Game Physics

A quick introduction to discrete time physics

The (discrete) laws of Physics

- Newton's equations of Motion
  - f = ma Force = Mass * Acceleration
- To display or render on screen, we need to know position and orientation, so
- Let's do some simple math physics. rewrite 2nd law as:
-
- a = f/m
-
- We know acceleration is change in velocity with time, ie.
-
- a = dv/dt
- so
- dv/dt = a = f/m

Discrete physics ➡ numerical integration

$$dv/dt = a = f/m$$

Velocity is change in position over time, so

$$dp/dt = v$$

This means, that if we know the current position, the current velocity, and the forces applied to an object, we can compute (simulate) future positions, velocities, and accelerations of the object. This same logic applies to the orientation as well.

In discrete physics, we do not solve differential equations!
we solve difference equations through numerical integration.

Euler Integration is easy!

Suppose we want to predict the position of a game entity (ent) at time 1, 2, 3, 4, and we know

- Position (p) of ent at time 0
- Force (f) on ent

Then

a = f/m

If we know acceleration, then we know velocity is change in acceleration. If we want to know velocity at time t = 1 then

dt = 1 - 0 = 1 (let's keep the math simple)

and we know

dv/dt = a

so

dv = a * dt so, we know the change in velocity

Computing change in position

Knowing velocity, we can compute the change in position because

$$dp/dt = v$$

so, change in position dp is

$$dp = v * dt$$

Simple Example

Let's do an example:

- One dimension
- Initial position, p is 0. Initial velocity v = 0
- Mass, m is 100
- Force, f is 50
- time, t = 0.0

To get position at time t = 1.0

dt = 1.0 - 0.0 = 1.0

a = f/m = 50/100 = 0.5

dv = a * dt = 0.5 * 1.0 = 0.5

newVelocity = v + dv = 0 + 0.5 = 0.5

dp = newVelocity * dt = 0.5 * 1.0 = 0.5

This is change in position, so the new position is p + dp

newPosition = p + dp = 0 + 0.5 = 0.5

New position at time t = 2

p = 0.5 and v = 0.5 (from last step)
dt = 2.0 - 1.0 = 1.0
a = f/m = 50/100 = 0.5 (constant acceleration)
dv = a * dt = 0.5 * 1.0 = 0.5
newVelocity = v + dv = 0.5 + 0.5 = 1.0
dp = v * dt = 1.0 * 1.0 = 1.0

newPosition = p + dp = 0.5 + 1.0 = 1.5

This was a simple example, with dt = 1. Just to see how it works, let's try dt = 2, that is we want to know the position of ent at time t = 2, t = 4, t = 6, and so on.

Example 2, dt = 2

p = 0, v = 0 (initial position = 0, initial velocity = 0)
m = 100
f = 50
dt = 2.0

so

a = f/m = 50/100 = 0.5
dv = a * dt = 0.5 * 2.0 = 1.0
newVelocity = v + dv = 0 + 1.0 = 1.0
dp = v * dt = 1.0 * 2 = 2.0

newPosition = p + dp = 0 + 2.0 = 2.0 !!!!
This is not the same as the newPosition you calculated using dt = 1? Why not ?

Exact

p = 0.5 a t^2

at t = 2:

p = 0.5 * a * t^2
p = 0.5 * 0.5 * 2^2
p = 0.5 * 0.5 * 4
p = 0.5 * 2
p = 1.0

Euler Python code. Try different dt values

```
t = 0.0
dt = 1.0

vel = 0.0
pos = 0.0
force = 50.0
mass = 100.0

while (t <= 10):
    pos = pos + vel * dt;
    vel = vel + (force/mass) * dt;
    t = t + dt;
    print "Time: ", t, "Position: ", pos, " Velocity: ", vel
```

Euler integration works for very small dt

- Euler assumes the rate of change is constant over dt.
- But velocity is changing over our dt, and the larger the dt, the larger the change, so our system violates Euler's assumptions

The error between Euler and exact (p = 0.5 a t^2) gets larger over time!

What do we do?

Typically, we don't need exact physics!

So, either

- Use simple Euler and know that the simulated physics in your game is not "real"
  - Your game is not real anyway
  - Simple code is easier to debug
  - Simple Euler integration code is FAST

Or

- Use Runge Kutta order 4 (RK4) integrator. Sufficiently accurate for the kind of timesteps used in our games
- Works by
  - Sampling the derivative at several different points in the timestep to detect change in (for example) velocity.
  - Combines these derivatives using a weighted average

The difference (starting pos = 100.0)

- dt = 0.1
  - Euler: Time: 10.0 Pos: 124.75 Vel: 5.00
  - RK4 : Time: 10.0 Pos: 125.00 Vel: 5.00
  - Exact: Pos: 125.00
- dt = 1.0
  - Euler: Time: 10.0 Pos: 122.50 Vel: 5.00
  - Rk4 : Time: 10.0 Pos: 125.00 Vel: 5.00
  - Exact: Pos: 125.00
- dt = 2.0
  - Euler: Time: 10.0 Pos: 120.00 Vel: 5.00
  - Rk4 : Time: 10.0 Pos: 125.00 Vel: 5.00
  - Exact: Pos: 125.00

Here's how RK4 works...

State and Derivative

```
class State:
    x = 0
    v = 0

class Derivative:
    dx = 0;
    dv = 0
```

Storage for position (x), velocity (v), dx/dt and dv/dt

RK4 integration

```
def integrate(state, t, dt):
    a = evaluateInitial(state, t)
    b = evaluate(state, t+dt*0.05, dt*0.5, a)
    c = evaluate(state, t+dt*0.05, dt*0.5, b)
    d = evaluate(state, t+dt, dt, c)

    dxdt = 1.0/6.0 * (a.dx + 2.0*(b.dx + c.dx) + d.dx)
    dvdt = 1.0/6.0 * (a.dv + 2.0*(b.dv + c.dv) + d.dv)

    state.x = state.x + dxdt * dt
    state.v = state.v + dvdt * dt
```

Sampling the derivative at several different points in the timestep to detect change in (for example) velocity.

evaluate - evaluates the derivative at multiple points in time
The key is that derivative a is used to evaluate b, which feeds into the evaluation of c, which feeds into d
The constants in the weighted average come from Lagrange's expansion of the Taylor series.

Position and velocity integration simultaneously

```
def evaluate(initialState, t, dt, d):
    state = State();
    state.x = initialState.x + d.dx * dt
    state.v = initialState.v + d.dv * dt

    output = Derivative()
    output.dx = state.v
    output.dv = acceleration(state, t+dt)
    return output

def acceleration(state, t):
    return force/mass        # constant acceleration
```

The difference (again)

- dt = 0.1
  - Euler: Time: 10.0 Pos: 124.75 Vel: 5.00
  - RK4 : Time: 10.0 Pos: 125.00 Vel: 5.00
  - Exact: Pos: 125.00
- dt = 1.0
  - Euler: Time: 10.0 Pos: 122.50 Vel: 5.00
  - Rk4 : Time: 10.0 Pos: 125.00 Vel: 5.00
  - Exact: Pos: 125.00
- dt = 2.0
  - Euler: Time: 10.0 Pos: 120.00 Vel: 5.00
  - Rk4 : Time: 10.0 Pos: 125.00 Vel: 5.00
  - Exact: Pos: 125.00
- Errors accumulate, so smaller errors accumulate slower.
- RK4 has lower error especially if you have larger dt
- Still simple enough to code
- Not very hard to understand, and modify, and extend to 3D