Introduction to Particle Simulations

Daniel Playne

Particle Simulations

Another major class of simulations are particle or n-body simulations.

In these simulations the state of the system is represented by particles.

Particle Simulations

Generally each particle is represented by a set of fields. These are often:

a position

a velocity

a mass

a radius

and possibly:

a rotation

an angular velocity

Particle Simulations

These particles can represent different entities depending on the simulation. This could be:

atoms

molecules

dust particles

snooker balls

asteroids

planets

galaxies

Particle Simulations

These particles all behave in fundamentally similar ways. This behaviour is based on Newton’s Laws of Motion.

These three laws are the basis for classical mechanics.

Particle Simulations

First Law: Every object continues in its state of rest, or of uniform motion in a straight line, unless compelled to change that state by external forces acted upon it.

Particle Simulations

Second Law: The acceleration **a** of a body is parallel and directly proportional to the net force **F** acting on the body, is in the direction of the net force, and is inversely propotional to the mass **m** of the body.

F = ma

Particle Simulations

Third Law: When two bodies interact by exerting force on each other, these forces are equal in magnitude, but opposite in direction.

Particle Simulations

Simple case – Hockey Pucks

Each particle represents a hockey puck sliding around on the ice. This is simple 2D system is relatively easy to simulate.

Particle Simulations

First step – Vectors.

In this sense a vector is referring to a mathematical or Euclidean vector. These consist of co-ordinates in space. We usually talk in vectors because it makes it easier to define something in 1D, 2D, 3D, 4D...

Particle Simulations

For our 2D hockey pucks a vector is as simple as (x,y). We need two vectors for each particle – position and velocity.

In this case we are going to assume that every hockey puck has the same mass and the same radius.

Particle Simulations

In this case each particle has a position *p* and a velocity *v* with mass *m=1* and radius *r=1.*

To simulate the particles moving, we must calculate a new position for the particle after some period of time has passed. This is the time-step of the simulation.

Particle Simulations

This can be written simply as:

$p_{t+h} = p_t + v_t{*}h$

This is known as the Euler integration method. Which is suitable for this type of particle simulation.

Particle Simulations

Our algorithm to update a particle simulation is as simple as:

- main loop
  - for all particles
    - ‘move’ particle
- repeat

Particle Simulations

Our algorithm to update a particle simulation is as simple as:

```
for(int i = 0; i < N; i++) {
   p[i] = p[i] + v[i] * h;
}
```

Particle Simulations

Unfortunately this is a rather boring simulation. Each particle moves in a straight line and will never stop. To make it slightly more interesting we will have to enforce collisions.

Collisions occur when a ‘hard’ particle hits into another ‘hard’ object.

Particle Simulations

The easiest collisions are particles colliding with immovable objects.

In this case our hockey pucks bouncing off the walls of the hockey rink.

Particle Simulations

In any collision system there are two parts, the collision detection and the collision response.

To check to see if a puck has collided with the sides of the rink, simply check to see if the puck is outside the bounds.

Particle Simulations

Once a collision has been detected the system must respond to the collision. For our hockey pucks, simply reverse the velocity in the direction of the collision.

Particle Simulations

- main loop
  - for all particles
    - ‘move’ particle
    - if ‘collision’ with boundary
      - respond to collision
- repeat

Particle Simulations

Our algorithm to update a particle simulation is as simple as:

```
for(int i = 0; i < N; i++) {
   p[i] = p[i] + v[i] * h;
   if(p[i].x-r < 0 || p[i].x+r > width) {
      v[i].x = -v[i].x;
   }
   if(p[i].y-r < 0 || p[i].y+r > height) {
      v[i].y = -v[i].y;
   }
}
```

Particle Simulations

More complex objects will require more complex collision detection and response systems.

For example – hockey pucks bouncing off each other.

Particle Simulations

This is now significantly more complicated. After each time-step the simulation must compare every pair of particles to see if they have collided by calculating the distance between them and checking to see if that distance is less than the combined radius of the hockey pucks.

Particle Simulations

- main loop
  - for all particles
    - ‘move’ particle
    - if ‘collision’ with boundary
      - respond to collision
  - for all particles
    - for all other particles
      - if ‘collision’ between particles
        - respond to collision
- repeat

Particle Simulations

```
...
for(int i = 0; i < N; i++) {
   for(int j = 0; j < N; j++) {
      if(distance(p[i],p[j]) < r*2) {
         collision!
      }
   }
}
```

Particle Simulations

Responding to a collision between particles is more complicated than an immovable wall.

In our example the mass of both particles is the same which makes the collision easier to calculate.

Particle Simulations

Calculating a collision in one dimension is simple if the masses are the same:

$v_1 = u_2$

$v_2 = u_1$

where:

u_1 and u_2 are the initial velocities

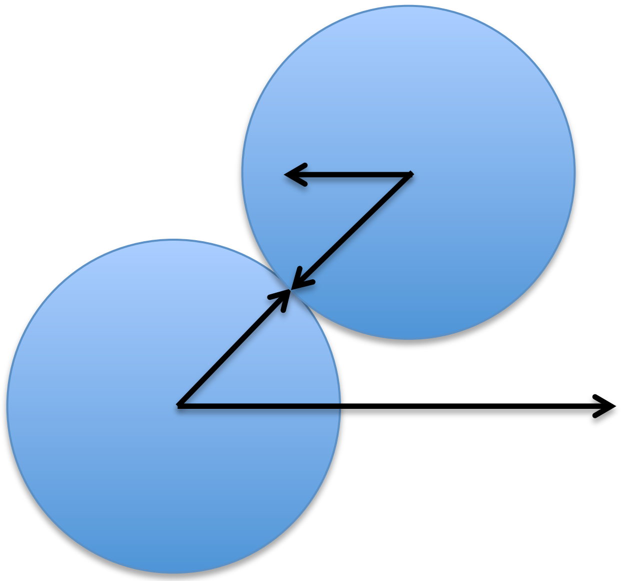
v_1 and v_2 are the final velocities

Particle Simulations

In two dimensions this is not as simple. The velocities of the particles must be split into the components that are in the direction of the collision.

Particle Simulations

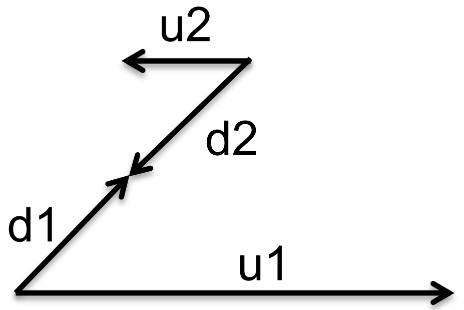
The ‘component’ of the velocity to be used in the collision is found from the dot product of the velocity and the unit vector.

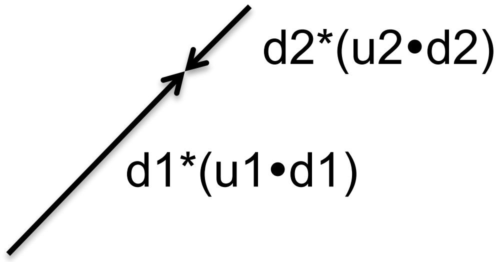


Particle Simulations

This component is the part used in the collision:

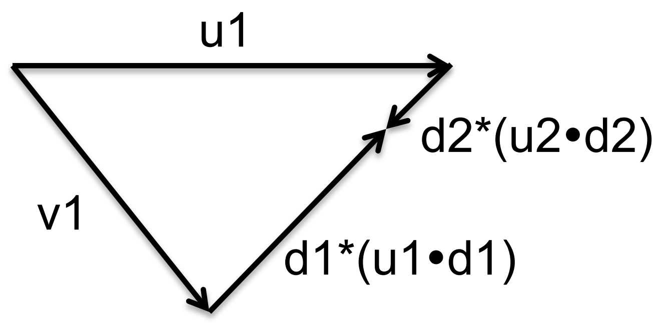
u1•d1 = u1.x*d1.x + u1.y*d1.y

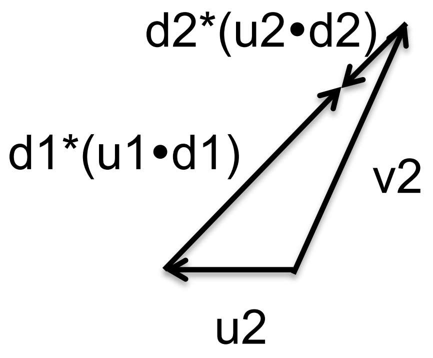




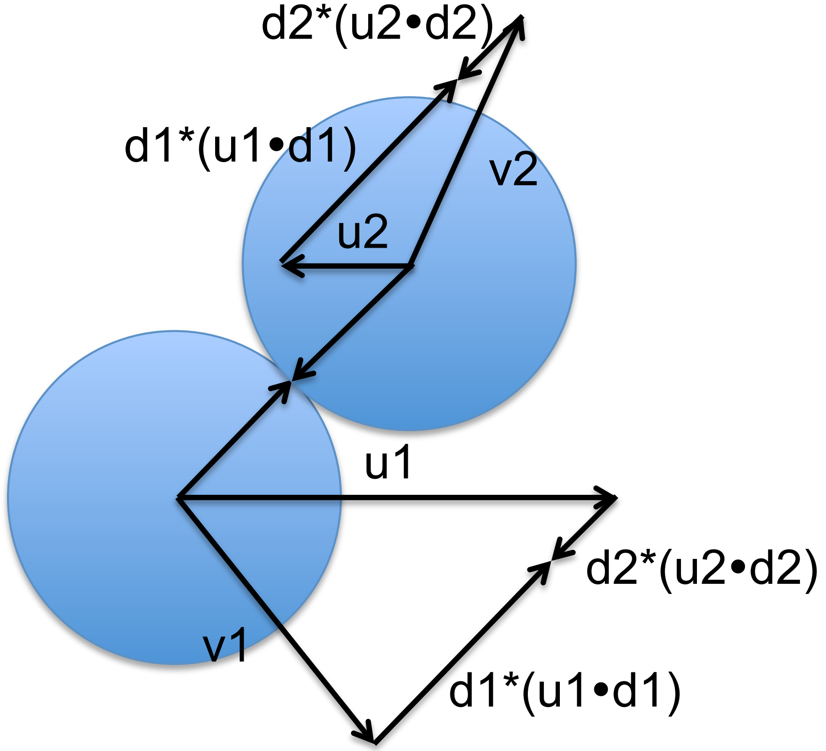
Particle Simulations

This is how the new value can be calculated.





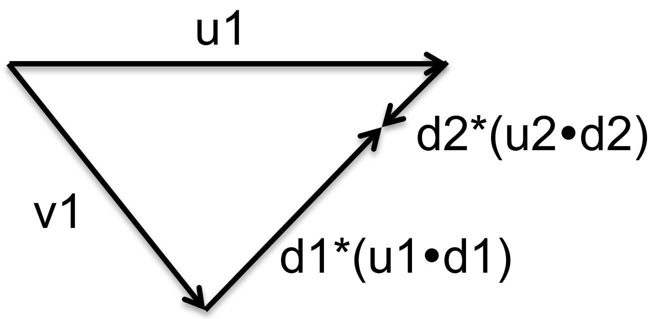
Particle Simulations

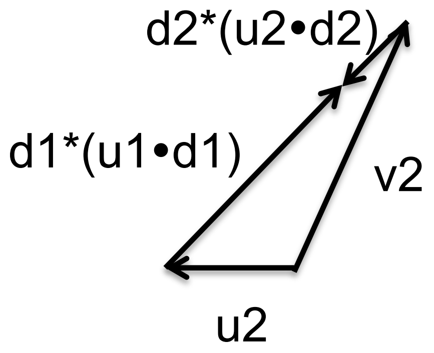


Particle Simulations

This calculation can be written as:

```
v1 = u1 – d1*(dot(u1,d1)) + d2*(dot(u2,d2))
v2 = u2 – d2*(dot(u2,d2)) + d1*(dot(u1,d1))
```





Particle Simulations

```
...
for(int i = 0; i < N; i++) {
   for(int j = i+1; j < N; j++) {
      if(distance(p[i],p[j]) < r*2) {
         v1 = u1 – d1*(dot(u1,d1)) + d2*(dot(u2,d2))
         v2 = u2 – d2*(dot(u2,d2)) + d1*(dot(u1,d1))
      }
   }
}
```

Particle Simulations

There is a certain degree of error in this calculation. This is caused by the fact that the particles are allowed to move inside each other before the collision occurs.

Particle Simulations

This can be solved by ‘reversing’ time to the point of the collision. Calculating the new velocities and then stepping the simulation back to the present time.

Particle Simulations

The calculation for this is:

```
if(distance(p[i],p[j]) < r*2) {
   t=(r*2-distance(p[i],p[j]))/dot(u1,d1)+dot(u2,d2))
   p1 = p1 – u1*t
   p2 = p2 – u2*y
   v1 = u1 – d1*(dot(u1,d1)) + d2*(dot(u2,d2))
   v2 = u2 – d2*(dot(u2,d2)) + d1*(dot(u1,d1))
   p1 = p1 + v1*t
   p2 = p2 + v2*t
}
```

Particle Simulations

This can lead to a new problem, because the particles are moving during the ‘collision’ phase. This movement can cause additional collisions.

In order to solve this problem the collisions must be resolved in the order they occur. The algorithm becomes:

Particle Simulations

- main loop
  - for all particles
    - ‘move’ particle
    - if ‘collision’ with boundary
      - respond to collision
  - detect collisions
  - while collision has occurred
    - find and resolve first collision
    - detect collisions
- repeat

Particle Simulations

In this way the collisions are always resolved in the order they would have occurred and there is no error introduced by our system.

Particle Simulations

Example