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

Jordan Spoooner

July 16, 2015
**Engine:** Computer software that performs a fundamental function, especially as part of a larger program

- Moves objects
- Detects collisions
- Resolves collisions
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 ⇒ \( \mathbf{a} = \frac{\mathbf{F}}{m} \)
- 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?

```cpp
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
- \( a = \frac{F}{m} \)
- \( v = \int a \, dt \)
- \( s = \int v \, dt, \ s = \Delta p \)
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

\[
\begin{align*}
  dx_1 &= \Delta t \, v_{x,n} \\
  dv_x 1 &= \Delta t \, a_x(x_n, y_n, t) \\
  dx_2 &= \Delta t \left( v_{x,n} + \frac{dv_x 1}{2} \right) \\
  dv_x 2 &= \Delta t \, a_x(x_n + \frac{dx_1}{2}, y_n + \frac{dy_1}{2}, t + \frac{\Delta t}{2}) \\
  dx_3 &= \Delta t \left( v_{x,n} + \frac{dv_x 2}{2} \right) \\
  dv_x 3 &= \Delta t \, a_x(x_n + \frac{dx_2}{2}, y_n + \frac{dy_2}{2}, t + \frac{\Delta t}{2}) \\
  dx_4 &= \Delta t \left( v_{x,n} + dv_x 3 \right) \\
  dv_x 4 &= \Delta t \, a_x(x_n + dx_3, y_n + dy_3, t + \Delta t) \\
  x_{n+1} &= y_n + \frac{dx_1}{6} + \frac{dx_2}{3} + \frac{dx_3}{3} + \frac{dx_4}{6} \\
  v_{x,n+1} &= v_{x,n} + \frac{dv_x 1}{6} + \frac{dv_x 2}{3} + \frac{dv_x 3}{3} + \frac{dv_x 4}{4}
\end{align*}
\]
Damping

- Why do we include damping?
- Issues caused by variable frame rate
Calculating the new position:

\[ p' = p + vt \]

Calculating the new velocity:

\[ v' = vd^t + at \]

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
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
```cpp
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);
}
```
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

**Frame 1:** Object accelerates downwards

**Frame 2:** Collision with ground detected: Object given upwards velocity
Dealing with Multiple Objects?

**Figure:** Interpenetration Resolution with Multiple Objects

\[ \Delta p_a = \frac{m_b}{m_a + m_b}dn \quad \text{and} \quad \Delta p_b = -\frac{m_a}{m_a + m_b}dn \]
Collision Resolution

**Figure:** The Impulse Method for Collision Resolution

- Newton’s Law of Restitution: $v = eu$
- Law of Conservation of Momentum:
  \[ m_a u_a + m_b u_b = m_a v_a + m_b v_b \]
- Equating Impulses: $j_a = -j_b$ where $j = m v - m u$
- Other Methods?
- Applications
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