CS 418: Interactive Computer Graphics

A Simple Physics Engine

Eric Shaffer

Newtonian Physics

- We will animate particles (aka point masses)
- Position is changed by velocity
- Velocity is changed by acceleration
- Forces alter acceleration

- Our physics engine will integrate to compute
  - Position
  - Velocity
- We set the acceleration by applying forces

Mass and Acceleration

- To find the acceleration due to a force we have

$$\ddot{\mathbf{p}} = \frac{1}{m}\mathbf{f}$$

- So we need to know the inverse mass of the particle
  - You can model infinite mass objects by setting this value to 0
- For the MP, you can use a uniform mass of 1
  - Or make the masses different if you want…

Force: Gravity

- Law of Universal Gravitation

$$f = G\frac{m_1 m_2}{r^2}$$

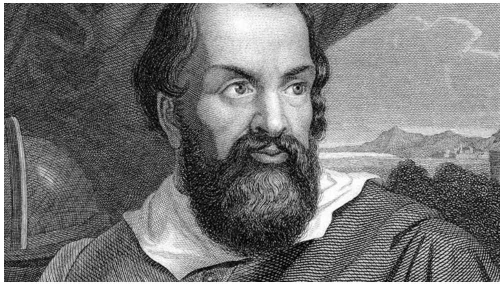
- G is a universal constant
- m_i is the mass of object I
- r is the distance between object centers
- f we care only about gravity of the Earth
  - m_1 and r are constants
  - r is about 6400 km on Earth
- We simplify to f = mg
  - g is about $10ms^{-2}$

Force: Gravity

- If we consider acceleration due to gravity we have

$$\ddot{p} = \frac{1}{m}(mg) = g$$

- So acceleration due to gravity is independent of mass



Force: Gravity

- In your MP

$$\mathbf{g} = \langle 0, -g, 0 \rangle$$

- For gaming, $10ms^{-2}$ tends to look boring
  - Shooters often use $15ms^{-2}$
  - Driving games often use $20ms^{-2}$
  - Some tune g object-by-object

Force: Drag

- Drag dampens velocity
  - Caused by friction with the medium the object moves through
- Even neglecting, you need to dampen velocity
  - Otherwise numerical errors likely drive it higher than it should be
- A velocity update with drag can be implemented as

$$\dot{\mathbf{p}}_{new} = \dot{\mathbf{p}} d^t$$

  - important to incorporate time so drag changes if the frame rate varies
  - for the MP, have all objects have the same drag, calculate once per frame

The Integrator

- The position update can found using Euler's Method:

$$P_{new} = P_{old} + \dot{p}t$$

- Note that is inaccurate, though good enough for the MP
  - Euler's error is O(t)
  - also position formula is inaccurate if acceleration is large
    - why?
- The velocity update is

$$\dot{\mathbf{p}}_{new} = \dot{\mathbf{p}}d^{t} + \ddot{\mathbf{p}}t$$

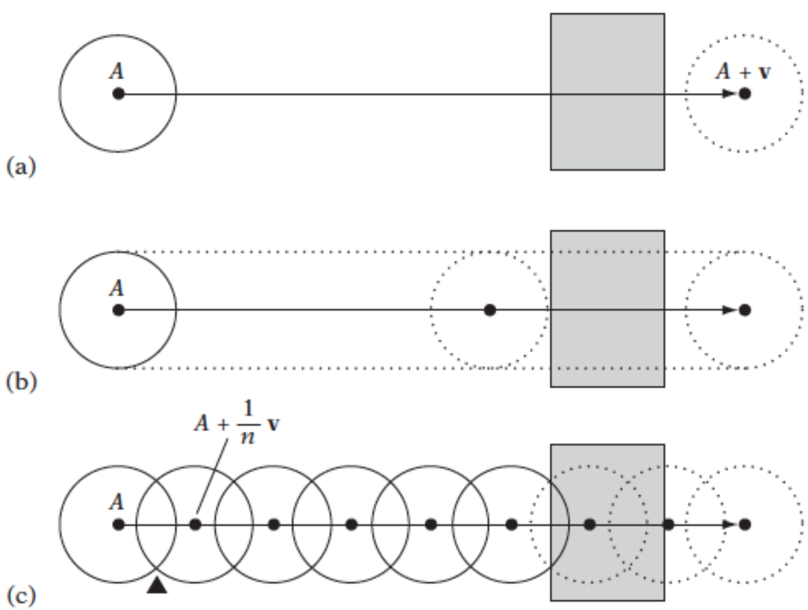
The Integrator

- You should ideally use actual time for *t*
  - or some scaled version of it
- In JavaScript, Date.now() returns current time in ms
  - so keep a previous time variable
  - each frame find out how much time has elapsesd
- …or you could use some uniform timestep you like

Collision Detection

- Surprisingly complex topic
  - Even a high-quality engine like Unity has issues
- We will simulate only two types of collision
  - Sphere-Wall
  - Sphere-Sphere
- We check for a collision when updating position
  - If a collision occurs the velocity vector is altered
  - Position is determined by the contact
  - Position and velocity update are completed with new values
    - over the remaining time

Dynamic Collision Detection



Dynamic collision tests can exhibit *tunneling*

if only the final positions of the objects are tested (a)

Or even if the paths of the objects are sampled (c)

A sweep test assures detection but may not be computationally feasible.

Sphere-Plane Collision

$$(\mathbf{n} \cdot X) = d \pm r \Leftrightarrow \qquad \text{(plane equation for plane displaced either way)}$$

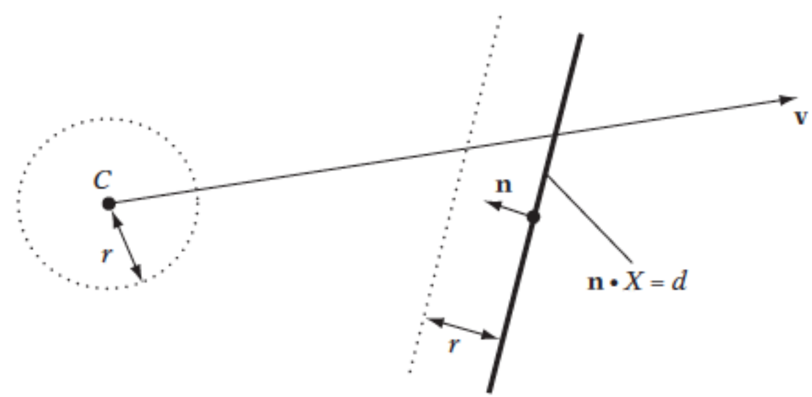
$$\mathbf{n} \cdot (C + t\mathbf{v}) = d \pm r \Leftrightarrow \qquad \text{(substituting } S(t) = C + t\mathbf{v} \text{ for } X\text{)}$$

$$(\mathbf{n} \cdot C) + t(\mathbf{n} \cdot \mathbf{v}) = d \pm r \Leftrightarrow \qquad \text{(expanding dot product)}$$

$$t = (\pm r - ((\mathbf{n} \cdot C) - d)) / (\mathbf{n} \cdot \mathbf{v}) \qquad \text{(solving for } t\text{)}$$

Can make it even simpler for the box walls in MP.

How?



Sphere-Sphere Collision

The vector $\mathbf{d}$ between the sphere centers at time t is given by

$$\mathbf{d}(t) = (C_0 + t\mathbf{v}_0) - (C_1 + t\mathbf{v}_1) = (C_0 - C_1) + t(\mathbf{v}_0 - \mathbf{v}_1)$$

$$\begin{aligned}
&\mathbf{d}(t) \cdot \mathbf{d}(t) = (r_0 + r_1)^2 \Leftrightarrow && \textit{(original expression)}\\
&(\mathbf{s} + t\,\mathbf{v}) \cdot (\mathbf{s} + t\,\mathbf{v}) = r^2 \Leftrightarrow && \textit{(substituting } \mathbf{d}(t) = \mathbf{s} + t\,\mathbf{v}\textit{)}\\
&(\mathbf{s} \cdot \mathbf{s}) + 2(\mathbf{v} \cdot \mathbf{s})t + (\mathbf{v} \cdot \mathbf{v})t^2 = r^2 \Leftrightarrow && \textit{(expanding dot product)}\\
&(\mathbf{v} \cdot \mathbf{v})t^2 + 2(\mathbf{v} \cdot \mathbf{s})t + (\mathbf{s} \cdot \mathbf{s} - r^2) = 0 && \textit{(canonic form for quadratic equation)}
\end{aligned}$$

This is a quadratic equation in t. Writing the quadratic in the form $at^2 + 2bt + c = 0$, with $a = \mathbf{v} \cdot \mathbf{v}$, $b = \mathbf{v} \cdot \mathbf{s}$, and $c = \mathbf{s} \cdot \mathbf{s} - r^2$ gives the solutions for t as

$$t = \frac{-b \pm \sqrt{b^2 - ac}}{a}.$$

Collision Resolution

- First, find closing (aka separating) velocity
  - Component of velocity of two objects in direction from one to another

$$v_c = \dot{\mathbf{p}}_\mathbf{a} \cdot (\mathbf{p}_\mathbf{b} - \mathbf{p}_\mathbf{a}) + \dot{\mathbf{p}}_\mathbf{b} \cdot (\mathbf{p}_\mathbf{a} - \mathbf{p}_\mathbf{b})$$

$$v_c = -(\dot{\mathbf{p}}_\mathbf{a} - \dot{\mathbf{p}}_\mathbf{b}) \cdot (\mathbf{p}_\mathbf{a} - \mathbf{p}_\mathbf{b})$$

$$v_s = (\dot{\mathbf{p}}_\mathbf{a} - \dot{\mathbf{p}}_\mathbf{b}) \cdot (\mathbf{p}_\mathbf{a} - \mathbf{p}_\mathbf{b})$$

- Collisions that preserve momentum are perfectly elastic
- We will use $v_{s\_after} = -cv_s$
  - c is the coefficient of restitution…a material property that you choose

Contact Normal

- When colliding with the ground use the contact normal

  <0, 1, 0>

  - Assuming the ground is level

- Contact normal in general is

$$\mathbf{n} = (\mathbf{p}_a - \mathbf{p}_b)$$