DEPARTMENT OF INFORMATION SYSTEMS AND COMPUTER SCIENCE

# Basic Applications of Physics II

Aiming for Collisions

# Lecture Time!

- Collision Response: Mass Effect
- Impulse: Because Force Takes Too Long

# Collision

- After determining acceleration and computing for new positions and velocities, check for collision

```
// the code template below is wrong
// what should it be?
for( int i = 0; i < numObjects; i++ )
{
     // move object[i]
     // collision check for object[i]
}
```

# Collision

- Would there be a problem moving the object, checking for its collisions, then moving the next object (and repeating)?

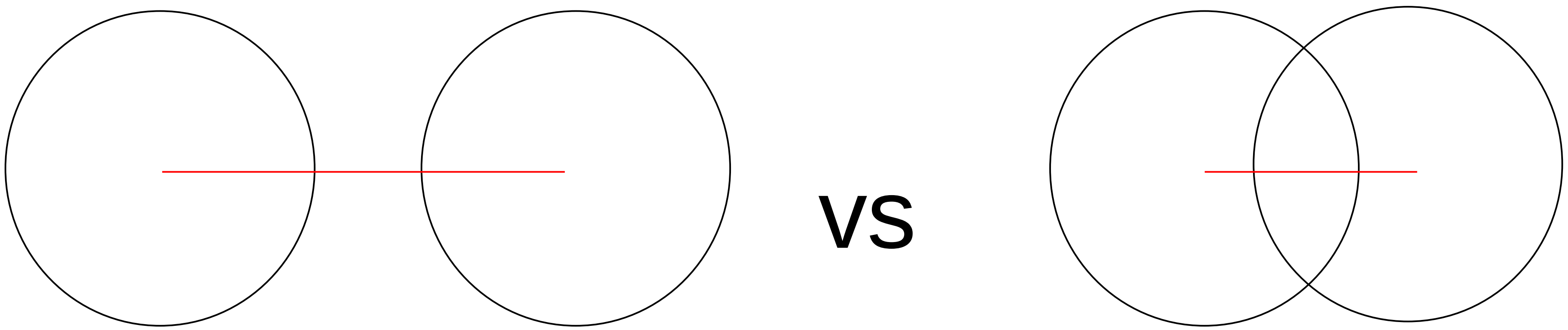
```
// the code template below is wrong
// what should it be?
for( int i = 0; i < numObjects; i++ )
{
     // move object[i]
     // collision check for object[i]
}
```

# Collision

- First, do a pairwise check for overlap
  - For shapes other than circles, you'll have to figure that out yourself (or wait for later lessons for convex polygons)
- Then, if there is overlap, check if the two objects are actually moving into each other
  - Both stationary or moving away from each other? No collision!

# Collision

- Check for overlap between circles by comparing radii and distance
  - If the sum of the radii is greater than the distance between centers, what does that mean for circle collision?



# Collision (After Overlap Check)

- First, determine a normal vector for the point of collision
  - Easy for circles – it's the vector we use to determine distance
  - Easy for edges – it's a vector perpendicular to the edge (facing away)
  - For other stuff – you'll have to look it up

# Collision (After Overlap Check)

- Next, determine relative velocity of one object to the other
  - Relative velocity of object A to object B = velocity_of_A – velocity_of_B
  - IMPORTANT: ALL equations (including the one above) in this slide set assume collision normal is pointing from object B to object A

# Collision (After Overlap Check)

- At this point, you have a collision normal vector and a relative velocity vector
  - What does it mean if they're pointing away from each other?
  - What does it mean if they're pointing in more or less the same direction?
  - And how do we determine where they're pointing (relative to each other) in code?

# Collision (After Overlap Check)

- If the relative velocity vector is pointing away from the collision vector (more than 90 degrees) then you have a collision
- Now we should determine what happens as a result of the collision
  - In other words, we need to use various collision response equations
  - No need to normalize collision normal for now

# Impulse

- Since we need to change object velocities instantaneously, we need to compute for impulse and apply it to the object velocities
  - Can’t rely on applying forces normally because it’ll take at least one timestep for the resulting change in accelerations to affect velocities

# Impulse

- However, just like friction, the effect of impulse depends on the objects' masses
  - Lighter objects should be more easily moved
  - Incredibly heavy objects (i.e. infinite mass) shouldn't budge at all
  - We're actually changing momentum (which affects velocity)

# Impulse

- How impulse (j) will be used:
  - new_velocity_of_A = old_velocity_of_A + ( ( j / mass_of_A ) * collision_normal
  - new_velocity_of_B = old_velocity_of_B – ( ( j / mass_of_B ) * collision_normal
  - Remember: Collision normal is assumed to point from B to A

# Impulse

- Solving for impulse:
  - –( (1 + elastic_coefficient)
    * ( relative_velocity • collision_normal ) )
    /
    ( ( collision_normal • collision_normal )
    *  (reciprocal_of_mass_of_A
    + reciprocal_of_mass_of_B) ) )

# Impulse

- Solving for impulse:
  - –( (1 + elastic_coefficient)
    * ( relative_velocity • collision_normal ) )
    /
    ( ( collision_normal • collision_normal )
    * (reciprocal_of_mass_of_A
    + reciprocal_of_mass_of_B) ) )
    - With circles, if you got this far, you already solved for this... what is it?

# Impulse

- Solving for impulse:
  - –( (1 + elastic_coefficient)
    * ( relative_velocity • collision_normal ) )
    /
    ( ( collision_normal • collision_normal )
    *  (reciprocal_of_mass_of_A
    + reciprocal_of_mass_of_B) ) )
    - You also already solved for this... when?

# Homework

- Modify your previous homework:
  - Your program should now require a command-line argument X that is an integer between 1 and 35, inclusive
  - In addition to the player-controlled circle, create X more circles
  - Positions should no longer be random to avoid initial overlapping

# Homework

- Modify your previous homework (continued):
  - These circles are also air hockey pucks, but the only time they should move is when another puck collides with them
  - Make sure none of the circles are initially overlapping (player-controlled circle included)

# Homework

- Modify your previous homework (continued):
  - All non-player-controlled pucks should have the same mass
  - Their mass can be the same as the mass of the player-controlled one, but provide a way to easily change the mass of either the player puck or the other X pucks in between compilations

# Homework

- Modify your previous homework (continued):
  - Friction should still be a toggle
  - Collision with the edges of the program window should still be totally elastic
  - Both friction (if enabled) and window edge collision should also apply to the non-player-controlled circles