriate collision detection subroutines.

***Sub*** *DetermineCollision (A as abstractshape,B as abstractshape)****If******TypeOf*** *A is CircleParticle and* ***TypeOf*** *B is CircleParticle* ***Then*** *CircleVsCircle(A,B)****ElseIf******TypeOf*** *A is RectangleParticle and* ***TypeOf*** *B is CircleParticle* ***Then*** *AABBvsCircle(A,B)****ElseIf******TypeOf*** *A is CircleParticle and* ***TypeOf*** *B is RectangleParticle* ***Then*** *AABBvsCircle(B,A)****End Sub***

As visible from the algorithm above, all this subroutine does is it passes the correct types of the shapes to the corresponding collision detection sub routines. The algorithms for CircleVsCircle and AABBvsCircle have been explained earlier. An if statement for AABBvsAABB can also be added in the future if rectangle to rectangle collisions are to be implemented.

**Generating Random Colours:**

As my end user has asked that the speeds of particles are to be visible and compared before and after collisions, I would need to have some visual representation of velocity, through arrows, and a numeric reorientation, through tables. However, I cannot have all particles looking the same, especially ones of the same type and size. Hence, I decided to assign particles a random colour when they are first created, this colour can be used to draw their outlines, making them different to other colours. Therefore, the algorithm to randomize colours in the program is as follows:

***Function GetRandomColor ()****Randomzie()  
Dim RandomColor As Random = New Random()****Return*** *Color.FromArgb(RandomColor.Next(0, 256), RandomColor.Next(0, 256), RandomColor.Next(0, 256))****End Function***

Randomize () is used at the start of the function to ensure the program uses a new seed when generating a random number, otherwise it will use the same seed and it might generate predictable numbers, hence not random. As the rest of the code is really self-explanatory, this algorithm is very simple, but it gets the job done.

One problem I found during implementation was that there are some rare occasions when a colour that is generated is too bright to be seen against the background. The reason why I did not fix this issue is because it seems like this is not a very common occurrence and the solution to the problem that I had found was rather long and complicated (explained in further detail in Evaluation); hence, I saw now real advantage of spending a lot of time fixing a very minute issue when I could be spending the same time developing my collision detection algorithms, one of the most important parts of the program.

**Validation and Exception Handling**

As my program does not rely on imputing a lot of data, the validation is quite minimal yet necessary. The only validation to avoid errors I would need is when creating particles, that is when a particle’s dimensions or mass set to 0 or a particle is created while another is being created already.

When spawning a particle, a Boolean value called spawning can be set to true until the particle is dropped into the desired position, through a mouse click event, where spawning can be set to false. A set if statements are then used to check if spawning is true or not, if spawning is true, then a message box is displayed asking the user to spawn their current particle first, otherwise, a check on the input data can be performed to see if they are zero or not, if they are zero a message box can be shown to the user asking them to enter valid values. The pseudo for the validation is as follows:

***Sub*** *AddCircleBtn\_Click()****if*** *InputRadius > 0 And InputMass > 0* ***Then******if*** *spawning = false  
 particles.add(circleparticle)   
 Spawning = true* ***b*** *messagebox.show(“Please spawn your current object first!”)* ***end if  
else*** *messagebox.show(“Please enter a suitable value for the mass and radius!”)****End Sub***

The same algorithm is used for the add rectangle button except the message box at the end shows “Please enter a suitable value for the mass, width and height!”.

It can be argued that a physics simulator should have lots of validation to avoid bugs and inaccurate physics. However, avoiding bugs or exceptions do not rely on try statements in my program, they rely on the equations being used, if all mathematics is implemented correctly, there should be no crashes. During coding, I faced a lot of exceptions and crashes, however most of these, if not all, have been fixed by tweaking algorithms.

**CLASS DESIGN**

This section aims to build upon the initial ideas for classes I had in the analysis. I decided to use all the classes I thought of in my analysis as I think they create and appropriate structure of the program and I did not add any additional classes as I didn’t feel the need for them.

**Circle, Rectangle and AbstractShape:**

Overall, I know that I will need to create two separate classes for circles and rectangles. This is because the two shapes are fundamentally different from each other, but in terms of their properties for the program, the key difference is that a circle will have a radius and a rectangle will have width and height. Furthermore, instead of having the particle being drawn in the form class, I have decided to draw them in their own classes because the physical appearance of a particle is its own property, so it should be in its own class. This means that in the picturebox’s paint methods, I will have to pass to PaintEventArgs to the circle/ rectangle class at one point. Since both a circle and rectangle are a type of shape, and both will have similar properties such as position, velocity and acceleration, I will need to create a separate abstract class that will be inherited by the circle and rectangle particles classes.

**Vector:**

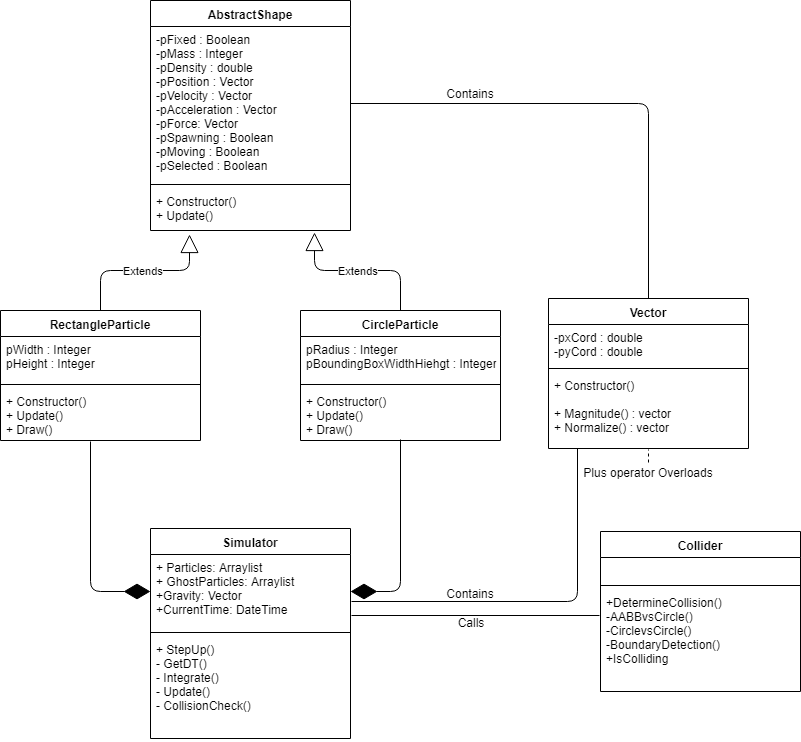
Furthermore, as VB.net does not have its own vector class, I will have to create my own vector class which is a good thing as it would give me another opportunity to demonstrate my coding skills. The vector class should mainly consist of two properties: the x coordinate and the y coordinate. However, additional methods, mainly ones involving vector maths that I plan to use in my algorithms as stated previously will also need to be added. These methods include magnitude, normalise, rotate and magnitude Squared. Furthermore, a vector should have some operators attached to it such as addition and subtraction, so my class will also contain operator overloads.

**Simulator:**

Since I aim to keep my program as modular as possible, considering it is a physics engine and it should have multiple applications, I will create a simulator class which will act as the backbone of all simulation in the program. The alternative approach would be to do all the simulation in the form class, however, if a user decided to create a different form for another application of the engine, they would need to rewrite the entire code for simulating particles which is neither programmer friendly nor modular. Hence the simulator class should be responsible for stepping up the simulation, applying velocities, accelerations or forces to particles and updating the particles’ positions.

**Collider:**

The reason why I didn’t include collisions in my reasoning above is that in my opinion collisions are not part of the physics update because the only part of a collision that would be actually using physics equations would the dynamic resolution, which is not the majority of the code. Therefore, to maintain cohesion, it can be argued that a collision is an inherent property of a particle. However, if another class called collider was to exist, then a collision would also be a property of the collider class. Hence, to keep my program modular, I have decided to handle all collisions in a separate collider class. This means that if another programmer wanted to make alterations to the way the collisions work, instead of finding the appropriate methods in the particle classes, they could just look at the collider class, which will manage all collisions. As mentioned above, to avoid simulation in the form class, the simulator class will also have a method that calls the collider class.

**UML DIAGRAM**

Drawn with Draw.io

As visible on the UML class diagram above, the main class of the program is the simulator. This is where the physics updates occur, and this is also where the particles are drawn. I did not choose the Form class to be the main class as I want the form class to only act as an interface between the user and the physics engine. Making the form class as the main class will only reduce the usability of the physcs engine as creating an application that makes use of the engine would require one to alter the physics engine itself rather than just creating a new form class and using the engine in that class.

As the Simulator class contains vectors and calls the collider class, it has a simple association with both the collider and vector classes, indicated by the solid lines. Furthermore, the clear arrows with “Extends” written across the lines show that both the CircleParticle and RectangleParticle inherit the AbstractShape class.

The RectangleParticle and CircleParticle classes will both have an ‘update’ method although their only purpose will be to call the update method of the AbstractShape class. However, I have still included them in my code as it allows me, or any other programmer, to implement more functionality to either of the classes in the future. Considering I am creating a physics engine, upgradability and being able to have different applications is important. One such example is that if I was to implement moments and rotation in rectangles, I would need to store all 4 corner points of the rectangle in an array, to allow efficient rotation, which would need to be updated every frame as the 3 other corner points added will rely on the top left corner (currently called position) + width or height.

Simulator, Collider and AbstractShape are all abstract classes as they will be declared as MustInherit which means they can’t be instantiated. This will improve performance and save memory, especially for the AbstractShape class.

The arrows with the dark diamonds indicate composition between the classes. If Simulator dies, then all the particles in the Arraylist will also die. Moreover, the “p” prefix is used for encapsulated data, each variable will have its own corresponding public property so other classes can access these values.

**CURRENT CLASSES:**

The following tables show the current class structure of the program, these classes were created with the class design section (explained above) in mind.

**Vector Class**

**Fields:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| pXcord | Double | To store the X-coordinate of the vector object | private | 12.00 |
| pYcord | Double | To store the Y-coordinate of the vector object | private | 12.00 |

**Properties:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Property Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| Xcord | Double | To **get** and **set**  the X-coordinate of the vector object | Public | 12.00 |
| Ycord | Double | To **get** and **set**  the Y-coordinate of the vector object | Public | 12.00 |

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| Rotate | Vector, Double | Vector | To apply a rotational matrix with value of theta = the double passed in on the vector passed in | Public |
| Magnitude | Vector, Vector | Double | To find the distance between the two vectors passed in using Pythagoras’ Theorem | Public |
| Magnitude | Vector | Double | To find the magnitude (size) of the vector | Public |
| MagnitudeSquared | Vector, Vector | Double | To find the distance2 between two vectors passed in. This is used to avoid using the Sqrt function unless needed. | Public |
| MagnitudeSquared | Vector | Double | To find the size2 of the vector. This is used to avoid using the Sqrt function unless needed. | Public |
| Normalise | Vector | Vector | Finds a vector of magnitude 1 but same direction as the argument vector. | Public |
| Normalise | Vector, Vector | Vector | Finds a vector of magnitude 1 but same direction as the argument vectors. | Public |

**Operator overloads:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| + | Vector, Vector | Vector | To add one vector to another | Public |
| - | Vector, Vector | Vector | To subtract one vector from another | Public |
| / | Vector, Double | Vector | To divide a vector by a scalar quantity | Public |
| \* | Vector, Double | Vector | To multiply a vector by a scalar quantity | Public |

Operator overloading is very important for the vector class as a vector is a data type which will be involved in a lot of calculations in my program. Furthermore, I implemented overloading as an example of polymorphism where different operators will have implementations depending on their arguments. While not all operators are frequently used in my program, creating them will be helpful if I, or another programmer, decided to expand this project in the future. The \* operator would need two implementations to ensure I stick to defensive programing, one if the values passed in are Vector, Double and the other if the values passed in are Double, Vector. While it can be argued that the programmer (me in this case) only passes in values in a certain format, such as passing the vector first and then the double, this is likely to cause bugs if the double is passed first as keeping track of what order the arguments need to be passed in can get confusing when coding mathematical equations.

**AbstractShape Class:**

**Fields:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| pFixed | Boolean | Holds whether or not the particle is fixed or not, if it is fixed, velocities will not be applied to it. | Private | True |
| pMass | Integer | Holds the Mass of the particle. | Private | 12 |
| pColor | Color | Holds the line colour of the particle. | Private | Red |
| pDensity | double | Holds the density of the particle. Density = Mass / Volume (Area in 2D space) | Private | 10.09 |
| pPosition | Vector | Holds the position of the top left corner of the shape. | Private | (10.00,12.00) |
| pForce | Vector | Holds the force applied to the particle | Private | (10.00,12.00) |
| pAcceleration | Vector | Holds the acceleration of the particle. | Private | (10.00,12.00) |
| pVelocity | Vector | Holds the velocity of the particle | Private | (10.00,12.00) |
| pSpawning | Boolean | Holds whether or not the particle is being spawned or not | Private | True |
| pBeingPushed | Boolean | Holds whether the particle is being applied a force with the mouse cursor or not. | Private | False |
| pDragged | Boolean | Holds whether the particle is being dragged by the mouse cursor or not. | Private | False |

**Properties:**

Each field listed above has its own corresponding properties. However, the properties for pDensity, pMass, pColor and pFixed are all **ReadOnly** properties as they are inherent properties of a shape that shouldn’t change in a real-life scenario. The rest of the properties will be both readable and writeable as they will be changing with every update cycle.

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| Update | Gravity (Vector) | NA | Used to update the value of gravity on the particle (depending on the value of the GravitySlider). Also used update the acceleration of each particle before it can be integrated. | Public |

**CircleParticle Class:**

**Fields:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| pBoundingBoxWidthHeight | Integer | Stores the dimensions of the virtual square outside the circle. It is equal to radius \* 2 | private | 12 |
| pRadius | Integer | Holds the radius of the circle | private | 6 |

**Properties:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Property Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| BoundingBoxWidthHeight | Integer | To **Get** the BoundingBoxWidthHeight of a circle | Public ReadOnly | 12 |
| Radius | Integer | To **Get** the Radius of a circle | Public ReadOnly | 6 |

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| Update |  |  | Mainly exists for possible future implementations. Calls the update methods in parent class: AbstractShape | Public |
| Draw | e (PaintEventArgs) |  | Calls the draw methods of all particles. | Public |

**RectangleParticle Class:**

**Fields:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| pWidth | Integer | Holds the width of a rectangle. | private | 7 |
| pHeight | Integer | Holds the height of a circle | private | 6 |

**Properties:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Property Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| Width | Integer | To **Get** the Width of a rectangle | Public ReadOnly | 12 |
| Height | Integer | To **Get** the Height of a rectangle | Public ReadOnly | 6 |

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| Update |  |  | Mainly exists for possible future implementations. Calls the update methods in parent class: AbstractShape | Public |
| Draw | e (PaintEventArgs) |  | Draws the particle and corresponding velocity line. | Public |

**Simulator Class:**

**Variables:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable Name** | **Datatype** | **Purpose** | **Scope** | **Example** |
| Particles | ArrayList | Holds all the particles. | Public shared | 12 |
| CurrentTime | DateTime | Holds the radius of the circle | private | 6 |

**Methods:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| StepUp | e (PaintEventArgs) | **NA** | This is the main simulation loop used to step up the simulation by calling the CollisionCheck and Update methods. | Public |
| GetDT | **NA** | Timestep as double | Used is to determine the semi-fixed timestep of the simulation unless the value of the speed of simulation is changed, in which case that value is used. | Private |
| CollisionCheck | AbstractShape | **NA** | Loops through each particle and checks for boundary or particle to particle collisions. | Private |
| Integrate | AbstractShape | **NA** | This is the implementation of semi-implicit Euler integration to create smoother and more realistic motion. | Private |

**Collider Class:**

The Collider class does not contain any fields or properties as all the required values will be passed in from the CollisionCheck method in the Simulator class.

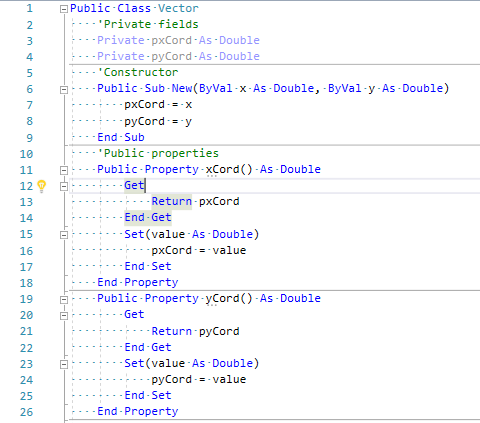
**Methods:**

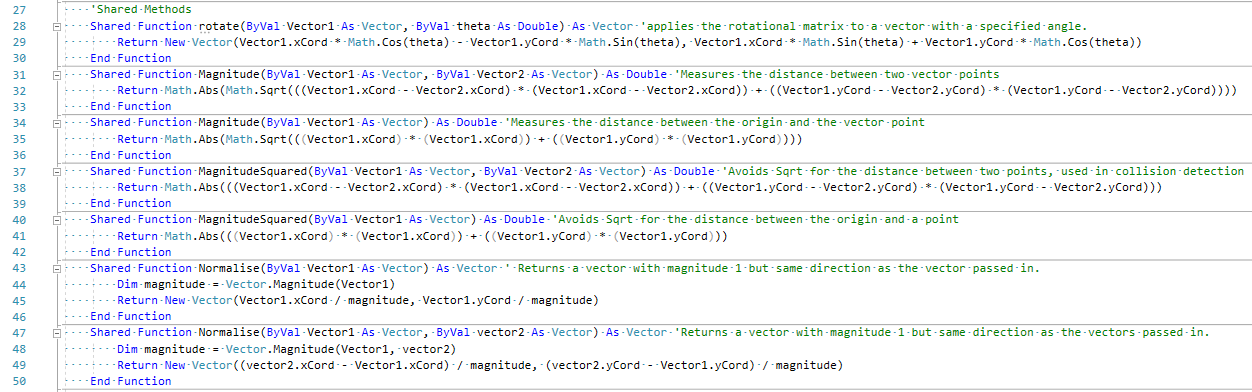
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| DetermineCollision | AbstractShape, AbstractShape | **NA** | Determines whether a circle vs circle or a rectangle vs circle collision test needs to take place. | Public |
| AABBvsCircle | RectangleParticle, CircleParticle | **NA** | This is the rectangle vs circle collision test. Determines if a collision has occurred and resolves it accordingly. | Private |
| CirclevsCircle | CircleParticle, CircleParticle | **NA** | This is the circle vs circle collision test. Determines if a collision has occurred and resolves it accordingly. | Private |
| BoundaryDetection | AbstractShape | **NA** | This is the circle vs boundary collision test. Determines if a collision has occurred and resolves it accordingly. | Private |

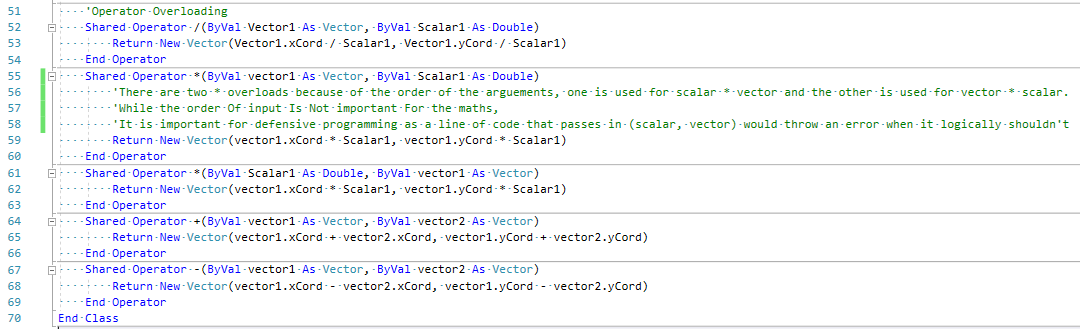
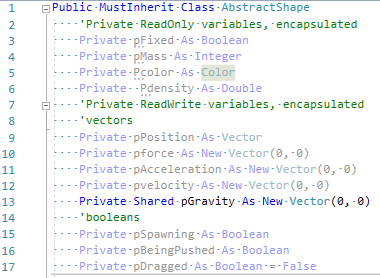
**SimForm Class:**

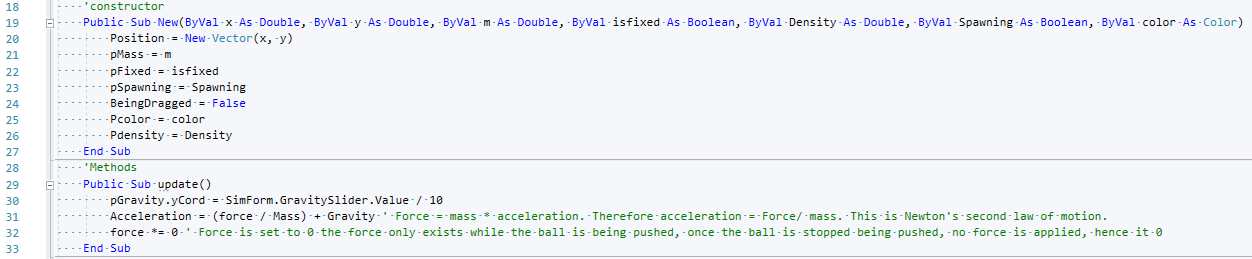
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method Name** | **Arguments** | **Return Values** | **Purpose** | **Scope** |
| SimForm\_Load | Object, EventArgs | **NA** | Initial Load event of the form. | Private |
| SimTime\_Tick | Object, EventArgs | **NA** | The timer used to invalidate (refresh) the DrawingSurface control, which causes it’s paint event to run. | Private |
| UpdateLists\_Tick | Object, EventArgs | **NA** | The timer used to update the listviews | Private |
| DrawingSurface\_Paint | Object, PaintEventArgs | **NA** | Calls the StepUP method in the simulator class to handle simulation. | Private |
| MouseDown, MouseUp, MouseMove **Events** | Object, MouseEventArgs | **NA** | For different mouse events. Clicking down the mouse button allows particles to be removed, dragged and pushed. MouseMove allows a force to be calculated and to drag particles around. MouseUp allows the force to be applied and to drop particles back into position. | Private |
| UpdateListViews() | **NA** | **NA** | Updates information in listviews. | Private |
| GetRandomColor | **NA** | Color | Generates a random colour for a particle’s outline. | Private |
| Slider\_Scroll **Events** | Object, EventArgs | **NA** | Updates the text of corresponding labels. Variables for CoR, Gravityt and Speed have not been used as they would need to be global for Simulator class to access them, hence, the simulator class looks at the value of the slider to get the values. | Private |

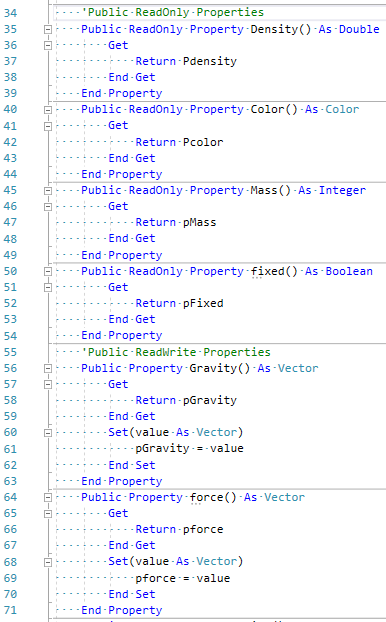
**Technical Solution**

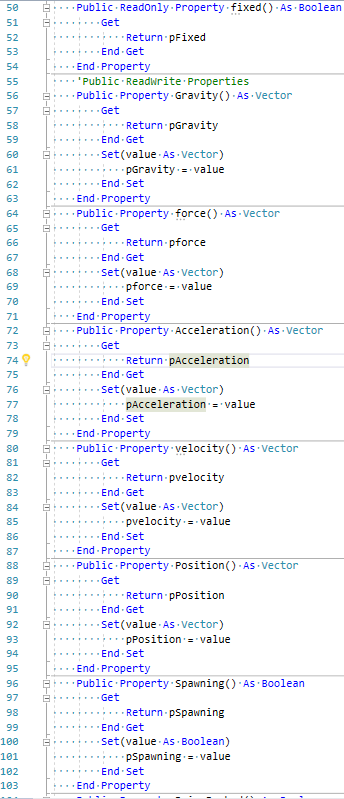
**Vector Class**

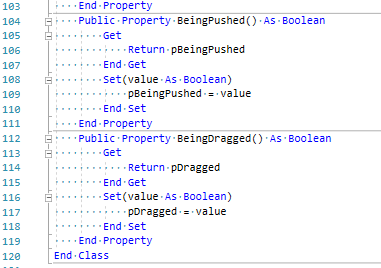


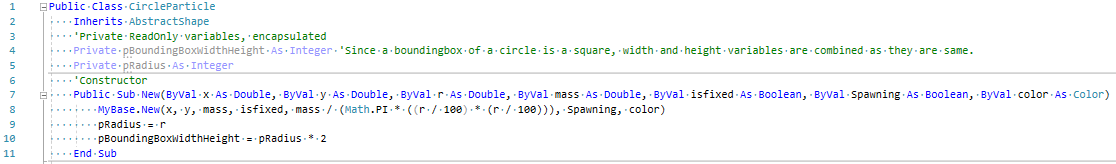
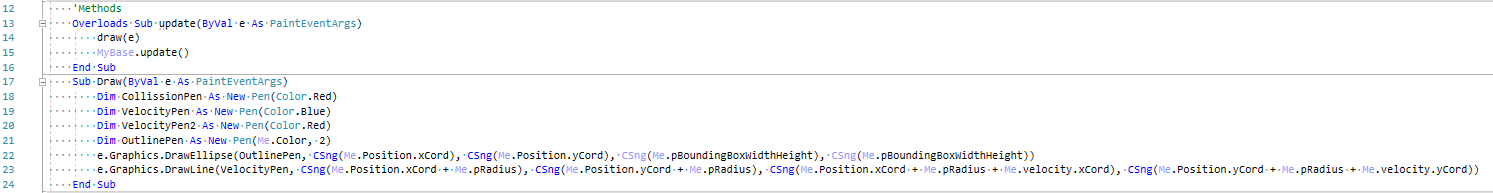
**AbstractShape Class:**

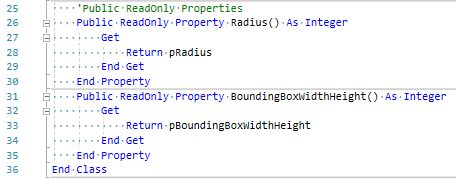


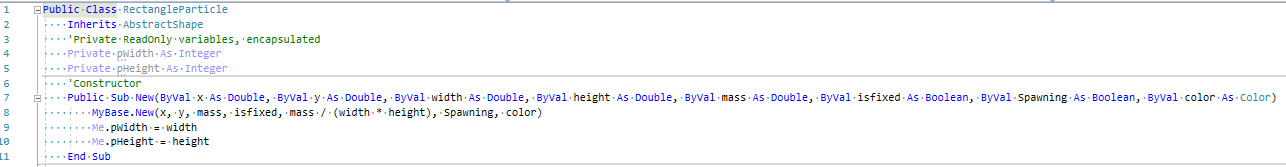


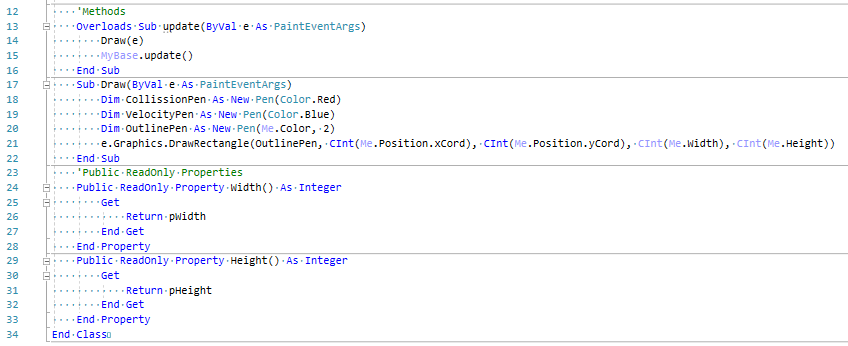


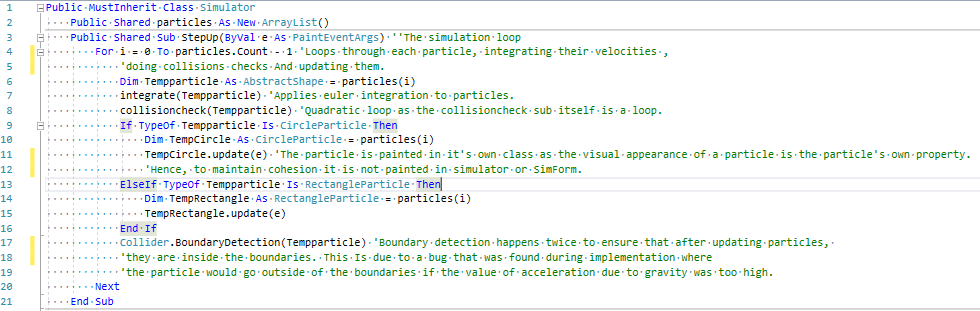
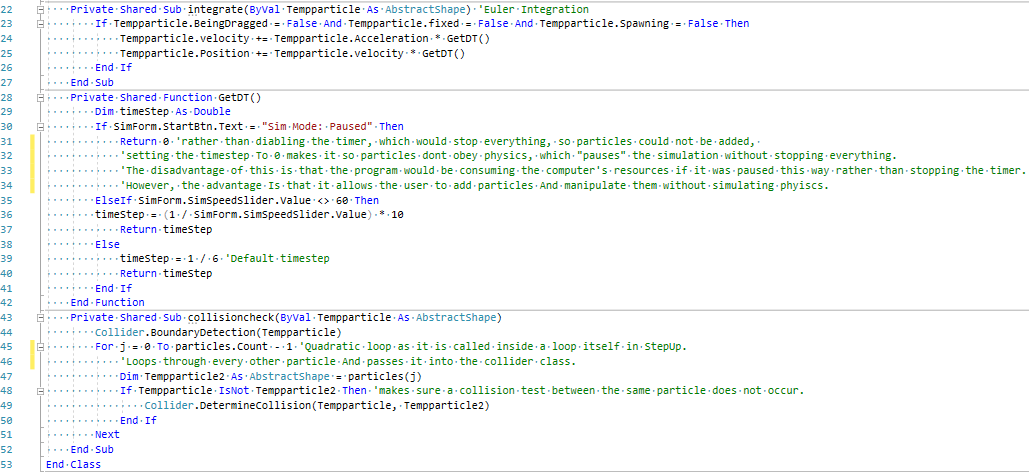


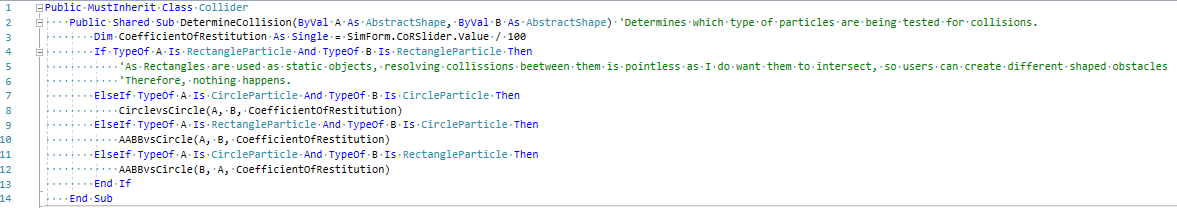
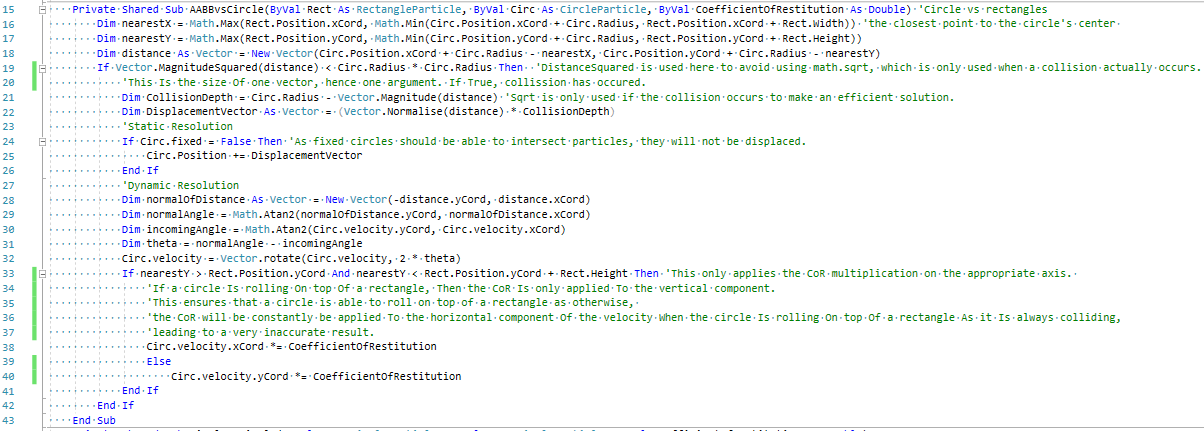
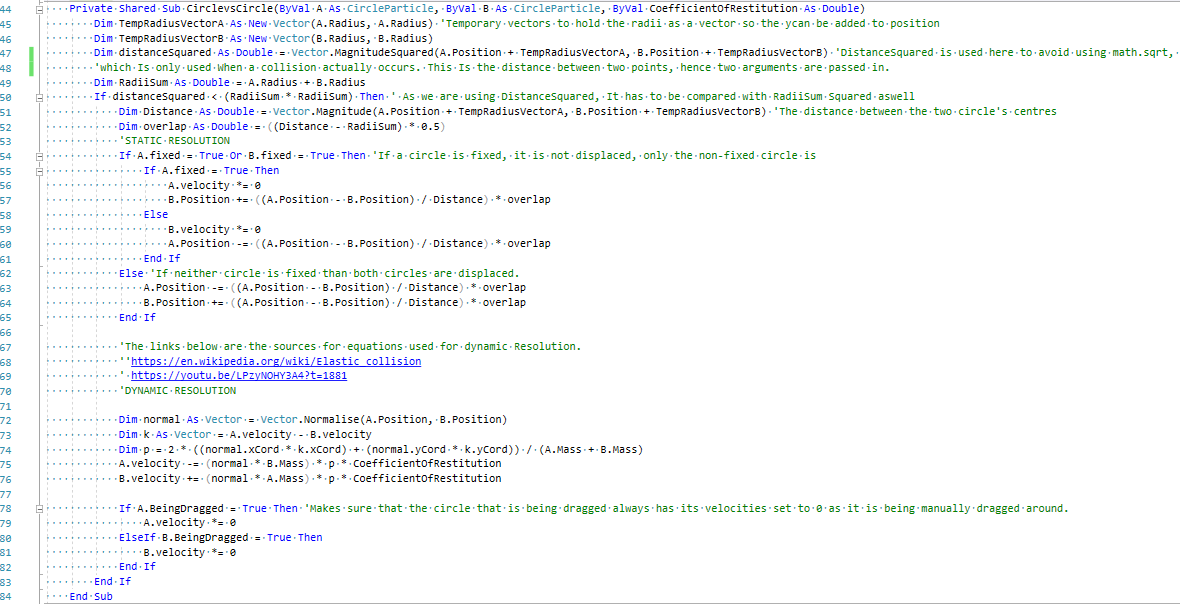
**CircleParticle Class:**

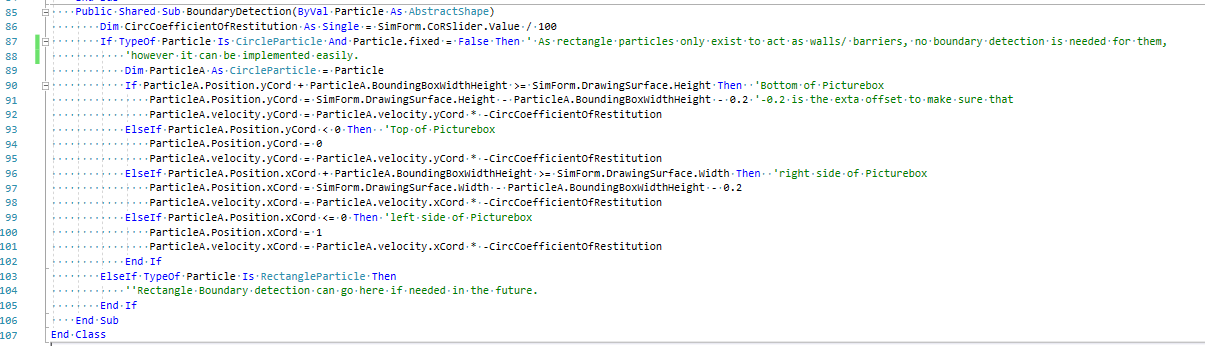


**RectangleParticle Class:**



**Simulator Class:**

**Collider Class:**

**Simform:**

**TutorialForm:**

**MathsForm:**

**Testing**

This section contains the final formal **black box tests** that I conducted after I felt like I had finished the program. While I tested my program multiple times during the implementation stage to get rid of bugs, these tests will demonstrate if I have met my initial objectives. If any objective is not met, I will try to find out the cause, fix it and document it down below. Since left and right mouse clicks have different functions in my program, it is important to show which mouse button is being clicked during the tests. To achieve this, a program called the NohBoard, a keyboard visualisation program, courtesy of ThoNohT through GitHub, has been used.

The testing video for these tables can be found by entering:  
“**https://www.youtube.com/watch?v=z3VKIkMFUKE**” in your browser or typing **Huzaifa Ahmed Physics Engine Black Box Testing Video** in the YouTube search bar (search results may vary due to other popular videos showing higher up).

**Test Tables**

**SimForm test:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Test** | **Purpose** | **Data Type** | **Explanation of test** | | **Expected Outcome from test** | **Pass/ Fail** | **Time in Video** |
| **1.1** | Adding a circle particle. | Erroneous | “Add Circle” is pressed, however radius and mass have not been specified so they default to 0. | An error message should be presented asking the user to input mass and radius. | | **Pass** | 0:20 |
| Erroneous | “Add circle” is pressed while spawning another particle already. | An error message “Please spawn your objects first!” should be shown. | | **Pass** | 14:35 |
| Normal | “Add Circle” is pressed when a radius between 1 and 200 (inclusive) and mass between 1 and 1000 (inclusive) has been input. | The circle should appear at the top left corner of the screen and should follow the mouse location. Left clicking on the screen will spawn the circle at the mouse location. The circle will not fall as the simulation is not running at default. The circle listview should contain the correct information regarding the circle. The rectangle’s line colour should be random | | **Pass** | 0:28 |
| Normal | “Add Circle” is pressed when a radius between 1 and 200 (inclusive) and mass between 1 and 1000 (inclusive) has been input. Fixed checkbox is checked. | The circle should appear at the top left corner of the screen and should follow the mouse location. Left clicking on the screen will spawn the circle at the mouse location. The circle will not fall as the simulation is not running at default and because it is fixed. The circle listview should contain the correct information regarding the circle. The rectangle’s line colour should be random | | **Pass** | 0:34 |
| Erroneous | A value of radius that is greater than 200 or a value of mass that is greater than 1000 has been input. | | The radius will be set to 200 and the mass will be set to 1000. | **Pass** | 01:11 |
| Boundary | A radius of 200 and a Mass of 1000 has been input. | | Nothing should happen. | **Pass** | 01:19 |
| **1.2** | Adding a rectangle particle. | Erroneous | “Add Rectangle” is pressed, however width, height and mass have not been specified. | | An error message should be presented asking the user to input width, height and mass. | **Pass** | 00:41 |
| Normal | “Add Rectangle” is pressed when the width, height and mass between 1 and 1000 inclusive have been input. | | The rectangle should appear at the top left corner of the screen and should follow the mouse location. Left clicking on the screen will spawn the rectangle at the mouse location. It will not fall as the simulation is not running at default and rectangles are modelled as fixed. The rectangle listview should contain the correct information regarding the rectangle. The rectangle’s line colour should be random. | **Pass** | 0:51 |
| Erroneous | A value of width, height or mass that is greater than 1000 has been input. | | The value will automatically be set to the upper bound, 1000. | **Pass** | 01:23 |
| Boundary | A value of width, height or mass that is 1000 has been input. | | Nothing should happen. | **Pass** | 01:31 |
| **1.3** | Starting the Simulation | Normal | “Sim Mode: Paused” (StartBtn button) is pressed to toggle the simulation so it starts running. | The simulation should start running so the non-fixed circle particle added before should start falling, the fixed circle particle should remain stationary. The text of the StartBtn button should change to “Sim Mode: Running” and the colour of the text should be green. The information corresponding to each particle should update in the list views. | | **Pass** | 01:44 |
| **1.4** | Pausing the Simulation | Normal | “Sim Mode: Running” (StartBtn button) is pressed to toggle the simulation so it pauses | The simulation, which should be currently running, should pause. Circle particles should stop moving. The text of the StartBtn Button should change to “Sim Mode: Paused” and the colour of the text should be red. | | **Pass** | 01:54 |
| **1.5** | Deleting particles | Erroneous | A circle particle is right clicked when the text of the DeleteModeBtn is “Mode: Moving” and the simulation is paused. | Since the simulation is not in delete mode, particles will not be deleted. Particles will have their velocities removed since they are being “grabbed” when they are right clicked. | | **Pass** | 02:20 |
| Erroneous | A rectangle particle is right clicked when the text of the DeleteModeBtn is “Mode: Moving” and the simulation is paused. | Nothing should happen as the simulation is not in delete mode | | **Pass** | 02:10 |
| Normal | A circle particle is clicked when the text of the DeleteModeBtn is “Mode: Deleting” and the simulation is running. | The circle particle will be removed from the simulation, the corresponding information from the circle particles listview will also be deleted. | | **Pass** | 02:37 |
| Normal | A rectangle particle is clicked when the text of the DeleteModeBtn is “Mode: Deleting” and the simulation is running. | The rectangle particle will be removed from the simulation, the corresponding information from the rectangle particles listview will also be deleted. | | **Pass** | 02:38 |
| Normal | A circle particle is clicked when the text of the DeleteModeBtn is “Mode: Deleting” and the simulation is paused. | The circle particle will be removed from the simulation, the corresponding information from the circle particles listview will also be deleted. | | **Pass** | 02:32 |
| Normal | A rectangle particle is clicked when the text of the DeleteModeBtn is “Mode: Deleting” and the simulation is paused. | The rectangle particle will be removed from the simulation, the corresponding information from the rectangle particles listview will also be deleted. | | **Pass** | 02:31 |
| **1.6** | Changing the value of the coefficient of restitution of collisions to test if it influences collisions. | Normal | The CoR slider is slid to the left. | As the value of the slider changes, the text of the corresponding label should also change, updating the value of CoR displayed. When the simulation is played, the particles should lose more kinetic energy in collisions, hence they should come to rest at a faster rate. | | **Pass** | 03:15 |
| Normal | The CoR slider is slid to the right. | As the value of the slider changes, the text of the corresponding label should also change, updating the value of CoR displayed. When the simulation is played, the particles should lose less kinetic energy in collisions, hence they should come to rest at a slower rate | | **Pass** | 03:23 |
| Boundary | The Cor slider is set to 0 and 1 | At CoR = 0, particles should stop immediately. | | **Pass** | 03:38 |
| At CoR = 1, particles should not lose any energy, hence they should not stop moving. | | **Pass** | 03:44 |
| **1.7** | Changing the simulation speed. | Normal | The Simulation speed slider is slid to the right. | | The speed of the simulation should slow down as the slider is slid to the right. | **Pass** | 05:04 |
| Normal | The Simulation speed slider is slid to the Left. | The speed of the simulation should speed up as the slider is slid to the left. | | **Pass** | 05:22 |
| Boundary | The simulation speed is set to 1x slower and 20x slower. | The simulation should behave normally as it did with normal data. | | **Pass** | 05:01 and 05:14 |
| **1.8** | Changing the value of acceleration due to gravity | Normal | The acceleration due to gravity slider is slid to the right. | Particles should have a higher acceleration applied to them, hence they should fall faster. | | **Pass** | 04:08 |
| Normal | The acceleration due to gravity slider is slid to the left. | Particles should have a lower acceleration applied to them, hence they should fall slower. | | **Pass** | 04:22 |
| Boundary | The acceleration is set to 0 and then at 100 | At acceleration = 0, particles should act like they are in a vacuum, there will be no effect of gravity. | | **Pass** | 04:28 |
| At acceleration =100, particles should act as they would with normal data, with extremely high acceleration due to gravity. | | **Pass** | 04:42 |
| **1.9** | Applying a force to particles | Erroneous | The left mouse button is held down when the cursor is not inside a particle, dragged mouse button is released. | Nothing should happen as the cursor is not inside a particle. | | **Pass** | 06:21 |
| Erroneous | The left mouse button is held down, when the simulation is running, with the cursor is inside a rectangle particle, dragged and the mouse button is released. | Nothing should happen as the cursor is inside a rectangle particle, which cannot have a force applied to them. | | **Pass** | 06:29 |
| Erroneous | The left mouse button is held down with the cursor is inside a fixed circle particle, dragged and the mouse button is released. | Nothing should happen as the cursor is inside a fixed circle particle, which cannot have a force applied to them. | | **Pass** | 06:25 |
| Erroneous | The left mouse button is held down, when the simulation is paused, with the cursor is inside a rectangle particle, dragged and the mouse button is released. | Nothing should happen | | **Pass** | 06:40 |
| Erroneous | The left mouse button is held down, when the simulation is paused, with the cursor is inside a fixed circle particle, dragged and the mouse button is released. | Nothing should happen | | **Pass** | 06:39 |
| Erroneous | The left mouse button is held down, when the simulation is paused, with the cursor is inside a non-fixed particle, dragged and the mouse button is released. | Nothing should happen | | **Pass** | 06:41 |
| Normal | The left mouse button is held down with the cursor is inside a circle particle, dragged and the mouse button is released. | A force should be applied to the circle particle; hence it should gain an acceleration. | | **Pass** | 06:31 |
| **2.0** | Dragging a particle around | Erroneous | The right mouse button is held down when the cursor is not inside a particle, dragged and let go. | Nothing should happen as the cursor is not inside any particle. | | **Pass** | 06:56 |
| Normal | The right mouse button is held down when the cursor is inside a particle (circle or rectangle), dragged around and the mouse button is released. | The particle’s centre should follow the cursor’s position. The particle should ignore gravity but should still be able to collide. Letting go of the right mouse button should drop the particle back into the simulation, so it is subject to gravity. | | **Pass** | 07:01 |
| **2.1** | Displaying the tutorial. | Normal | The “Show tutorial” button is pressed | The Tutorial form should open. | | **Pass** | 07:18 |
| **2.2** | Resetting the simulation. | Normal | The “RESET simulation” button is pressed. | All current particles in the simulation should be removed. | | **Pass** | 05:57 |
| The listviews should be cleared. | | **Pass** |
| The values of sim speed should be set to “1x slower” and the corresponding label should be updated. | | **Pass** |
| The values of acceleration due to gravity should be set to “9.8” and the corresponding label should be updated. | | **Pass** |
| The values of the CoR should be set to “0.5” and the corresponding label should be updated. | | **Pass** |

**Collisions test:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test** | **Purpose** | **Data Type** | **Explanation** | **Expected Outcome** | **Pass/ Fail** | **Time in Video** |
| **2.3** | Checking collisions between circle particles. | Normal | Two circles are allowed to collide by applying a force when the CoR is 1 | The circles should collide with each other and should apply an appropriate force to each other, conserving momentum. | **Pass** | 10:05 |
| Normal | Two circles are allowed to collide by applying a force when the CoR is 0 | The circles should not intersect each other and should apply an appropriate force to each other. Momentum should not be conserved. | **Pass** | 10:38 |
| Normal | A circle is manually dragged into another circle | If the circle which is not being dragged is fixed, then it should not move away, and the particles should collide. | **Pass** | 12:59 |
| If the circle which is not being dragged is not fixed, it should be pushed away. | **Pass** | 13:11 |
| **2.4** | Checking collisions between a circle and a rectangle particle. | Normal | A circle is allowed to collide with a rectangle by applying a force. | Since rectangles are fixed, it should not move. The circle should bounce in the opposite direction. Its momentum depending on the CoR. | **Pass** | 11:04 |
| Normal | A non- fixed circle is manually dragged into the rectangle. | The particles should collide, the rectangle should not move as it is fixed | **Pass** | 12:04 |
|  | A fixed circle is manually dragged into the rectangle. | Should not collide. | **Pass** | 13:23 |
| Normal | A rectangle is manually dragged into a non-fixed circle. | The particles should not intersect. The circle particle should be pushed away by the rectangle particle. | **Pass** | 12:32 |
| Normal | A rectangle is manually dragged into a fixed circle. | Should not collide. | **Pass** | 13:21 |
| **2.5** | Checking collisions between rectangle particles. | Normal | A rectangle particle is manually dragged into another rectangle particle. | The rectangle particle should intersect, allowing the user to create custom shapes. | **Pass** | 14:19 |
| **2.6** | Checking the collisions between boundary and particles. | Normal | A circle particle is launched towards the boundaries. | The circle should bounce back in the opposite direction. | **Pass** | 12:42 |
| Normal | A non-fixed circle is manually dragged towards the boundaries. | The circle should not go beyond the boundaries, if it tries to, it should be sent back inside. | **Pass** | 13:26 |
| Normal | A fixed circle is manually dragged towards the boundaries. | The circle should be able to go through the boundaries. This is to allow the user to create custom shapes. | **Pass** | 13:51 |
| Normal | A rectangle is manually dragged towards the boundaries. | The rectangle should be able to go through the boundaries. This is to allow the user to create custom shapes. | **Pass** | 13:45 |

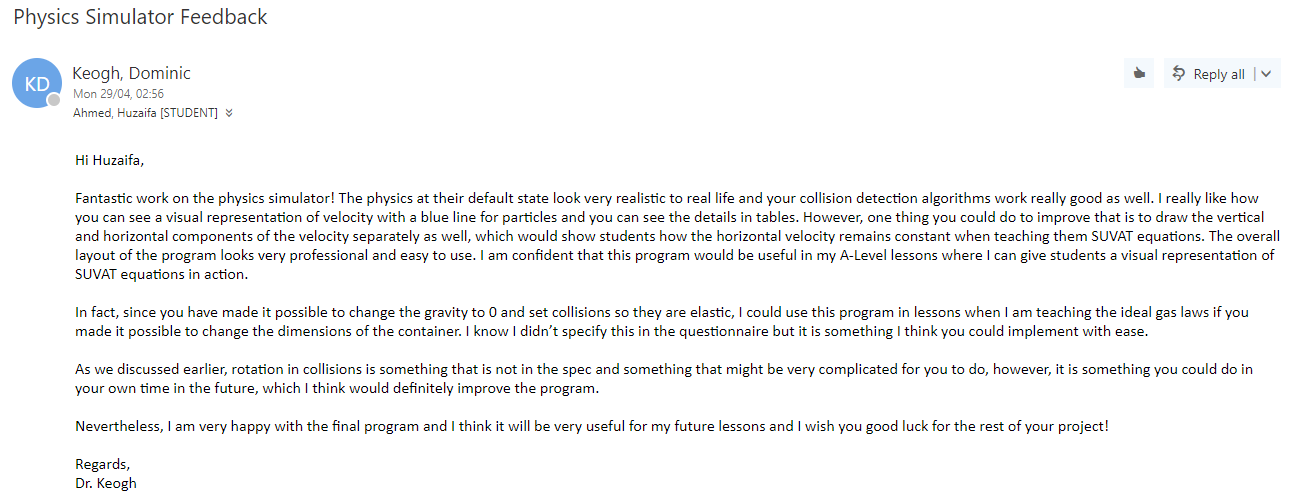
**TutorialForm:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test** | **Purpose** | **Data Type** | **Explanation** | **Expected Outcome** | **Pass/ Fail** | **Time in Video** |
| **2.7** | Displaying tutorial #1: adding particles. | Normal | The linklabel with the text: “1. Adding particles to the simulation” is clicked. | A messagebox containing text that explains how particles can be added should pop up. | **Pass** | 07:24 |
| **2.8** | Displaying tutorial #2: removing particles. | Normal | The linklabel with the text: “2. removing particles from the simulation” is clicked. | A messagebox containing text that explains how particles can be removed should pop up. | **Pass** | 07:30 |
| **2.9** | Displaying tutorial #3: manipulating particles. | Normal | The linklabel with the text: “3. Manipulating particles in the simulation” is clicked. | A messagebox containing text that explains how particles can be manipulated should pop up. | **Pass** | 07:39 |
| **3.0** | Displaying tutorial #4: manipulating simulation. | Normal | The linklabel with the text: “4. Manipulating the simulation” is clicked. | A messagebox containing text that explains how the simulation can be manipulated should pop up. | **Pass** | 07:48 |
| **3.1** | Displaying tutorial #5: Limitations of the model. | Normal | The linklabel with the text: “5. Limitations of the model” is clicked. | A messagebox containing text that explains the limitations of the model should pop up. | **Pass** | 07:53 |
| **3.2** | Displaying tutorial #6: Maths used in the simulation. | Normal | The linklabel with the text: “6. Maths used in the simulation. | MathsForm should open. | **Pass** | 08:01 |

**MathsForm:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test** | **Purpose** | **Data Type** | **Explanation** | **Expected Outcome** | **Pass/ Fail** | **Time in Video** |
| **3.3** | Opening sources for equations and pictures. | Normal | The link label with text “Blue line is an unknown curve,  red line is the approximation” is clicked | The corresponding website should open on the internet browser. | **Pass** | 08:11 |
| The link label with text “Source: https://en.wikipedia.org/wiki/Euler\_Method” is clicked | The corresponding website should open on the internet browser. | **Pass** | 08:24 |
| The link label with text “Gif source: https://en.wikipedia.org/wiki/Elastic\_collision#/media/File:Elastischer\_sto%C3%9F\_2D.gif  ” is clicked | The corresponding website should open on the internet browser. | **Pass** | 08:37 |
| The link label with text “Angle Equations from Wikipedia” is clicked | The corresponding website should open on the internet browser. | **Pass** | 08:48 |
| The link label with text “Angle free equations from Wikipedia  ” is clicked | The corresponding website should open on the internet browser. | **Pass** | 09:02 |

**Evaluation**

**End User Feedback**

Overall, I felt very satisfied with the result of the project as I believe it was an effective solution to my end user’s needs. Looking back at it, while programming collisions was a challenging and frustrating task at times, it felt very rewarding at the end seeing it all work without crashing the program. As someone who is passionate about computer games development, creating the engine is an accomplishment for me as I took the time and effort to learn the logic and mechanics behind physics engine. Furthermore, my end user liked my project, he believed that the simulation with the default values seemed very realistic, which means I have met my overall goal of creating a physics engine since the physics have been described as realistic. My end user was also very pleased with drawing of velocity lines, however, he stated that breaking the velocity lines into its respective vertical and horizontal components would aid the students understanding more as the horizonal component remains the same in projectile motion. This wouldn’t be very difficult to implement as it would only require basic Pythagoras. Moreover, he also mentioned that program was already able to demonstrate the ideal gas and allowing him to change the dimensions of the container will allow him to explain it to students. This is also an easy task as sliders can be included to change the picturebox’s widths and heights. Lastly, he reiterated that rotation is something that could improve the program, which I agree with I will try to implement it in the future.

**Confirmation of meeting the Initial Objectives**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objective:** | **Pass/Fail** | Explanation | Rating (0-5) | How to improve |
| 1.1: Particles should be able to move in any direction at a velocity. | **Pass** | This objective has been achieved as it can be seen in the testing video that the particle is able to fall downwards, and applying a force in a direction makes the particle move in that direction. This can be seen in test 1.9. | 4 | The program could be made 3D, meaning it particles will be able to move in any direction. |
| 1.2: Non-fixed particles should be subject to gravity unless they are static objects. | **Pass** | This objective has been met as it can be seen in the testing video during test 1.3. | 5 | **NA** |
| 2.1: Particles should not be able to intersect other particles unless they are both fixed, if they do, they both should be displaced accordingly so that they do not intersect anymore. | **Pass** | This object has been met as it can be seen in the Collisions tests, which show particles colliding and being displaced. | 4 | This objective refers to collision detection and resolution, which can be vastly improved by implementing collision between any convex shape. Including rotation between collisions would make the simulation more realistic as a whole and it would give a purpose to the current semi-useless rectangle particles. |
| 2.2: If a particle is “launched” towards another particle and they collide, a force should be applied on both particles in appropriate directions. The momentum in the collision should also be conserved. | **Pass** | Tests 2.3 and 2.4 in the collisions testing show particles being launched and allowed to collide. Since both tests passed in the testing video, this objective has been met. | 5 | **NA** |
| 2.3: Elastic and inelastic collisions will rely on a coefficient of restitution (CoR) variable, meaning a CoR of 1 will result in no energy loss of the particular particle and CoR of 0 will mean the particle will stop after any collision. | **Pass** | This objective has been met because firstly, the value of CoR is hard coded in the program to be between 0 and 1 and secondly, test 1.6 demonstrates the CoR being used and the test passed. | 5 | **NA** |
| 2.4: Non-fixed particles should stay within the bounds of the drawing area, if one attempts to leave, it shall bounce of/ displace accordingly. | **Pass** | This objective has passed because test 2.6 passed, which tests collisions between particles and boundaries. | 5 | Including collisions between convex shapes would naturally improve this objective. |
| 3.1: The user is able to spawn a particle at the x and y position of their cursor. | **Pass** | This objective has been met as shown in test 1.1 as the user was able to spawn the particle wherever they wanted with their cursor. | 5 | **NA** |
| 3.2: The user is able to specify the radius and mass of the particle before spawning it in. | **Pass** | This objective has been met as shown in test 1.1 as the user was able to specify radius and mass. | 5 | This objective refers to inputting the properties of a particle, which would also be naturally improved if the user is allowed to create a custom convex shape. |
| 3.3: The user should be able to delete any specified particle. | **Pass** | This objective has been met as shown in test 1.5 as the user was able to delete any particle they wanted. | 5 | **NA** |
| 4.1: If the simulation is running, it should be able to pause, and vice versa. | **Pass** | This objective has been met as shown in test 1.4 | 5 | **NA** |
| 4.2: The end user should be able to control the speed of the simulation. I.e. make it slower through a slider. | **Pass** | This objective has been met as shown in test 1.7 | 4 | The current slider doesn’t allow the simulation to be made faster because that proved to be rather glitchy as particles would go through other particles or walls at high speeds. However, this can be improved by implementing a more robust collision detection system could potentially solve this problem. |
| 4.3: The user can completely reset the simulation. (All particles should be removed and the values for the coefficient of restitution, speed of the simulation and acceleration due to gravity should be set to default, which are 1, 1x slower and 9.81 respectively.) | **Pass** | This objective has been met as shown in test 2.2 | 5 | **NA** |
| 5.1: The end user can change the value of acceleration due to gravity with a slider. The minimum value should be 0, so they can simulate collisions in space. | **Pass** | This objective was met as shown in test 1.8 | 5 | The maximum value of acceleration due to gravity is 100 m/s2 at this stage as at higher values, particles start to go through other particles. This will be inevitably fixed when a more robust collision detection system is created. |
| 5.2: The end user can change the value of the CoR with a slider. The minimum value should be 0 to demonstrate complete loss of energy and the maximum should be 1 to demonstrate complete conservation of energy. | **Pass** | This objective was met as shown in test 1.6 | 5 | **NA** |
| 6.1: The user is able to pick up a particle, drag it around and drop it wherever their mouse cursor is. | **Pass** | This objective was met as shown in test 2.0 | 5 | **NA** |
| 7.1: The user should be able to apply a force to particles with their mouse. | **Pass** | This objective passed as shown in test 1.9 | 5 | **NA** |
| 7.2: Newton’s second law of motion: F=ma should be obeyed when applying a force, so the same size of force on a heavier particle should apply a smaller acceleration than a lighter particle. | **Pass** | This objective passed as shown in test 1.9 in the testing video as the acceleration added from the force in the code = force / mass | 5 | **NA** |
| 8.1: A listview should include information about each particle such as the velocities of the particles, their positions, mass, and density. | **Pass** | This objective is met as the listviews with the appropriate headers are visible in the testing video. | 5 | **NA** |
| 8.2: The listview should automatically add items when a new particle is created | **Pass** | This objective was met as shown in test 1.1 and 1.2. When add circle/particle is pressed, a new item in the list view is added which contains information about the particle’s position, velocity, mass and density and dimensions. | 5 | **NA** |
| 8.3: Anytime a property of a particle is updated, such as its position, the corresponding value in the listview should update as well. | **Pass** | This objective is met as the listviews refresh every at every timer tick, the effect of this can be seen throughout the testing video when any particle moves. | 3\* | This objective has been given a 3/5 because while current implementation does the job, there is an annoying “flicker” that occurs. I tried solving this problem however I was not able to as it might be a bug with list boxes in VB.net as using the same logic and similar code, I was able to program this in C# and there was no flicker. However, rewriting the entire program in C# is not a good solution so this problem can’t be fixed. Furthermore, due to generating random colours, some colours can be generated that are too light to see against the background, this can be solved by only generating colours that are dark. |
| 8.4: Each particle should have an arrow drawn on it to represent its velocity. | **Pass** | This objective is met as it is visible throughout the testing video when a particle is moving, it has a blue line representing its velocity. | 5 | This objective can be improved by acting on the end-user’s feedback which is to also show the separate vertical and horizontal components of the velocity. |

**Implementing improvements:**

* **Making the program 3D**

The approach I could have taken to make my program three dimensional was to use Open-GL for graphics due to its far superior rendering capabilities when compared to GDI+. However, as I mentioned in my analysis, I avoided this approach as it would mean I would have to learn how to use Open-GL, which is more complicated than GDI+ itself and it means the program would get less development time and so a less functional but better-looking program would have been developed. However, after completing this project, I feel like a more knowledgeable programmer compared to when I started on this programmer and I would enjoy converting this physics engine into a 3D program with OpenGL. It would be a very time-consuming and challenging task; however, I have a lot of time during the summer holidays and this is something I can do to keep me busy.

* **Adding user-specified convex shapes (and creating a more robust collision engine)**

Adding user-defined convex shapes is not much of a challenge in the program. It can be simply done by allowing the user to create a shape by connecting lines together. In terms of programming, if a user clicks on the screen while they are creating a shape, a Boolean value can be assigned for this, a node will be created which will be added to a list, These nodes are essentially corner points and an if statement with some trigonometry can be used to see if the shape as an internal angle that is greater than 180 degrees, and if it does, then is a concave shape and so is not a valid. It can be argued that concave shapes should be included as that would make the simulation more realistic, however, this should be an extension task to this one as collision detection between concave shapes is very difficult to program.

The closest point approach that I currently use will not work for collision detection with convex polygons as it only works on axis-aligned bounding boxes (rectangles) so I would need to use the separating axis theorem (SAT) which would make my collision engine more robust. I found a YouTube video by the OneLoneCoder [10] that explains how to implement convex polygon collisions which I could use, however, he doesn't explain dynamic resolution which means I would need to study that further to program the physics. Furthermore, to add rotation to the current rectangles, I would need to rework the RectangleParticle class by adding an array of 4 vectors, each representing a corner point. The Draw method would need to be updated as well so instead of just drawing a rectangle, 4 lines would be drawn, each connecting to the corner points. This is to make rotation easier as all the physics would be handled by the simulation which would affect the top left corner point, and the rest of the points can be determined in the update method, which was created to be able to extend the functionality of the program, of the rectangle class.

* **Only generate dark colours.**

This can be done by following an approach described I found on an article by Martin Ankerl[11] which is to use HSV (hue, saturation, value) rather than RGB (red, green, blue) when generating random colours. Randomizing the saturation and value while keeping the hue the same would generate colours that are darker, hence easier to see. However, the problem is that VB.net generates colours from ARGB (Adobe RGB) so I would need to first randomize the colour using HSV, then convert it to RGB, which can be done so with the help of an article on Wikipedia [12] that explains HSV to RGB conversions.

* **Drawing vertical and horizontal components**

This is an improvement suggested by the end user a it allows him to better demonstrate the fact that the horizontal component of velocity does not change in projectile motion. This can be simply done by adding two additional line being drawn in the draw methods of the CircleParticle class. The horizontal velocity line can be drawn by creating a line from the circle’s centre point (centre.x, centre.y) to the point, (velocity.x, centre.y) and the vertical velocity line can be drawn by creating a line from the point (velocity.x, velocity.y) to the point (velocity.x, centre.y)

**Additional features:**

* **Adding air resistance/ Drag**

As a lot of the times, particles are falling, it would make sense to implement air resistance (or drag). This feature can simply be implemented by calculating the value of drag, which is:

**Drag = Coefficient x (Density x Velocity^2) x 0.5 x area of particle**

*Where the* ***coefficient*** *is an experimental value.*

***Density*** *is the density of the surrounding, such as air or water.*

This value can simply be taken away from the velocity of the particle. There are several values of the coefficient available on the internet for different shapes, or I could try values until one produces realistic results for different shapes.

* **Adding liquids**

While it is very tough, to program visually realistically flowing fluid, fluid can still be implemented in a very simple manner. Creating a separate class, which inherits RectangleParticles should do the job. When calculating the drag (as described previously), a simple collision check can be used, as it is already programmed for circles and rectangles, and if the particle is inside this rectangle, the value of density would be changed to the corresponding density of the liquid such as 997 kg/m³ for water, else the density of air will be used: 1.225 kg/m³.

**Usability of the Engine**

As I have created a general-purpose physics engine rather than a program that is specifically only tailored towards classroom use, it is important that the engine has many other applications.

**Platformer Shooter Game.**

For example, one application would be to develop a basic platformer game, while the current program does not have the relevant objects to do so as it was created for the needs of my end user, a simple class called Player can be created, which inherits the circle or rectangle class and has methods that apply a force to it in the appropriate direction when an arrow key is pressed. As collision detection is already in place, the player will now be able to move and collide with walls. Enemies can be added by creating an Enemy class that inherits circle/rectangle class, a simple bullet class can be created by inheriting the circle class which can be launched from the player’s position to where the cursor is, colliding the enemy could cause the enemy to die.

**8-ball pool game**

An 8-ball pool would arguably require more work than the platformer game mentioned above as one would need to program in all the rules of the game. However once that is done, everything else is relatively easy. Additional properties can be added to the circle particle which dictates whether the ball is stripped, solid, pocket or a cue ball. Value of acceleration due to gravity would be set to 0 and the CoR would be set to 1. When applying a force to the cue ball, a line could be drawn representing the magnitude of the force. When a solid, striped or cue ball collides with a pocket circle, which are fixed around the table, the ball is pocketed, and the rules follow.

**Simulating the ideal gas laws**

This is something that could be created very easily already, setting the value of acceleration due to gravity to 0 and the value CoR to 1 will ensure all collisions are elastic. Adding two sliders that adjust the dimensions of the picturebox would allow demonstrating the kinetic theory of gasses as well.

**Conclusion**

I am happy to say that I have met all objectives, as visible from the objectives table above, that I had created in the analysis section after the discussion and questionnaire from my end user.

As my project in its entirety was a success, I believe it would have been even better given more time as I would have liked to add many things and features, some of which I explained earlier in the evaluation. Some of the other things I could have added would have been more types of particles such as wheel particles, that rotate and move, bridge particles, which could have been a set of particles attached to each other by a string and pendulums, which would have covered the pendulums, an ordinary circle attached to a string. Furthermore, I could have delved into my mathematics specification and added pulleys, ladders and moments. However, all of these tasks require a substantial amount of time and effort to complete to an acceptable standard which is why I decided to not include them. Prioritising the core of the engine compared to additional features undoubtedly created a more complete program.

One major issue with my program is the flicker in the list view when updating a particle’s information. As described in the objectives table above. However, despite trying various solutions such as using addRange, Begin/End Update and Suspend/Resume Layout, there was no fix to the problem. The only way this program could have fixed was to recode my entire program in C#, which seems like a lot of work to just fix a very small inconvenient bug.

Regardless, any project can have small bugs and lots of aspects to improve upon, but I am proud of how well mine turned out. Both I and my end user with the overall quality and scope of the project and I hope any aspiring physicist or computer scientist can use my program to learn more about projectile motion and collisions.

**References**

**[1] = (Quora, 2015 )** [**https://www.quora.com/What-are-the-benefits-and-disadvantages-of-each-CS-programming-language**](https://www.quora.com/What-are-the-benefits-and-disadvantages-of-each-CS-programming-language)  
**[2] = (Dreamincode, 2010 )**[**https://www.dreamincode.net/forums/topic/27087-advantagesdisadvantages-of-programming-languages/**](https://www.dreamincode.net/forums/topic/27087-advantagesdisadvantages-of-programming-languages/)  
**[3] = (Codeproject, 2003 )**[**https://www.codeproject.com/Articles/4659/Introduction-to-GDI-in-NET**](https://www.codeproject.com/Articles/4659/Introduction-to-GDI-in-NET) **[4] = (VB Forums, 2016) )**[**http://www.vbforums.com/showthread.php?840001-A-question-about-GDI-a-comparison-between-vb-and-c-and-overall-responsiveness**](http://www.vbforums.com/showthread.php?840001-A-question-about-GDI-a-comparison-between-vb-and-c-and-overall-responsiveness) **[5] = (VB Forums, 2005)** [**http://www.vbforums.com/showthread.php?342255-Flicker-Free-Listview-VB-NET**](http://www.vbforums.com/showthread.php?342255-Flicker-Free-Listview-VB-NET) **[6] = (YouTube, 2018)** [**https://www.youtube.com/watch?v=LPzyNOHY3A4&t=705s**](https://www.youtube.com/watch?v=LPzyNOHY3A4&t=705s) **[7] = (Stackoverflow, 2017)** [**https://stackoverflow.com/questions/45370692/circle-rectangle-collision-response**](https://stackoverflow.com/questions/45370692/circle-rectangle-collision-response) **[8] = (OneLoneCoder's Github)** [**https://github.com/OneLoneCoder/videos/blob/master/OneLoneCoder\_Balls1.cpp**](https://github.com/OneLoneCoder/videos/blob/master/OneLoneCoder_Balls1.cpp) **[9] = (YouTube, 2019) https://youtu.be/7Ik2vowGcU0   
[10] = (Stackoverflow, 2017)** [**https://stackoverflow.com/questions/45370692/circle-rectangle-collision-response**](https://stackoverflow.com/questions/45370692/circle-rectangle-collision-response) **[11] = (martin.ankerl.com, 2009)** [**https://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/**](https://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/)  
**[12] = (Wikipedia)** [**https://en.wikipedia.org/wiki/HSL\_and\_HSV#HSV\_to\_RGB**](https://en.wikipedia.org/wiki/HSL_and_HSV#HSV_to_RGB) **[13] = (Gafferongames, 2004)** [**https://gafferongames.com/post/integration\_basics/**](https://gafferongames.com/post/integration_basics/)