Optimising the Computational Simulation of Mechanical Models: Designing a Real-Time Physics Engine

Jordan Spooner

July 15, 2015

Contents

1 Vectors

The first task in the development of a physics engine is to implement a class for vectors, which have significant applications including (but not limited to) defining the co-ordinates of any point in 3D space, as well as changes in space.

When defining a change in position, using vectors, it can be expressed as the product of two components:

$$\mathbf{a} = d\mathbf{n}$$

In this case, the vector, $\mathbf{a}$ is expressed in terms of its magnitude, d (found by Pythagoras' theorem and notably a scalar value), multiplied by a unit vector, which has magnitude 1 in the direction of motion.

The vector, $\mathbf{a}$ may also be multiplied by a scalar (changing the magnitude of a but having no influence on its direction) or added to (or subtracted from) another vector (which will give you the resultant of the two vectors).

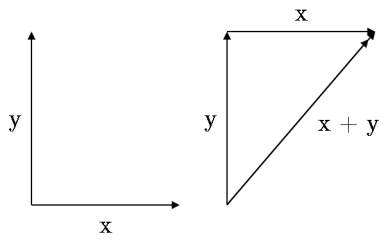

Figure 1: A Geometric Representation of Vector Addition

Finally, we need to implement a function to find the scalar product of two vectors, say $\mathbf{a}$ and $\mathbf{b}$, which is given by

$$\mathbf{a} \bullet \mathbf{b} = \mathbf{a}_x\mathbf{b}_x + \mathbf{a}_y\mathbf{b}_y + \mathbf{a}_z\mathbf{b}_z$$

The major application of the scalar product is to find the angle between two vectors. It is fairly easily proven, using Pythagoras Theorem, that:

$$\theta = \arccos\left(\frac{\mathbf{a} \bullet \mathbf{b}}{|\mathbf{a}| \times |\mathbf{b}|}\right)$$

where θ gives the angle between the two vectors.

I also chose to include some other functions, which include the vector and component products of two vectors, as well as a function to find the square of the magnitude (since we can then avoid calling the `sqrt` function, and in many cases we will find the square of the magnitude is sufficient for calculations).

The code that deals with these functions is found in `include/engine/core.h`.

2 Particle Kinematics

We can use Newton's Laws of Motion, which describe, to great accuracy, the motion of a point mass (a particle), in order to simulate a great proportion of movements within our

engine. For each particle, we of course need to define its position, velocity and acceleration, all of which are vectors (hence the importance of first defining a class for vectors).

Newton's First Law states that an object will continue with constant velocity unless acted upon by an external force. In reality, however, this is rarely ever the case and in the majority of cases, objects will decelerate due frictional forces. Hence we should also add a fourth property to the particle class, which roughly accounts for the effects of drag (and also prevents objects from accelerating due to mathematical inaccuracy). This property is called `damping` in my engine, and simply removes a proportion of a particle's velocity for each subsequent calculation.

Furthermore, Newton's second law states that the resultant force acting upon an object produces acceleration that is proportional to the object's mass. Hence, we should also include the mass of the particle, meaning we can calculate the acceleration, $\mathbf{a}$, using:

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

It should be clear from the given equation that the mass of an object should never be 0. Furthermore, there may be cases where an infinite mass is required (some objects will be immovable). Hence, we instead store the inverse of the mass, to rectify these two problems.

We also need to take into account the effect of gravity. Since the acceleration due to gravity, g, is constant (roughly equal to $9.8ms^{-2}$), there is no need to calculate the force and then convert it back into an acceleration. Instead, g can just be added directly to the acceleration of each particle. Furthermore, this will allow the value of g to easily be changed, which may be required for certain simulations.

To calculate the velocity, we of course need to find $\int a \; dt$. Meanwhile, to calculate the change in position (displacement), we, using our previously found velocity, find $\int v \; dt$.

For this we use the semi-implicit Euler integration method. This means that we take the duration of each frame, and add to the previous velocity the acceleration during that frame:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \Delta t$$

For the new position, however, we use the previously calculated (future) velocity, since this helps to stabilise the displacement[1]:

$$\mathbf{p}_{n+1} = \mathbf{p}_n + \mathbf{v}_{n+1} \Delta t$$

The code responsible for these calculations can be found in `include/engine/particle.h` and `src/particle.cpp`, and with the engine in its current state, the `projectile` simulation will work.

3 Forces

So far, our physics engine is able to simulate gravity and (crudely through the `damping` property) drag. There are of course many other forces that we may wish to simulate (including

[1] Alternatively, one could implement the (simpler and more intuitive) explicit Euler method - however, it is both numerically unstable without a high sampling rate and also rather inaccurate. (See [6]). If we required more a more accurate simulation (say for a scientific or mathematical investigation), we would be more likely to choose a high-order implementation of the Runge-Kutta methods (of which the Euler method can be thought of as simply a first-order implementation). (See [4]).

a more sophisticated and accurate versions of gravity and drag).

We can use D'Alembert's Principle[10] for combining multiple forces. It allows us to represent the resultant of several forces as a single force, found from their vector sum:

$$\mathbf{F} = \sum_i \mathbf{F}_i$$

This is very easily implemented in our engine by creating a vector object that holds the sum of all forces. Each time through the main update loop, we can then add any new forces that arise.

Since the range of forces that we might need to simulate may be extensive, we can implement a standard interface for all forces, and then deal with specific details later on (this is called polymorphism and is the main benefit of using an object-oriented programming language). The interface, the "force generator", is responsible for adding new forces to our objects and updating them, as well as keeping a registry to store all the different kinds of forces we may wish to implement.

3.1 Gravity

Before this point, we have always added gravity through applying a constant acceleration. Although this is a perfectly valid method for doing so, we now have the ability to introduce gravity as a force (a much cleaner and more sophisticated implementation). We can introduce a class to generate gravity, such that it has the sole property of the acceleration due to gravity, and calculates the force given the mass of the object:

$$\mathbf{F} = mg$$

A single instance of this class could of course be shared among any number of objects.

3.2 Drag

Our force generator class is also capable of adding a drag force. Although in reality, drag can be found from

$$F_D = \frac{1}{2}\rho\mu^2 C_D A$$

where ρ gives the density of the fluid, μ the flow velocity relative to the object, A the reference area, and C_D the drag coefficient, in reality this is too complex and computationally demanding to be simulated accurately in real time.

Hence in many engines, we find the drag equation is often significantly simplified to give:

$$\mathbf{F}_D = -\hat{\mathbf{v}}(k_1 |\mathbf{v}| + k_2 |\mathbf{v}|^2)$$

where $\hat{\mathbf{v}}$ gives a unit vector in the direction of the velocity, $|\mathbf{v}|$ gives the speed of the object, and k_1 and k_2 are both constants that determine the strength of the drag force.

Hence, where there is a k_2 value, the drag will grow more quickly at higher speeds, as in the case with aerodynamic drag (where doubling the speed will cause the magnitude of the drag to almost quadruple).

I should note that despite the new implementations for gravity and drag using our "force generator", I will continue to use damping for drag and add apply gravity directly as an acceleration in my engine. This is because these will be much more efficient and given our uses for the engine, a more complex simulation of gravity and drag is not required.

4 Springs

Springs and spring-like objects have many applications in a physics engine. They allow us to connect objects, in addition to simulating soft (deformable) objects, and can also be used for a number of (otherwise very complex) effects, such as cloth effects and for fluids (e.g. water ripples).

We can use Hooke's Law to implement springs and spring-like objects in our engine, which states that:

$$F = kx$$

where k gives the spring constant, x the extension, and F the force.

However, we need to manipulate this law so that we can implement it in our (3-dimensional) physics engine, giving:

$$\mathbf{F} = k(|\mathbf{d}| - l_0)\hat{\mathbf{d}}$$

where $\mathbf{d}$ gives the distance between the two ends of the spring in vector form.

Although real springs will have a limit of elasticity (beyond which permanent deformation will occur), I choose not to implement this in my engine since it is not required in any of my simulations.

We can introduce springs to our engine through the interface we developed for forces earlier. The generator will obviously require the length of the spring and the spring constant, in addition to a pointer that points to the object to which the spring is connected (or a defined point in space). This means that, unlike for our gravity and drag force generators, unfortunately we cannot reuse this instance for multiple objects (instead a we must define new properties for each given spring/ spring-like object).

4.1 Buoyancy

The force resulting from the buoyancy of an object can quite easily be found using Archimedes' Principle, which states that it is simply given by the weight of the fluid that has been displaced.

However, using this method exactly would require us to calculate the exact volume of every object suspended in a fluid (which, given irregular shapes, may be complex). This kind of calculation is not required as it is unlikely we will need to find the buoyant forces on objects to a high accuracy. An estimation using springs will likely be sufficient.

We can implement this using Hooke's Law by stating that an object near to the surface of the liquid will experience a force that is proportional to its depth. In reality, this is only true for a partially submerged object with a uniform cross-sectional area. When an object is fully submerged, it will instead experience a constant force, and we can implement this by defining a fixed force that acts on an object should it be below a given depth. Likewise,

when the object is above a given depth (such that is likely completely unsubmerged), we need not apply any force.

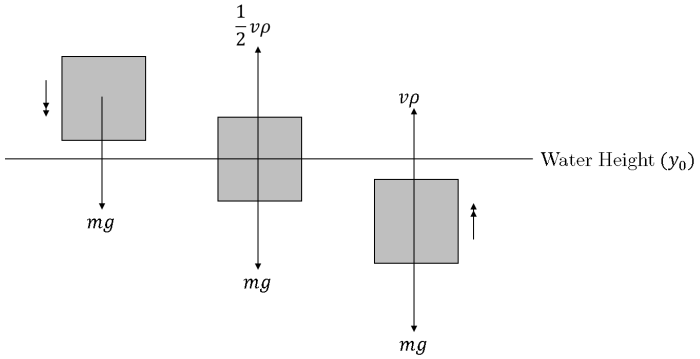

Figure 2: The Buoyancy Force in its Different Cases

We can therefore define the force as:

$$F = \begin{cases} 0 & \text{where } d \leq 0 \text{ (completely unsubmerged)} \\ dv\rho & \text{where } 0 < d < 1 \text{ (partially submerged)} \\ v\rho & \text{where } d \geq 1 \text{ (completely submerged)} \end{cases}$$

where d gives the proportion of the object submerged, v gives the volume of the object, and ρ gives the density of the liquid. We can calculate the value of d given the y value of the object, y, the y-value of the liquid plane, y_0 and h is the height of the object (i.e. $y_0 - s$ gives the value below which the object is considered to be completely submerged):

$$d = \frac{y - (y_0 + s)}{2s}$$

4.2 Stiff Springs

It would be possible to simulate all collisions using stiff springs, and many physics engines implement this method (referred to as the "penalty method"). However, it is very difficult to do so. If the spring constants are set to small values, then everything will bounce and the scenarios will look unnatural. Meanwhile, if very high values are chosen, there will be a variety errors, from objects accelerating at significant rates, such that they disappear from view almost instantly, to crashes caused by numerical errors as a result of such large values.[2] For this reason, I have not implemented this functionality in my engine.

[2]The reason for this is that as we increase the spring constant, the force applied will of course also increase. We will therefore reach a certain stiffness, where the force applied to the spring will be great enough such that it passes its rest length before the next update. If it is now more compressed than it was originally extended, it will accelerate even faster but now in the opposite direction. At the next update, the spring therefore has a greater extension than it did originally! Clearly, as this cycle continues, an ever greater force will be produced and so the end of the object's position will tend towards a position infinitely far away from its original position. Of course this scenario becomes significantly more likely with a slower frame rate. If we need to implement stiff springs, we could use simple harmonic motion, such that the position of one end of the spring obeys the equation:

$$\mathbf{a} = \frac{k}{m}\mathbf{p}$$

where k gives the spring constant as before. We can then solve this equation to give an expression that links the object's position to its time, and generate the required force according to this. (Although even this method still has several limitations).(For the best possible implementation, which involves damped harmonic motion, see [12]).

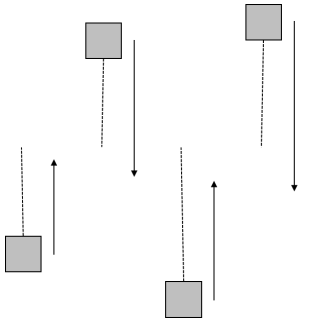

Figure 3: Stiff Spring over Time

5 Collisions

The final feature I have chosen to implement in my engine is collision detection and resolving.

The most common method for implementing collision detection is to introduce a function that looks through a set of objects and checks whether any two objects are interpenetrating.

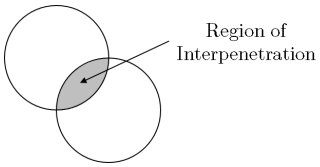

Figure 4: Widely-Used Method for Collision Detection

It then has to first resolve the interpenetration before resolving the collision itself (otherwise, if the approach speed is very small, the two objects may become stationary at a point where they are still interpenetrating). Although this is a very simple process with one object interacting with the scenery (we just move the object back in the direction of the contact normal until it no longer overlaps), it becomes very much more complex when we begin to involve multiple objects. Take, for example, a small box colliding with the surface of a planet. If we chose to move each of the objects the same amount to resolve the interpenetration, we would obviously see an unrealistic movement of the planet. Hence instead we instead take the masses of the objects into account, giving:

$$\Delta \mathbf{p}_a = \frac{m_b}{m_a + m_b} d\mathbf{n}$$

and

$$\Delta \mathbf{p}_b = -\frac{m_a}{m_a + m_b} d\mathbf{n}$$

where d gives the interpenetration 'depth' and $\mathbf{n}$ gives the direction of the contact normal.

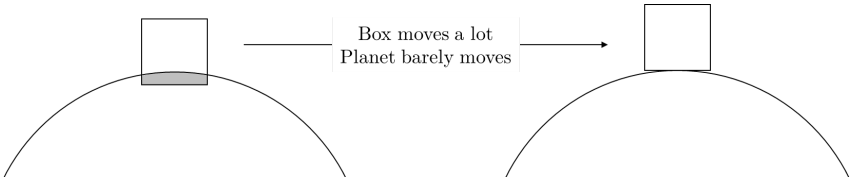

Figure 5: Resolving Interpenetration

There are two obvious limitations to our collision detection method so far. The first is a result of high-speed collisions. Where objects are moving very quickly towards each other (especially when combined with a low frame rate), they may simply pass through each other without a collision being detected (since the frame only refreshes after they have passed each other). Even if a collision is detected, if the objects have passed halfway through each other, then they will be seen to have a positive separation speed and so no impulse is generated. This issue is particularly difficult to resolve, and as such is beyond the scope of this paper.

The second limitation occurs when we have two particles which are at rest or with very small velocities. Take a particle resting on the ground. Each frame, a downwards acceleration will be provided due to gravity. Hence the particle interpenetrates with the ground, and an impulse will be applied. This will lead to an unnatural movement of the particle such that it will vibrate and sometimes even experience a significant upwards acceleration. This issue is much easier to solve. We simply work out the velocity produced by acceleration and remove it from the velocity we use in our collision resolution. A more sophisticated approach would be to simulate the normal reaction force of the ground on any object. However, when considering irregularly-shaped objects/ ground, one can see this very quickly becomes a complex and difficult task.

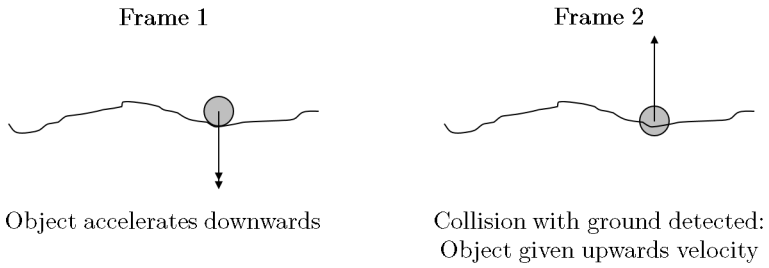

Figure 6: Collision Detection and Resolving on Objects at Rest

There are multiple widely-used methods for implementing collision resolving (such as the penalty method, as mentioned earlier), however I will use an impulse-based solution. First of all, we assume that momentum will always be conserved:

$$m_a\mathbf{u}_a + m_b\mathbf{u}_b = m_a\mathbf{v}_a + m_b\mathbf{v}_b$$

We can then calculate the resulting (separation) velocities following a collision using the equation:

$$v = eu$$

where v gives the separation speed (following the collision), u gives the approach speed (prior to the collision) and e gives the coefficient of restitution between the two colliding objects. We can alter this coefficient to give more elastic ($e \approx 1$) or inelastic ($e \approx 0$) collisions.

We now use the new separation velocity we found earlier to calculate new velocities given the masses of the involved particles - by equating impulses. The impulse is simply the change in momentum of an object, and it is calculated by the equation:

$$j = m\mathbf{v} - m\mathbf{u}$$

We also need to take into account the direction of the impulse produced. Where we have two particles, we can find the contact normal by considering their positions:

$$\hat{\mathbf{n}} = \widehat{\mathbf{p}_a - \mathbf{p}_b}$$

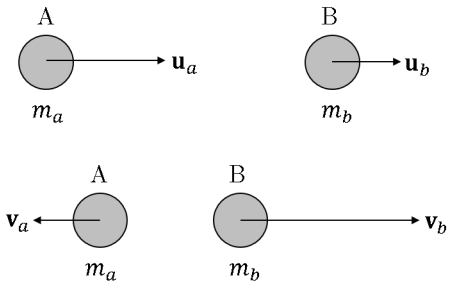

Figure 7: Impulse Method for Collision Resolution

We keep collisions separate to our other force calculations, instead adding an instantaneous change in velocity each time a collision is detected:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \frac{j}{m}$$

5.1 Rods and Cables

We can also use the collisions system we have built to introduce rods and cables to our engine. A rod keeps two particles at a given length from each other whilst a cable simply gives a maximum bound of how far apart they can be. The code for implementing these connectors can be found in `pconnector.h` and `pconnector.cpp`.

6 Simulations

We now have a fairly complete real-time mass-aggregate physics engine[3], capable of simulating a wide variety of scenarios. Our final task is to build the actual simulations, so that we can test graphically the engine behaves as expected. A lot of the code required to do so is generic, involving the rendering of 3D graphics (for which I have used OpenGL) and the calculation of the property, `duration`, which gives the time between two frames.

This leaves us only with the task of building the actual applications and giving each of their objects values for the required variables:

6.1 `projectile`

The first simulation, `projectile`, was largely built to test the particle engine developed in Section 2. The code is not especially complex, and there is only one object (apart from the application itself), `projectile1`.

Hence the main task was to provide values for the several variables that are required by the `particle` class. In this case, I have greatly exaggerated the value of g to $20ms^{-2}$. Although it is possible to use the actual value of g, $9.8ms^{-2}$, this is rarely ever used in

[3]Sadly, despite having written over 2500 lines of code across more than 20 different files, I have not yet implemented any physics for rotation or rigid bodies, so more sophisticated simulations involving, for example, objects toppling are not (yet!) possible with my engine.

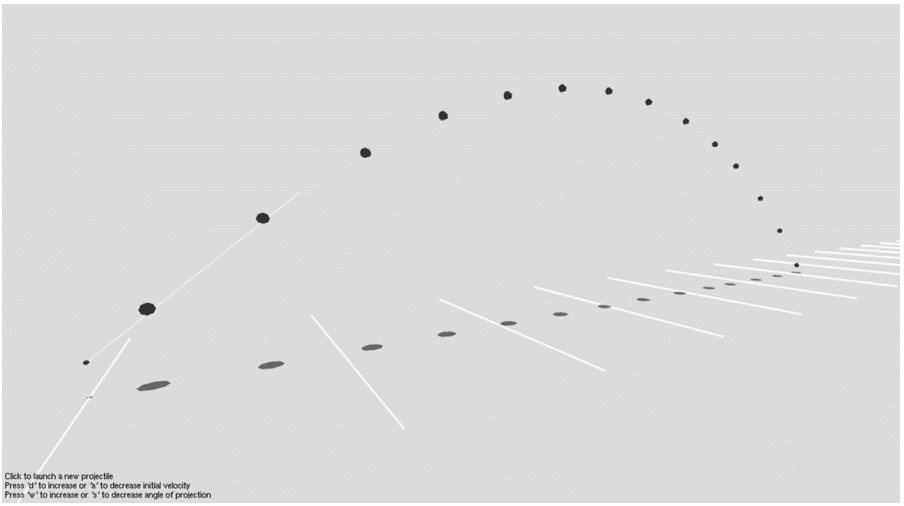

Figure 8: Projectile Simulation

simulations, especially in games where the smaller acceleration can make the simulation less exciting. Given the significant number of approximations and short-cuts we have made during the development of our physics engine so far (in addition to many other factors that we have simply ignored), it would be difficult to achieve a truly realistic simulation, and so in this case I simply chose to select the value of g that gives the best-looking simulation. I have given the projectile a mass of $200kg$ (although this has no effect in this particular simulation - it will be required later for collisions) and a damping factor of 0.99 (i.e. 99% of the projectile's velocity is retained each second), as the air resistance against the projectile is likely to have a very small effect on its acceleration. The angle of projection and initial velocity are defined by the user. This is achieved by the use of a unit vector, multiplied by a magnitude. The user may define the magnitude (within reasonable bounds) and the y-value of the unit vector. The z value is then simply calculated to produce a unit vector, using Pythagoras' Theorem (given that x is kept constant).

6.2 `springForces`

This simulation demonstrates the spring forces explored in section 4.

6.3 `bridge`

The `bridge` simulation makes use of the `pcollision` code (including the use of collisions for cables), discussed in section 5.

All of the source code for this project, in addition to executables for the simulations, annotated slides, and other resources can be found at ***`juddcompsci.tk`****.*

References

[1] Song Ho Ahn. *OpenGLProjection Matrix*. URL: http://www.songho.ca/opengl/gl_projectionmatrix.html.

[2] Jonathan Brenner. *PhysicsEngine: C++ Physics Engine in OpenGL*. URL: http://github.com/JonathanBrenner/PhysicsEngine.

[3] Prof. Mark Claypool. *Video Game Technologies: Game Physics*. URL: http://www.di.ubi.pt/~agomes/tjv/teoricas/10-physics.pdf.

[4] Richard Fitzpatrick. *Runge-Kutta Methods*. URL: http://farside.ph.utexas.edu/teaching/329/lectures/node35.html.

[5] Stanley B. Lippman. *C++ Primer, 2nd Edition*.

[6] Zdzislaw Meglicki. *Euler Method*. URL: http://beige.ucs.indiana.edu/B673/node50.html.

[7] Ian Millington. *Game Physics Engine Development*. URL: http://shiba.hpe.cn/jiaoyanzu/wuli/soft/Physics/Game_Physics_Engine_Development.pdf.

[8] Newcastle University. *Physics Lecture Slides*. URL: http://research.ncl.ac.uk/game/mastersdegree/gametechnologies.

[9] M. Sadraey. *Drag Force and Drag Coefficient*. URL: http://faculty.dwc.edu/sadraey/Chapter%203.%20Drag%20Force%20and%20its%20Coefficient.pdf.

[10] Dr C T. Whelan. *The Principles of Dynamics*. URL: http://www.maths.cam.ac.uk/studentreps/ftp/podv0.1.pdf.

[11] Ted Yin. *A Brief Intro to Game Physics*. URL: http://acm.sjtu.edu.cn/ppca/w/images/c/c0/Brief_intro_game_physics.pdf.

[12] Multiple Authors. *Damped Harmonic Motion*. URL: http://philschatz.com/physics-book/contents/m42246.html.

[13] Multiple Authors. *StackOverflow*. URL: http://stackoverflow.com.