The Design Behind Physics Engines

Jordan Spoooner

The Judd School

June 2015

Balancing:
- optimisation (efficiency)
- accuracy
- limitations

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

- Applies forces
- Newtonian mechanics
- Integration
- Linear / Angular

- Checks to see if objects intersect

- Momentum + Impulses

'Moves Objects'

Note that this is rarely ever the case

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 $\Rightarrow \mathbf{a} = \frac{\mathbf{F}}{m}$ — By far the most important
- For every action there is an equal and opposite reaction

└ collisions (but note forces not accelerations)

Storing the Data

Point masses rather than rigid or soft bodies

Particles - No volume, no angular motion

All expressed as vectors (in terms of x, y, z)

- Position, Velocity, and Acceleration - What about speed/ direction of motion? ($\mathbf{a} = d\mathbf{n}$) — Can be worked out by normalising vector
- Mass — Stored as inverse
- Volume/ Any other variables?

Rigid bodies → Vol, Angular Motion

Soft → deformation

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

defined as a float → essentially a decimal number

Implementation: The Update Loop

main role

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

Later

Usually 4-5x more updates (improves quality of integration)

can be computationally expensive only with large numbers of objects - so some need to be removed

The Integrator

Simply sum vectors of forces

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

by Newton's 2nd law

since $a = \frac{dv}{dt} = \frac{d^2 s}{dt^2}$

since $v = \frac{ds}{dt}$

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 $s_{n+1} = 2S_n - S_{n-1} + a_n \Delta t^2$

- Most common, simple
- Uses first derivative by calculati
  at current time
- Tends to be unstable without
  high sampling

- Need to predict future
  acceleration (controlled by player

- Uses future velocity – stabilis
- No prediction required
- Fast to compute
- Calculation order is vital

- Uses the second derivative
- no velocity – uses last two
  positions + acceleration
- stable
- reversible
- needs initial conditions

Runge-Kutta Methods

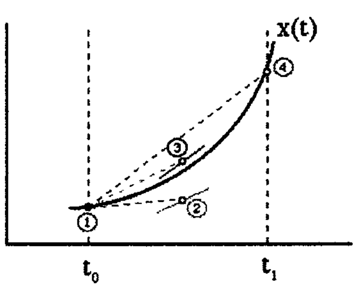
– For scientific (not real-time) applications

By Solving Differential Equations:

- Divides the time step into a number of sections
- Generally 4th order

i.e. Euler integration is a low order Runge-Kutta implementation

Figure: Runge-Kutta Method for Numerical Integration



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

needs to be an estimate from iterative methods – since not solvable algebraically

Damping

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

- Stops objects from speeding up randomly due to inaccuracy of calculations
- can be used to simulate drag - Newton's first Law
  -> crude but fast

- Slows at different rates - and very quickly
  -> so taken to the power of frame duration to give damping factor/sec

Implementation

Using explicit Euler integration

Calculating the new position:

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

Calculating the new velocity: damping factor (per second)

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

```
if (inverseMass <= 0.0f) return;  --> immovable object
assert(duration > 0.0);  -> negative duration causes problems!
position.addScaledVector(velocity, duration);
velocity.addScaledVector(acceleration, duration);
velocity *= real_pow(damping, duration);
```

Implementation of formulae above

Testing: Projectiles

5 lines of
about 250
(for simulation alone!

mass at 200kg

```
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);
```

user defined
(by magnitude and
length of y in unit vector)

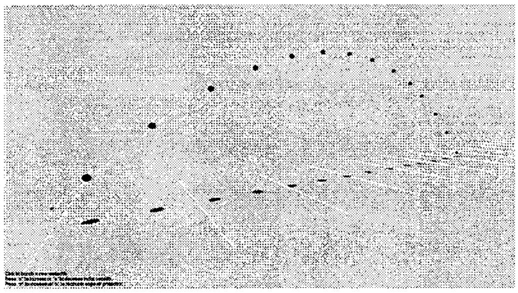
g greatly exaggerated
to $20ms^{-2}$

retains 99% of
velocity
each second

Defines original
position (above
ground so as to not
be removed instantly)

Testing: Projectiles

Figure: Projectile Simulation



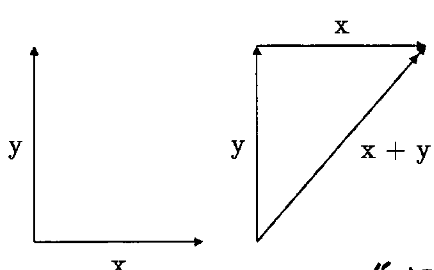
```
projectileNumber->particle.setMass(200.0f);
projectileNumber->
particle.setVelocity(projectileVelocity);
projectileNumber->
particle.setAcceleration(0.0f,-20.0f,0.0f);
projectileNumber->particle.setDamping(0.99f);
projectileNumber->particle.setPosition(0.0f,1.5f,0.0f);
```

Forces

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

Implies vector addition possible to add forces

Figure: Geometrical Representation of Vector Addition



- Interfaces and Polymorphism

"Force generator"
- standard way to implement forces
- deal with specific forces later

Implementation

STANDARD INTERFACE:

Force Registry class
=> updateForces function

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

ForceGen

update particle
- goes to
force accumulator

begin at beginning
of force registry,
end at end

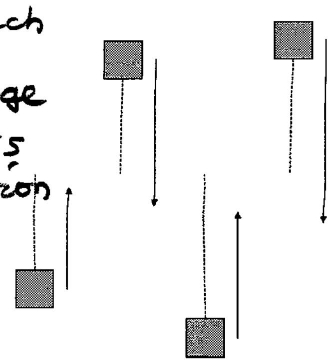
Springs

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

- Buoyancy
- Cloths
- Fluids
- Soft / deformable objects
  "blobs"

- setting k too small
  ⇒ bounces too much
- setting k too large
  ⇒ massive errors
  - acceleration
  - numerical

Figure: Stiff Springs over Time



Implementation

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

our force interface

define the force

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

sets force to position of first object

take away position of other object

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

define magnitude as length

Finds extension

multiplied by spring constant

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

done know why I did this!

adds force to a accumulator

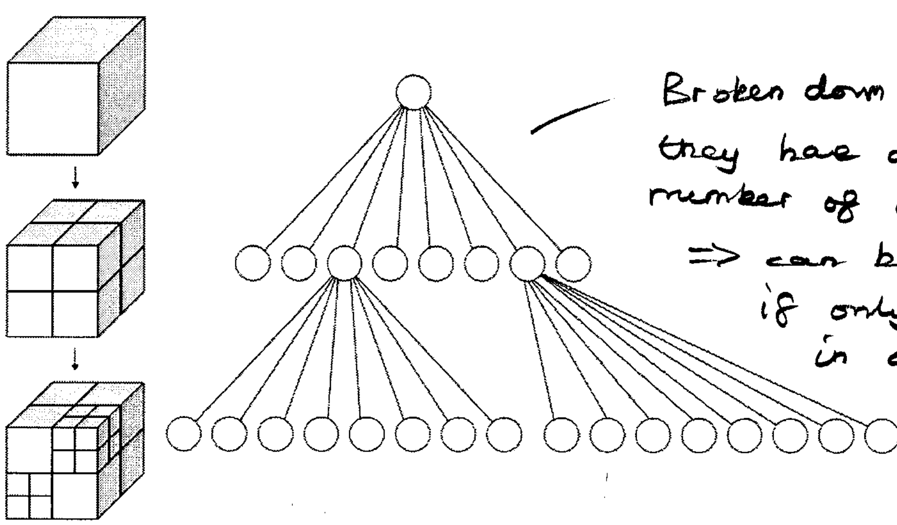
Broadphase Collision Detection Methods

Detection: BROADPHASE (otherwise n^2 combs to look at)

BSP (Octrees/ Quadtrees)

Binary Space Partitioning

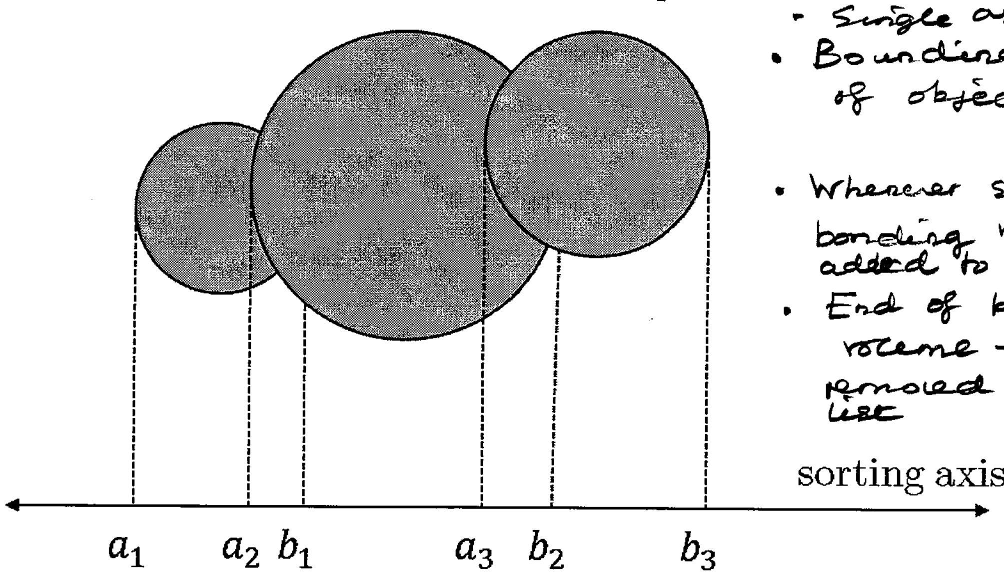
Figure: BSP Method for Broadphase Collision Detection



Broken down only when they have a set number of objects
⇒ can be ignored if only one object in a given space

Sort and Sweep

Figure: Sort and Sweep Method for Broadphase Collision Detection

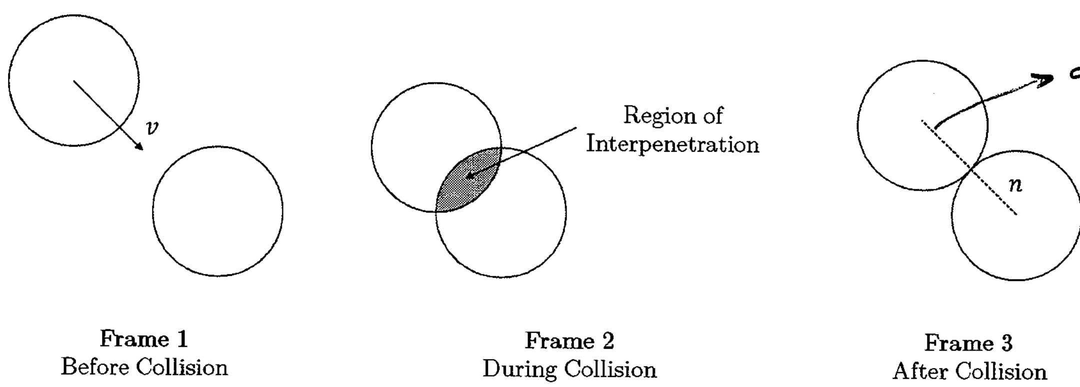


- Single axis chosen
- Bounding volume of objects projected
- Whenever start of bounding volume → (a) added to active list
- End of bounding volume → (b) removed from active list

Narrowphase Collision Detection and Resolving Interpenetration

First need to resolve interpenetration before resolving actual collision → could only do this (projection method)

Figure: Resolving Interpenetration



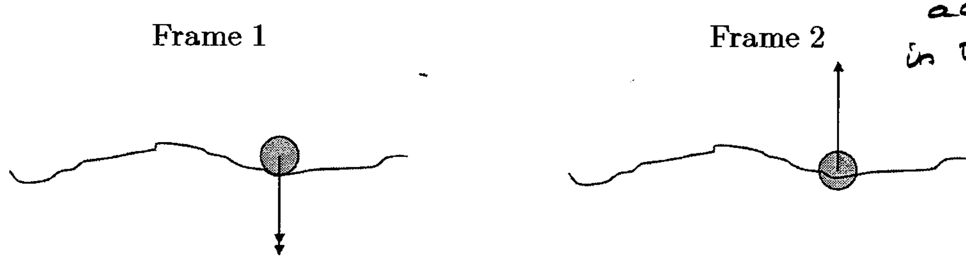
direction vector along which object is moved back

collision detected when they interpenetrate

Limitations

- High-Speed? — pass through each other without detection → difficult to fix
- Objects at Rest? — vibrate or even given significant upwards velocity — we just ignore velocity produced by acceleration in the last frame

Figure: Errors Caused by Objects at Rest



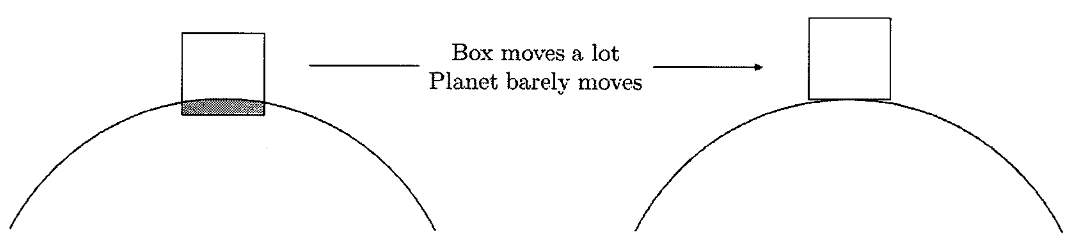
Object accelerates downwards

Collision with ground detected: Object given upwards velocity

---

Dealing with Multiple Objects? — Problems?

Figure: Interpenetration Resolution with Multiple Objects

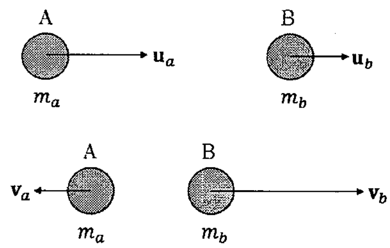


$$\Delta \mathbf{p}_a = \frac{m_b}{m_a+m_b} d\mathbf{n} \text{ and } \Delta \mathbf{p}_b = -\frac{m_a}{m_a+m_b} d\mathbf{n}$$

instead we take mass into account

Collision Resolution

Figure: The Impulse Method for Collision Resolution



Coefficient of restitution
0 → inelastic
1 → elastic

- 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 $j = m\mathbf{v} - m\mathbf{u}$
  by LCM
- Other Methods? — Penalty (a)
  — Projection (s)
- Applications
  Rods and cables

Implementation

code quite complex - so algorithmic description :

- Get approach speed
- $v = eu$ — Find separation speed by NLR
- $\Delta v_{total} = v - u$ — Find change in velocity
- $j_{total} = m_{total} \Delta v_{total}$ — Find overall change in momentum (impulse)
- $\Delta v_a = \frac{j_{total}}{m_a}$
- $\Delta v_b = \frac{j_{total}}{m_b}$ — By LCM - impulses are the same so we split by mass to find ΔV

Limitations of the Real-Time Mass-Aggregate Physics Engine

- Rigid Bodies
- Soft Bodies

Toppling

Proper shapes with actual volumes — kind of possible with collisions

Deformable objects — kind of possible with springs