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

Jordan Spoooner

July 16, 2015

What is a Physics Engine?

Engine: Computer software that performs a fundamental function, especially as part of a larger program

- Moves objects
- Detects collisions
- Resolves collisions

‘Moves Objects’

Newton’s Laws of Motion

- A body will remain at rest or continue with constant velocity unless acted upon by an external force
- The acceleration of the body is proportional to the resultant force acting upon the body $\Rightarrow \mathbf{a} = \frac{\mathbf{F}}{m}$
- For every action there is an equal and opposite reaction

Storing the Data

Particles - No volume, no angular motion

- Position, Velocity, and Acceleration - What about speed/ direction of motion? ($\mathbf{a} = d\mathbf{n}$)
- Mass
- Volume/ Any other variables?

```
class Particle{
protected:
 Vector3 position;
 Vector3 velocity;
 Vector3 acceleration;
 real damping;
 real inverseMass;}
```

Implementation: The Update Loop

- 'The integrator'
- Separate to graphics
- Calculates change in position, p
- Damping

The Integrator

- Resolves forces
- $\mathbf{a} = \frac{\mathbf{F}}{m}$
- $\mathbf{v} = \int \mathbf{a} \; dt$
- $\mathbf{s} = \int \mathbf{v} \; dt$, $\mathbf{s} = \Delta\mathbf{p}$

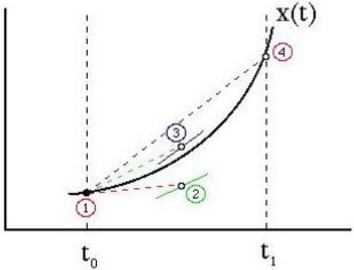
Algorithms for Numerical Integration

- Explicit Euler Integration
  - $v_{n+1} = v_n + a_n \Delta t$
  - $s_{n+1} = s_n + v_n \Delta t$
- Implicit Euler Integration
  - $v_{n+1} = v_n + a_{n+1} \Delta t$
  - $s_{n+1} = s_n + v_{n+1} \Delta t$
- Semi-Implicit Euler Integration
  - $v_{n+1} = v_n + a_n \Delta t$
  - $s_{n+1} = s_n + v_{n+1} \Delta t$
- Verlet Integration
  - $s_{n+1} = s_n + v_n \Delta t + a_n \Delta t^2$ and $v_n = \frac{s_n - s_{n-1}}{\Delta t}$
  - gives $s_{n+1} = 2S_n - S_{n-1} + a_n \Delta t^2$

Runge-Kutta Methods

By Solving Differential Equations:

Figure: Runge-Kutta Method for Numerical Integration



$$
\begin{aligned}
dx1 &= \Delta t\, v_{x,n} \\
dv_x1 &= \Delta t\, a_x(x_n, y_n, t) \\
dx2 &= \Delta t\, (v_{x,n} + \frac{dv_x1}{2}) \\
dv_x2 &= \Delta t\, a_x(x_n + \frac{dx1}{2}, y_n + \frac{dy1}{2}, t + \frac{\Delta t}{2}) \\
dx3 &= \Delta t\, (v_{x,n} + \frac{dv_x2}{2}) \\
dv_x3 &= \Delta t\, a_x(x_n + \frac{dx2}{2}, y_n + \frac{dy2}{2}, t + \frac{\Delta t}{2}) \\
dx4 &= \Delta t\, (v_{x,n} + dv_x3) \\
dv_x4 &= \Delta t\, a_x(x_n + dx3, y_n + dy3, t + \Delta t) \\
x_{n+1} &= y_n + \frac{dx1}{6} + \frac{dx2}{3} + \frac{dx3}{3} + \frac{dx4}{6} \\
v_{x,n+1} &= v_{x,n} + \frac{dv_x1}{6} + \frac{dv_x2}{3} + \frac{dv_x3}{3} + \frac{dv_x4}{4}
\end{aligned}
$$

Damping

- Why do we include damping?
- Issues caused by variable frame rate

Implementation

Calculating the new position:

$$\mathbf{p}' = \mathbf{p} + \mathbf{v}t$$

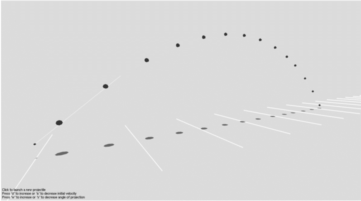
Calculating the new velocity:

$$\mathbf{v}' = \mathbf{v}d^t + \mathbf{a}t$$

```
if (inverseMass <= 0.0f) return;
assert(duration > 0.0);
position.addScaledVector(velocity, duration);
velocity.addScaledVector(acceleration, duration);
velocity *= real_pow(damping, duration);
```

Testing: Projectiles

Figure: Projectile Simulation




```
projectileNumber->particle.setMass(200.0f);
projectileNumber->
particle.setVelocity(projectileVelocity);
projectileNumber->
particle.setAcceleration(0.0f,-20.0f,0.0f);
projectileNumber->particle.setDamping(0.99f);
projectileNumber->particle.setPosition(0.0f,1.5f,0.0f);
```

Forces

- D'Alemberts Principle: $\mathbf{F} = \sum_i \mathbf{F}_i$

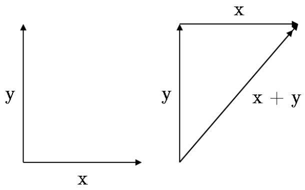


Figure: Geometrical Representation of Vector Addition

- Interfaces and Polymorphism

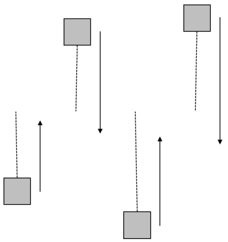
Implementation

```
void PForceReg::updateForces(real duration){
 Registry::iterator i = registrations.begin();
 for(; i != registrations.end(); i++){
 i->fg->updateForce(i->particle, duration);}}
```

Springs

- Hooke's Law: $f = kx$
- Applications
- Stiff Springs

Figure: Stiff Springs over Time



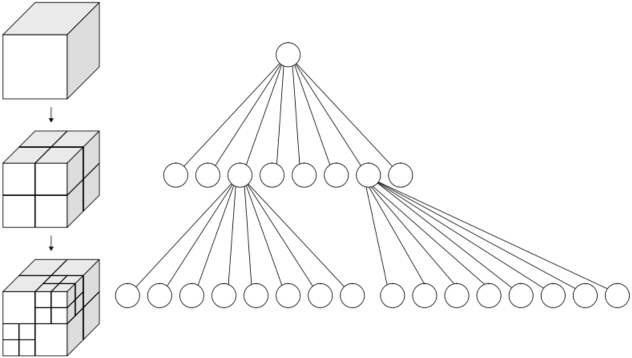
Implementation

```
void PSpring::updateForce(Particle *particle, real
duration){

 Vector3 force;
 particle->getPosition(&force);
 force -= other->getPosition();

 real magnitude = force.magnitude();
 magnitude = real_abs(magnitude - restLength);
 magnitude *= springConstant;

 force.normalize();
 force *= magnitude;
 particle->addForce(force);}
```

Broadphase Collision Detection Methods

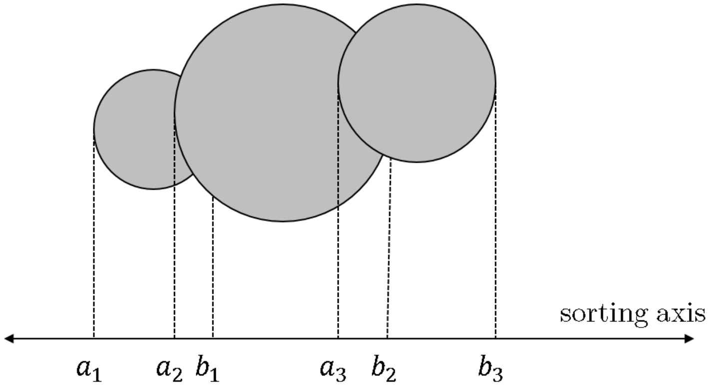
BSP (Octrees/ Quadtrees)

Figure: BSP Method for Broadphase Collision Detection



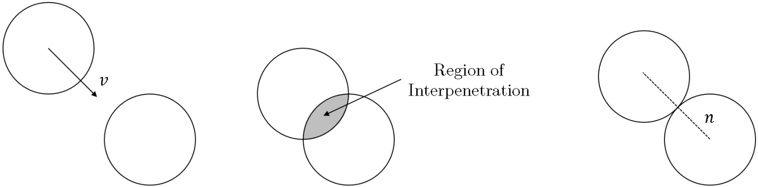
Sort and Sweep

Figure: Sort and Sweep Method for Broadphase Collision Detection



Narrowphase Collision Detection and Resolving Interpenetration

Figure: Resolving Interpenetration



Frame 1
Before Collision

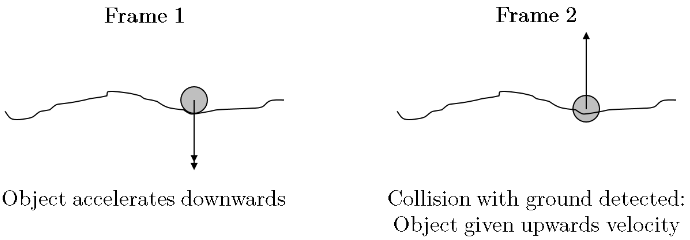
Frame 2
During Collision

Frame 3
After Collision

Limitations

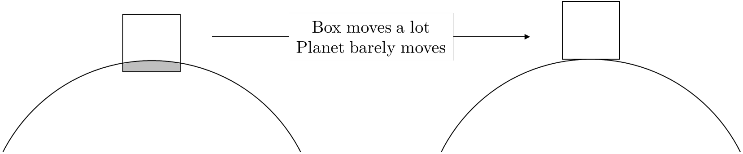
- High-Speed?
- Objects at Rest?

Figure: Errors Caused by Objects at Rest



Dealing with Multiple Objects?

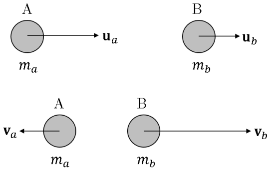
Figure: Interpenetration Resolution with Multiple Objects



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

Collision Resolution

Figure: The Impulse Method for Collision Resolution



- Newton's Law of Restitution: $v = eu$
- Law of Conservation of Momentum: $m_a\mathbf{u}_a + m_b\mathbf{u}_b = m_a\mathbf{v}_a + m_b\mathbf{v}_b$
- Equating Impulses: $j_a = -j_b$ where $j = m\mathbf{v} - m\mathbf{u}$
- Other Methods?
- Applications

Implementation

- Get approach speed
- $v = eu$
- $\Delta v_{total} = v - u$
- $j_{total} = m_{total} \Delta v_{total}$
- $\Delta v_a = \frac{j_{total}}{m_a}$
- $\Delta v_b = \frac{j_{total}}{m_b}$

Limitations of the Real-Time Mass-Aggregate Physics Engine

- Rigid Bodies
- Soft Bodies