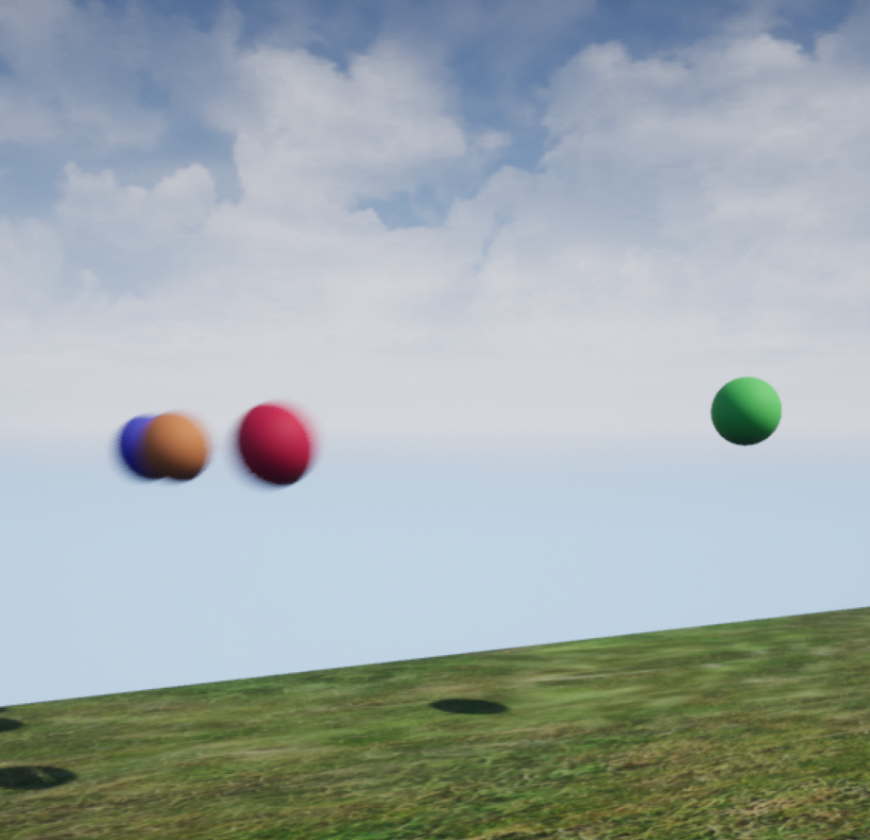
Implementation of Physics Simulation for Computer Games

GAV-3019-N



Bethan Symes | t7099112 | 2020-21

# Part 1: Ballistic Trajectory Using Explicit and Solver Calculations.

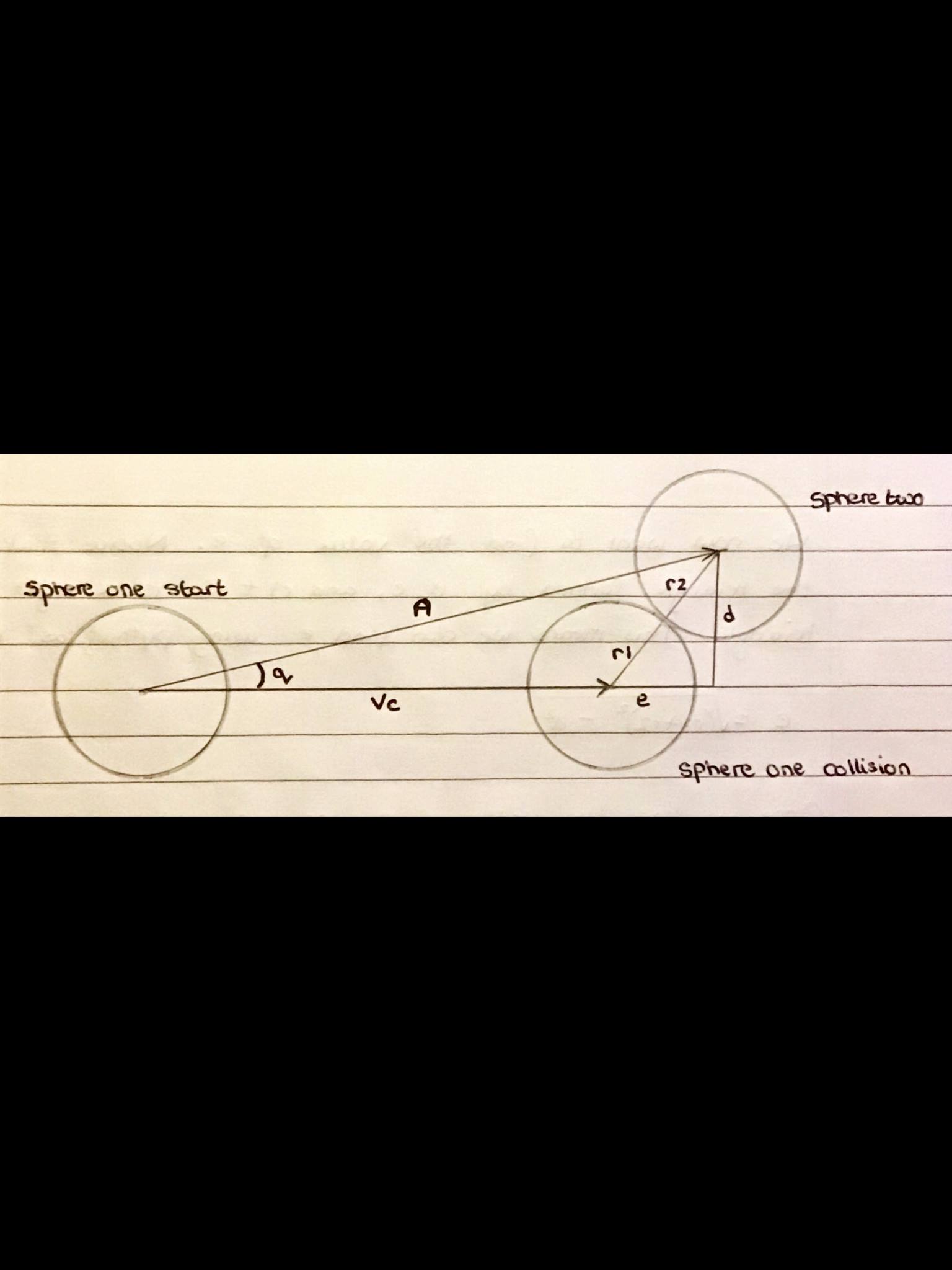
**Comparison between Newton, and Euler’s Method**

When the same initial velocity is applied to both Euler and Newtons calculations, the product is different for each. This is due to the fact that Euler’s method, in this case, is attempting to calculate in real time using discreet time steps. As reality does not move in discreet time steps, a small error is produced after each calculation. These errors add up over the course of the simulation, making the end result less accurate, sometimes by a significant margin depending on the length of the simulation and the number of time steps used in calculation. For example, the initial velocity vector pertaining to each identical sphere at the beginning of the simulation is {750, 0, 750} and both spheres start at position (0, 0, 0). In the simulation, the end position of the Newton sphere is 12 units less in the x axis than the Euler sphere which is calculated in discreet time steps which does not follow the way real time flows. On a small scale such as this, the error appears to be minor. However, due to the necessity for these simulations to be in real time, the step size must be limited and therefore the accuracy will also be limited. This makes Newtons calculations more appropriate for everyday physics simulation. This is visible in the simulation output in Unreal Engine.

# Part 2: Sphere to Sphere, and Sphere to Plane – Collision Detection and Response.

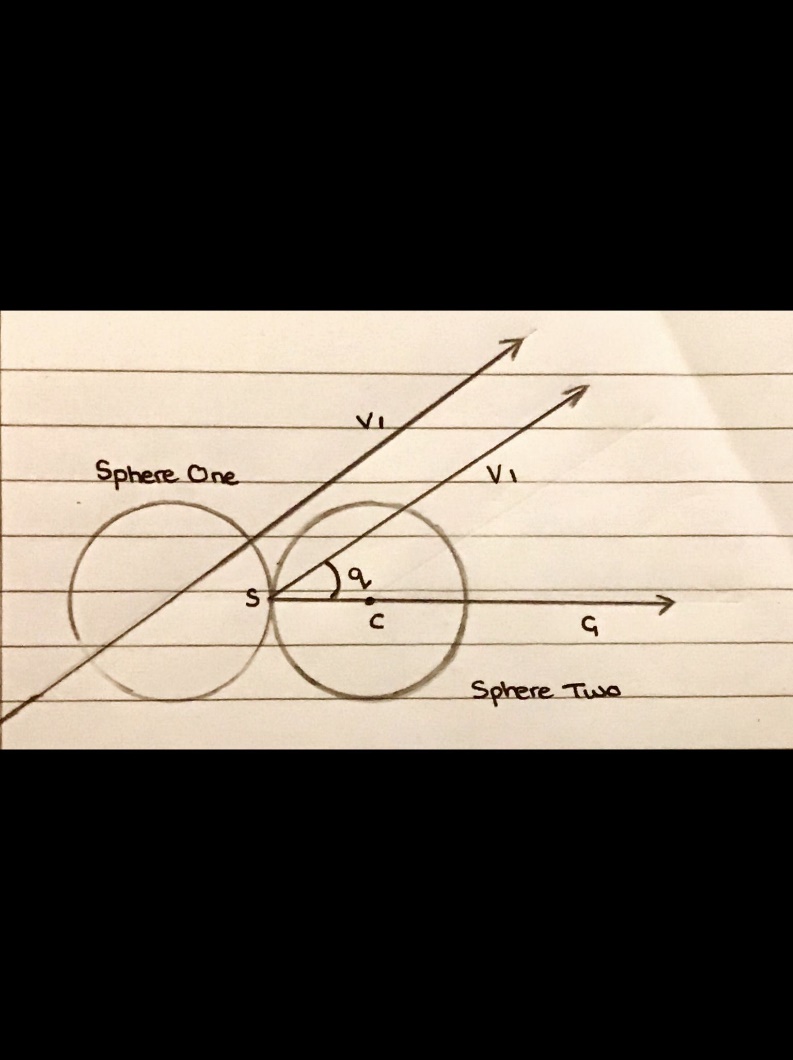
**Collision Detection- Sphere to Sphere**

When calculating sphere to sphere collision, in this case, we are talking about one sphere moving towards another static sphere. In order to calculate sphere to sphere collision, we need to consider the velocity at which sphere one is moving, while comparing the distance between the centres of spheres one and two, and the sum of their radii, at d, the distance between the two spheres at their closest possible approach. To determine whether the moving sphere is on course to collide with the static sphere, we look to find the point along the vector *v* where to two spheres meet, otherwise known as the point of contact. Using the known velocity vector, we can calculate the point at which a collision occurs by checking the distance and the radii while the sphere moves along *v*. The distance at which sphere one has travelled along v from the start position to the point of collision is noted as *vc* in the diagram. If the sum of the radii is equal to the distance between the sphere centres, the two will have collided. If the sum of the radii is more than the distance between the sphere’s centres, then they will be overlapping. While trying to implement the collision detection in C++, I came across an unexplained error meaning I could not take the implementation any further. This issue was due to the degrees value. Both the A and the V vectors were correct, and the dot product function was executing correctly as tested using multiple libraries (UKismetMathLibrary, FVector, cmath) and getting the same inexplicably high value for each. I attempted to fix this by performing a conversion from radians to degrees which gave me a more appropriate value of 21, however, this value was subject to drastic change at various points in the simulation and after having to rewrite this code multiple times due to unexplained Unreal Engine C++ errors, and my PC struggling to run Unreal Engine, I did not have time to successfully solve this issue, which in turn, meant I could not implement the sphere to plane collision.



**Collision Response – Sphere to Sphere**

To calculate each spheres reaction to the collision, we must find the angle at which the moving sphere hits the static sphere as this will determine how much force is applied to each sphere and in which direction.



The collision point, noted in the digram as *s,* and its relation to the center of the static sphere, *c,*  produces a vector, *G,* for the direction that the resulting force will be applied to sphere two. *V1* is the velocity vector at which the moving sphere is travelling, and also the direction that the applied force is acting, the direction that sphere two will move after collision will be directly associated with this.

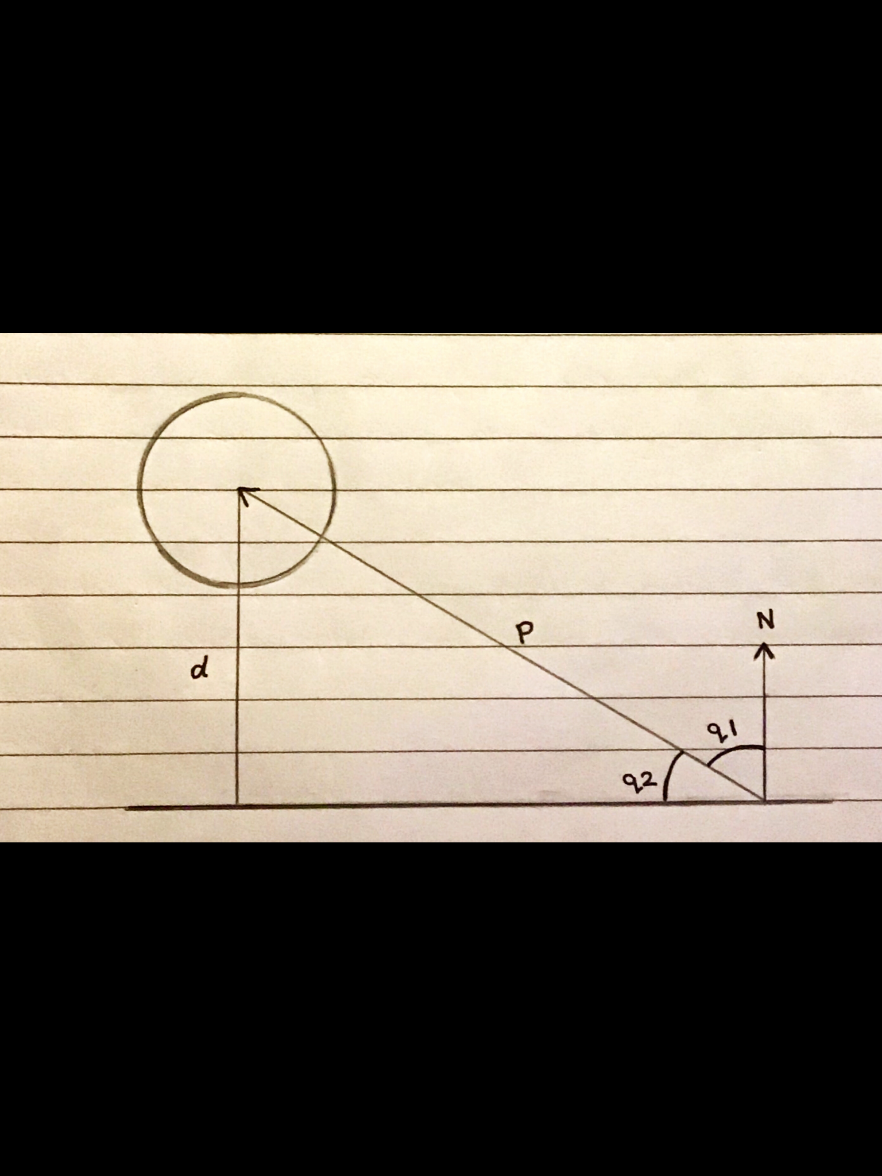
Now the direction in which the force will act in is known, a calculation for the amount of force being applied to sphere two is needed. To do this, we find the angle, *q*, between vectors *G* and *V1*. Cos(*q*) is then calculated to find the proportional force to be applied to the static sphere in the direction of the vector *G*.

In order to calculate collision response accurately, the coefficient of restitution needs to be considered. This is the relation between the spheres final velocity and their initial velocity, after a collision has occurred. So in the case of the simulation, sphere one, our moving sphere, has an initial velocity of {750, 0, 750}, and sphere two, our static sphere, has an initial velocity of {0, 0, 0}. After the event of collision has occurred, the impact force is then distributed between the two spheres based on the angle of impact and the restituation of each sphere. This value can be anywhere between zero, and one, where one is a complete elastic reaction with no loss of kinetic energy[2]. Assuming this is the case for our simulation, we can determine the acceleration, and therefore velocity, of our static sphere as a result of the collision by using the formula Cos(*q*) *F* / *m2.* Using the resulting values, the calculation of sphere one’s velocity post collision is now possible using the conservation of momentum law. This law poses that, the total momentum within a system, in this case, the simulation, will remain at a constant throughout. The momentum is calculated by multiplying the mass of the given object- the moving sphere, by the initial velocity at which it is travelling.

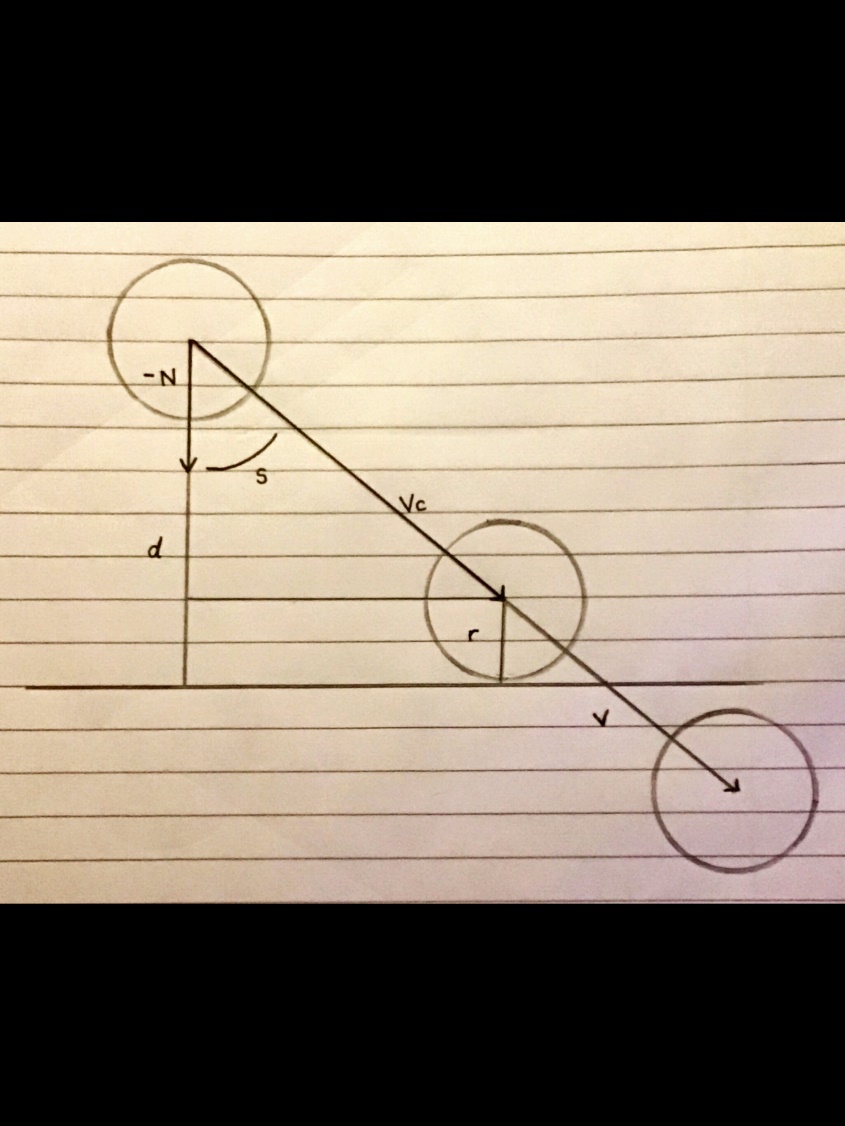
**Collision Detection- Sphere to Plane**

To calculate the sphere to plane collision, first the angle between *N*, where *N* is the surface normal at an arbitrary point on the plane, and *-V*, where *V* is the velocity of the moving sphere, is found to determine whether the sphere is on a collision course with the plane. For this to be true, the resulting angle of this calculation must be less than 90⁰.

As with the sphere to sphere collision detection, we need to find the distance between the moving sphere and its closest point to what it could potentially collide with, in this case, the plane. This is done using trigonometry, using our vector *P*, the vector between the arbitrary point on the plane and the start position of the moving sphere, and the angle between this vector and the plane itself. To do this we multiply the Sin of *q2* by the magnitude of *P*; *d = Sin(q2) |P|.*



The next step is to find the point of contact between the sphere and the plane along the vector V. This will be true when the sphere’s position + radius is equal to the distance to the plane. As with the sphere to sphere collision, the distance at which sphere one has travelled along v from the start position to the point of collision is noted as *vc* in the diagram. We find this value by flipping the normal of the plane at the arbitrary point and moving it to the start position of the moving sphere, noted in the diagram as *-N*. This allows for us to be able to use trigonometry for our calculation of *vc,* thus giving us the point of contact between the moving sphere and the plane.

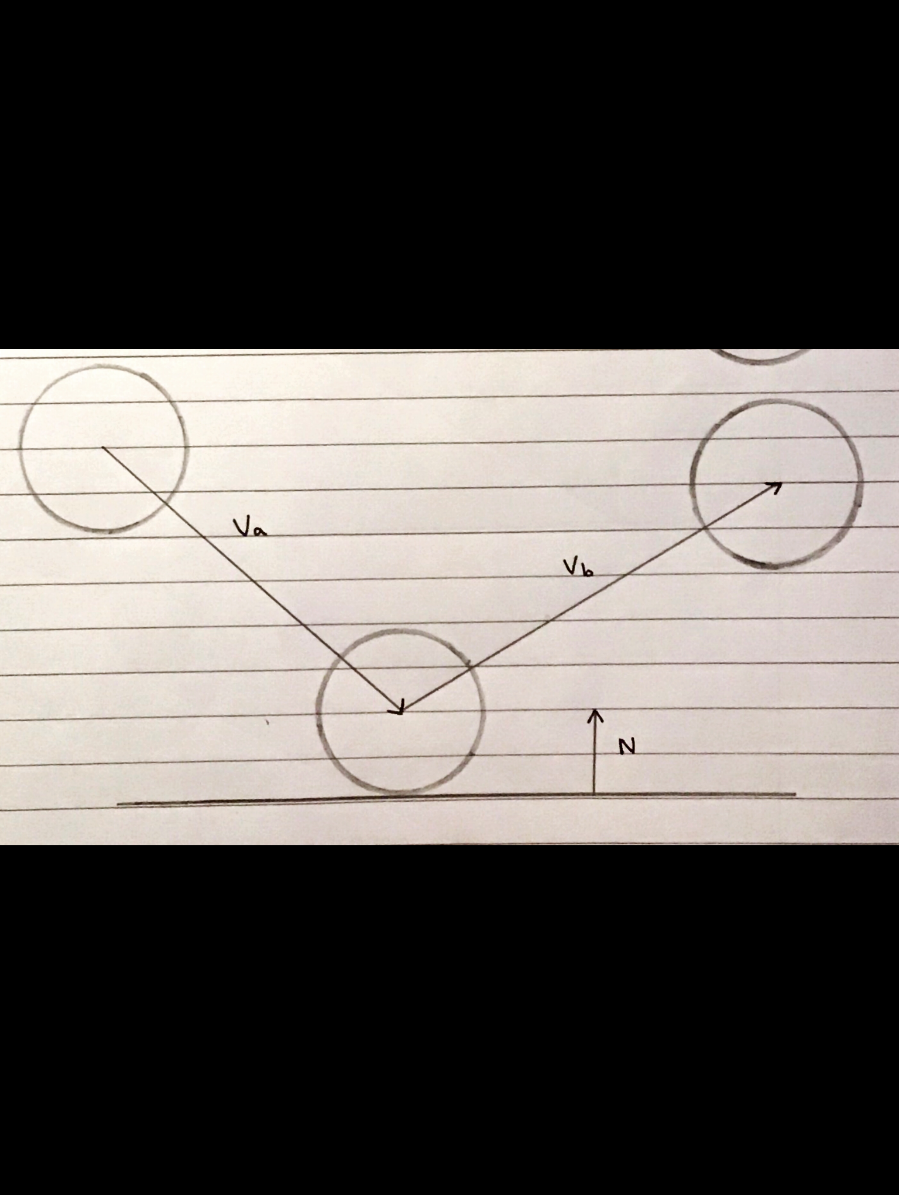


I intended to use the moving sphere for the sphere to sphere collision for the plane collision, however, the technical difficulties and limitations mentioned previously meant that this was not completed, however, the C++ implementation I did manage remains in the code files despite the fact that it’s functionality is not visible in the simulation other than in debugging text.

**Collision Response – Sphere to Plane**

As before, we will assume our object to be collided with, the plane, is static and therefore we only have one velocity and momentum to contend with during the collsion response calculations.

In the diagram below, we can see the sphere moving towards the plane along the velocity vector *Va*, in the case of the simulation, this is just the velocity of the moving sphere, {750, 0, 750}. The spheres post collision velocity is noted as *Vb*. As mentioned before, in a perfectly elastic collision, the restitution will be one, meaning no momentum energy is lost during the collision. Due to this, we can say that |*Va*| = |*Vb*| as the pre collision velocity is equal to the post collision velocity.



This also means that we only need to know the direction of the vectors[7], so we create two unit vectors based on our velocity vectors, by dividing the vectors by their respective lengths as shown below [8].

With the known unit vector , we can now take the plane normal and calculate the ‘bounce’ of the sphere off of the plane after collision using vector geometry[8].

The resulting value is then converted to the appropriate length by multiplying it by the magnitude of *Va,* and applied to simulate a bounce off of the plane surface.

# Part 3: Sandbox – Fluid Simulation

The study of fluid mechanics applies not only to liquids, but to gases as well. These mechanics apply to Newtonian fluids, which, have a principal feature of conforming to Newtons laws when acted upon by an external force such as gravity. These mechanics do not accurately apply to non-Newtonian fluids. These are named as such due to the fact that when under influence from external forces, they do not behave in accordance with Newtons laws as Newtons law of viscosity applies to a fluid whose shear stress and shear rate remain at a constant [1] which is only true in Newtonian fluids.

The basic principles of fluid dynamics consist of the conservation of mass and momentum as well as energy. In order to simulate fluid, we look to computational fluid dynamics which is used to predict the way the particles flow when exposed to a set of predetermined conditions. I will be exploring the particle based method of fluid simulation in this research report and the merits and downfalls with regards to real time simulation and possible use in games.

Fluids, in this sense, can be categorized into one of two groups; Incompressible fluids, meaning fluids where the density is constant such as water, and compressible fluids, meaning fluids where the density changes with pressure, like gases [3].

From here forward, the term ‘particle’ will be used to describe a point in a simulated fluid. The fundamental idea behind particle based fluid simulations, otherwise known as The Lagrangian approach is that we follow all particles and identify changes in the particle properties and apply these within the system [4]. In some ways this is much like the sphere-to-sphere calculations except for this simulation, we need to store a more varied set of data for each particle. The position, velocity, viscosity, pressure, density and temperature are some of the possible data requirements for this type of simulation. In order to determine how these properties effect the particles we must look to the Navier Stokes partial differential equations. These allow us to determine the behaviour of a particle based on its velocity in relation to its other properties; Primarily, density, pressure, viscosity [5], and external forces such as gravity as seen in the equation below [6].

Once the Navier Stokes equations are satisfied for each particle, the simulation will begin to behave as a fluid.

In order to implement this, I would create a STRUCT in which to store all of the relevant particle variables, and a function to execute the Eulerian method. I would then make each particle an instance of the struct, and apply the function to each to allow for easy monitoring of variables over time. For each particle in the simulation, assuming the simulation was of an incompressible fluid, most the variables would need to be recalculated each frame. This allows the particles behaviour to be manipulated to a moderately accurate degree in real time, should the application require it.

This method is not as accurate or complex as other methods such as the field based or Lattice Boltzman methods, but it does deduct from the overall complexity of the simulation, making it more computationally efficient on a small scale and much easier to comprehend. In addition to this, this method can be used in real time which makes it ideal for video games.

Simulation created in Unreal Engine 4.21.

Link to the GitHub Repository: https://github.com/Kittyjelly19/PhysicsSimulation

# References

[1] George H.F., Qureshi F. (2013) Newton’s Law of Viscosity, Newtonian and Non-Newtonian Fluids. In: Wang Q.J., Chung YW. (eds) Encyclopedia of Tribology. Springer, Boston, MA. <https://doi.org/10.1007/978-0-387-92897-5_143>

[2] Clapham, C & Nicholson, J (2009). *The Concise Oxford Dictionary of Mathematics$ The Concise Oxford Dictionary of Mathematics (4 ed.)*. 4th ed. Oxford University: Oxford University Press. coefficient of restitution.

[3] Chegg Study. (). *Incompressible And Compressible Flow.* Available: https://www.chegg.com/homework-help/definitions/incompressible-and-compressible-flow-5. Last accessed 23rd Dec 2020.

[4] Verma, Dr. N. (2012). *Lecture 9: Lagrangian and Eulerian approaches; Euler's acceleration formula.* Available: https://nptel.ac.in/content/storage2/courses/103104043/Lecture\_pdf/Lecture9.pdf. Last accessed 23rd Dec 2020.

[5] Friedlander, S. (2006). *Stability of Flows.* Available: https://www.sciencedirect.com/topics/mathematics/navier-stokes-equation. Last accessed 2nd Jan 2021.

[6] Holton, M. (2020). *Physics Simulation: Fluids.* Available: https://eat.tees.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content\_id=\_3654975\_1&course\_id=\_83466\_1&framesetWrapped=true. Last accessed 2nd Jan 2021

[7] Pranshu Gaba, Margaret Zheng, Nihar Mahajan. (). *Unit Vectors.* Available: https://brilliant.org/wiki/unit-vectors/#:~:text=These%20unit%20vectors%20are%20commonly,to%20V%20with%20unit%20length. Last accessed 3rd Jan 2021.

[8] Holton, M. (2020). *Physics Simulation: Sphere-Sphere and Sphere-Plane Collision Reactions – Useful Referents.* Available: https://eat.tees.ac.uk/webapps/blackboard/content/listContent.jsp?course\_id=\_83466\_1&content\_id=\_3615109\_1. Last accessed 2nd Jan 2021.