Algorithms

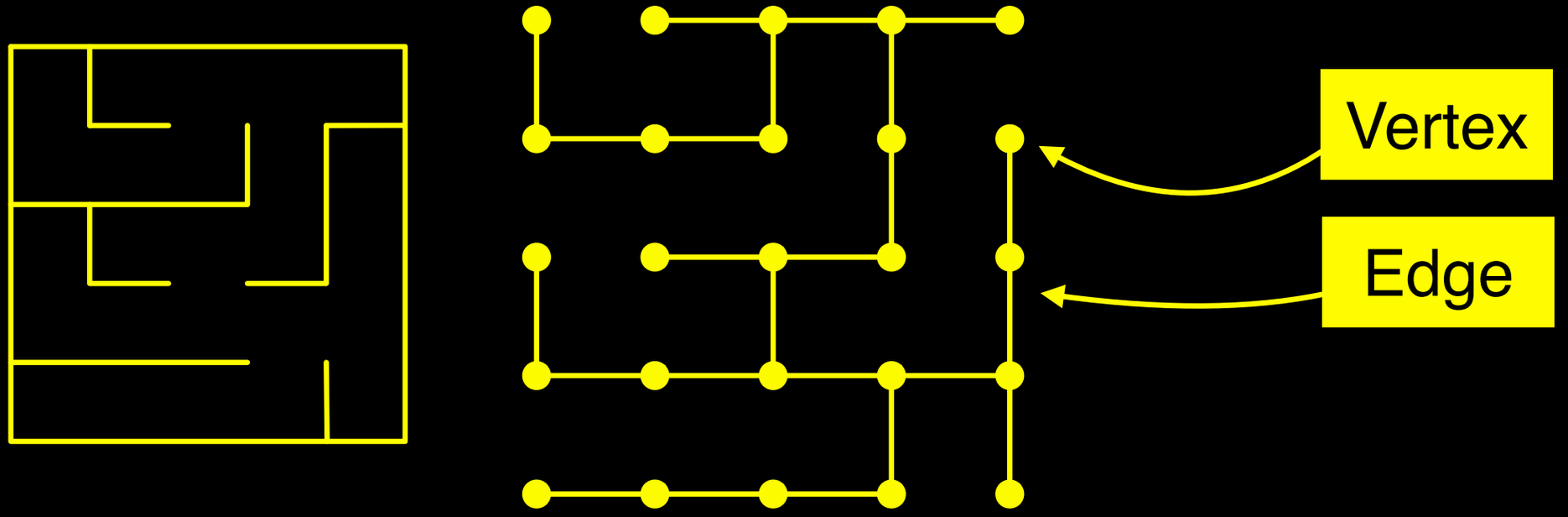
(for Game Design)

Session 5: Collision Detection

Last time

- Navigation
- Breadth First Search
- Dijkstra's Algorithm
- Greedy Best First Search
- A*
- Navmeshes (Funnel Algorithm)

Mazes are Graphs



- Each cell in a maze ➙ vertex in a graph
- Linked cells ➙ edge in graph

Breadth First Search

- We've seen this before
  - Flood fill using a frontier (FIFO queue)
- Continue until frontier is empty
  - Pick the first location from the frontier
  - Mark it as "visited"
  - Add unvisited neighbors to the frontier

Dijkstra's Algorithm

- Calculates the minimum cost path from one vertex to all other vertices
- Implies that edges have costs
  - Terrain costs
  - Does it cost more to go through a door? Or to cross a river?

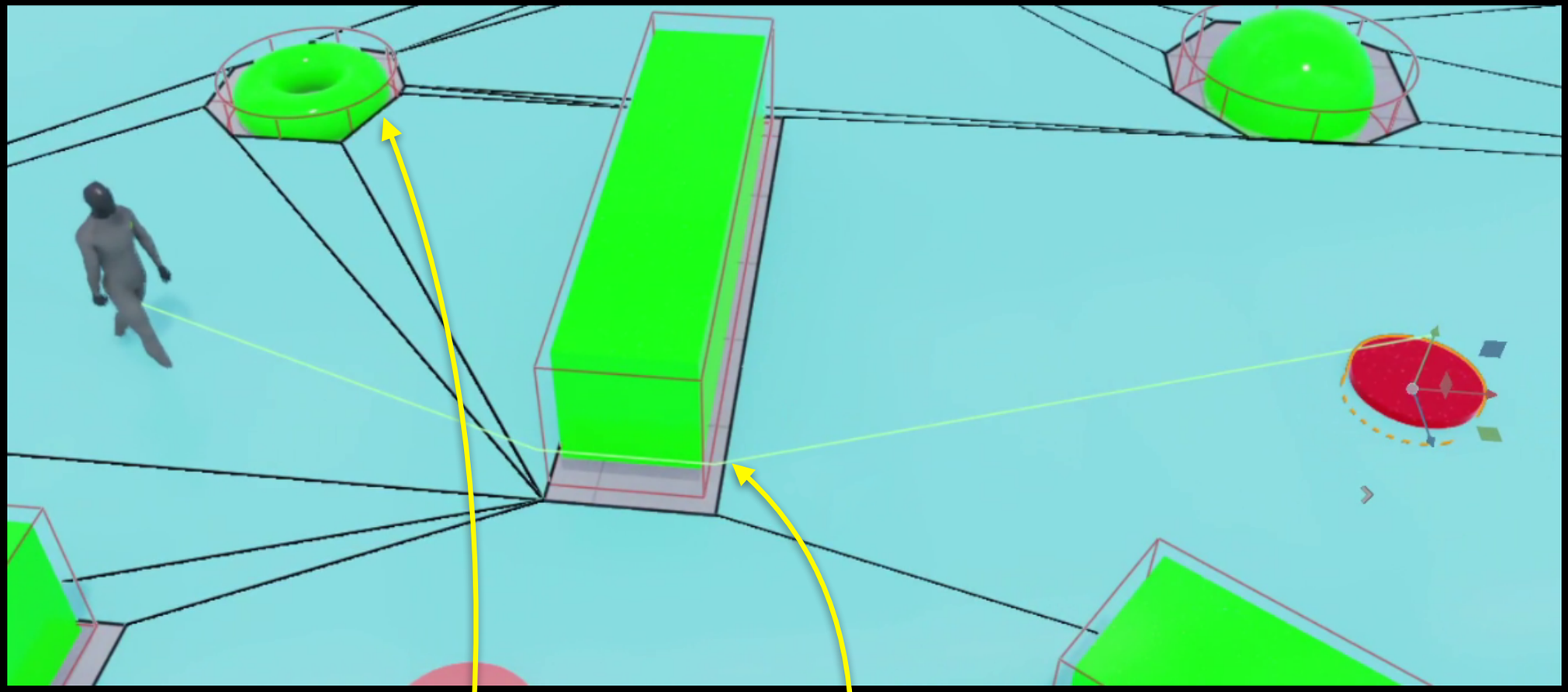
Efficiency!

- BFS and Dijkstra's spread out to search in all directions
  - If there is only a single goal, this is inefficient
  - Just head towards the goal
- But, if we knew which step was the best way to the goal, we'd already know the entire path

A* Algorithm

- A graph search algorithm
  - Developed at SRI in 1962
  - Developed for robotics
- A best-of-both worlds algorithm, combining Dijkstra's and GBFS

Example



Notice that obstacles are not included -- they are just a hole in the mesh

Image credit: NVIDIA Omniverse Documentation

Today

- Collsions, with some history
- Problems
  - Scale
  - Move through
- Solutions: a 4-step plan

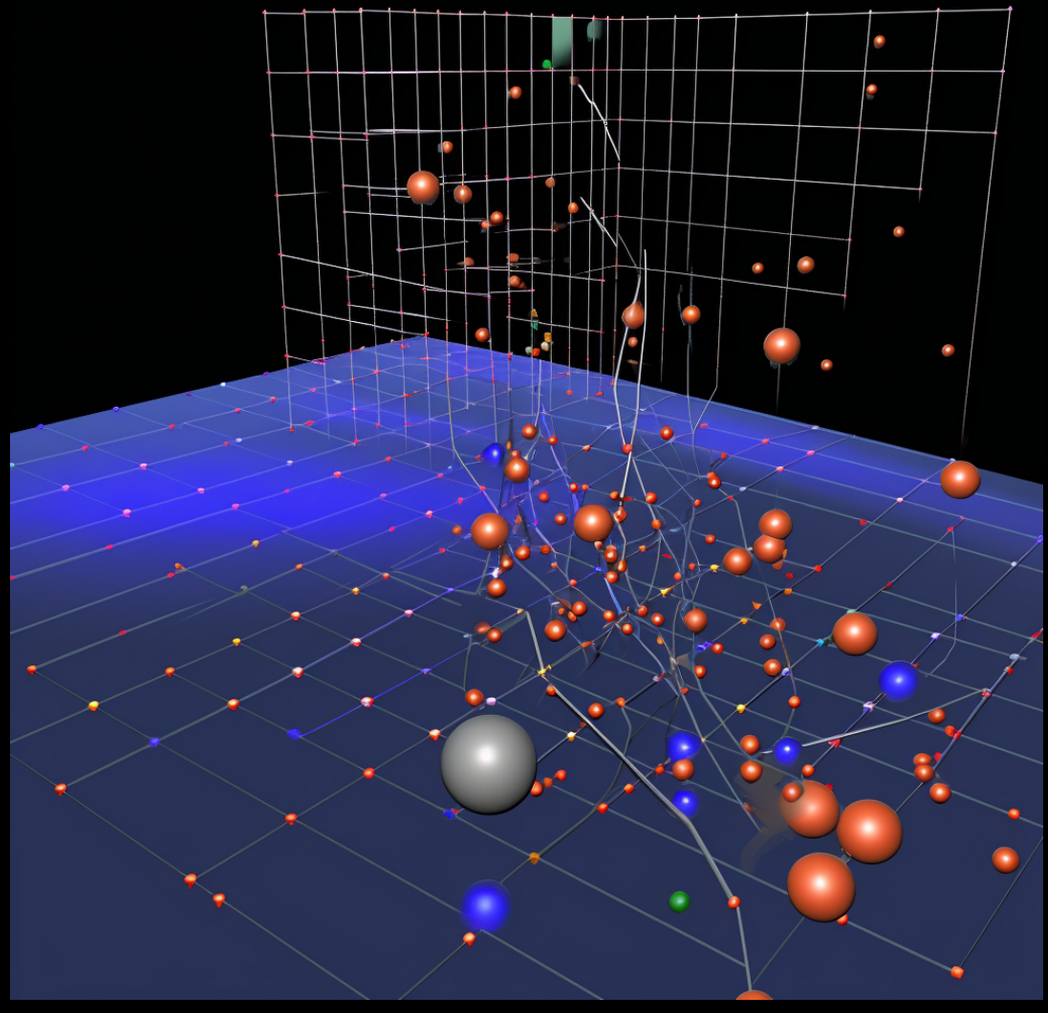
collision detection

Collision Detection

- We know how to draw a sprite
  - blit the sprite.image into a rect on the display surface
- But, how should sprites interact with each other?
  - What if one sprite "walks" into another?
  - How do we know if they "bounce" into each other?

Collision Detection

- Also known as physics simulation
- Determining if/how objects interact
- Based solely on mathematical notations about those objects



Remember the Game Loop?

- Each object (sprite) is a collection of positional data
- Each time around the game loop, that data is updated

```
mario_rect.bottomleft = (mario_rect.left   + velocity[0],
                         mario_rect.bottom + velocity[1])
```

- A naive algorithm would check each pixel and see if it is drawn in two or more sprites
- Better: for each pair of sprites, s_1 and s_2:

  p_1 is the set of pixels drawn by s_1

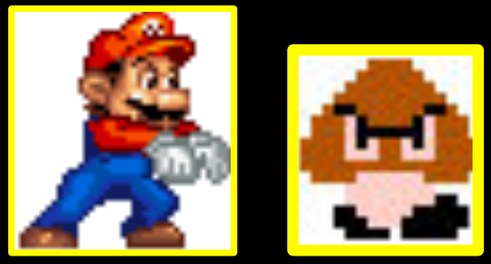
  p_2 is the set of pixels drawn by s_2

  is $p_1 \cap p_2 = \emptyset$? (in which case, no collision)
- That's still lots of CPU power!

TMBABW!

Abstraction!

- What if we just check if their rects overlap?
  - Much simpler than creating the sets and calculating intersection



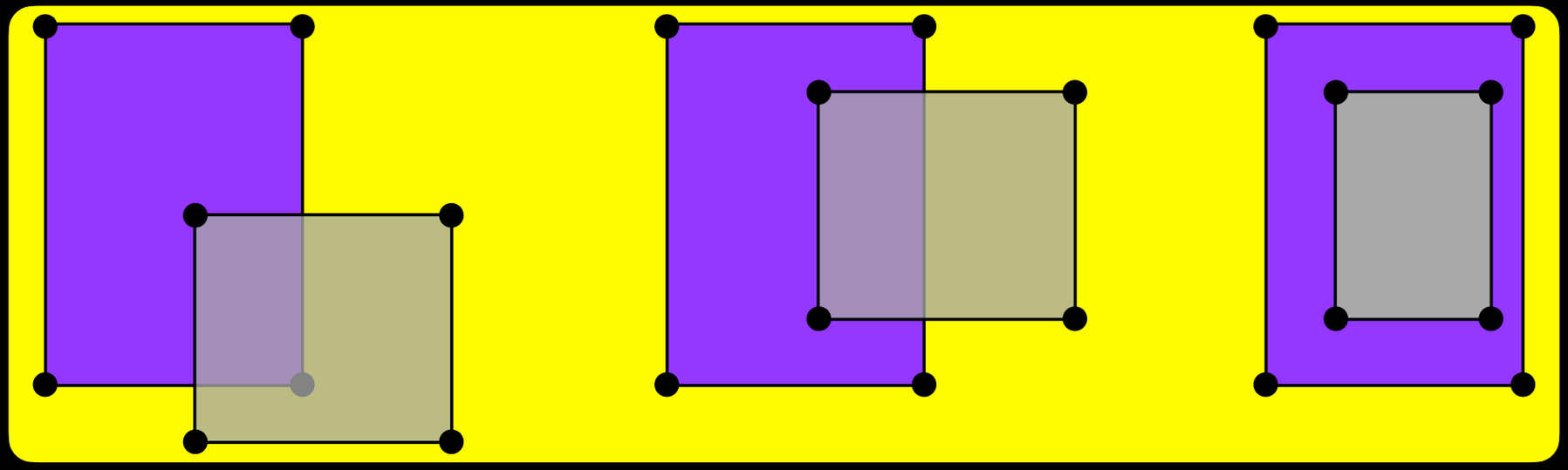
- Occasionally incorrect

Rect Collision Algorithm

- Test each corner of R_1 to see if it is *inside* the perimeter of R_2
- Pygame's `rect` has `collidepoint` method
  - Easy to imagine what it is doing
  - Checks if point's x coordinate is in range [`rect.x` to `rect.x + rect.width - 1`]
  - Repeat for y coordinate

```
def collision_2rects(r1, r2):
    ''' Return True if the two rects overlap.  Test corners
        of each rect against the other rect.
    '''
    if (r1.collidepoint(r2.topleft) or
        r1.collidepoint(r2.topright) or
        r1.collidepoint(r2.bottomleft) or
        r1.collidepoint(r2.bottomright)):
        return True
    # Need to test r2 for collisions with r1 now
    if (r2.collidepoint(r1.topleft) or
        r2.collidepoint(r1.topright) or
        r2.collidepoint(r1.bottomleft) or
        r2.collidepoint(r1.bottomright)):
        return True
    return False
```

- Why is the test bi-directional? (so many tests!)
- There are situations where one rect's corners aren't actually inside the other rect



- The `rect` class has a `colliderect( )` method
  - Implemented differently, functions the same

Summary

- Model each sprite as a rect
- Each time around the game loop, after updating locations ...
- Check each sprite to see if its rect collides with each other sprite's rect

This is a bad approach, for two reasons

Tyranny of numbers

Move-through

Two Problems

- Tyranny of Numbers
  - Scale: Too many pairwise collision checks
- Move-through
  - Time is quantized by the game loop
  - Stuff might happen between frames

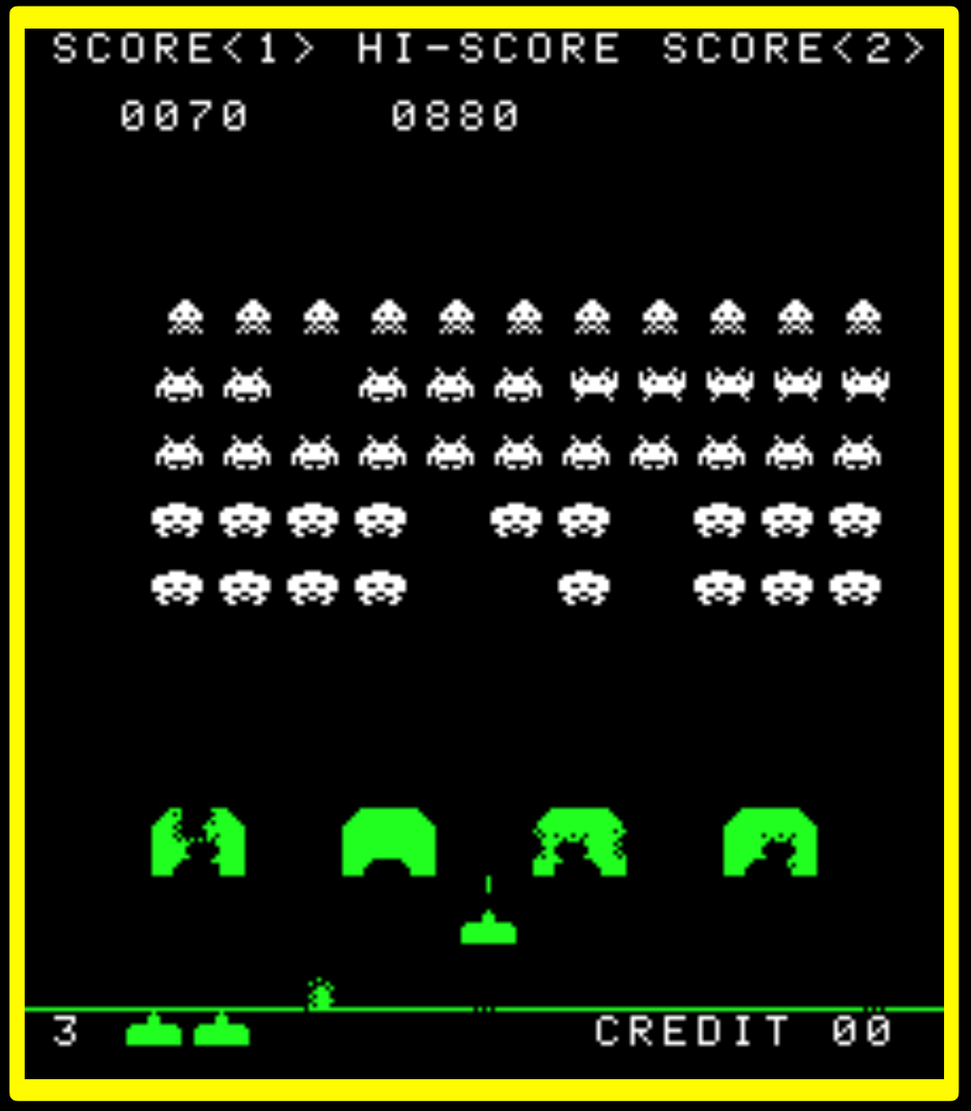
tyranny of numbers

Tyranny of Numbers

- Scale!
- There are a lot of sprites
- Checking collisions between every pair takes too much time and CPU power

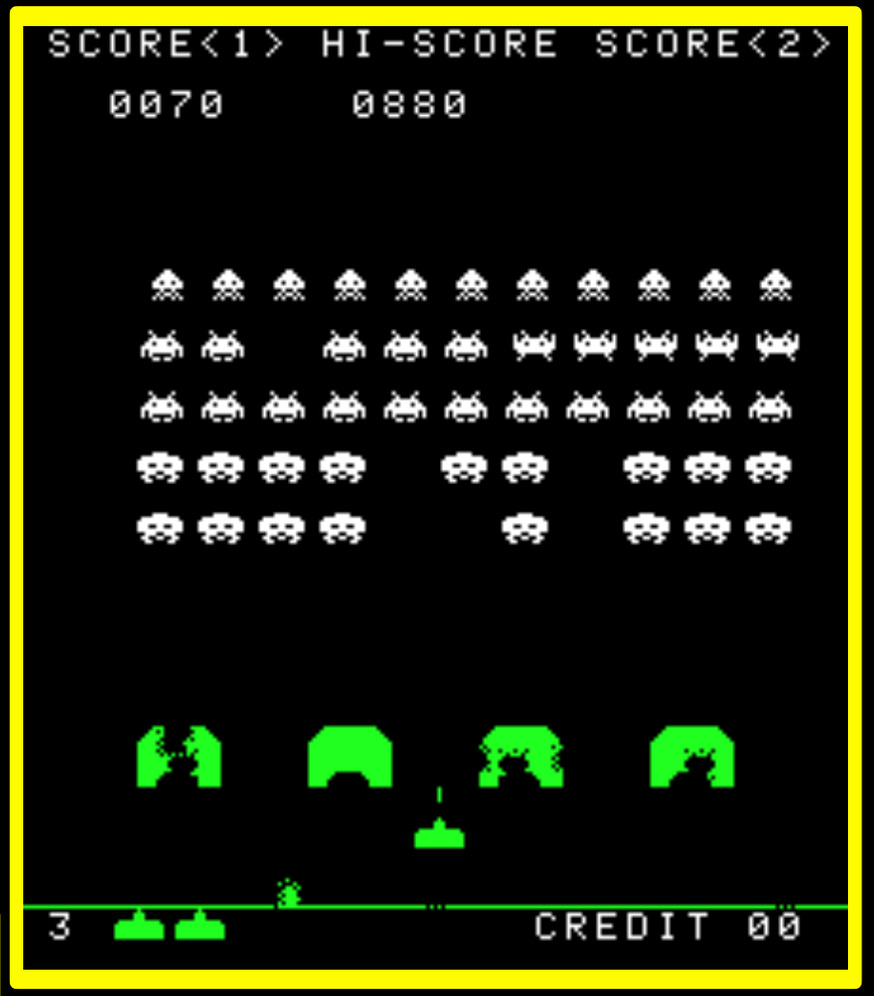
Historical Example

- Space Invaders
- 1978 Arcade Game
- Most money earned (infl. adj.)
- Simple rules



Object Count

- Aliens: 50-70, level dependent
- ~12 alien bombs
- 4 or 5 bunkers
  - Can be blasted into pieces
- Player's cannon and bullet
- Flying saucer



Total: about 100 objects

Checking 100 objects

- With *n* objects, testing each object for a collision with all the others will require $n(n-1)$ tests
  - The tests are bidirectional, so half that
- Space Invaders: n=100 → $\frac{1}{2}n(n-1) = 4950$
- All tests for each frame (60fps)

Thats 1/4 million tests per second

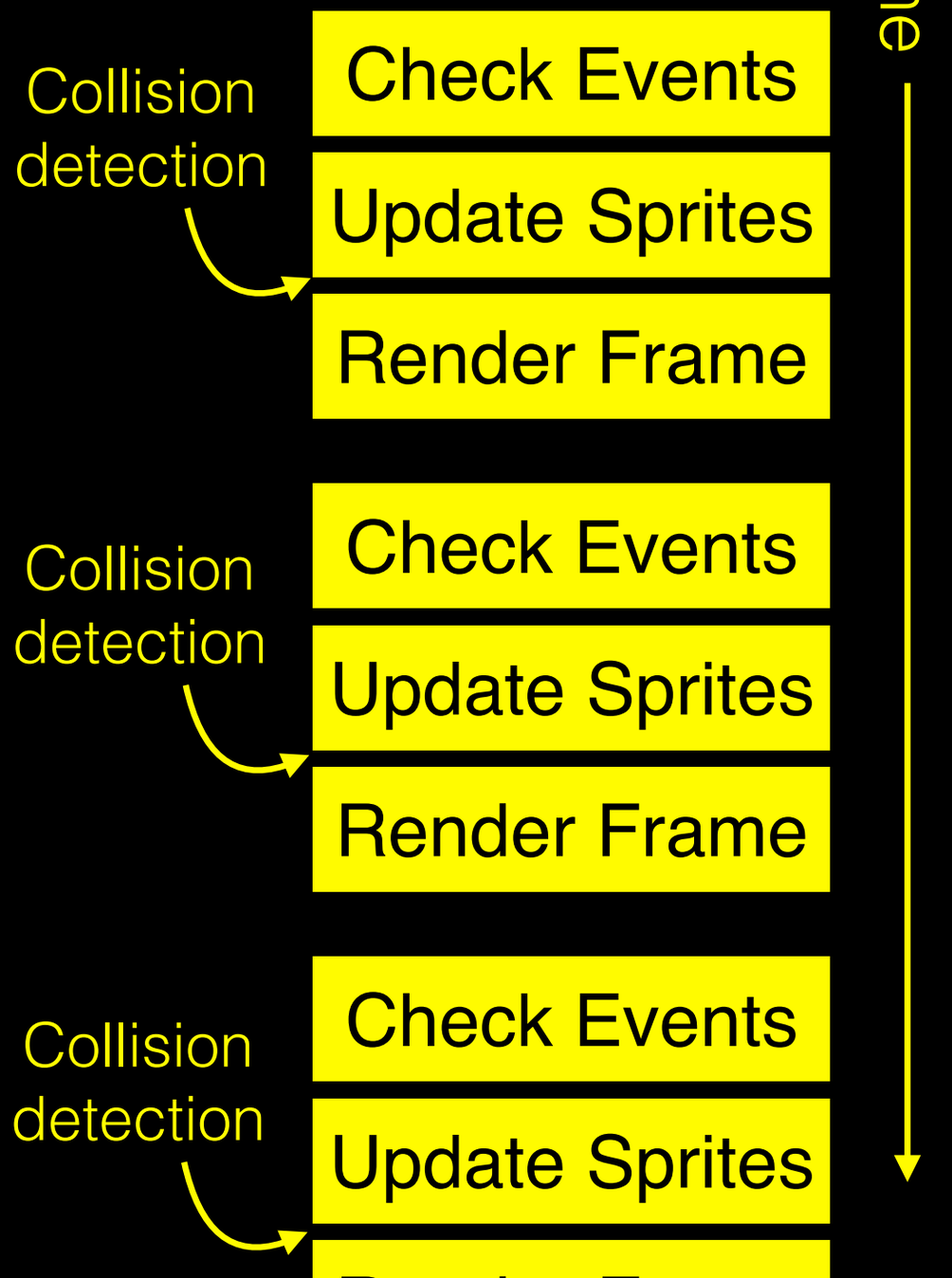
.25M Tests?

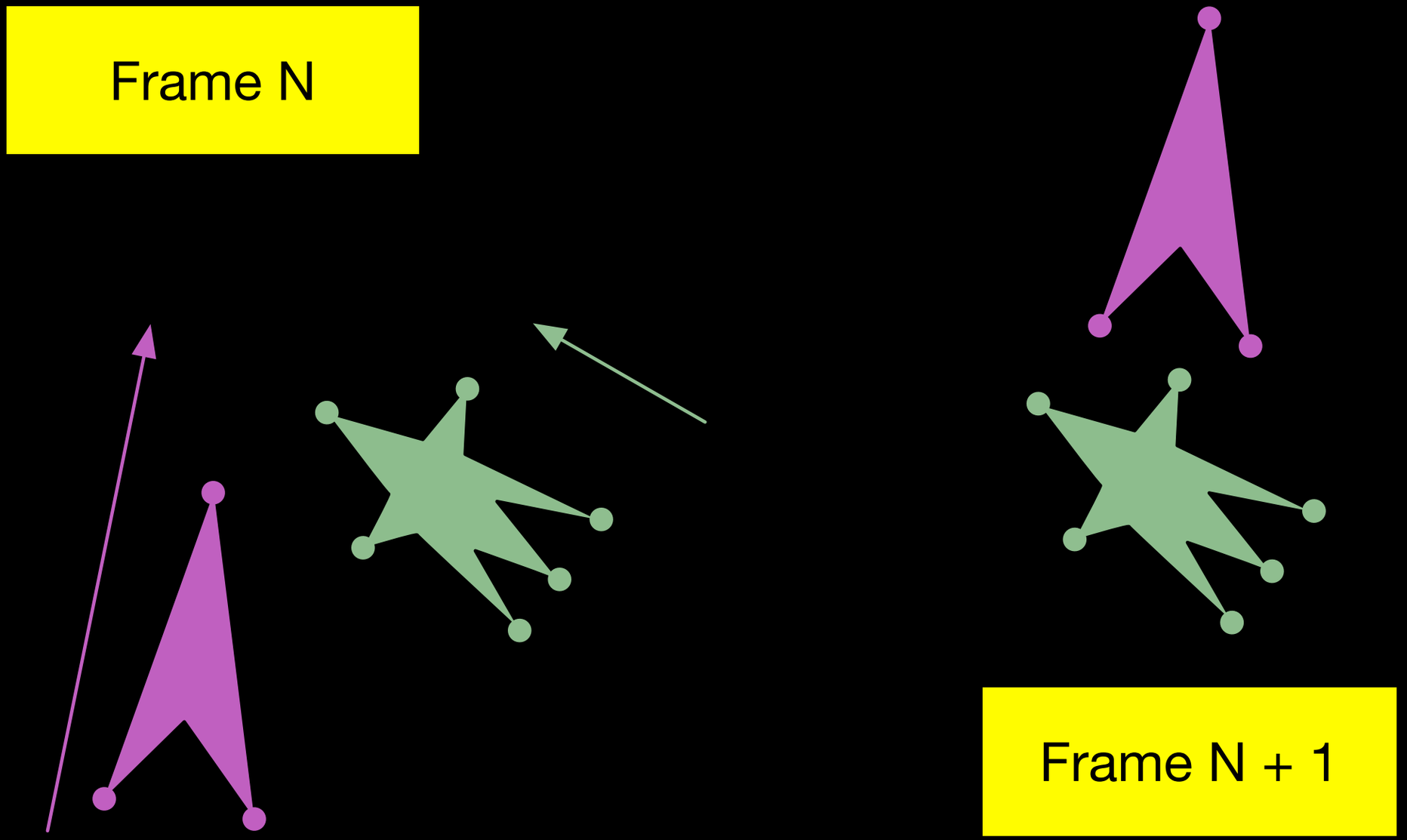
- On modern hardware, that's not many
  - Today's games have lots more objects
- But, on 1978 machines, they only had ~1M operations per second
- Bottom line: We need to find ways to cut down on the number of tests performed

move through

Time in Our Game

- Collision detection checks happen at "quantized" times
- What happens if collisions happen quickly *between* these points in time?




Frame N
Frame N + 1

Solution

- A better detection algorithm is necessary!
  - But, that algorithm is going to be more complex and involve more code
  - And thus take longer

Strategy

1. **Reduce** the number of checks that need to be made
    - This can be for *operational* reasons
    - Or because sprites are *geographically separated*
2. **Quickly test** sprites (pairwise): discard those that can't possibly collide
    - A very quick, but not very precise test
3. **Thoroughly test** remaining sprites to detect collisions
    - An in-depth test that is very precise
4. **Resolve collisions:** Do whatever -- explosions, bounces, etc.

step 1: reduce checks

Operational Reasons

- These are game-specific checks
- Examine how the game works at a general level to find efficiencies

Look For

- Objects that don't move
  - Don't test them against other motionless objects
- Objects that are allowed through certain other objects
  - Example: "Friendly" Fire

Reductions

- This examination will usually find huge numbers of checks that don't need to be made
- In general: you save these checks by not writing the code to do the check in the first place
  - Organize your sprites into groups and only test certain ones

- Example
  - All alien ships move with the same vector
  - Bunkers are stationary
  - Alien missiles are "friendly" to alien ships

Geographic Reduction

- Another "game-specific" type of check
- **Distant objects don't need to be checked for collision**
- What "distant" means is dependent upon the game

Off-screen Objects

- In many games, the screen represents a portion of the overall environment
  - Perhaps the maze is so huge, you only see part of it
  - In a FPS, you can't see what's behind you

Off-screen collisions?

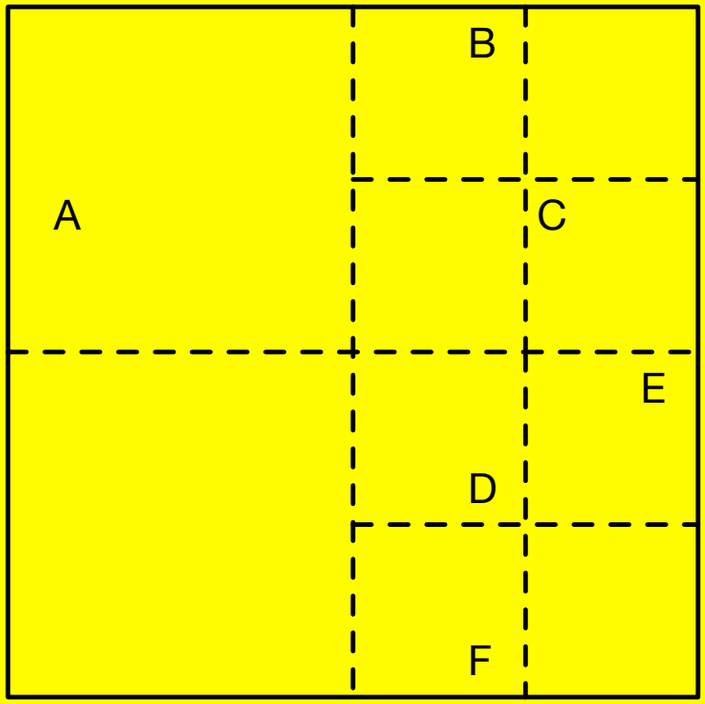
- Maybe, maybe, maybe, off-screen objects need not be tested for collisions
- Much care is needed here, as ignoring the results of off-screen objects can lead to strange situations should a player glance away and then glance back

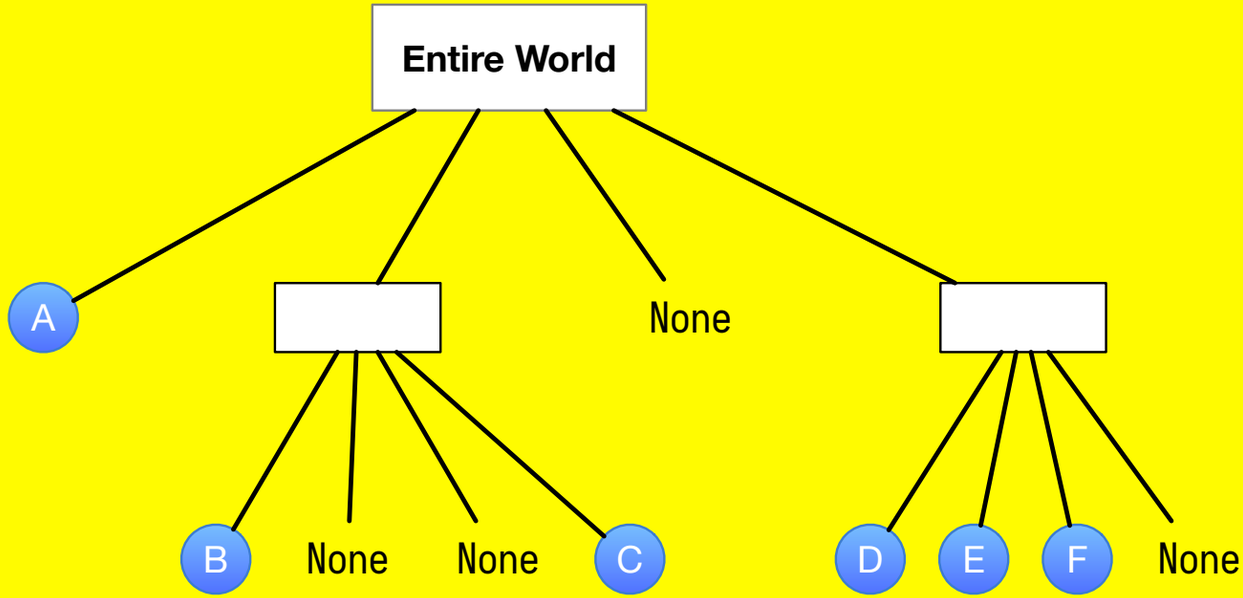
Restricted Locations

- Some objects can only exist in certain areas
- Thus not subject to collision with objects that can't enter those areas
- Example: Space Invaders flying saucer can't collide with anything other than the player's bullet
- Example: player's gun cannot collide with bunkers, players bullet, flying saucer

Partitioning Objects

- Perhaps keep track of objects in a special data structure based on geographic locations
- A **quadtree** is a data structure that partitions the world into quarters
  - Each quarter of which is also a quadtree


B
A
C
E
D
F


Entire World
A
None
B
None
None
C
D
E
F
None

Quadtrees

- The quadtree might be managed based on classes of objects, for operational reasons
  - i.e. a quadtree of all the alien fighter ships
- Quadtrees are also useful to find the nearest-neighbor
  - ex: If you want to know which player a monster will attack
  - An O(log n) search

Other partitions

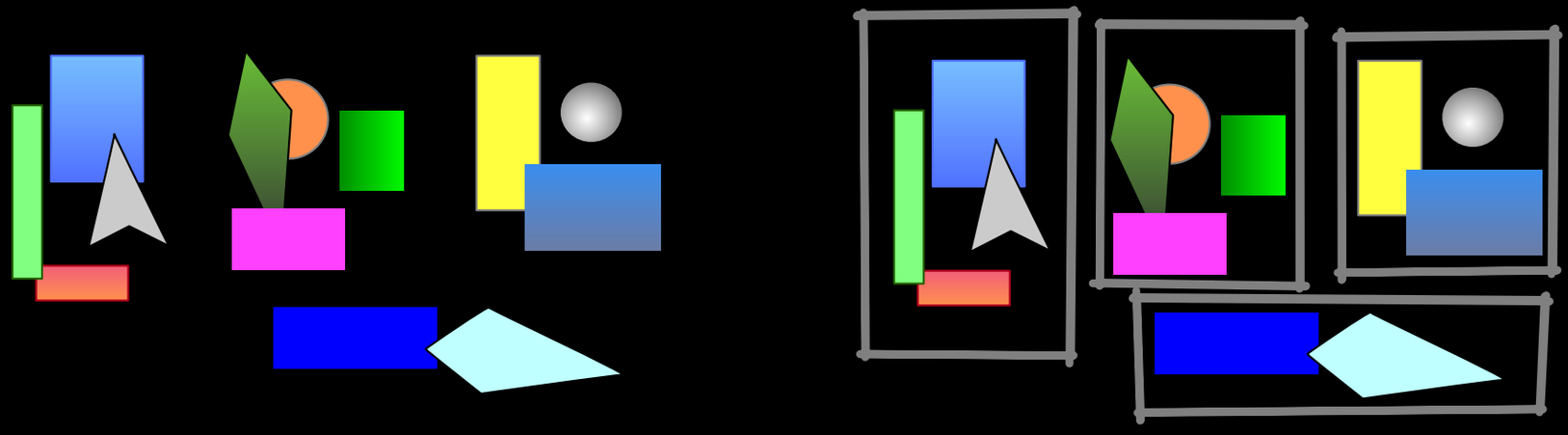
- Octrees are a 3d version
- Binary Space Partitioning (BSP) splits space in two at each level, but doesn't have to be in half
  - Representation is a BSPtree
- K-d trees are another technique, a special case of BSPtrees

Game-specific

- Depending on the game, you can also have specific partitioning schemes
- Ex: In a race game, you might split the track into portions
  - Keep track of which cars are in each portion
- Analyze the speed and size of objects to decide how big each portion should be
  - Ideally, each portion should only contain objects that can potentially collide

Automated Partitioning

- There are algorithms to do more automated partitioning
  - Goldsmith-Salmon Incremental Construction Method
  - Bottom-up n-ary Clustering



step 2: quick tests

Strategy

1. **Reduce** the number of checks that need to be made
    - This can be for *operational* reasons
    - Or because sprites are *geographically separated*
2. **Quickly test** sprites (pairwise): discard those that can't possibly collide
    - A very quick, but not very precise test
3. **Thoroughly test** remaining sprites to detect collisions
    - An in-depth test that is very precise
4. **Resolve collisions:** Do whatever -- explosions, bounces, etc.

Is a Collision Possible?

- Quick test ➔ discover if a collision is possible, not if a collision actually happened
  - Most potential collisions don't actually happen
  - So, spend just a few CPU cycles to discard most of the potentials

Which Test?

- There are many different quick tests
- Use the one(s) based on what the shapes are and how you represent them in your game

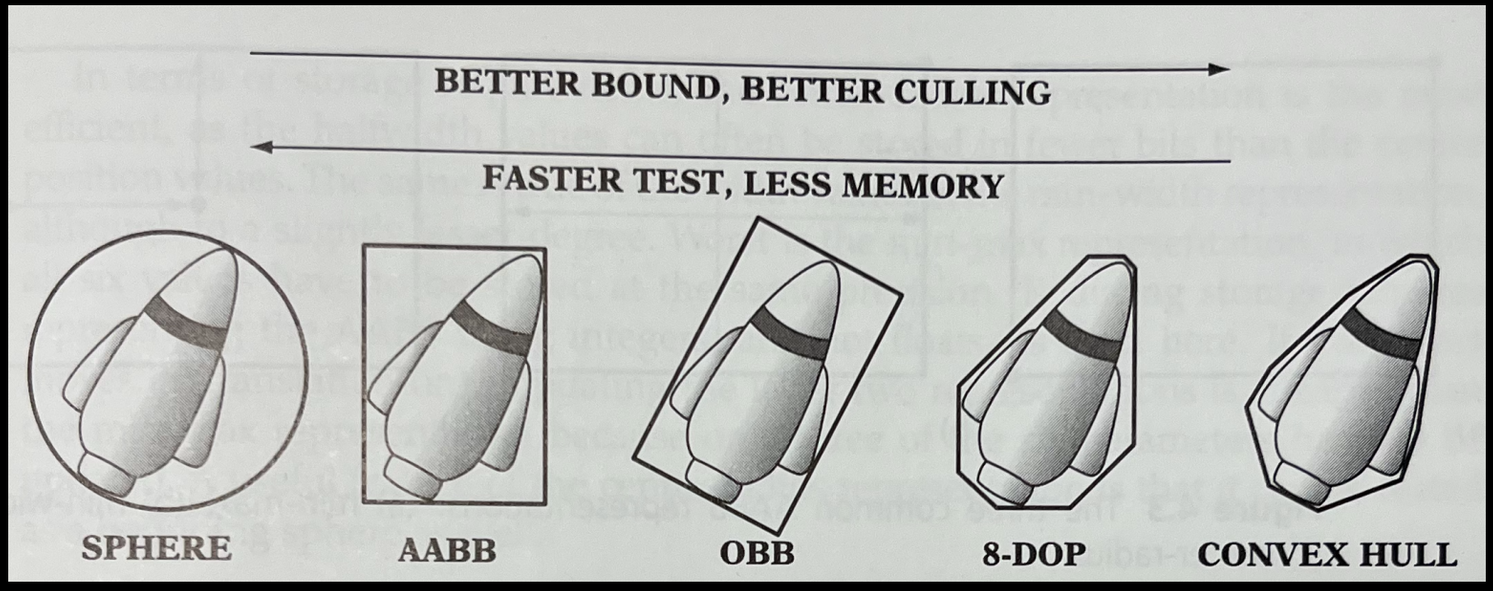
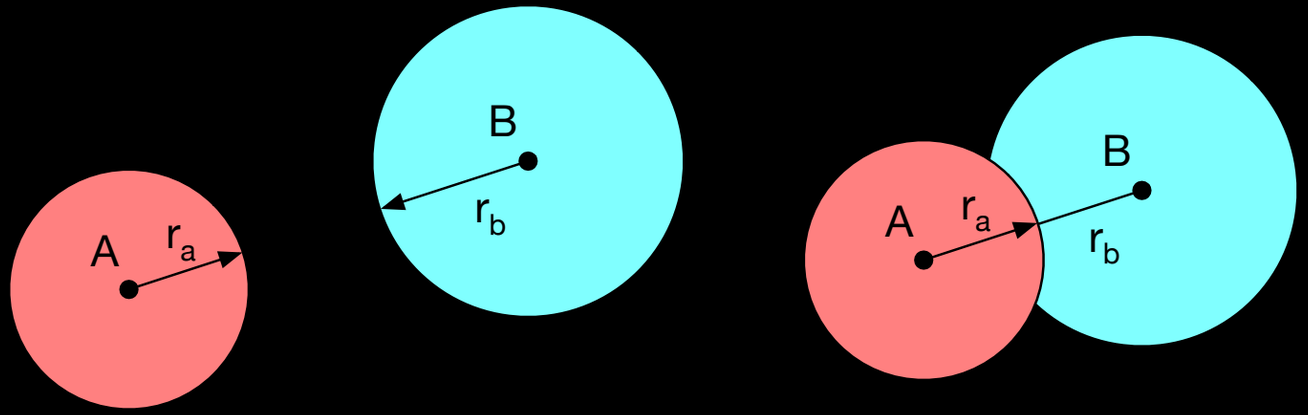


Image credit: *Real-Time Collision Detection*, by Christer Ericson

Circle / Sphere Test

- Enclose the sprits in circles
  (or, for a 3d game, in a sphere)
- Many objects fit well within a circle anyway:
  asteroids, Mario shells, centipede segments, etc.
- If two circles collide, the distance between their centers must be less than the sum of their radii

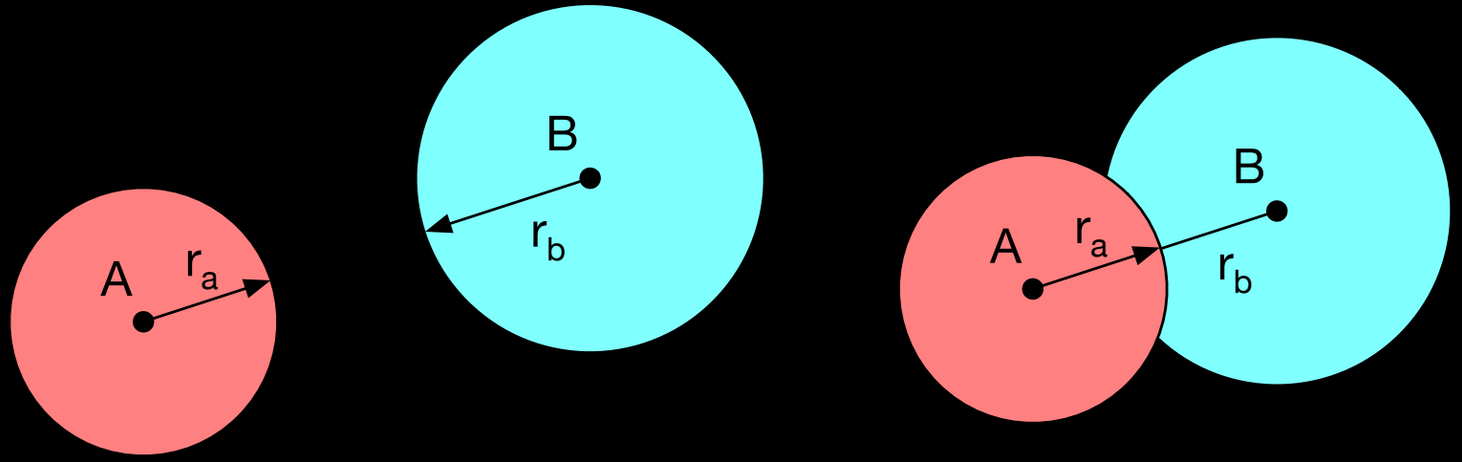


- The distance between the center of A and B is the square root of the sum of the distances squared

$$\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} < r_a + r_b$$

- Square roots are a very, very expensive calculation, so we prefer:

$$(a_x - b_x)^2 + (a_y - b_y)^2 < (r_a + r_b)^2$$



Bounding Box Test

- A **bounding box** is a rectangle that just barely fits the object
  - Some are aligned with the axis (AABB) like a `rect`
  - Some are aligned with the object (OOBB)
    - Object-oriented bounding box has nothing to do with OO programming

Bounding Boxes

Axis-aligned

Also AABB

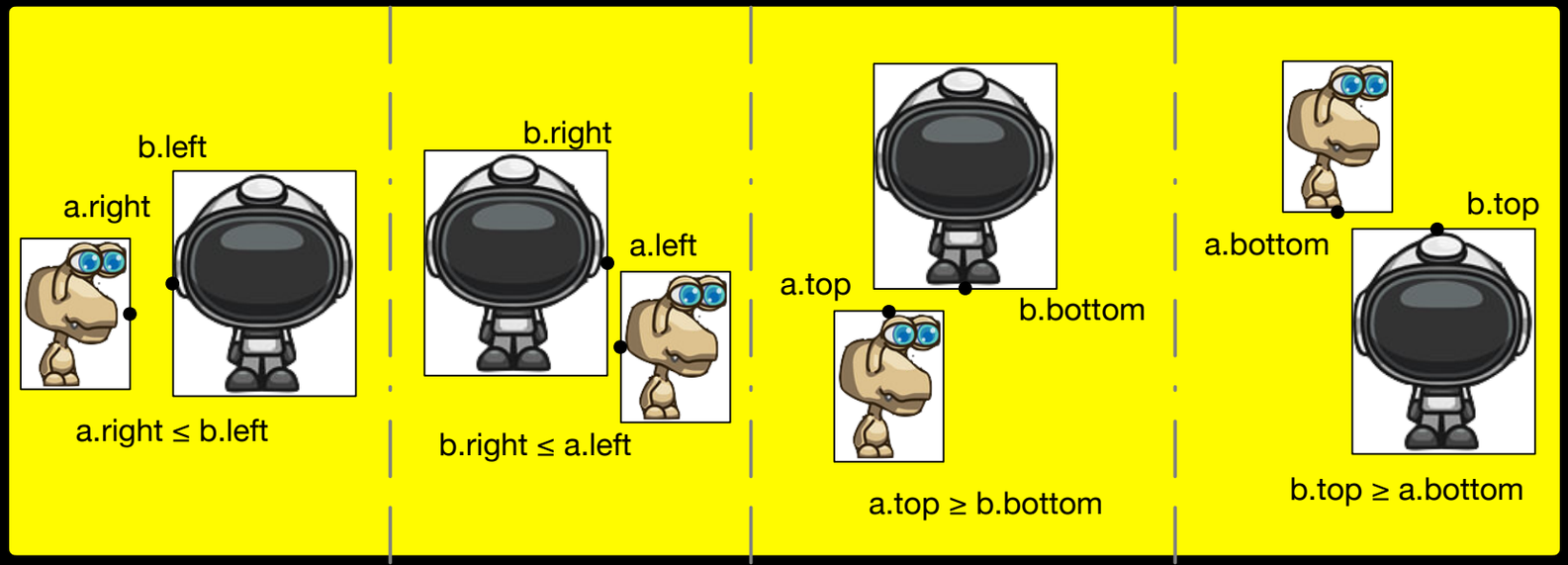
Object-oriented

Must adjust when object rotates

A Better Collide Rect

- I showed code to do an AABB collision test earlier (the `collision_2rects` function)
  - Relied upon the `Rect.collidepoint` method
  - Big `if` statements with lots of comparisons
- There is a better way!

```
def collision_2rects_primitive(r1, r2):
    ''' Return True if the two rects overlap.
    '''
    if (r1.right <= r2.left or
        r2.right <= r1.left or
        r1.top   <= r2.bottom or
        r2.top   <= r1.bottom):
        return False
    return True
```



Separating Axis

- This code is actually an example of the separating axis theorem:
  - If there exists an axis by which the objects do not overlap, then the objects don't intersect
- Several collision detection tests depend on this theorem

Capsules

- People (or humanoids) are a common shape
- Humanoids are not rectangular and they aren't circular
  - We can model them as a combination of rectangle and circle -- well, semicircles

Capsule Test

- To test collisions with capsules:
  - Bounding box test on the rectangular area
  - Two circle tests on the ends
- A bit more expensive than either circle or bounding box. But, more likely to be accurate.

step 3: precise tests

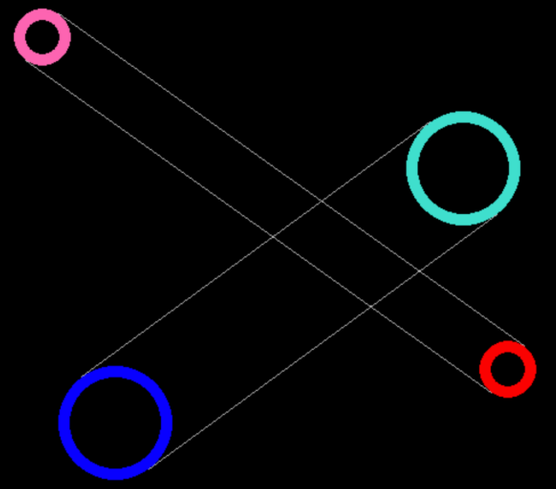
Strategy

1. **Reduce** the number of checks that need to be made
    - This can be for *operational* reasons
    - Or because sprites are *geographically separated*
2. **Quickly test** sprites (pairwise): discard those that can't possibly collide
    - A very quick, but not very precise test
3. **Thoroughly test** remaining sprites to detect collisions
    - An in-depth test that is very precise
4. **Resolve collisions:** Do whatever -- explosions, bounces, etc.

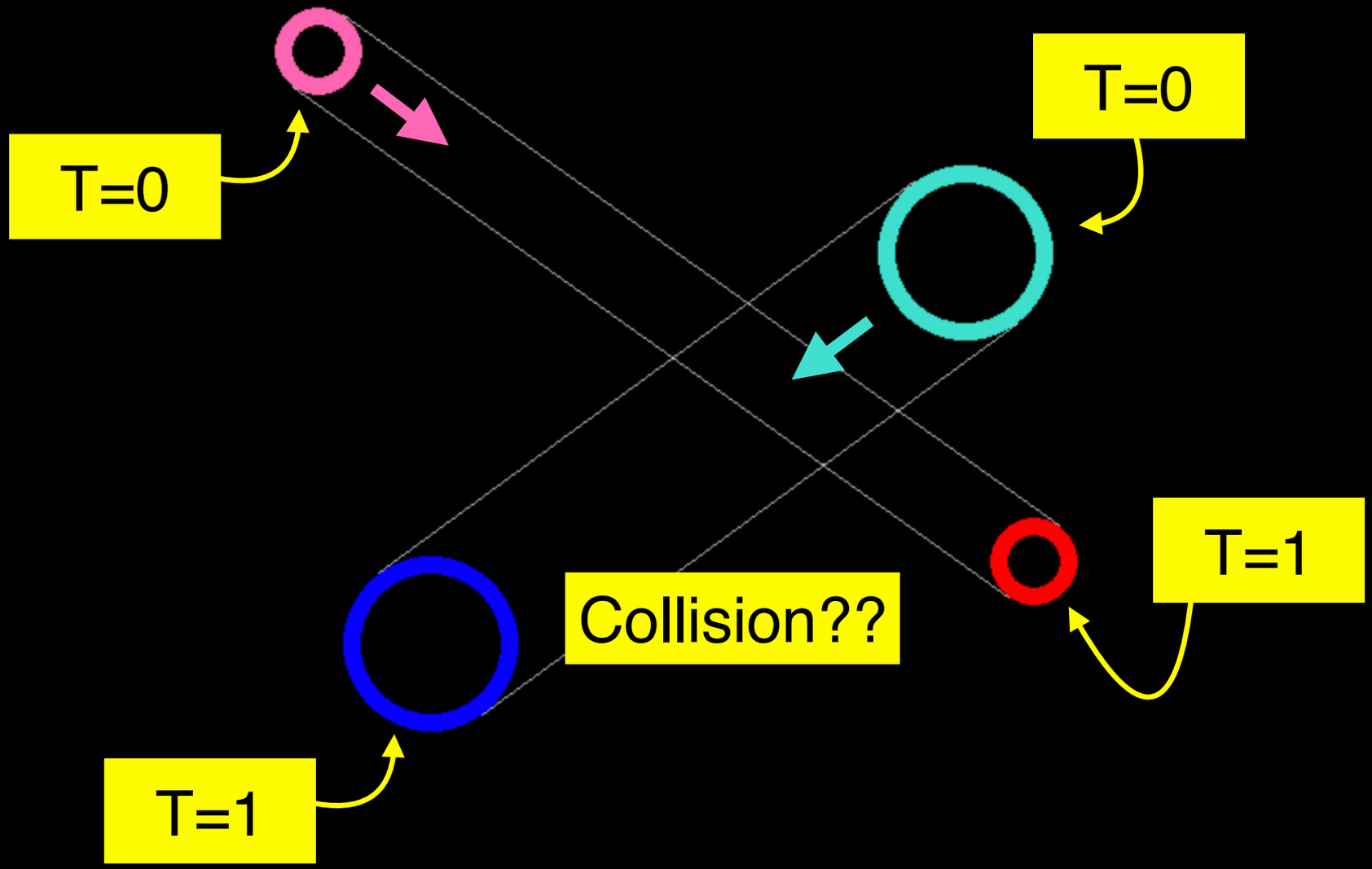
Swept Circle / Sphere

- Handout is from "Game Programming: Algorithms and Techniques" by Sanjay Madhav, ISBN 0-321-94015-6
  - This is an EXCELLENT book on game algorithms and I highly recommend it
  - The code is written in a pseudo-code that looks similar to Lua or C++, but it is very readable
- The book also has far fewer typos than our current textbook

Swept Sphere Algorithm



- Given two circles / spheres
  - So, you know their radius
- Given the location of the circle centers in this frame and the previous one
- Find: If they intersect
  - If so, where


T=0
T=0
T=1
Collision??
T=1

The Math

- Full derivation is in the handout
- Start with a parametric equation for each sphere position (such that t=0 is first frame position and t=1 is the second frame)

$$P(t) = P_0 + \vec{v}_p t$$

$$Q(t) = Q_0 + \vec{v}_q t$$

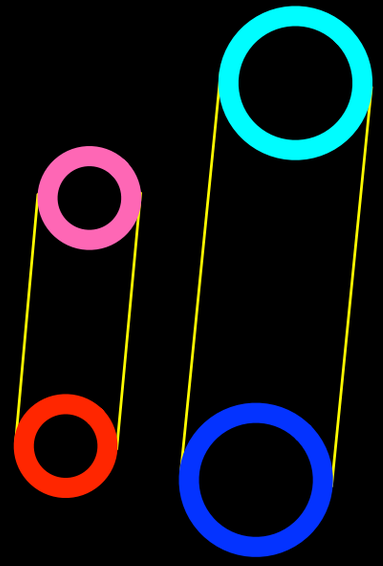
- Solve for when the distance between is equal to sum of the radius

$$||P(t) - Q(t)|| = r_p + r_q$$

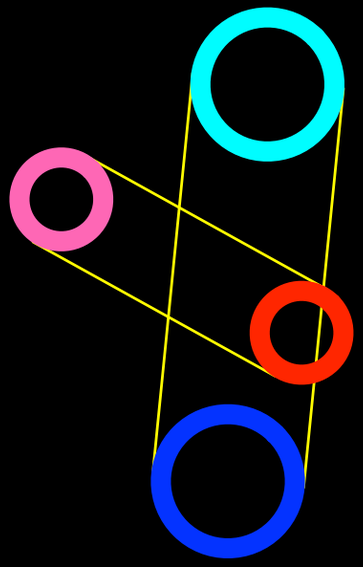
The Solution

$$(B \cdot B)t^2 + 2(A \cdot B)t + A \cdot A - (r_p + r_q)^2 = 0$$

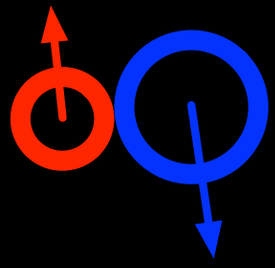
- This is a quadratic expression, with three classes of solutions



Discriminant < 0



> 0



= 0

Solve for t

- If discriminant > 0, then the value of t specifies the parametric time when the first touch occurs
  - If 0 < t < 1, then the collision happens between the frames you are worried about
- You can use the parametric equations to determine the position of the spheres at the collision point

Full Details

- If you are curious, full details are in the handout
- And, in the code for `SweptSphere.py`

response

Strategy

1. **Reduce** the number of checks that need to be made
    - This can be for *operational* reasons
    - Or because sprites are *geographically separated*
2. **Quickly test** sprites (pairwise): discard those that can't possibly collide
    - A very quick, but not very precise test
3. **Thoroughly test** remaining sprites to detect collisions
    - An in-depth test that is very precise
4. **Resolve collisions:** Do whatever -- explosions, bounces, etc.

Response to Collision

- Perhaps it is enough to know the collision happened
  - Common for fired shots
- If a bullet collides with an enemy spaceship
  - merely make the spaceship explode
- If a bullet collides with the player
  - perhaps health-bar is reduced

Bounces

- Negating the velocity works for horizontal and vertical walls
- But, not off general objects
  - Slow objects get stuck to each other
  - Real-world collisions don't work this way

Elastic Collisions

- Calculate the point of collision
- Determine a plane of collision
- Reflect the object's velocities off the plane

For Circles / Spheres

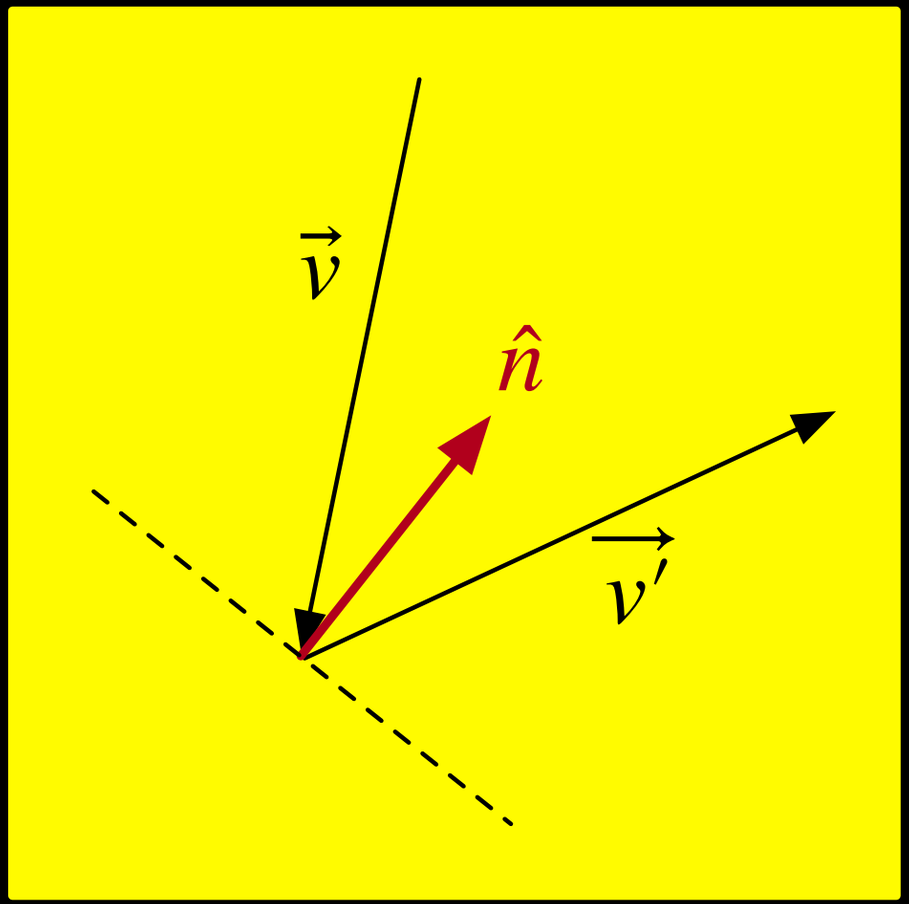
- Use swept-sphere algorithm to determine t
- Figure out each sphere's position at time t
  - At this time, the spheres must be touching
- The touch point is directly between the sphere centers
  - Linearly interpolated based on their radius

$$\frac{A\,.\,radius}{A\,.\,radius + B\,.\,radius}$$

- The vector between the circles is the normal to the plane they must bounce off
  - Normalize this vector
- Calculate the bounce vector with a touch of vector math

- Incoming object with velocity $\vec{v}$ bounces with velocity $\overrightarrow{v'}$ off a plane with normal $\hat{n}$ such that:

$$\overrightarrow{v'} = \vec{v} - 2\hat{n}(\vec{v}\cdot\hat{n})$$



Great Reference

- *Real-Time Collision Detection*, by Christer Ericson
- 632-page authority
- Unfortunately, out-of-print and expensive

What did you learn today?

- Collision Detection Problems
- Collision Detection Strategy
- Lots and lots of tests