Game Physics

John E. Laird

with a few modifications by Matt Evett

Based on *The Physics of the Game, Chapter 13 of Teach Yourself Game Programming in 21 Days,*

pp. 681-715

Why Physics?

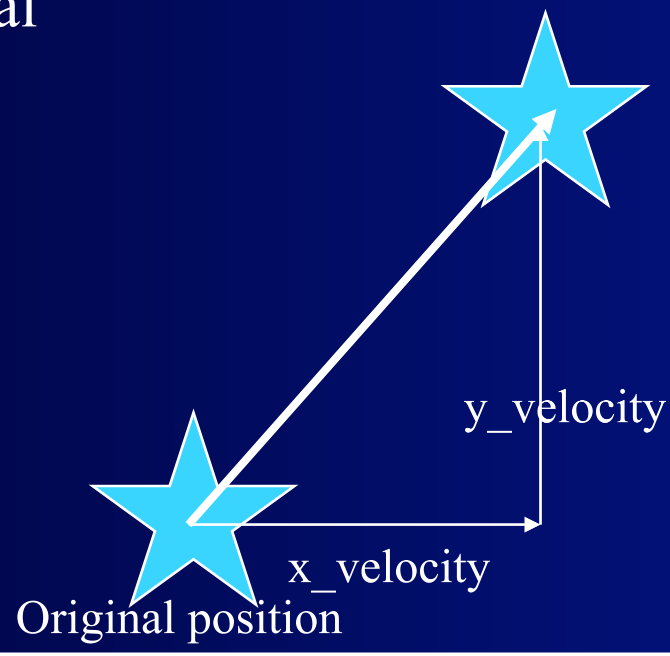
- Some games don't need any physics.
- Games based on the real world should look realistic, meaning realistic action and reaction.
  - More complex games need more physics:
    - sliding through a turn in a racecar.
  - Running and jumping off the edge of a cliff.
- If you try to do it from scratch, you might get it wrong.
- It is easy to get it right *approximately*
  - For Newtonian physics where f=ma
  - Rigid bodies
- Not easy:
  - Clothes, pony tails, a whip, chain, volcanoes, boomerang, organics

Computational Physics

- We don't want just the equations
- We want efficient ways to compute new values
  - Assume fixed discrete simulation – constant time step
  - Add t_n * for variable simulation
- Approach in talk:
  - 2D physics, usually easy to generalize to 3D (add z)
  - Rigid bodies (no deformation)
  - Will just worry about center of mass
    - Not accurate for all physical effects
  - Give basic equations at beginning
  - Give calculations need in discrete, constant step simulation.

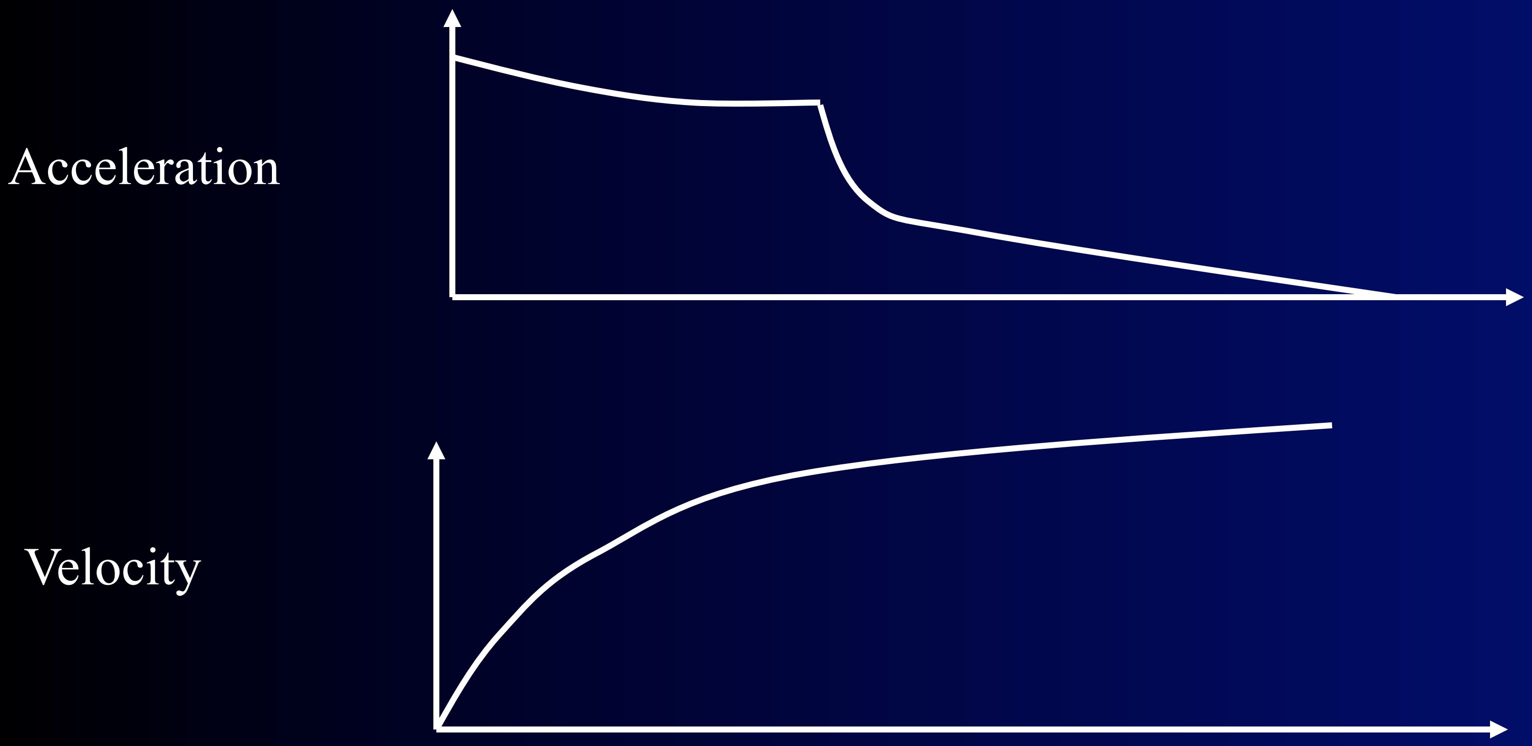
Position and Velocity

- Modeling the movement of objects with velocity
  - Where is an object at any time t?
  - Assume our metric is pixels
- Equations:
  - player_x(t) = t * x_velocity + x_initial
  - player_y(t) = t * y_velocity + y_initial
- Computation:
  - player_x=player_x + x_velocity
  - player_y=player_y + y_velocity



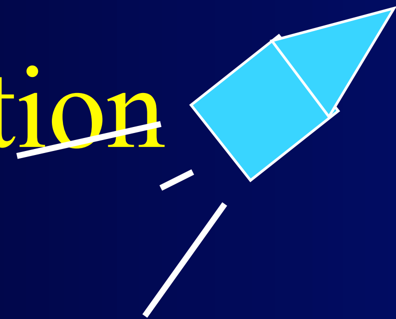
Acceleration

- Acceleration is change in velocity per unit time

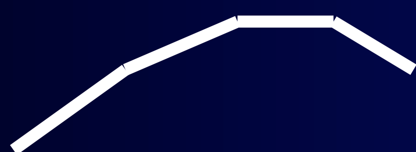


Approximate

Acceleration



- Computation:
  - x_velocity=x_velocity + x_acceleration
  - y_velocity=y_velocity + y_acceleration
- Changing acceleration:
  - use a table based on other factors:
  - acceleration = acceleration_value(gear, speed, pedal_pressure)
    - Cheat a bit acceleration = acceleration_value(gear, speed) * pedal_pressure
  - x_acceleration = cos (v) * acceleration
  - y_acceleration = sin (v) * acceleration
- Piece-wise linear approximation to continuous functions



Gravity

- Gravity is a force between two objects:
  - Force F = G * (M1*M2)/ D^2
    - G = Gravitational constant
    - D = Distance between the two objects
  - So both objects have same force applied to them
    - F=MA --> A=F/M
- On earth, assume mass of earth is so large it doesn't move, and D is relatively constant
  - Assume uniform acceleration

Gravity

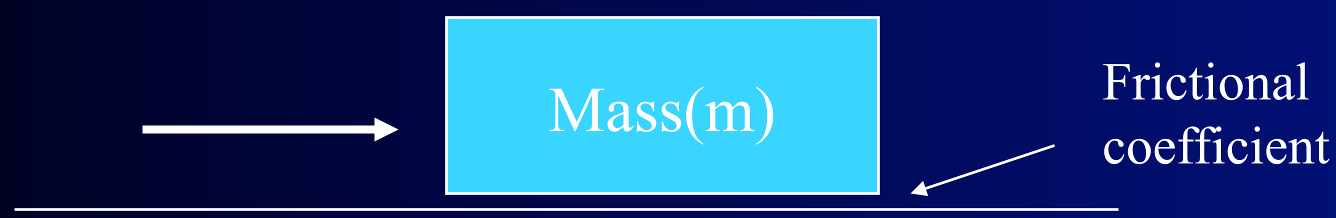
- Equation:
  - V(t) = 1/2g*t^2
  - g = 9.8 m/s^2 or 32ft/s^2
- Computation
  - x_velocity = x_velocity + 0
  - y_velocity = y_velocity + gravity
    - gravity must be normalized to time slice of game (or so that it looks “good”)

Space Game Physics

- Gravity
  - Influences both bodies
  - Can have two bodies orbit each other
  - Only significant for large mass objects
- What happens after you apply a force to an object?
- What happens when you shoot a missile from a moving object?
- What types of controls do you expect to have on a space ship?
- What about a flying game?

Friction

- Conversion of kinetic energy into heat
- Frictional Force = C * G * M
  - C = frictional coefficient = amount of force to maintain a constant speed
  - G = gravity
  - M = mass



- velocity = velocity - friction
  - For velocity > friction!
- Usually two frictional forces
  - Static friction when at rest. If velocity = 0. No movement unless overcome.
  - Kinetic friction, when moving (< static friction)

Race Game Physics

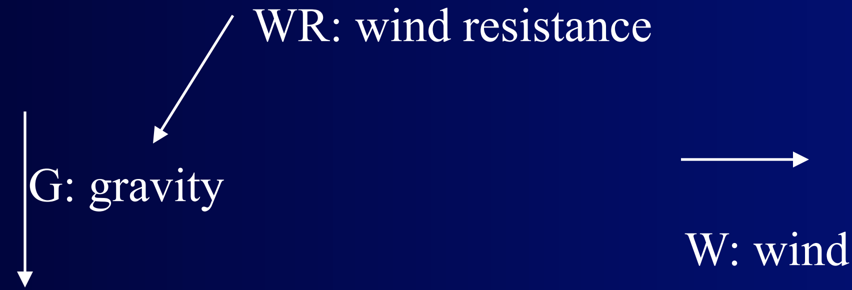
- Non-linear acceleration
- Resting friction > rolling friction
- Rolling friction < sliding friction
- Centripetal force?

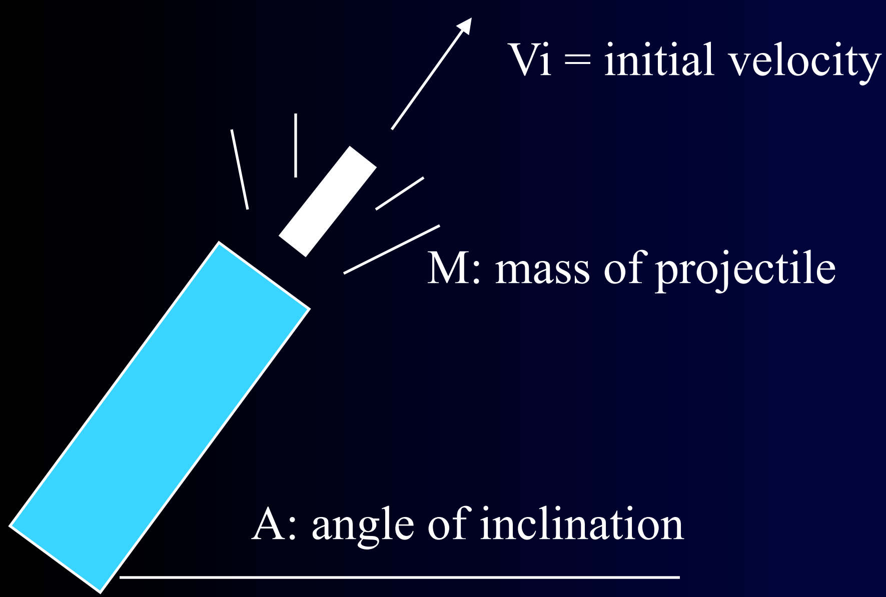
- What controls do you expect to have for a racing game?
  - Turning requires forward motion!

- What about other types of racing games
  - Boat?
  - Hovercraft?

Projectile Motion

- Forces





X= x + Vx + W

Y = y +Vy

Vxi = cos(A)*Vi

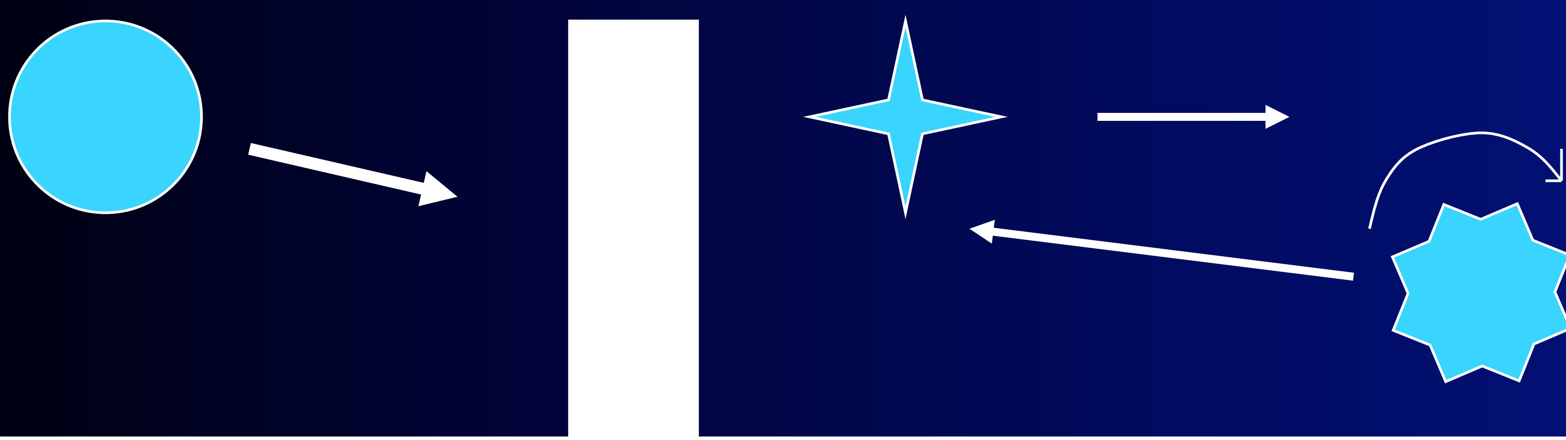
Vyi = sin(A)*Vi

Vx = Vx -WR(Vx)

Vy = Vy - WR(Vy) + G

Back to Collisions

- Steps of analysis for different types of collisions
- Different types of collisions
  - Circle/sphere against a fixed, flat object
  - Two circles/spheres
  - Rigid bodies
  - Deformable
- Model the simplest - don't build a general engine



Collisions: Steps of Analysis

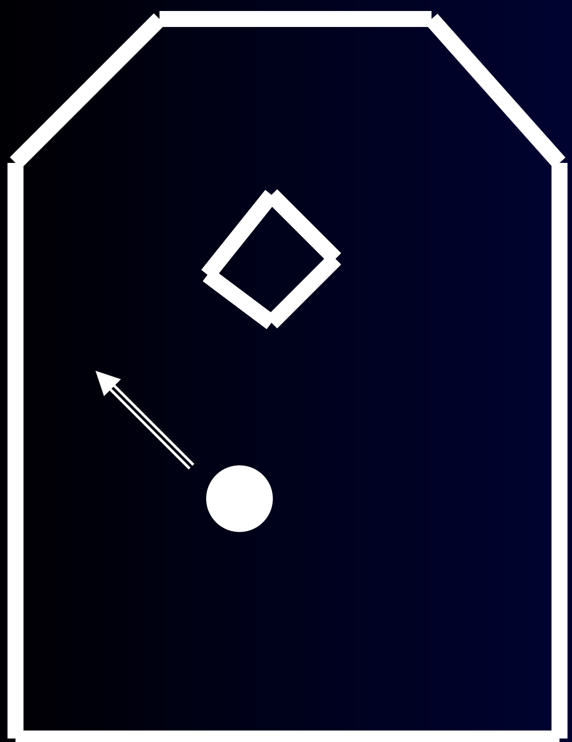
- Detect that a collision occurred
- Determine the time of the collision
  - So can back up to point of collision
- Determine where the objects are when they touch
- Determine the collision normal
- Determine the velocity vectors after collision
- Determine changes in rotation

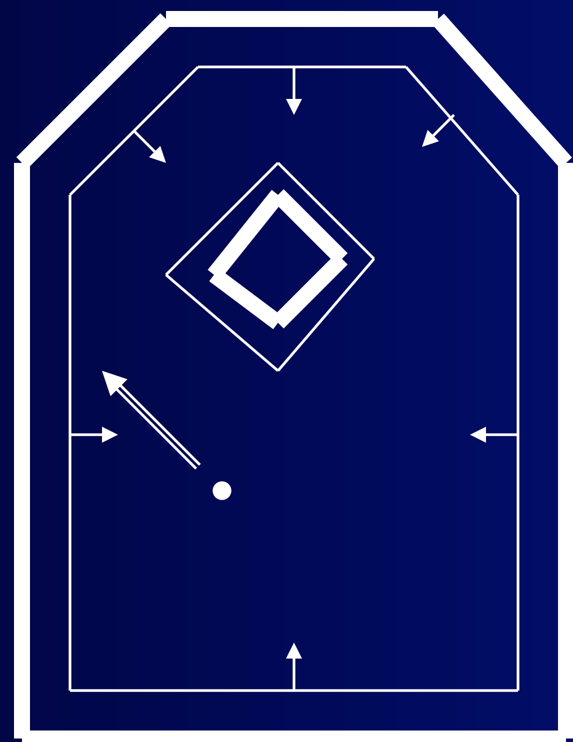
Circles and Lines 1

- Simplest case
  - Good step for your games - pinball
  - Assume circle hitting an *immovable* barrier
- Detect that a collision occurred
  - If the distance from the circle to the line < circle radius
  - Reformulate as a point about to hit bigger walls
  - If vertical and horizontal walls, simple test of x, y.

Circles and Lines 2

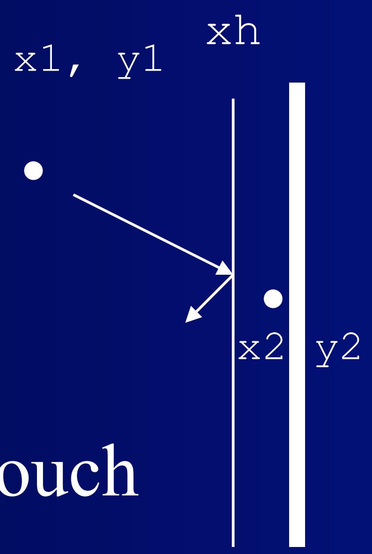
- What if more complex background: pinball?
  - For complex surfaces, pre-compute and fill an array with collision points (and surface normal).





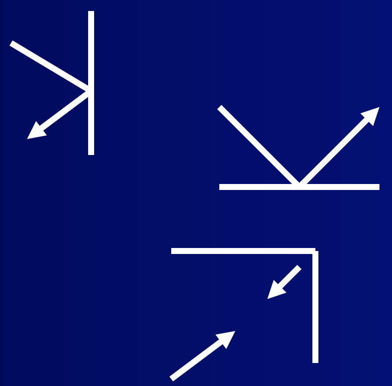
Circles and Lines 3

- Determine the time of the collision
  - tc = (xh-x1)/(x2-x1)*dt + ti
  - dt = delta time = time increment
  - ti = initial time
  - tc = collision time
- Determine where the objects are when they touch
  - yc = y1- (y1-y2) * tc; xc = xh
- Determine the collision normal
  - Angle of line using (x1-xh) and (y1-yc)



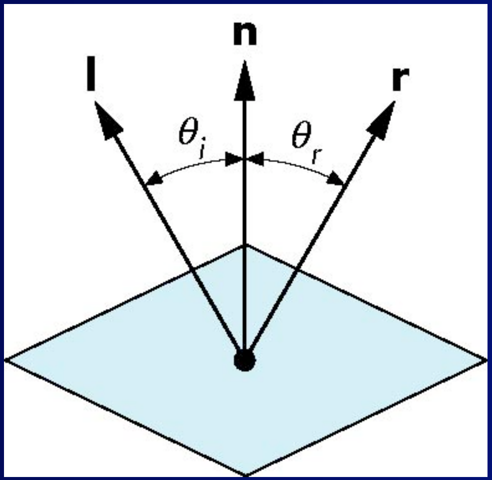
Circles and Lines 4

- Determine the velocity vectors after collision
  - Orthogonal to collision normal
  - Vertical - change sign of x velocity
  - Horizontal - change sign of y velocity
  - Corner - change sign of both
  - Other - invert velocity at collision normal



- Compute new position
  - Use dt - tc to calculate new position from collision point
- Determine changes in rotation
  - None! Unless we really want to add complexity of rotational dynamics
- How to "invert velocity" at collision…

Angle of Reflection



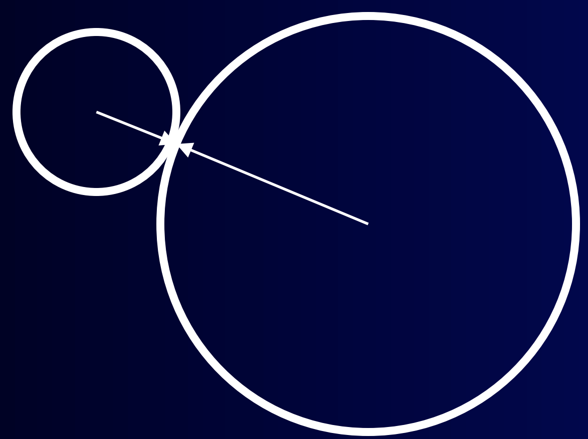
- Angle of incidence = angle of reflection
- Normalize the vectors (divide vector by its own length) so |l|=|n|=1
- We want |r|=1
- Since $\theta_i=\theta_r$, $\cos\theta_i=\cos\theta_r$
- Dot product: $\cos\theta_i = l \bullet n = \cos\theta_r = n \bullet r$
  - $V1 \bullet V2 = (x1x2+y1y2)$
- Because the vectors are coplanar we can rewrite $r = \alpha l + \beta n$, dot product with n=
- $n \bullet r = \alpha l \bullet n + \beta = l \bullet n$

More Reflection…

- $\mathbf{n} \bullet \mathbf{r} = \alpha \mathbf{l} \bullet \mathbf{n} + \beta = \mathbf{l} \bullet \mathbf{n}$
- Because r should be unit length
- $1 = \mathbf{r} \bullet \mathbf{r} = \alpha^2 + 2\alpha\beta\, \mathbf{l} \bullet \mathbf{n} + \beta^2$
- Two equations and unknowns; solving yields
- $\mathbf{r} = 2(\mathbf{l} \bullet \mathbf{n})\mathbf{n} - \mathbf{l}$
- **r** is new unit velocity, “denormalize” by multiplying by length of original velocity, **l**
  - **Elastic collision**

Circles and Spheres 1

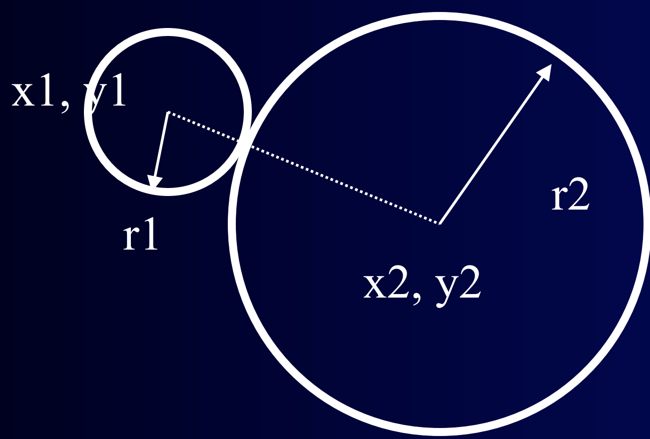
- Another important special case
  - Good step for your games.
  - Many techniques developed here can be used for other object types



- Assume elastic collisions

Detect that a collision occurred

- If the distance between two objects is less than the sum of their radii
  - Trick: avoid square root in computing distance!
  - $(r1 + r2)^2 > ((x1-x2)^2 + (y1-y2)^2)$



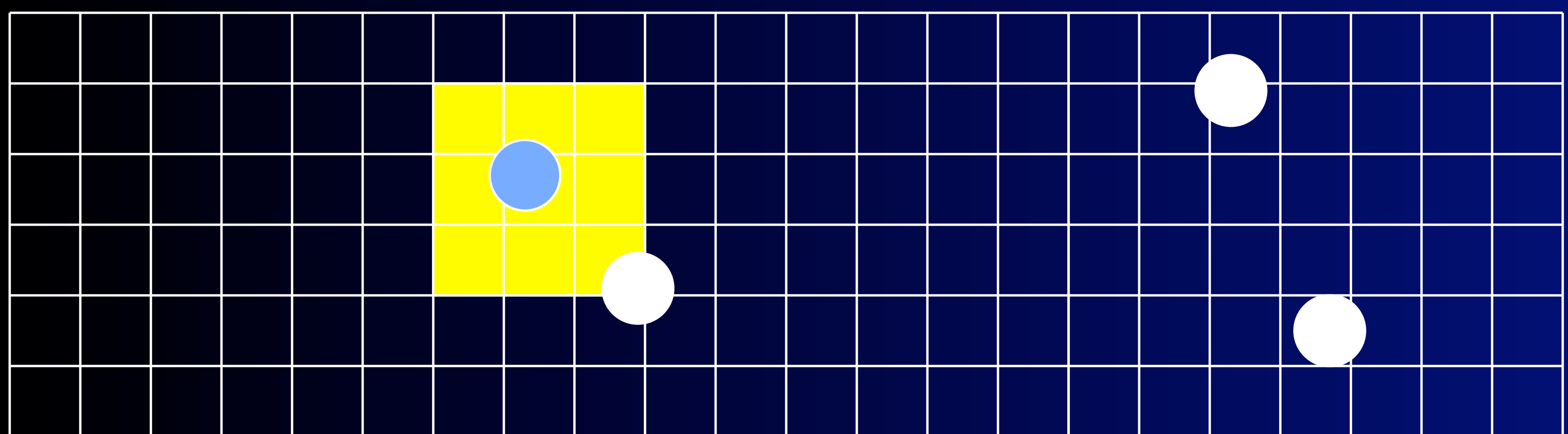
- Unfortunately, this is N^2 in number of objects

Detect Collision

- With non-circles, gets more complex and more expensive for each pair-wise comparison
- General approach:
  - Observations: collisions are rare.
    - Most of the time, objects are not colliding
  - Create series of filters so that only need to do expensive tests on very few pairs.

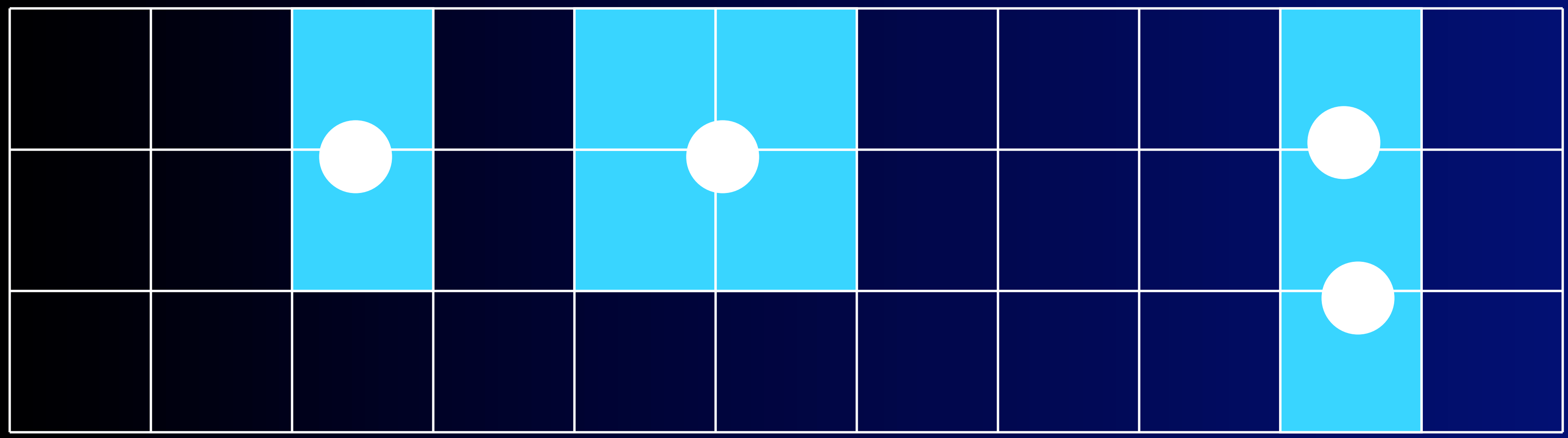
Detect that a collision occurred

- Avoid most of the calculations by using a grid:
  - Size of cell = diameter of biggest object
- Test objects in cells adjacent to object's center
  - Can be computed using mod's of objects coordinates:
    - bin sort (do you recall run-time of this alg.?)
- Linear in number of objects (usually—if no big clusters)
- For non-circles, use bounding circle/sphere



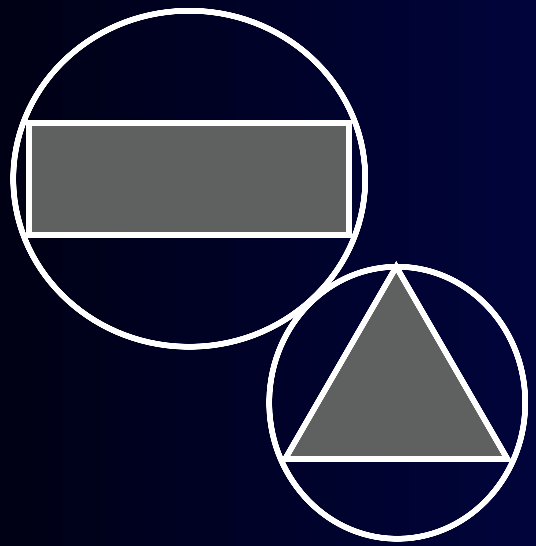
Detect that a collision occurred

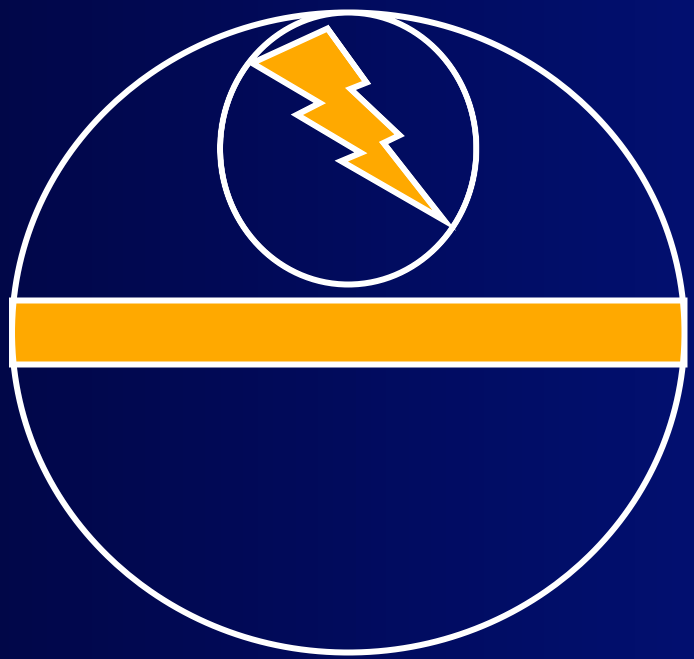
- Alternatively, if many different sizes:
  - Size of cell is arbitrary.
  - Here I used twice size of average object
- Test objects in cells touched by object.
  - Must determine cells object is in.
  - Works for non-circles too.



Detect that a collision occurred

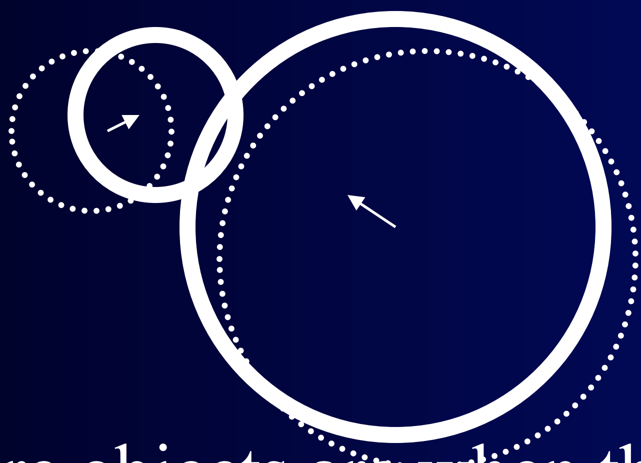
- For non-circles, next test could be to see if bounding circles/spheres overlap
  - Pretty cheap
  - Not great for thin objects
  - If circles overlap, can then test if the objects truly overlap





Circles and Spheres 2

- Determine the time of the collision
  - Interpolate based on old and new positions of objects.



- Determine where objects are when they touch
  - Backup positions to point of collision
- Determine the collision normal
  - Bisects the centers of the two circles at position where they collided

Circles and Spheres 3

- Determine the velocity
  - Assume elastic, and no friction.
  - Assume head on (can generalize to more dimensions)
- Conserve Momentum: Mass * Velocity
  - M1*Vi1 + M2*Vi2 = M1*Vf1 + M2*Vf2
- Conservation of Energy (Kinetic Energy)
  - M1*Vi1^2 + M2*Vi2^2 = M1*Vf1^2 + M2*Vf2^2
- Final Velocities
  - Vf1 = (2*M2*Vi2 + Vi1*(M1-M2))/(M1+M2)
  - Vf2 = (2*M1*Vi1 + Vi2*(M1-M2))/(M1+M2)
    - What if equal mass, M1 = M2
    - What if M2 is infinite mass?

Alternative Formulation

- Let u1 and u2 be the initial velocities, and v1 and v2 the final

$$v_1 = \left(\frac{m_1 - m_2}{m_1 + m_2}\right) u_1 + \left(\frac{2m_2}{m_1 + m_2}\right) u_2$$

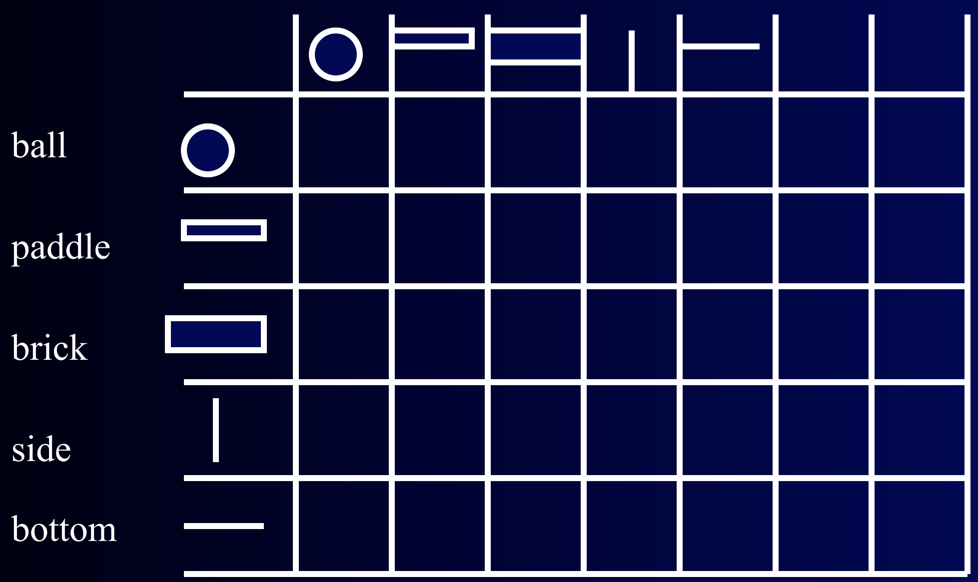
$$v_2 = \left(\frac{m_2 - m_1}{m_1 + m_2}\right) u_2 + \left(\frac{2m_1}{m_1 + m_2}\right) u_1$$

Must be careful

- Problems with round-off error in floating-point arithmetic.
  - Careful with divides
- Especially when have objects of very different masses
  - (can you see why this might be problematic?)

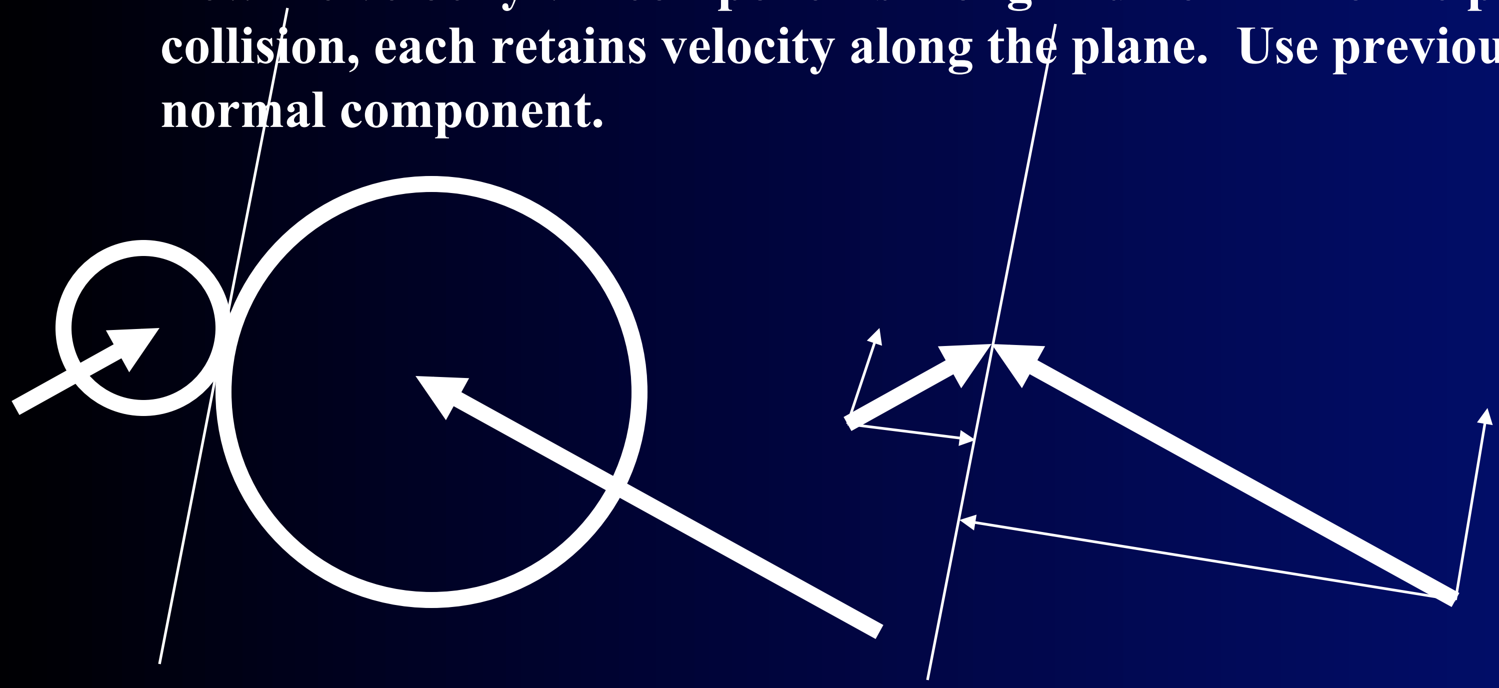
Avoiding Physics in Collisions

- For simple collisions, don’t do the math
  - Two identical balls swap velocities
- For collisions between dissimilar objects
  - Create a collision matrix



Circles and Spheres 4

- Non-head on collision – but still no friction
- Velocity change:
  - **Maintain conservation of momentum**
  - **Change of velocity orthogonal to the collision normal**
  - **Rewrite velocity via components along and normal to the plane of collision, each retains velocity along the plane. Use previous calc for normal component.**

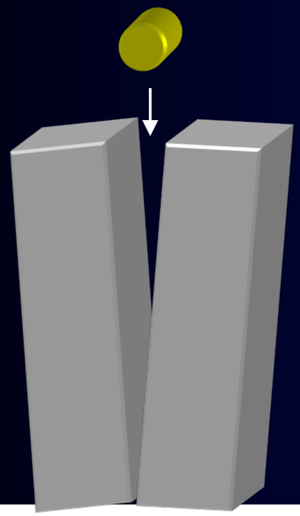


Internet Resources

- Nice discussion of the mathematics of elastic collisions at http://www.mcasco.com/p1lmc.html

Physics Engines

- Havok
- Strengths
  - Do all of the physics for you as a package
- Weaknesses
  - Can be slow when there are many objects
  - Have trouble with small vs. big object interactions
  - Have trouble with boundary cases



Particle System Explosions

- Start with lots of point objects (1-4 pixels)
- Initialize with random velocities based on velocity of object exploding
- Apply gravity
- Transform color intensity as a function of time
- Destroy objects when collide or after fixed time

- Can add vapor trail (different color, life, wind)

PseudoCode Particle Animation

- *Modified from devmaster.net*

<u>Set up particle</u>

```
While Animation In Progress
    If Particle Not Dead Then
        Particle Position += Particle Direction * Speed
        Particle Speed += Particle Acceleration
        Modify Particle's Speed
        Modify Particle's Energy

        If Particle's Energy < Threshold Then
            Mark Particle As Dead
        End If

        If Particle Hits Object Then
            Modify Particle's Position, Direction, Speed and Energy
        End If

        Display Particle
    End If
End While
```

Example Particle System

- See http://www.jhlabs.com/java/particles.html
- Can be used for explosions, sparkler trails, exhaust

Advanced Physics

- Modeling liquid
- Movement of clothing
- Movement of hair
- Fire/Explosion effects
- Reverse Kinematics