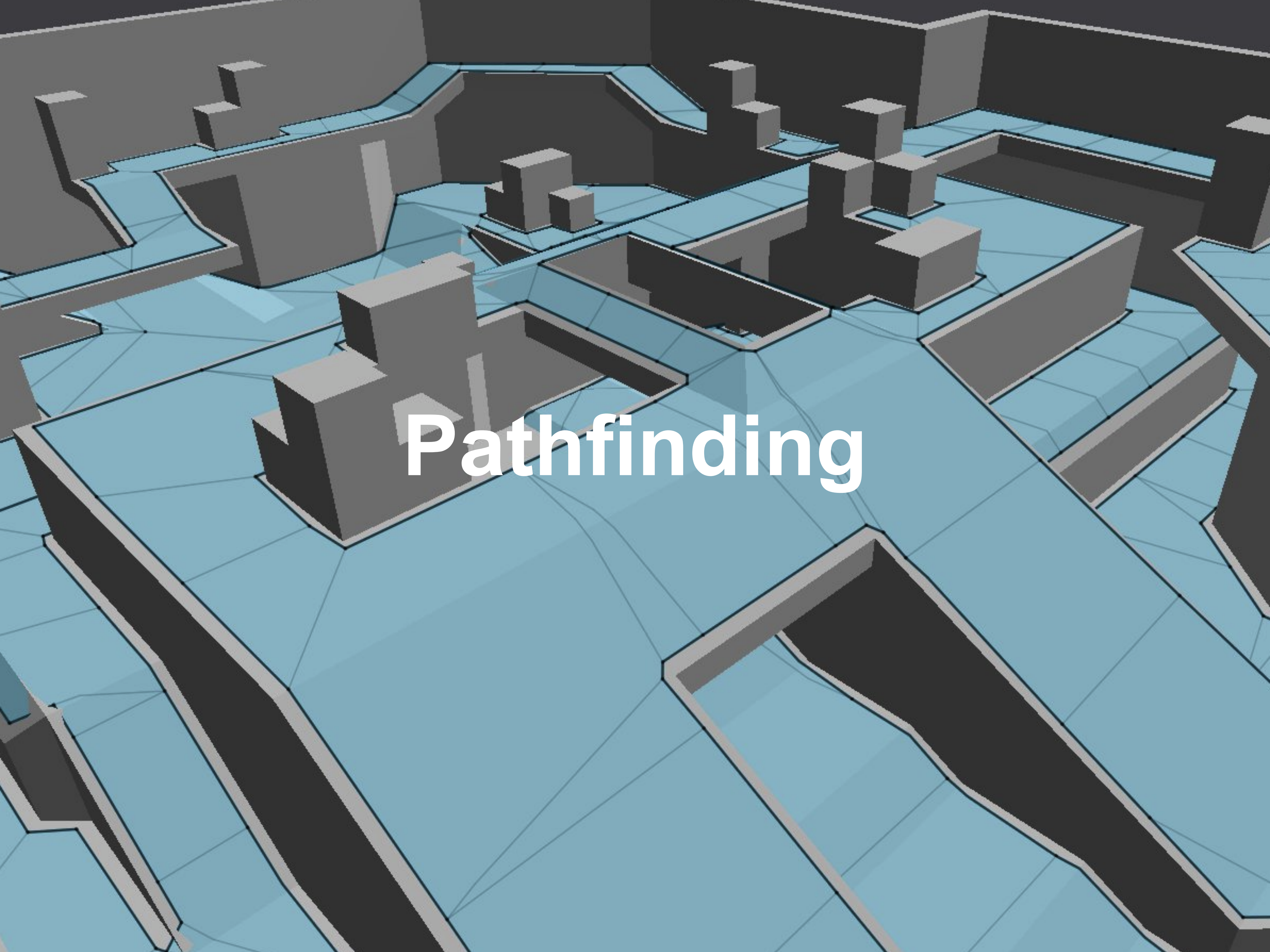
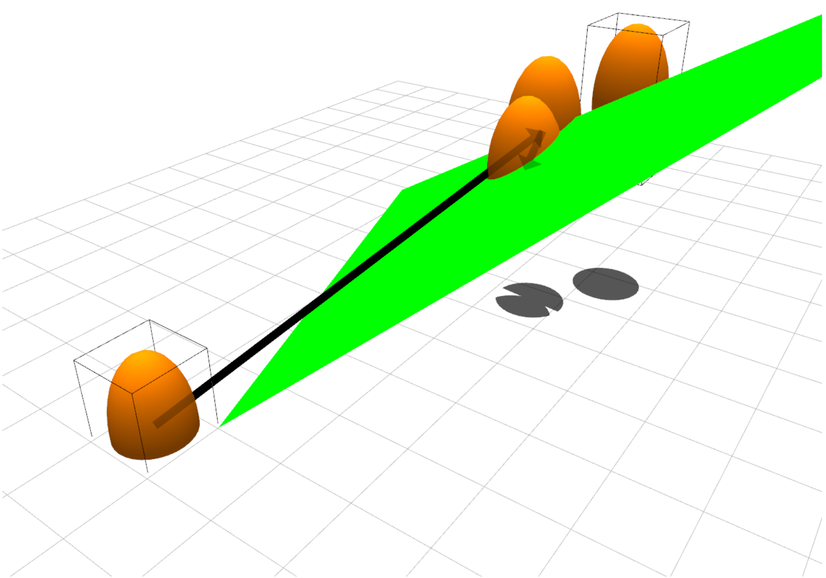
Pathfinding



Collision Detection Comments

- There were a couple common pitfalls
- Most common bug: short-circuiting on edge collisions
  - Doesn't work - interior is always first, but vertex collision could happen before edge collision



Collision Detection Comments

- Most common inefficiency: Using the greater t-value from quadratic equations
  - These are in the interior of the triangle, so there's no reason to use them
  - Just use (-b - sqrt(b^2 - 4ac)) / (2a) since a will always be positive, ignore larger term

Pathfinding

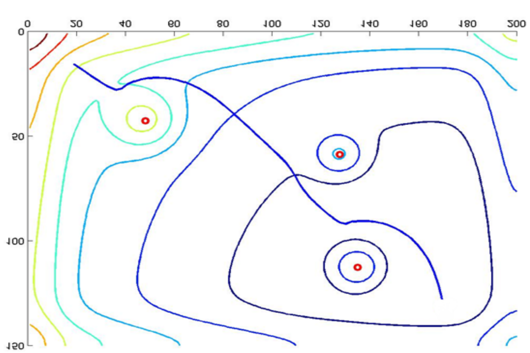
- Pathfinding is the most common primitive used in game AI
  - A path is a list of instructions for getting from one location to another
  - Not just locations (jump, climb ladder)
- A hard problem!
  - Bad path planning breaks the immersive experience
  - Many games get it wrong

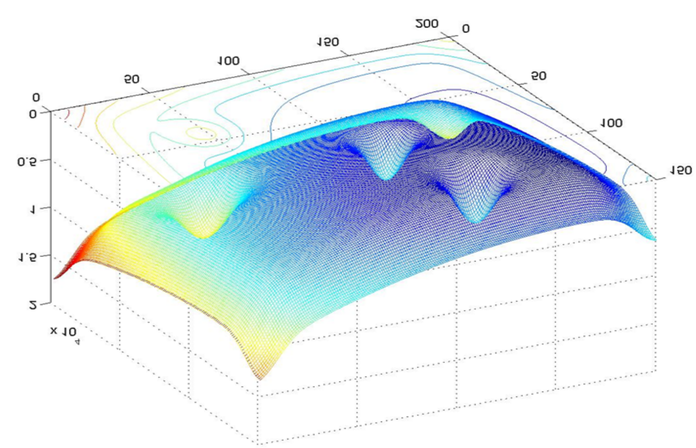
3D World Representation

- Need an efficient encoding of relevant information in the world
  - Navigable space
  - Important locations (health/safety/bases)
- Field-based approaches
  - Potential fields
- Graph-based approaches
  - Waypoints
  - Navigation meshes

Pathfinding with Potential Fields

- The potential at a location represents how desirable it is for the entity to be there
  - Obstacles have low potential
  - Desirable places have high potential
- Potential field: a region of potential values
  - Usually a 2D grid in games
  - Good paths can be found by hill climbing



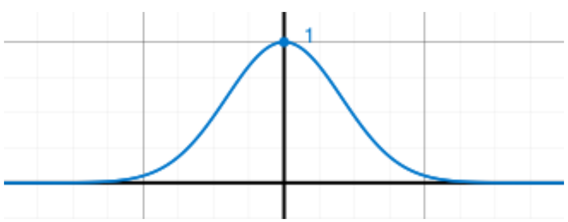


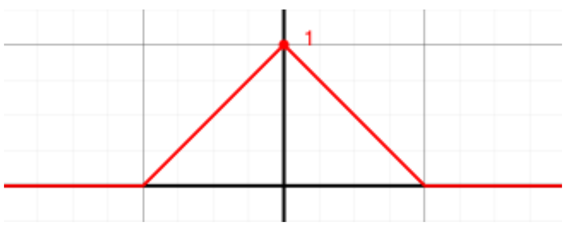
Pathfinding with Potential Fields

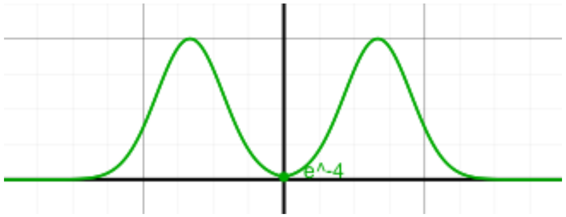
- Algorithm details
  - On startup
    - Generate potential fields for static objects
  - Periodically (~5 times a second)
    - Generate potential fields for dynamic objects
    - Sum static and dynamic potential fields to get the final potential field
  - Each update
    - Pathfinding entities move towards direction of greatest potential increase (hill climbing)

Pathfinding with Potential Fields

- Potential function for a single entity
  - Usually defined radially
  - Non-radially symmetric ones useful too
    - Example: cone of negative values ahead of player to encourage enemies to stay out of view
- Example potential functions (radial):

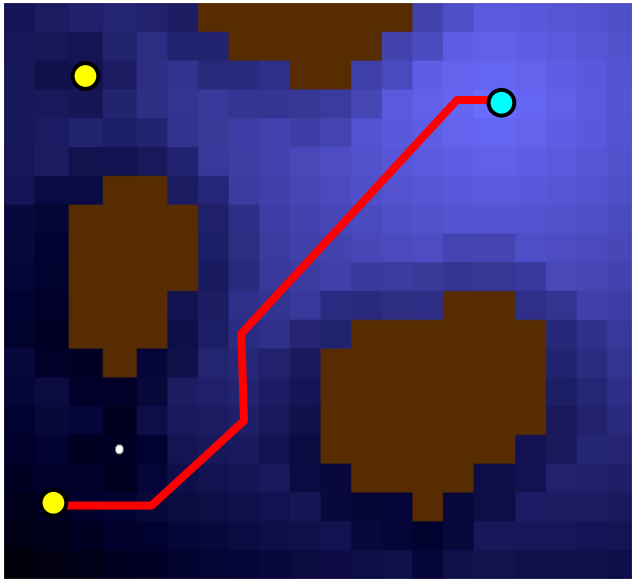




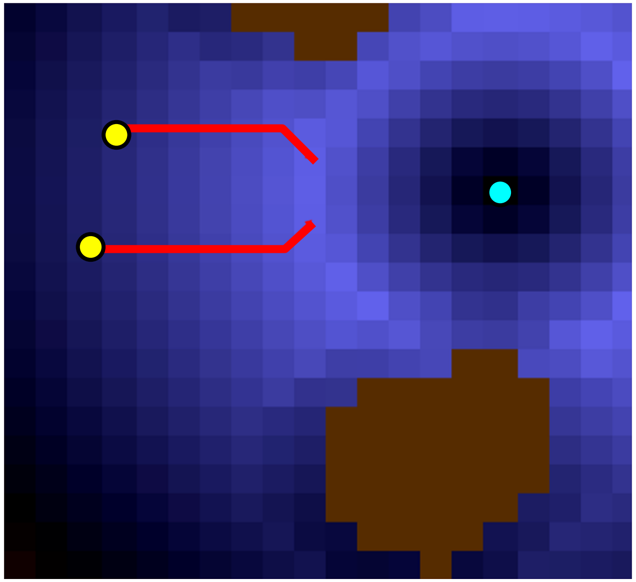


Pathfinding with Potential Fields

- Potential functions don't need linear falloff



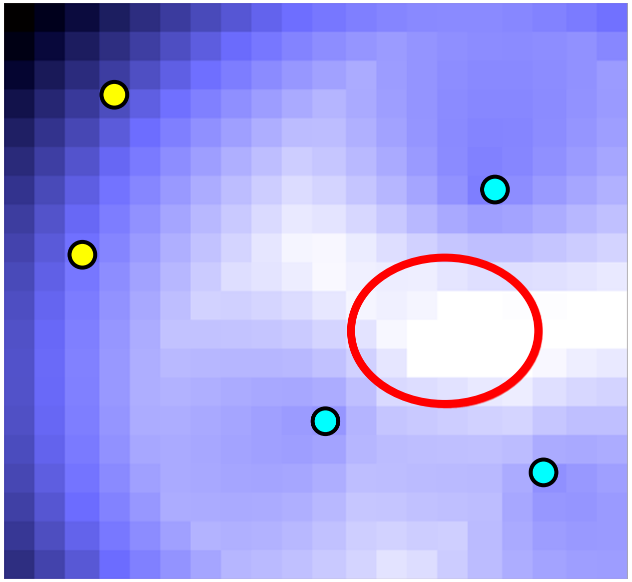
Linear falloff leads to a target



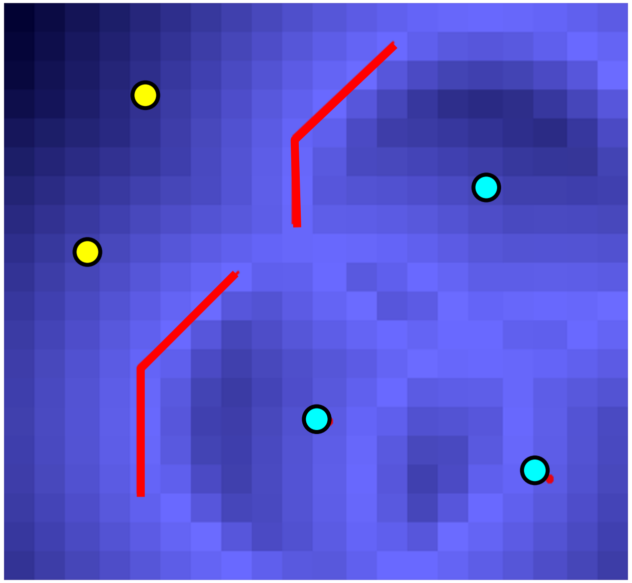
Rise then fall leads ranged units to a safe distance away

Pathfinding with Potential Fields

- Multiple ways of combining potentials
  - Maximum sometimes works better than sum



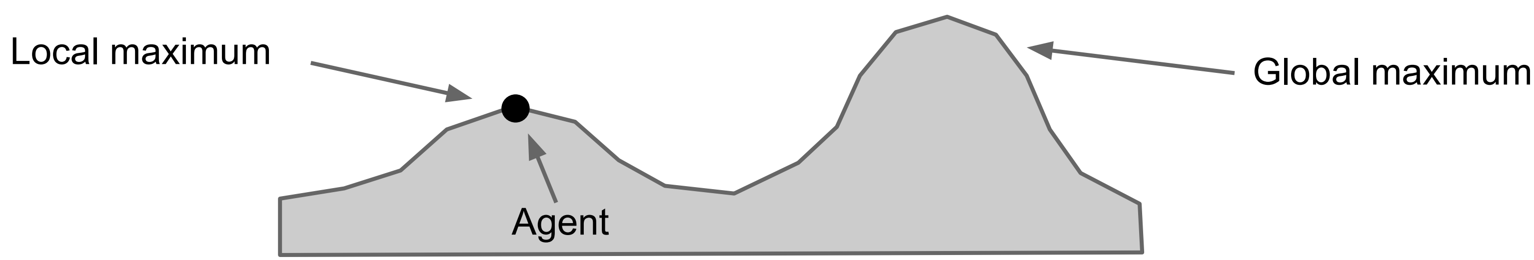
Summing creates false desirable spot for ranged units



Maximum correctly identifies desirable areas for ranged units

Pathfinding with Potential Fields

- Advantages
  - Able to represent fully dynamic world
  - Hill climbing doesn't need to generate and store the entire path
  - Naturally handles moving obstacles (crowds)
  - Can be efficiently implemented on the GPU
- Drawbacks
  - Tuning parameters can be tricky
  - Hill climbing can get stuck in local maxima



Avoiding Local Maxima

- Agents drop negative-potential "pheromone trail"
  - Bulge behind them pushes them forward
  - Doesn't prevent agents from turning around in corners
- Still doesn't avoid all local maxima
  - Potential fields are better suited for dynamic worlds with large open areas
  - Classical graph-based pathfinding works better for complex terrain with lots of concave areas

Reconsidering Potential Fields

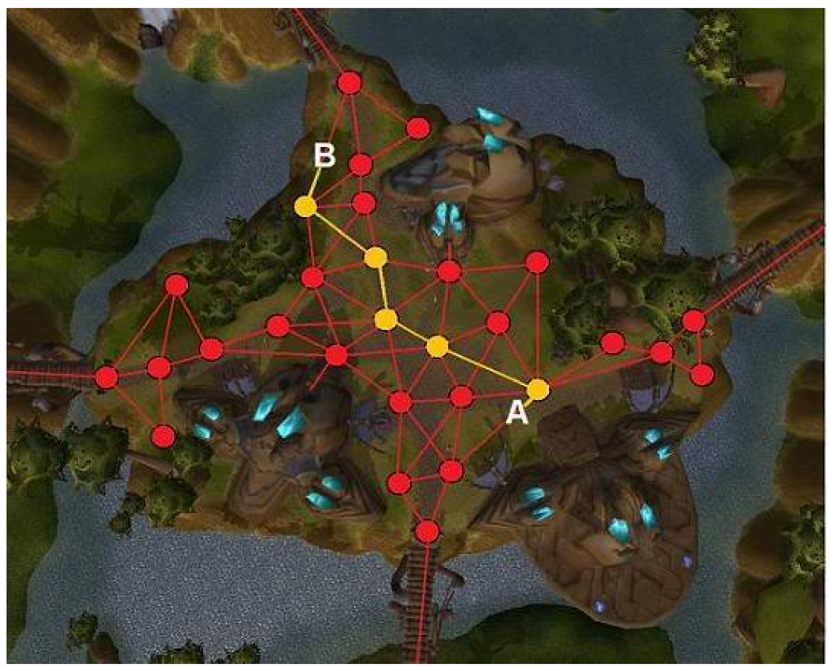
- Not actually used in many real games
  - We couldn't find any commercial releases that use them
  - But there are at least custom Starcraft bots that do
- Instead, most games use graph-based path planning

Graph-Based Path Planning

- World represented as a graph
  - Nodes represent open space
  - Edges represent ways to travel between nodes
  - Graph search algorithms used to find paths
- Two common types
  - Waypoint graphs
  - Navigation meshes

Waypoint Graphs

- Represent a chosen set of paths through the world as a web of line segments
  - Nodes are waypoints, edges connect waypoints



Disadvantages of Waypoint Graphs

- No model of space in between waypoints
  - No way of going around dynamic objects without recomputing the graph
- Optimal path is likely not in the graph
  - Paths will zig-zag to destination
  - Good paths require huge numbers of waypoints and/or connections
- Awkward to handle entities with different radii
  - Have to turn off certain edges and add more waypoints

Navigation Meshes

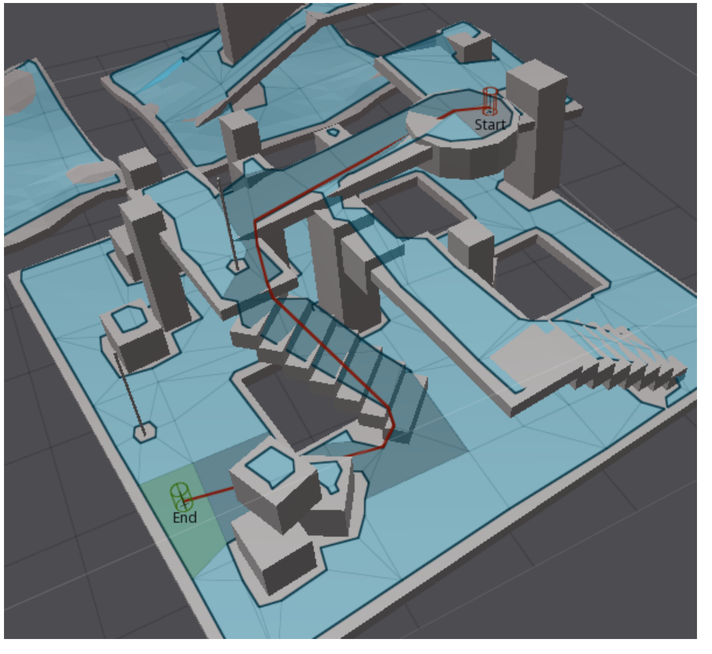
- Convex polygons as navigable space
  - Nodes are polygons, edges are shared edges between polygons

Advantages of Navigation Meshes

- Models entire navigable space
  - Can plan path from anywhere inside nav mesh
  - Paths can be planned around dynamic obstacles
  - Zig-zagging can be avoided
- More efficient and compact representation
  - Equivalent waypoint graph would have many more nodes and would take longer to traverse
- Naturally handles entities of different radii
  - Don't go through edges less than 2 * radius long
  - Leave at least a distance of radius when moving around nav mesh vertices

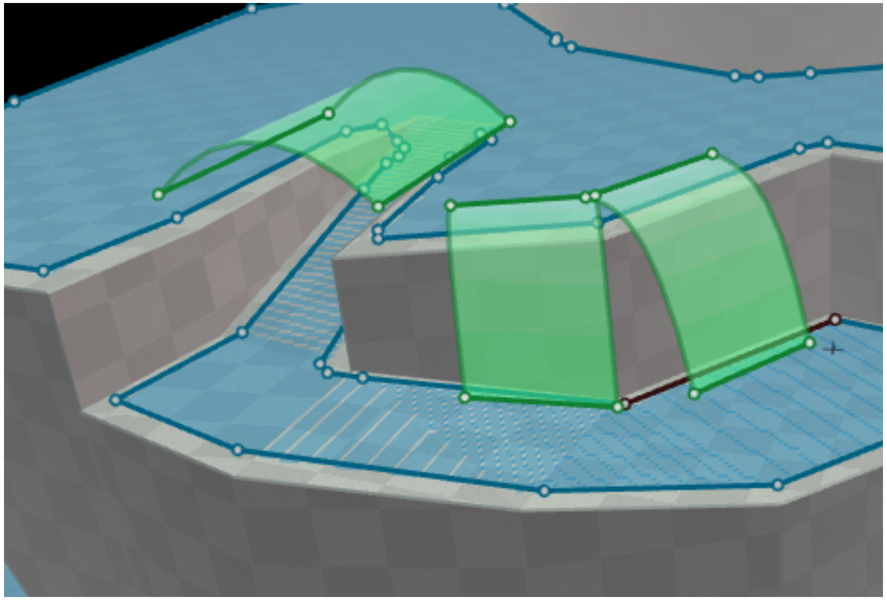
Navigation Meshes

- Different from collision mesh
  - Only contains walkable faces
  - Stairs become a single, rectangular polygon
  - Polygons usually shrunk to account for player radius



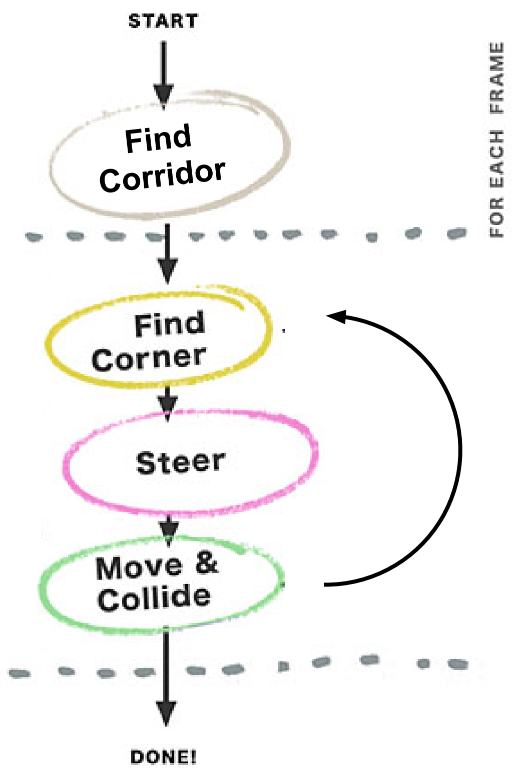
Navigation Meshes

- Annotate special regions
  - Regions for jumping across, falling down, crouching behind, climbing up, ...
  - Regions usually computed automatically



Navigation Loop

- Process for robust path navigation on a navigation mesh

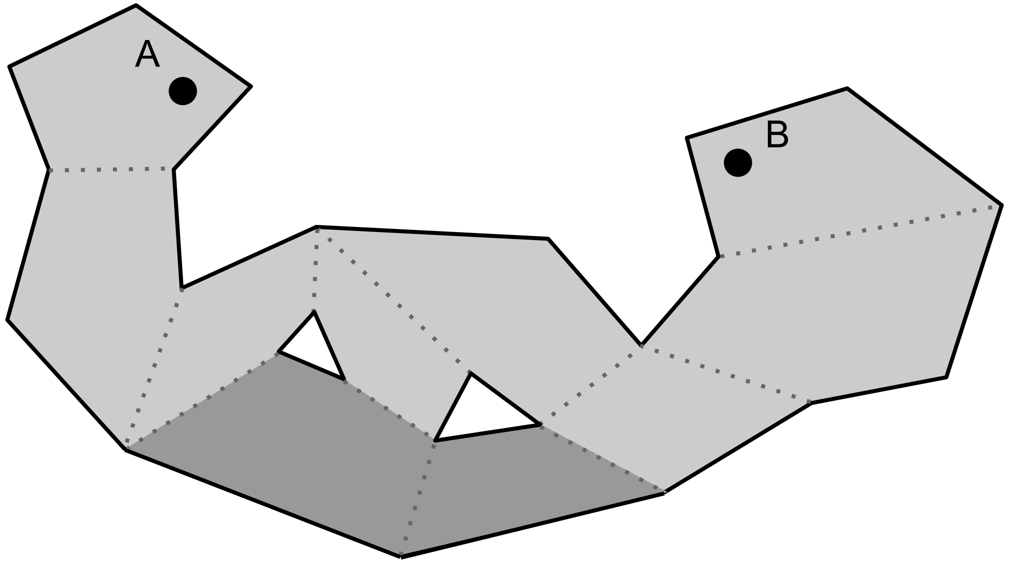


Graph Search

- First step in finding a path
- Graph search problem statement
  - Given starting point A, target point B and a nav mesh
  - Generate a list of nav mesh nodes from A to B (called a *corridor*)
- Simplest approach: Breadth-first search
  - Keep searching until target point is reached
  - Each edge has equal weight
- Most common approach: A-star
  - Variable edge weights
    - e.g. steep surfaces may have higher cost
  - Uses a heuristic to arrive at an answer faster

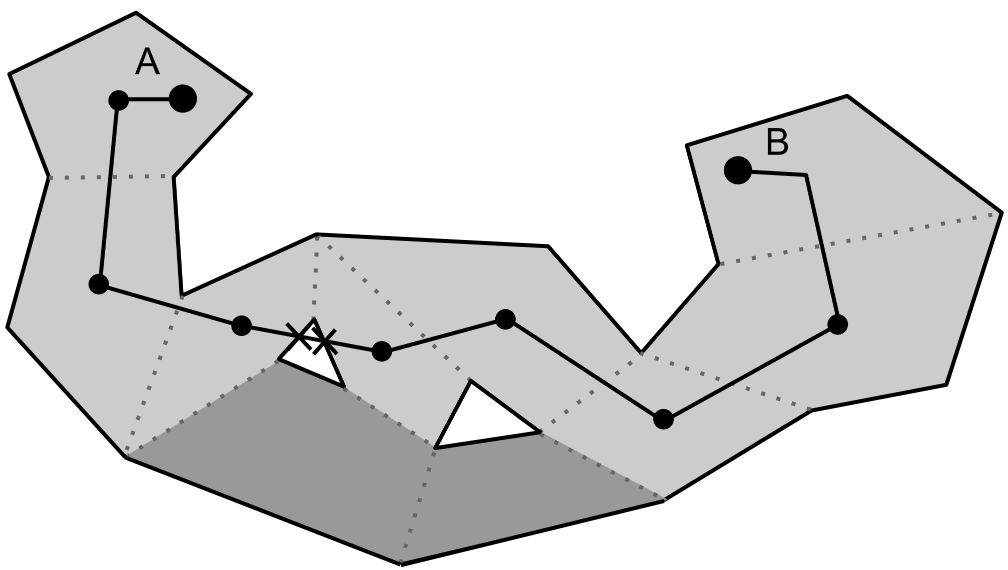
Path Generation

- Path generation problem statement
  - Given a list of polygons from a graph search
  - Construct the shortest path for the agent
- Path will be inside light colored polygons



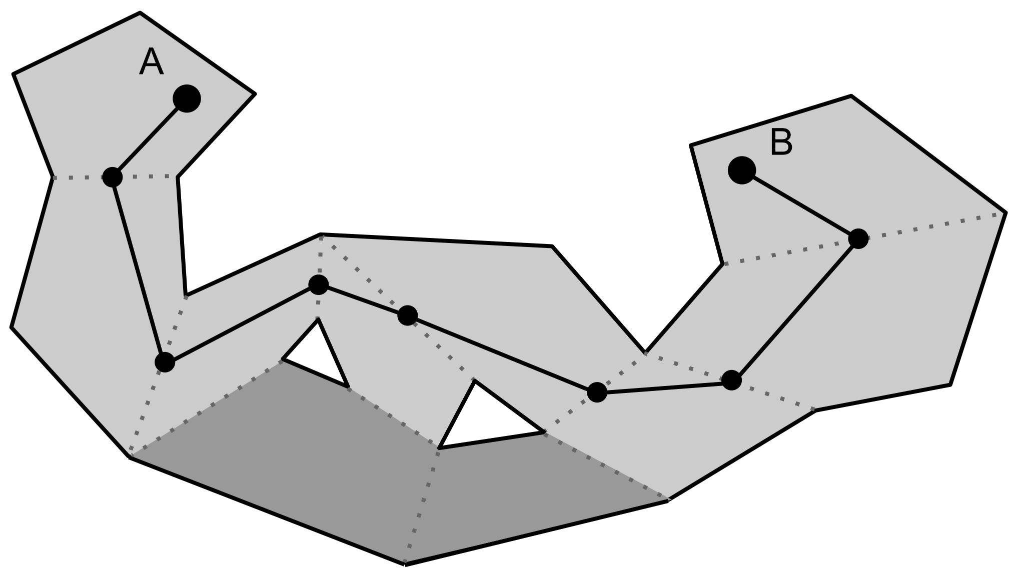
Path Generation: First Attempt

- Method: Connect polygon centers
  - Centers of adjacent polygons don't always work
  - Only guaranteed paths are from any point in a polygon to any other point within that polygon



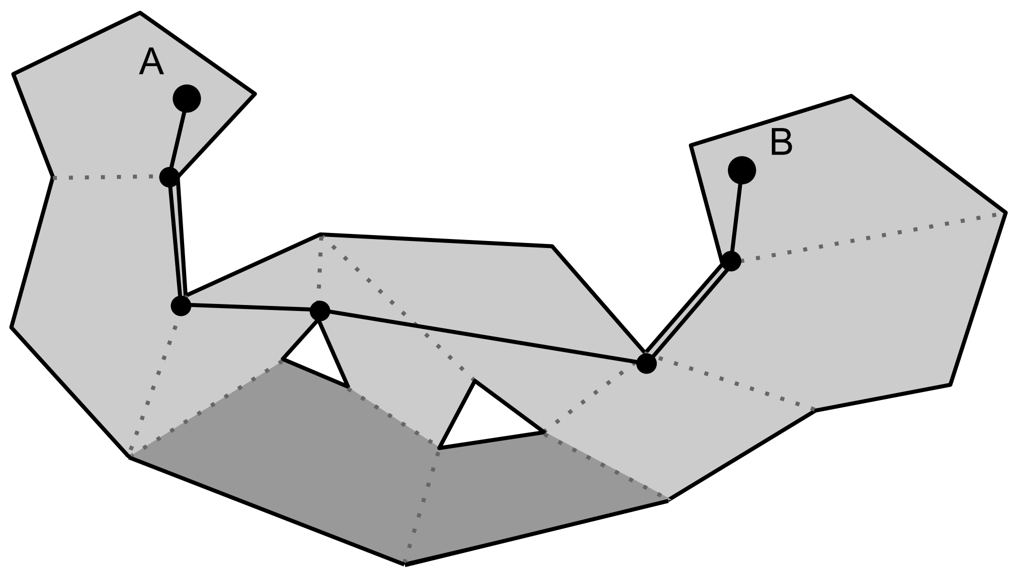
Path Generation: Second Attempt

- Method: Connect edge centers
  - Actually just a waypoint graph!
  - Will still lead to undesirable zig-zagging



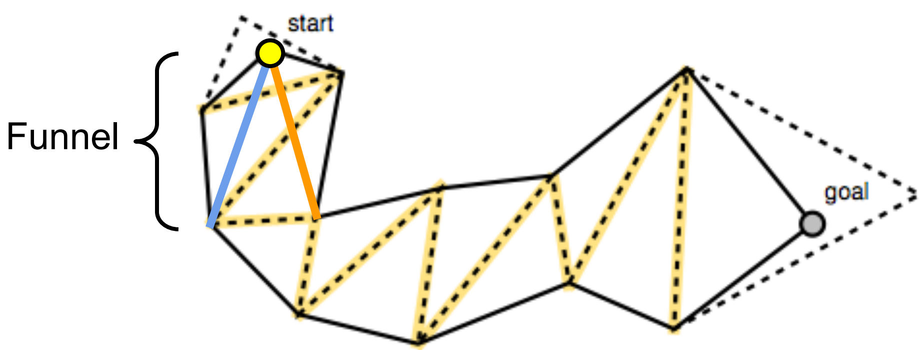
Path Generation: Third Attempt

- Method: Funnel algorithm
  - Computes shortest distance around corners

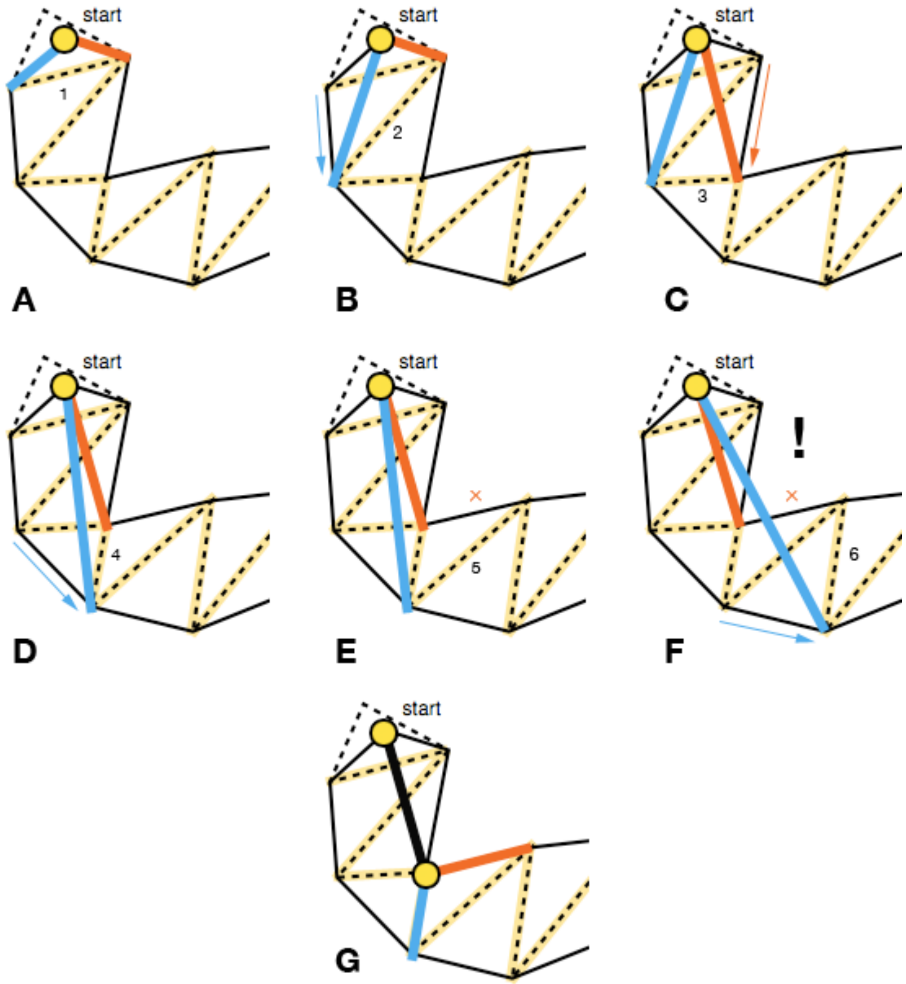


Funnel Algorithm

- Finds the shortest path
  - Traverses through a list of polygons connected by shared edges (portals)
  - Keeps track of the leftmost and rightmost sides of the "funnel" along the way
  - Alternates updating the left and right sides, making the funnel narrower and narrower
  - Add a new point to the path when they cross



Funnel Algorithm

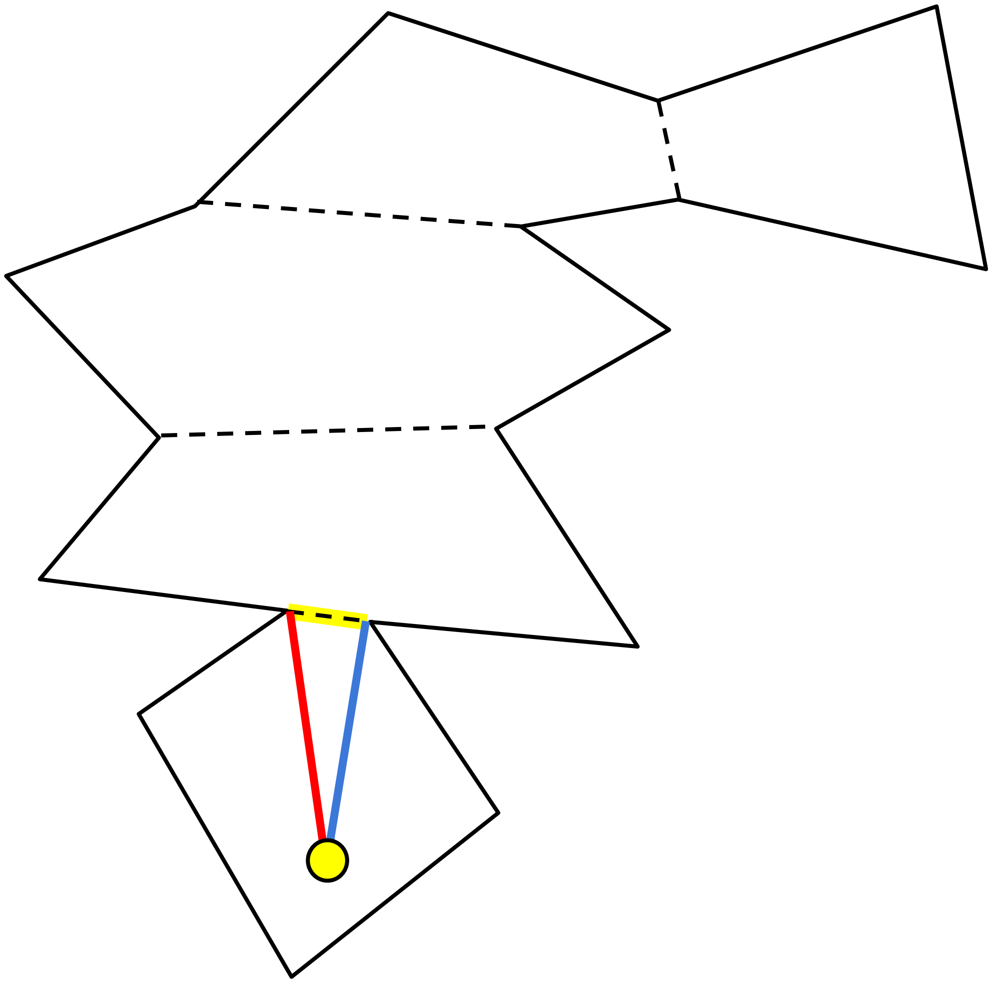


Funnel Algorithm

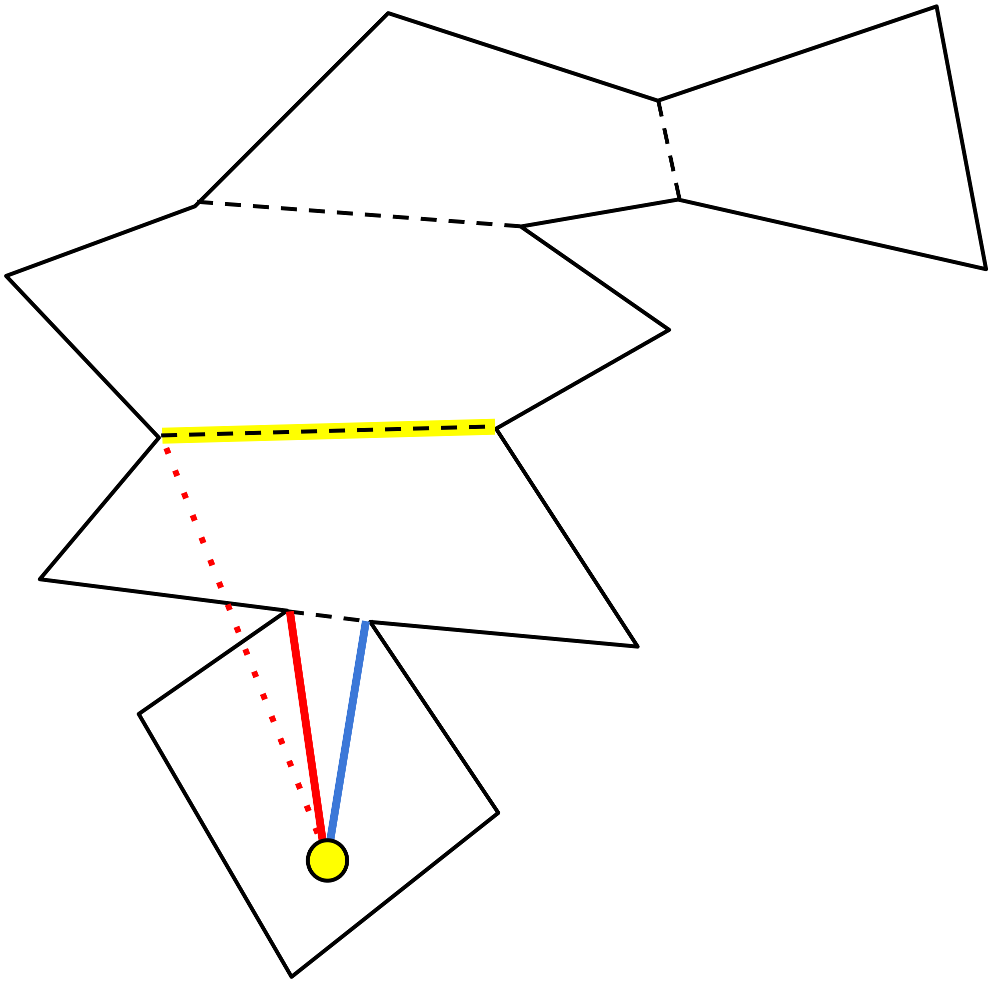
- Start
  - Apex point = start of path
  - Left and right points = left and right vertices of first portal
- Step
  - Advance to the next portal
  - Try to move left point to left vertex of next portal
    - If inside the funnel, narrow the funnel (C-D in picture)
    - If past the right side of the funnel, turn a corner (E-G in picture)
      - Add right point to path
      - Set apex point to right point
      - Restart at portal where right point came from
  - Try to move right point to right vertex of next portal
    - Similar to left point

Details at http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html

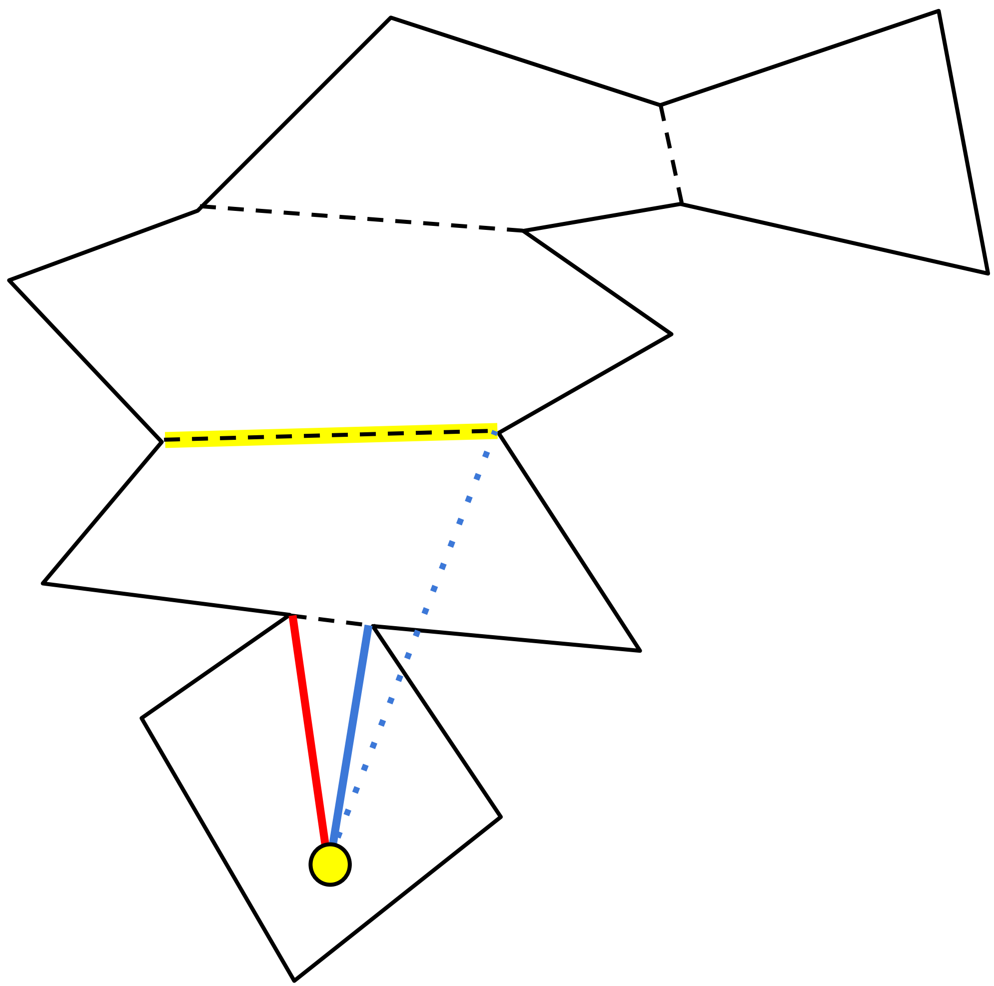
Specific Funnel Algorithm Case



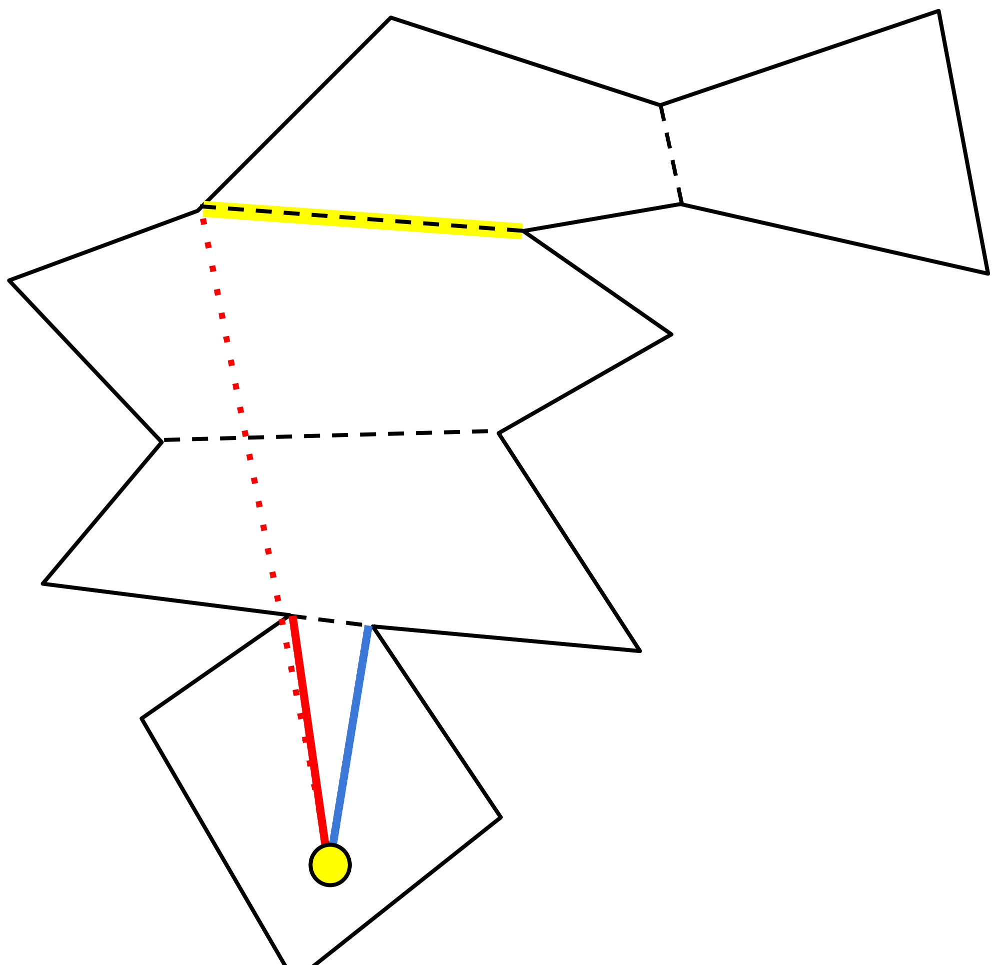
Specific Funnel Algorithm Case



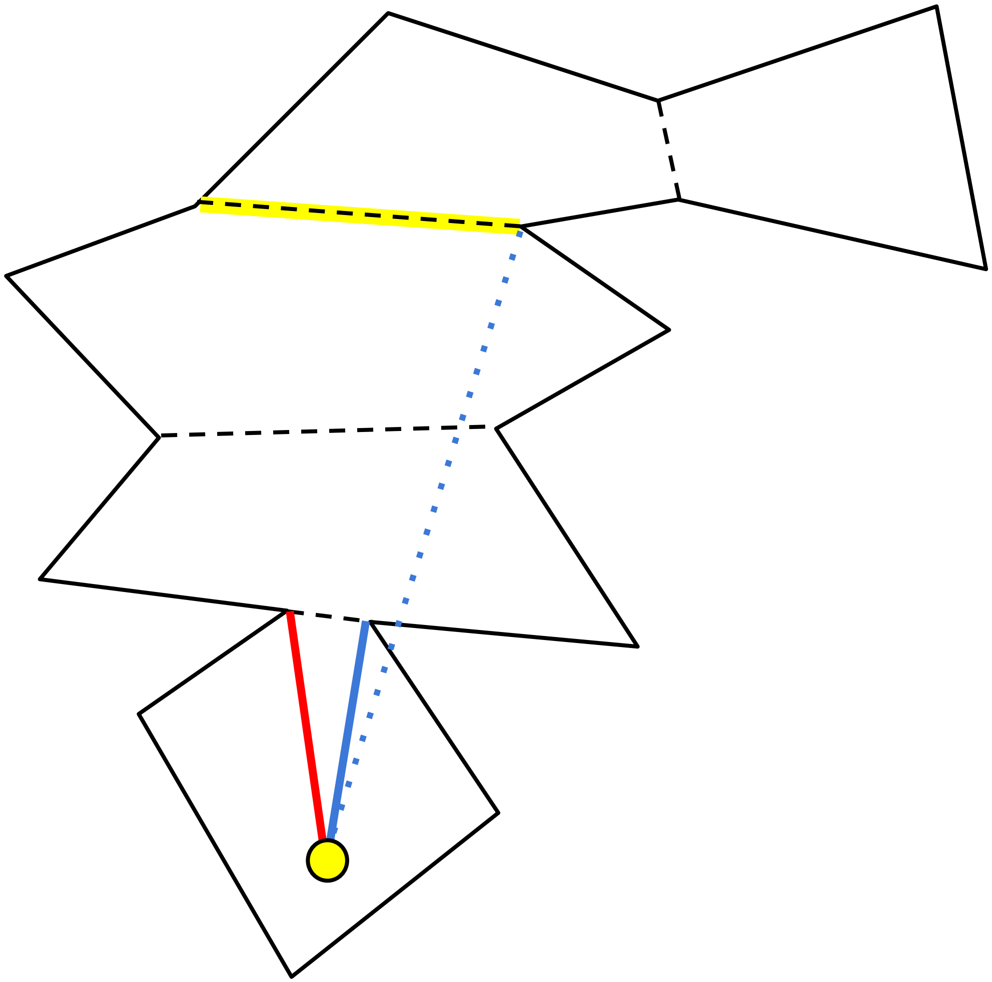
Specific Funnel Algorithm Case



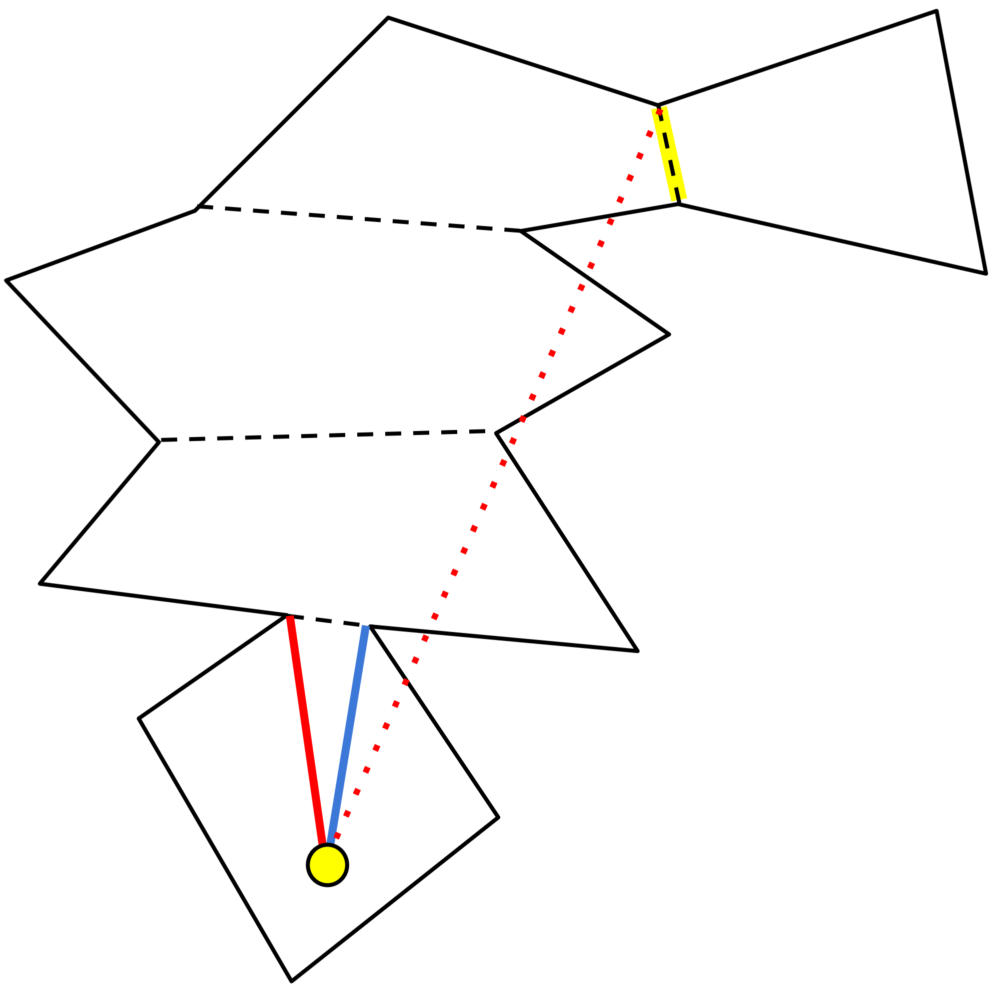
Specific Funnel Algorithm Case



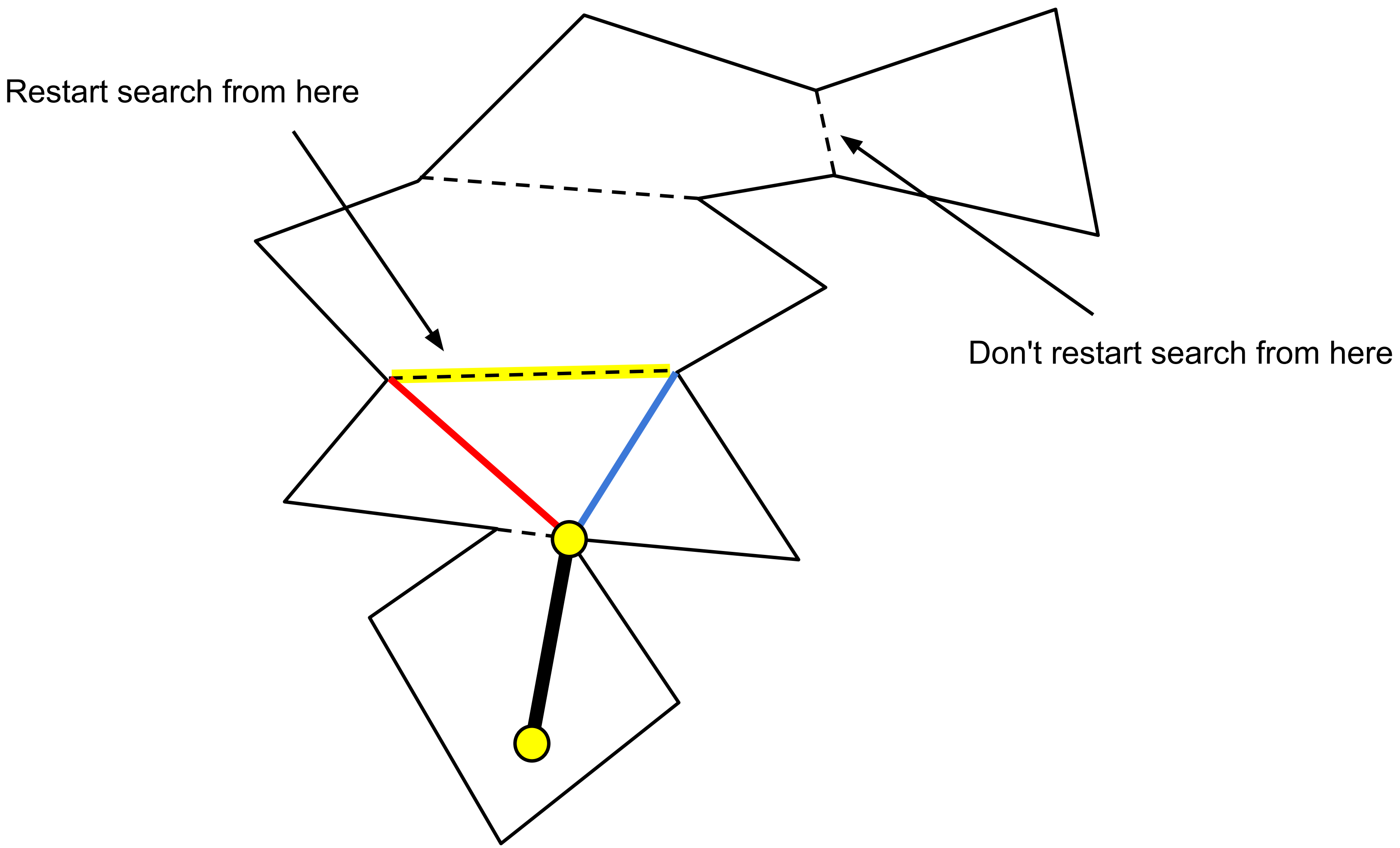
Specific Funnel Algorithm Case



Specific Funnel Algorithm Case



Specific Funnel Algorithm Case



Steering

- There are many different ways for an entity to move towards a point
- Moving in straight lines towards each destination gives robotic look
- Many alternatives exist: use depends on the desired behavior
  - Seek, arrive, wander, pursue, etc.
- Steering behaviors may be influenced by a group
  - Queue, flock, etc.

Steering Example: Arrival

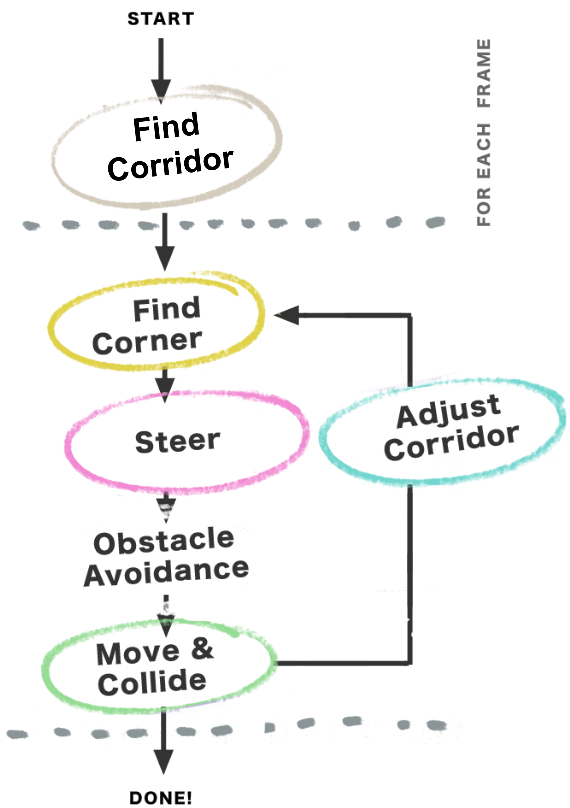
- When approaching the end of a path, we may want to naturally slow to a halt
- *Arrival* applies a deceleration force as the entity approaches its destination

Moving and Colliding

- If no collisions, simple as using the destination and steering to move
- Collisions can cause a variety of issues
  - May need to re-plan path if collision is impeding movement
  - Can detect getting stuck if the entity stays in roughly the same spot for a few seconds

But Wait, There's More!

- Shouldn't walk directly towards obstacles



Obstacle Avoidance

- Static obstacles can be avoided by generating the right navigation mesh
- Dynamic obstacles are trickier
- Baseline approach for dynamic obstacles
    - Use raycast or sweep test to determine if in obstacle is in the way
    - Apply steering force away from obstacle
    - Adjust force based on distance to obstacle

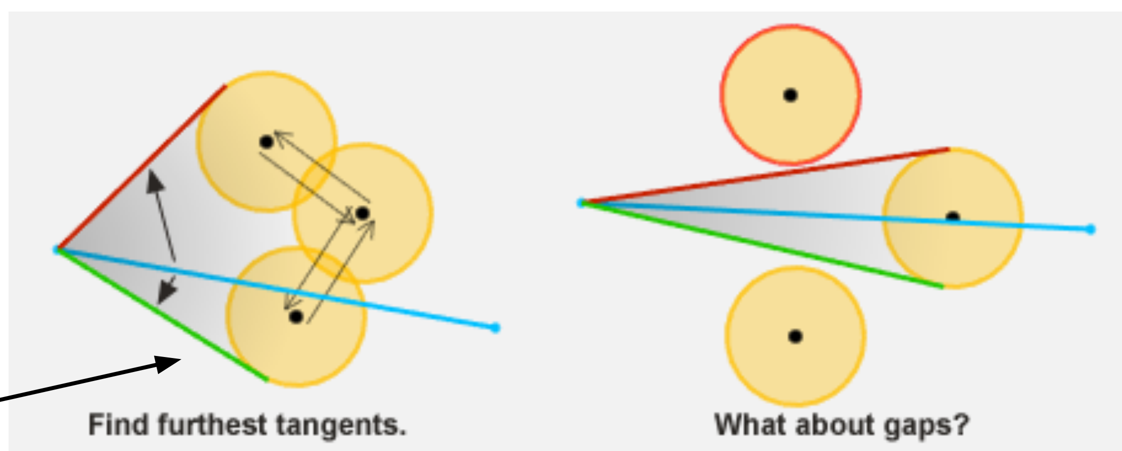
Dynamic Obstacle Avoidance

- If we consider each obstacle individually, this is purely *local* avoidance
  - Can easily get stuck in local minima
    - U shape formed by 3 crates
  - Remember, this step is added on top of *global* pathplanning
- Need approach between purely local and global for handling temporary obstacles
  - Will not perfectly handle all cases
  - Only perfect solution is to adjust navigation mesh
  - Example approach: "Very Temporary Obstacle Avoidance" by Mikko Mononen

Very Temporary Obstacle Avoidance

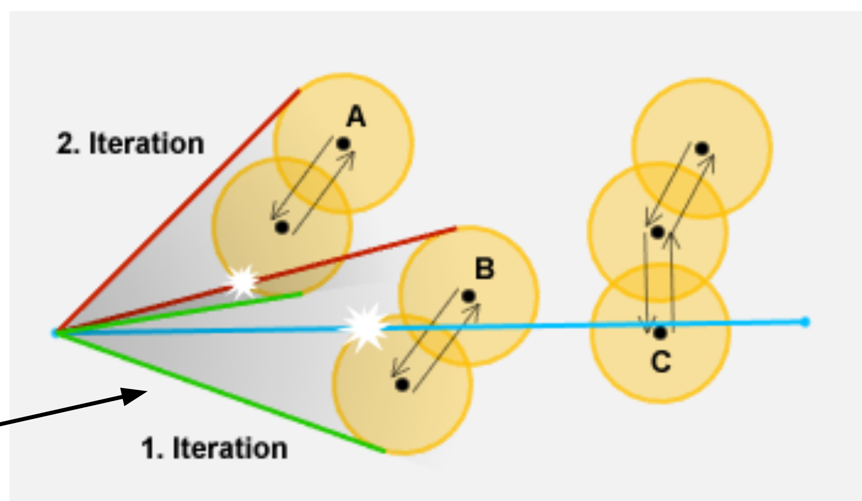
- For the obstacle blocking the path
  - Calculate tangent points
  - Choose tangent that generates a shorter path from the start position to the goal through the tangent
- Cluster overlapping objects into one object

Shortest path



Very Temporary Obstacle Avoidance

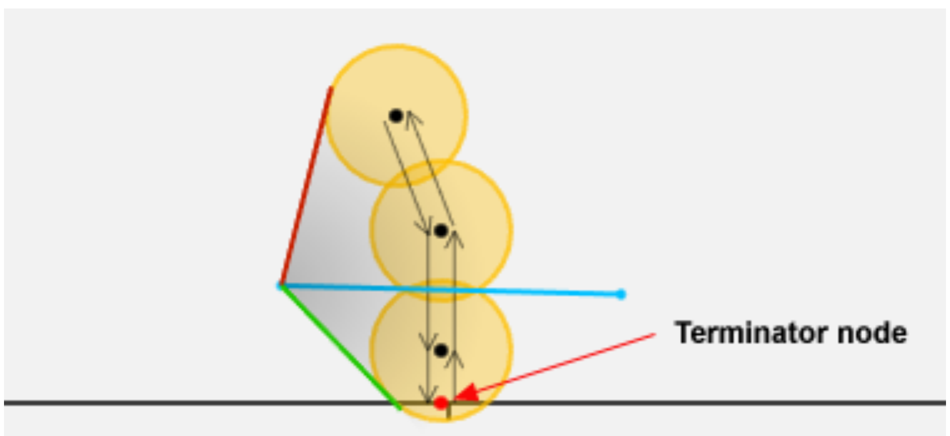
- Handling multiple obstacles
  - Check for obstacles on newly chosen path
  - Iterate until path is clear
    - Might take many iterations to converge
    - Only run 2-4 iterations, usually good enough



Shortest path

Very Temporary Obstacle Avoidance

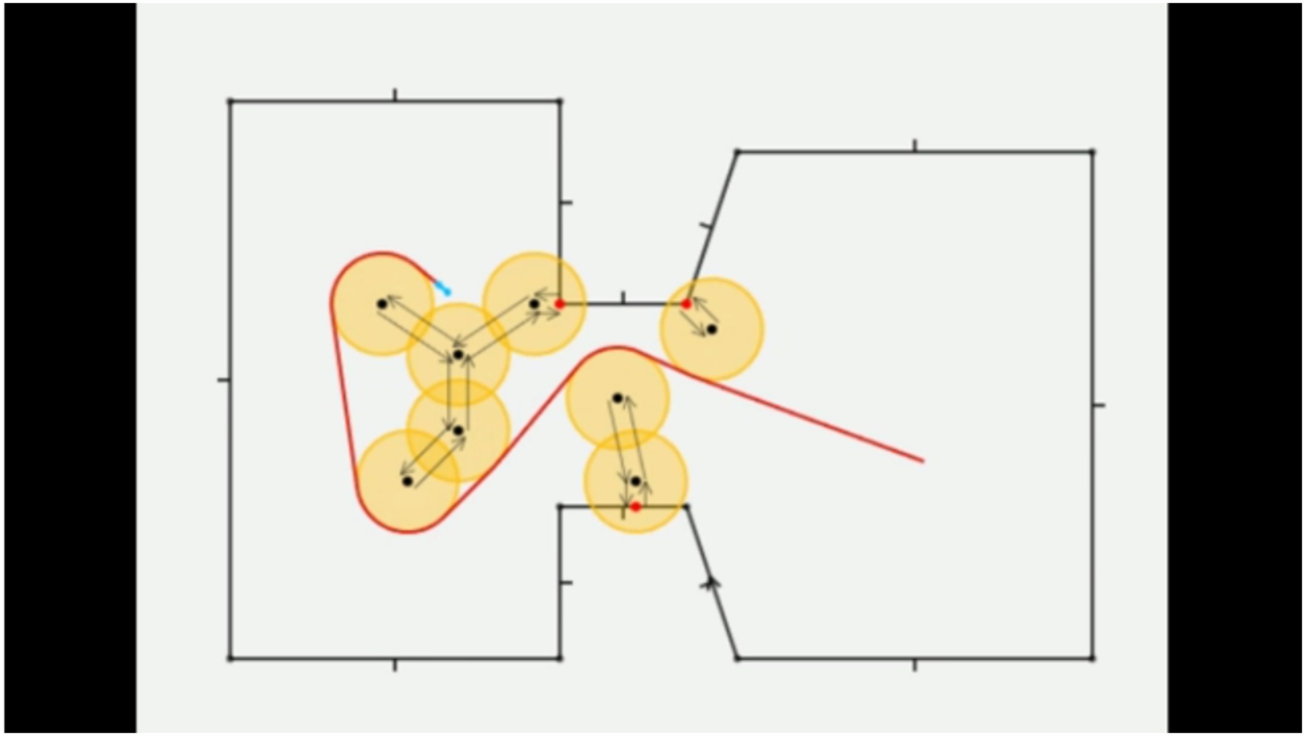
- Handling objects along walls
  - Check for intersections along navigation mesh boundary
  - If one is hit, exclude that path



Avoiding the obstacle cluster from below would be faster but leads to collision with navigation mesh boundary.

Very Temporary Obstacle Avoidance

- Demonstration video



http://vimeo.com/19858753

Robustness

- Can't find path from off the navigation mesh
  - Clamp agents inside boundary of navigation mesh
  - Special-case climbing up ledges
- Crowds can't all follow the same path
  - Don't precompute the path, assume it's wrong
  - Use a more loose structure of path (polygons)
  - Just navigate to the next corner
  - Use local object avoidance to handle crowds

Case Study: Recast and Detour

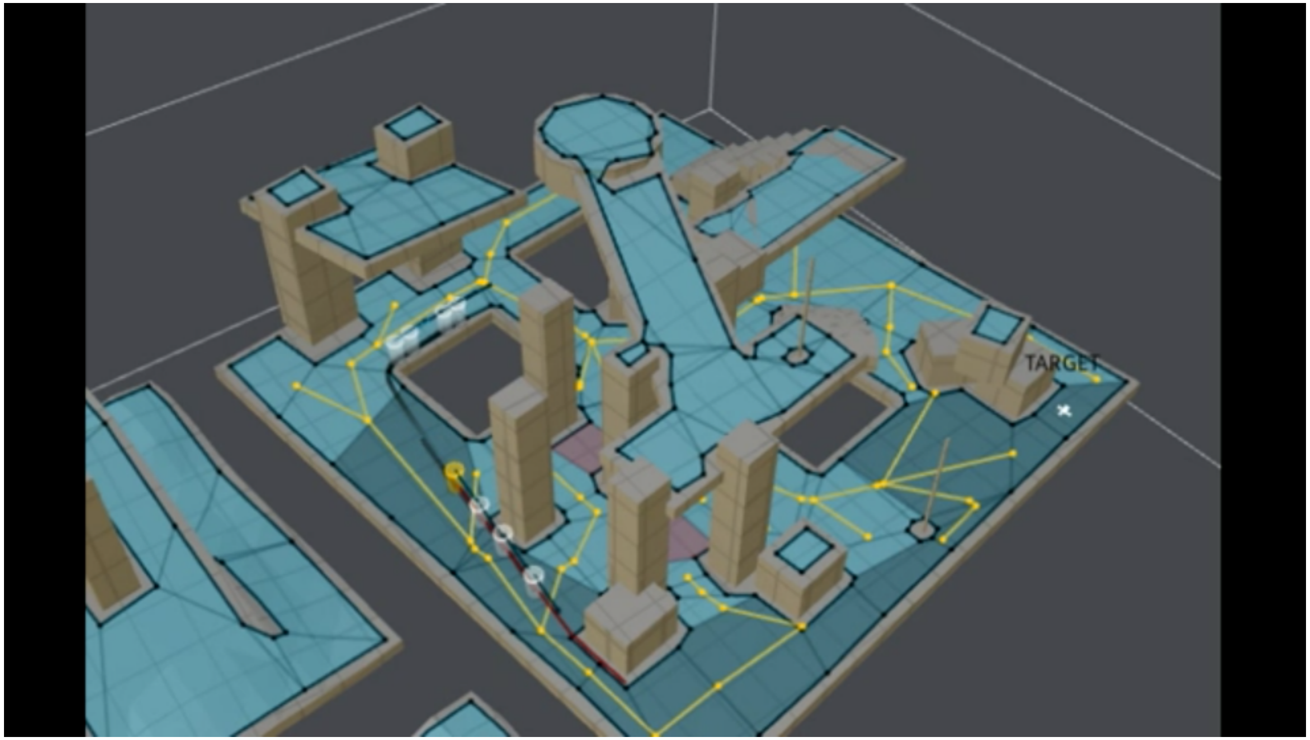
- Open source middleware
  - Recast: navigation mesh construction
  - Detour: movement over a navigation mesh
  - Developed by Mikko Mononen (lead AI on Crysis)
- Widely used in AAA games
  - Killzone 3
  - Bulletstorm
  - Halo Reach

Case Study: Recast and Detour

- Recast: navigation mesh generation
- Start with arbitrary mesh
  - Divide world into tiles
  - Voxelize a tile at a time
  - Extract layered heightfield from voxels
- Extract walkable polygons from heightfield
  - Must have minimum clearance
  - Merge small bumps in terrain and steps on stairs together
  - Shrink polygons away from edge to account for radius of agent

Case Study: Recast and Detour

- Detour: navigation mesh pathfinding



http://vimeo.com/27143809

References

- Recast and Detour
  - http://code.google.com/p/recastnavigation/
- Funnel algorithm
  - http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html
- Obstacle avoidance
  - http://digestingduck.blogspot.com/2011/02/very-temporary-obstacle-avoidance.html
- Potential fields
  - http://aigamedev.com/open/tutorials/potential-fields/

Platformer: Week 3

- Load a pre-made navigation mesh
  - Generate graph from triangle adjacency
- Find a set of nodes using breadth-first search
  - Or A-star, though this is optional
- Generate a path using the funnel algorithm
  - Pretend polygons are 2D in the xz-plane
  - Collision response will handle the y coordinate
- Local obstacle avoidance is not required

Platformer: Week 3

- Hand in
  - Navigation mesh is visualized
  - Path is visualized from player to a target position
  - Target position can be set to the player's current position by pressing a key
- In week 4, you will create at least one enemy that uses pathfinding
  - So starting thinking about that, too...

C++ Tip of the Week

- "Most vexing parse"
  - C++ has an ambiguity in its declaration syntax

```
std::string foo(std::string());
```

- Is this:
  - A variable of type std::string initialized to std::string()?
  - The forward declaration of a function that returns a std::string and has one argument, which is a pointer to a function with no arguments that returns a std::string?

C++ Tip of the Week

- "Most vexing parse"
  - C++ has an ambiguity in its declaration syntax

```
float x(3.1);
int bar(int(x));
```

- Is this:
  - A variable of type int initialized to int(x)?
  - The forward declaration of a function that returns an int and has one argument, which is an int named x?

C++ Tip of the Week

- C++ actually uses the second interpretation
  - The forward declaration of a function is the default
  - To define a variable, enclose the initial value in more parentheses:

```
// forward declarations
std::string foo(std::string());
int bar(int(x));

// variable declarations
std::string foo((std::string()));
int bar((int(x)));
```

Weeklies