

7 Physical Simulation

7.1 Aims

Simple Newtonian mechanics and a little numerical integration will provide suitably realistic physical simulations for most games. The underlying maths requires a basic knowledge of vectors and calculus. We will explore the 2D cases in this module, although the ideas all extend readily into three dimensions. We will look at some simple examples of how forces interact with a body. The module covers the summation of forces on a body into a single force and a single torque. The module then deals with the calculation of movement of the body subject to a single force and torque. Finally we have a quick look at collisions.

7.2 2D Vectors

In living in a 2D world, forces, accelerations, velocities have two components: a magnitude and a direction. That means that you need to treat these components as vectors. Usually, this means that you break down the magnitude and direction into two orthogonal magnitudes, the x component and the y component. Once you have the vectors represented as components, operations such as addition and subtraction are trivial. The other operation that is important is the dot product. The dot product is found by simply multiplying each of the components together and adding them. You won't have to worry about much other vector math than this, so make sure you understand at least the basics.

7.3 Dynamics

Often the simplest way to think of the dynamics problem is that interactions between objects happen by forces. The forces can be resolved into a single overall effect on the mass that can then be used to calculate motion.

7.3.1 Forces

Most interactions that affect motion can be described as forces (or their rotational cousins: torques). Forces occur from effects like gravity, contact between surfaces, motors, springs and friction. Near a large mass, the force from gravity is usually considered constant: mg , which makes calculations of motion pretty simple. When

calculating gravity effects for situations like orbits, gravity is proportional to $\frac{m_1 m_2}{r^2}$.

This means that the magnitude of the force is affected by the positions of the moving masses. This situation appears more complex as the motion affects the forces. You'll see this same kind of feedback effect from springs (where the force is proportional to the displacement), and friction (where the force is proportional to the velocity). The trick is to consider only the instant: what is the force NOW, based on the current position and velocity? Once you have the instantaneous force, you can work out the new velocity and position at the next instant, then recalculate the force. Calculating the total force on a mass is accomplished by vector addition. Just make sure you have all the components of the forces in the same units and in the same coordinate frame. After that you can simply add the components.

7.3.2 Finding the Position and Velocity

From the force calculation, it is trivial to calculate the acceleration from $F = ma$, remembering that the force is a vector, and so will the acceleration be. Now acceleration is the rate of change of velocity, $a = \frac{\Delta v}{\Delta t}$. From a code perspective this means:

```
vx_new = vx_old + ax * delta_t;
vy_new = vy_old + ay * delta_t;
```

When you consider that velocity is the rate of change of position, $v = \frac{\Delta s}{\Delta t}$, then the next line of code is much the same:

```
x_new = x_old + vx_new * delta_t;
y_new = y_old + vy_new * delta_t;
```

Surely it can't be that simple? Well, if you're worried about really accurate dynamic calculations then it's a little more complicated but otherwise, that's it.



Create a simple moving square that can be subject to forces up, down, left, right from the arrow keys. Add a friction force that is proportional to speed. Now add a strong force that attracts the square to the centre of the screen. See if you can make your square orbit the centre of the screen.

7.4 Collisions

It can be tricky to calculate the forces involved in a collision. Consequently, the best way can often be to simply reset the velocities based on known collision properties. You've done this kind of thing before when bouncing blitted objects off the side of the screen. This is easy to do when the reflective surface is along the coordinate frame that is used to resolve the components of the velocity vector. You can use this same trick for any angle surface if you swing the velocity vector into the coordinate frame of the surface. We can do this by taking the dot product of the velocity vector with the tangent and normal vectors of the surface as the text suggests, or you can use transformation matrices as we did in the 2D Graphics module.