# A Simple Physics Engine

CS 418: Interactive Computer Graphics

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

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

# Force and Mass and Acceleration

- How do we update acceleration when force is applied?
- 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

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

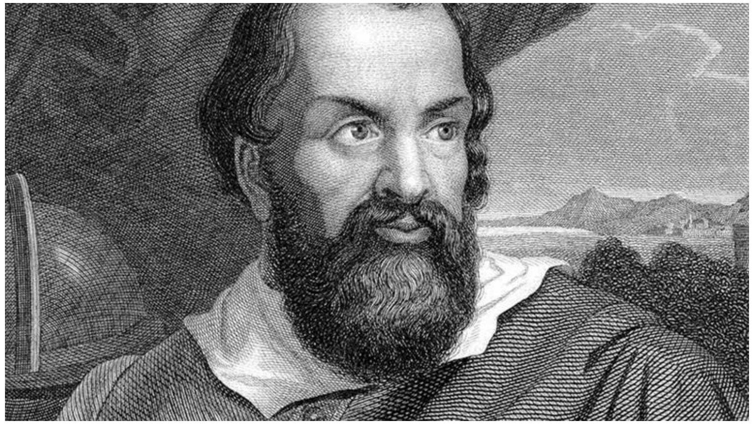
- G is a universal constant
- $m_i$ is the mass of an object
- r is the distance between object centers
- if we care only about gravity of the Earth
  - m1 and r are constants
  - r is about 6400 km on Earth
- We simplify to f = mg
  - g is about $10ms^{-2}$

# Acceleration due to 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



# Acceleration due to Gravity

- In your MP the magnitude and direction of acceleration would be

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

- 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 drag, 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
- What range should ***d*** be in?

# The Integrator

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

$$\mathbf{P_{new}} = \mathbf{P_{old}} + \dot{\mathbf{P}}\mathbf{t}$$

- This is a pretty inaccurate approximation of analytical integration
  - formula gets more inaccurate as acceleration gets larger
    - why?
  - In general we can characterize Euler method error as O(t)
  - ...good enough for the MP
- The velocity update is computed using Euler integration as well

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

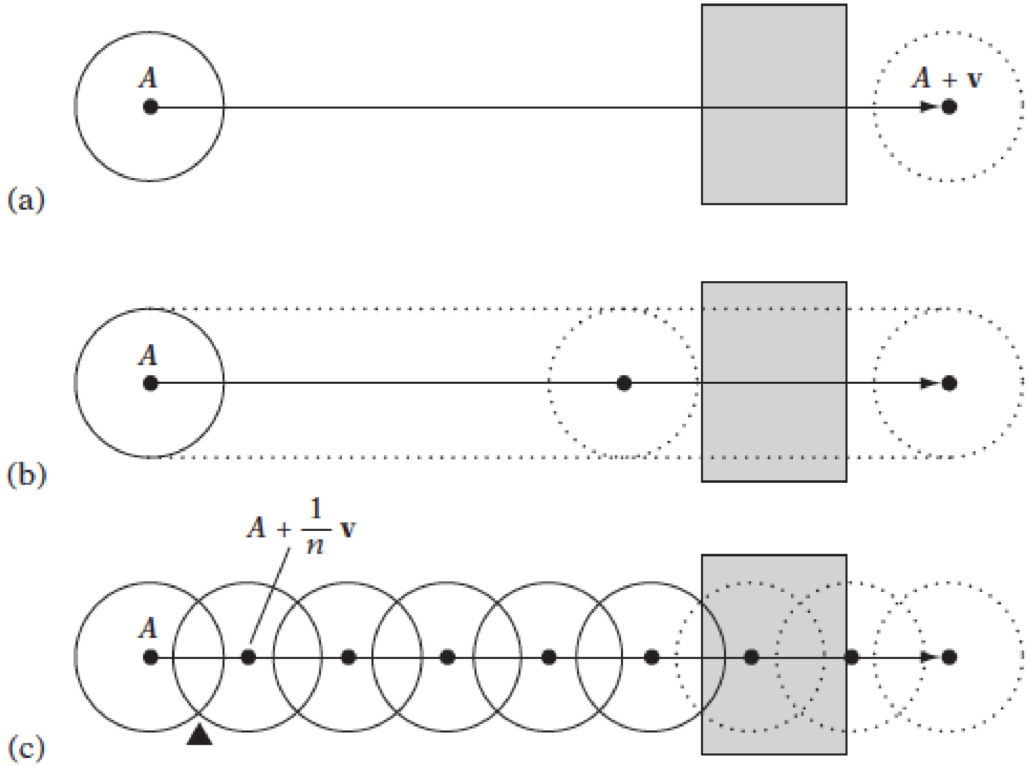
# The Integrator

- 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 discuss how to 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 an 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
….may not be computationally feasible.

# Sphere-Plane Collision

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

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

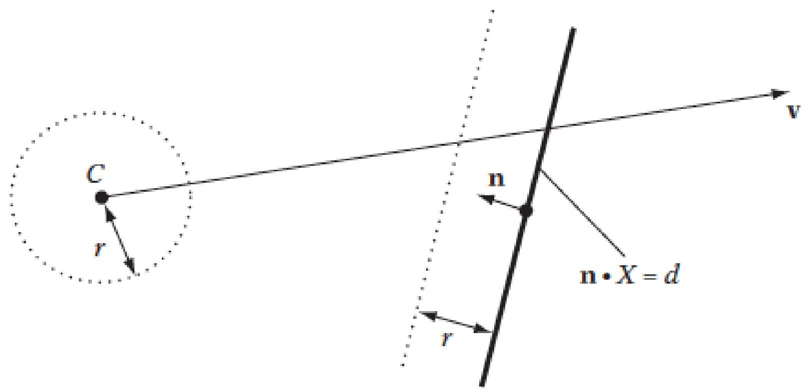
$$(\mathbf{n} \cdot C) + t(\mathbf{n} \cdot \mathbf{v}) = d \pm r \Leftrightarrow \qquad \textit{(expanding dot product)}$$

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

Why is it $\pm r$ ?

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

How?



# Sphere-Sphere Collision

The vector **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

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

# Resolving Particle-Particle Collisions

Simplest model of dynamic collision uses particles

particle is a mass located at a point

We need to compute an *impulse* that will be applied to both particles

An impulse is a change in momentum…we will use it to change velocity

$$Impulse = mass * Velocity$$

$$Velocity = \frac{Impulse}{mass} \therefore V' = V + \frac{j * n}{mass}$$

- *V′* is the velocity of a particle after collision
- *j* is the scalar magnitude of the impulse
- *n* is a unit vector expressing the direction of the impulse…the contact normal
- *mass* is the mass of the particle

# The Contact (or Collision) Normal

The contact normal between two particles a and b is given by

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

This is a unit length vector derived from the difference in positions

This will be the direction of the separating velocity for Particle a

The separating velocity changes the velocity of Particle a

# Separating Velocity

Need to compute the magnitude of the separating velocity

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

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

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

$v_c$ is the closing velocity before collision…positive for objects closing in on each other

$v_s$ is the separating velocity before collision…negative for objects closing in on each other

- Collisions that preserve momentum are perfectly elastic
- We will use $v_{s\_after} = -cv_s$
  - C in [0,1] is the coefficient of restitution…a material property that you choose
  - The negative sign is a result of the collision flipping the direction

# Separating Velocity

Still need to find the magnitude of the impulse for each particle…
Can solve for it:

$$(\dot{p}_a' - \dot{p}_b') \cdot n = -cv_s$$

…without going into details we can find the value j for the velocity updates

$$\dot{p}_a' = \dot{p}_a + \frac{jn}{mass_a}$$

$$\dot{p}_b' = \dot{p}_b - \frac{jn}{mass_b}$$

will be

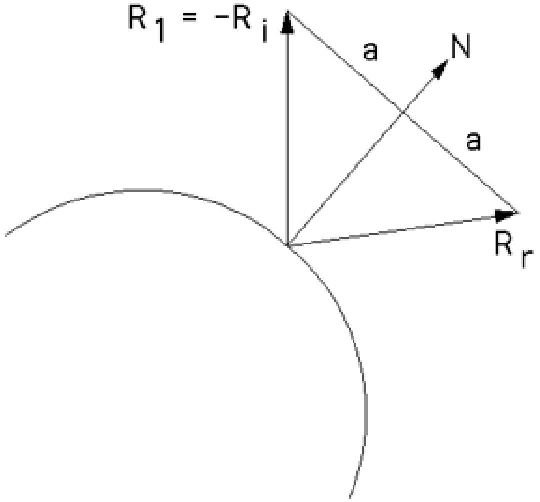
$$j = \frac{-(1+c)((\dot{p}_a - \dot{p}_a) \cdot n)}{\frac{1}{mass_a} + \frac{1}{mass_b}}$$

# Resolving Particle-Wall Collisions

- What direction to use after a sphere collides with a plane?
- It's the same as calculating the direction of a reflected ray

$$R_r = R_i - 2N(R_i \cdot N)$$

- The diagram below is in 2D…does anything change for 3D?



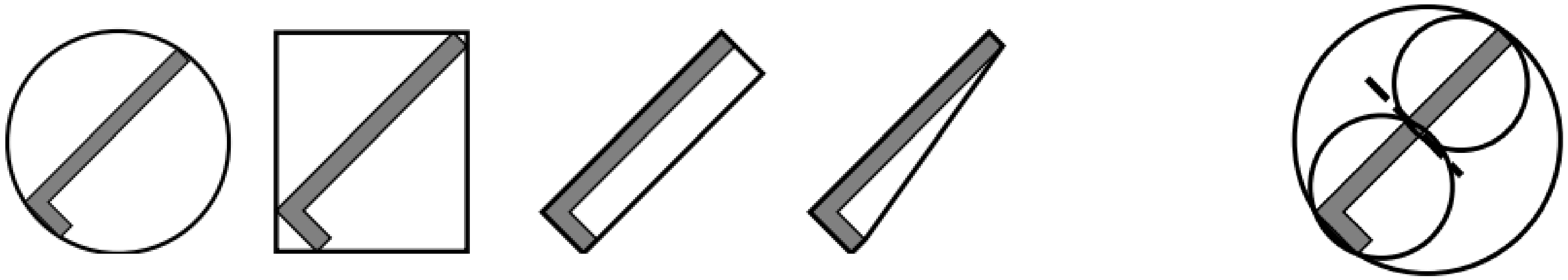
# General Collision Detection

Often requires use of bounding volumes or spatial data structures

1. **Broad Phase:** Uses bounding volumes to quickly determine which objects need to be checked closely for collision

2. **Narrow Phase:** Perform careful, expensive collision checks, such as triangle-triangle intersection for meshes

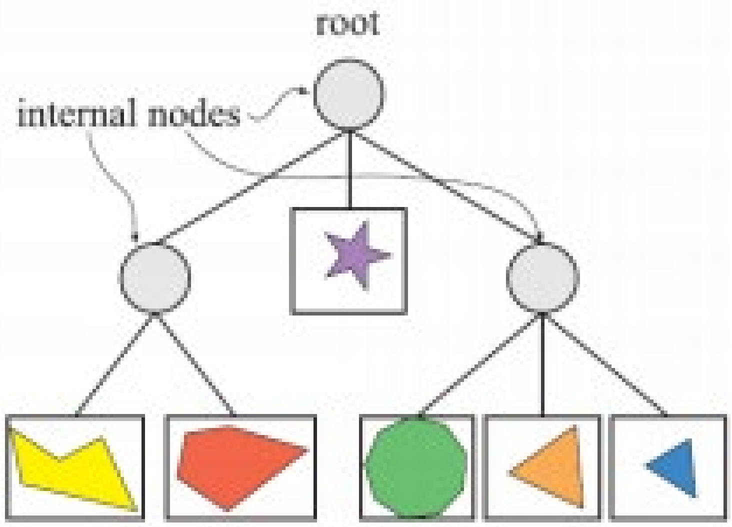
# Bounding Volumes



Sphere AABB OOBB Convex Hull Hierarchical (BVH)

AABB=Axis Aligned Bounding Box
OOBB = Object-Oriented Bounding Box
BVH = Bounding Volume Hierarchy

# BVH Construction

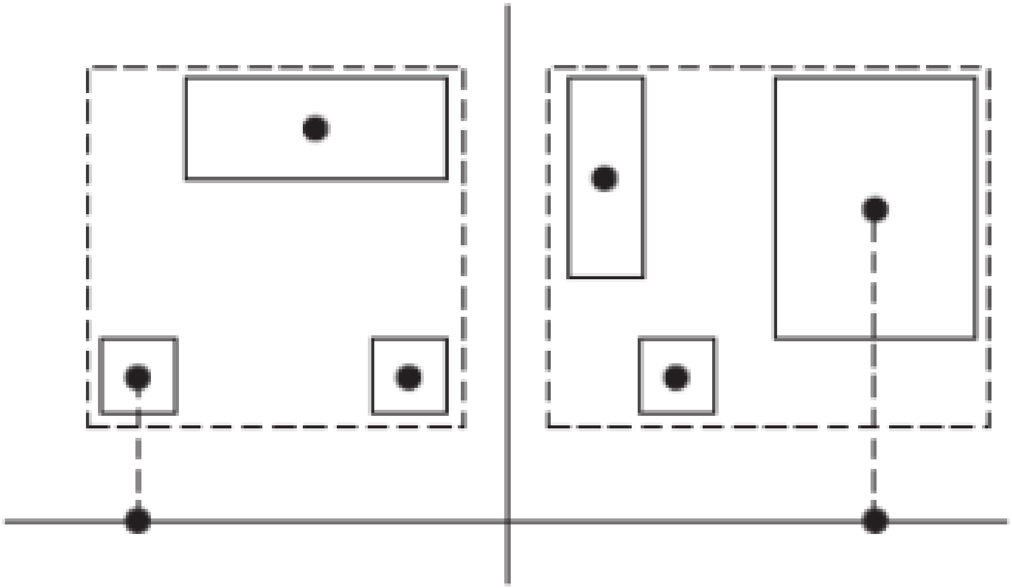



Can be constructed top down:

1. Compute bounding volume enclosing all of the geometry
2. Split the geometry into two or more groups
3. Compute bounding volume for each group
4. Recurse
5. Leaf nodes will enclose only one geometric primitive

Can also be built by bottom up merging which offers better parallelism

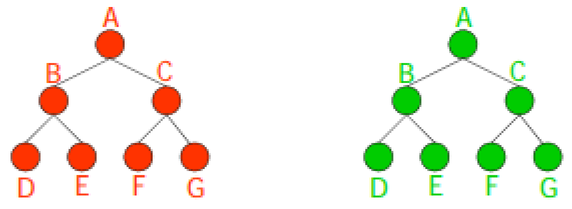
# BVH: How to Split



- Can compute a centroid for each geometric primitive
  - Split  on median centroid, along longest axis
  - Split on average centroid, along longest axis
- More sophisticated splitting criteria can be used
  - E.g. Surface Area Heuristic used on BVHs for ray-tracing

# BVH: How to Collide

- In a single BVH for scene
  - Two geometric primitives can overlap only if their volumes overlap
- Or, BVHs can be used for each composite object (e.g. mesh)
  - A search tree is constructed that records descent into each BVH
  - Determines if any cells overlap




Two objects described by their precomputed BVHs