
LECTURE 6
Thanks, Obama!
Pathfinding
Funnel Algorithm
Obstacle Avoidance
Tips for Platformer 3

Pathfinding

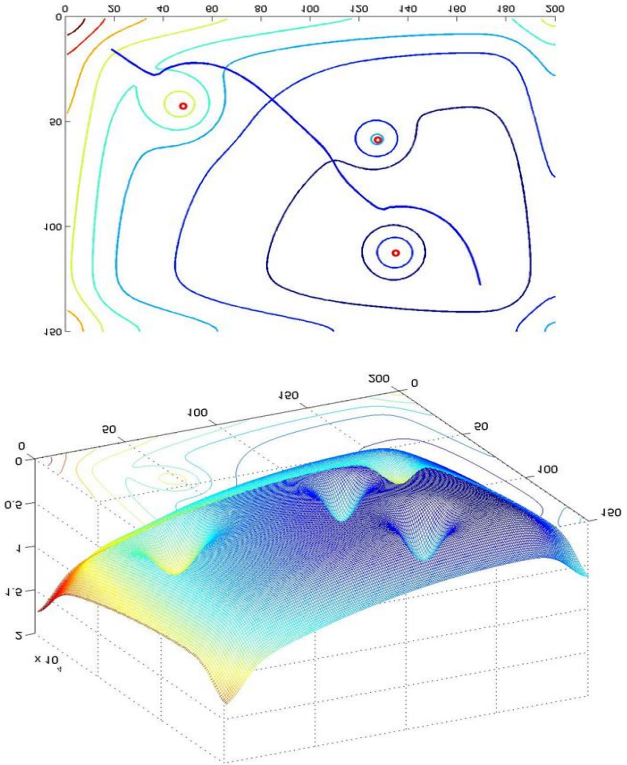
- Pathfinding is the most common primitive used in game AI
- A path is a list of instructions for getting from one location to another
  - Not just locations: instructions could include “jump” or “climb ladder”
- A hard problem!
  - Bad path planning breaks the immersive experience
  - Many games get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

3D world representation

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

Pathfinding with potential fields

- Potential field: a region of potential values
  - Usually a 2D grid in games
  - Good paths can be found by hill climbing
- 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

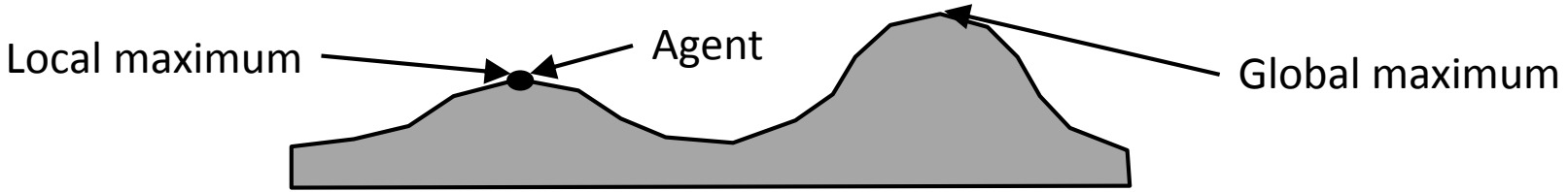


Potential fields: algorithm details

- 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 tick (~60 times a second)
  - Pathfinding entities move towards direction of greatest potential increase (hill climbing)

Pros and cons of 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

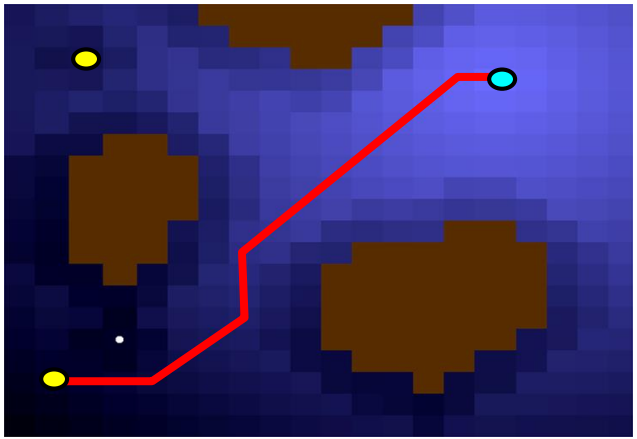


Reconsidering potential fields

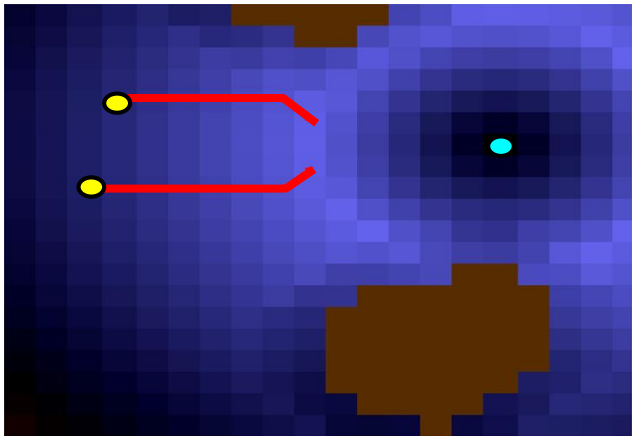
- Not actually used in many real games
  - We couldn't find any commercial releases that use them
  - But there are at least custom Starcraft bots that do
- Instead, most games use graph-based path planning

Choosing a potential function

- Potential functions don't need linear falloff



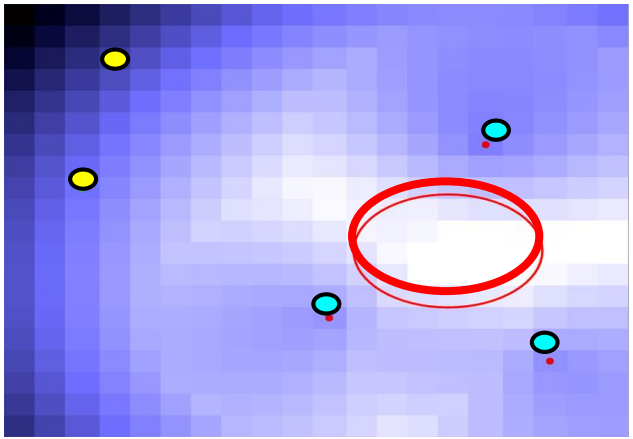
Linear falloff leads to a target



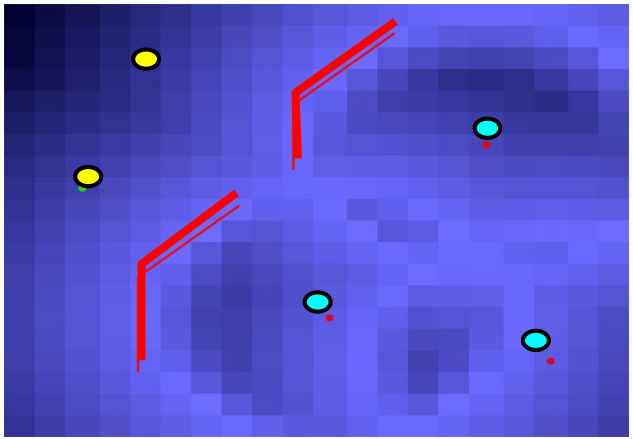
Rise then fall leads ranged units to a safe distance away

Pathfinding with Potential Fields

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



Summing creates false desirable spot for ranged units



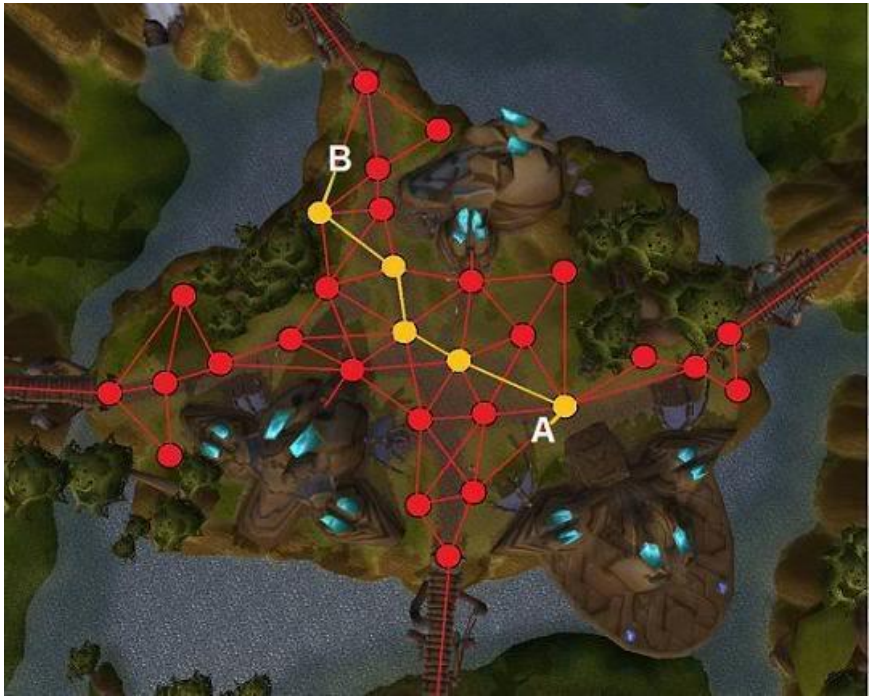
Maximum correctly identifies desirable areas for ranged units

Graph-based path planning

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

Waypoint graphs

- Represents a fixed set of paths through the world
- Nodes are waypoints
- Edges represent a path between adjacent nodes



Disadvantages of waypoint graphs

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

Navigation meshes

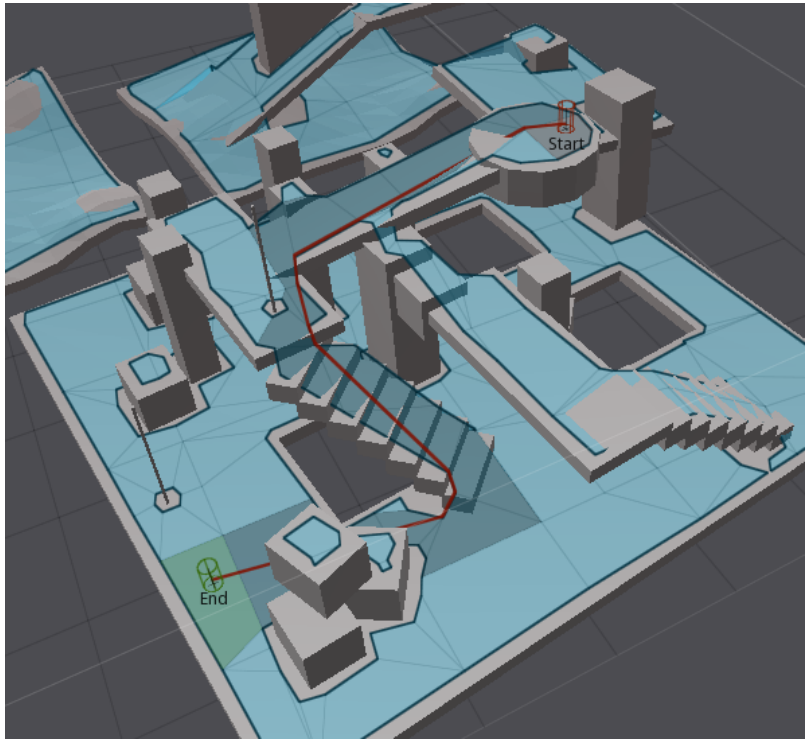
- Convex polygons as navigable space
- Nodes are polygons
- Edges show which polygons share a side

Advantages of navigation meshes

- More efficient and compact representation
  - Equivalent waypoint graph would have many more nodes and would take longer to traverse
- Models entire navigable space
  - Can plan path from anywhere inside nav mesh
  - Paths can be planned around dynamic obstacles
  - Zig-zagging can be avoided
- 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

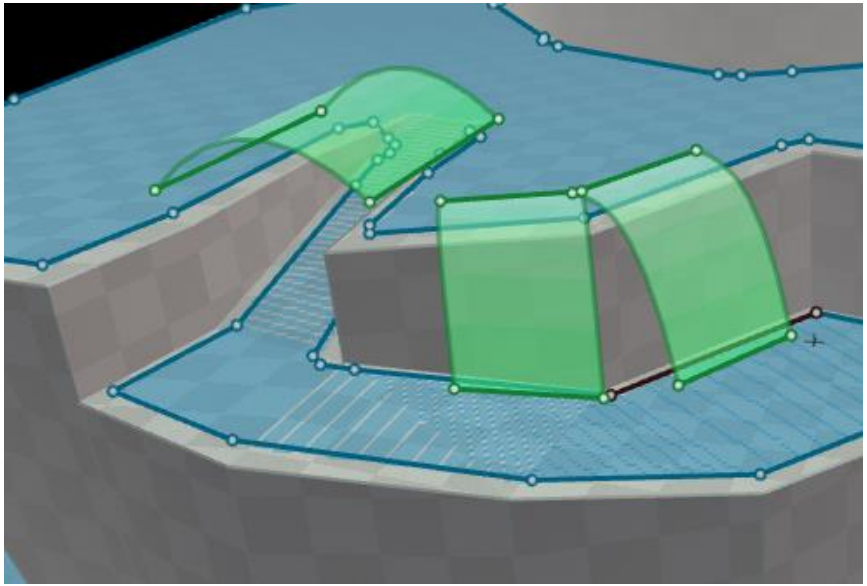
Navigation meshes

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



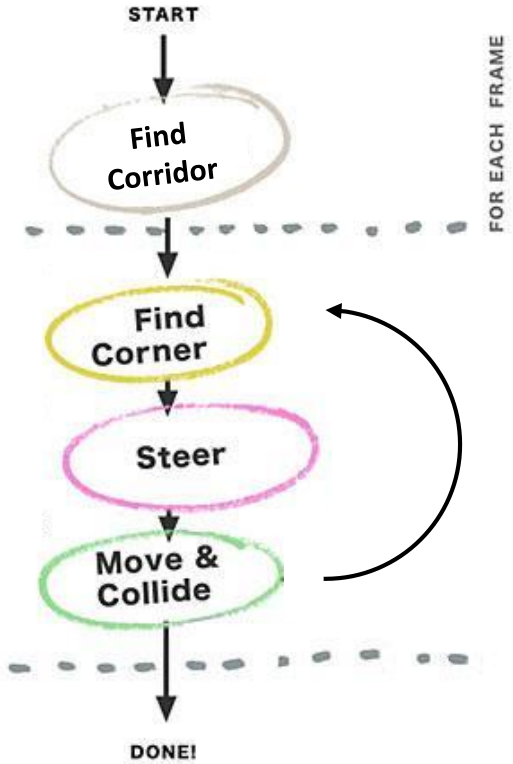
Navigation meshes

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



Navigation loop

- Process for robust path navigation on a navigation mesh:
  - Find sequence of polygons (corridor) using graph algorithm
  - Find corner using string pulling (funnel algorithm)
  - Steer using smoothing
  - Actually move/collide entity

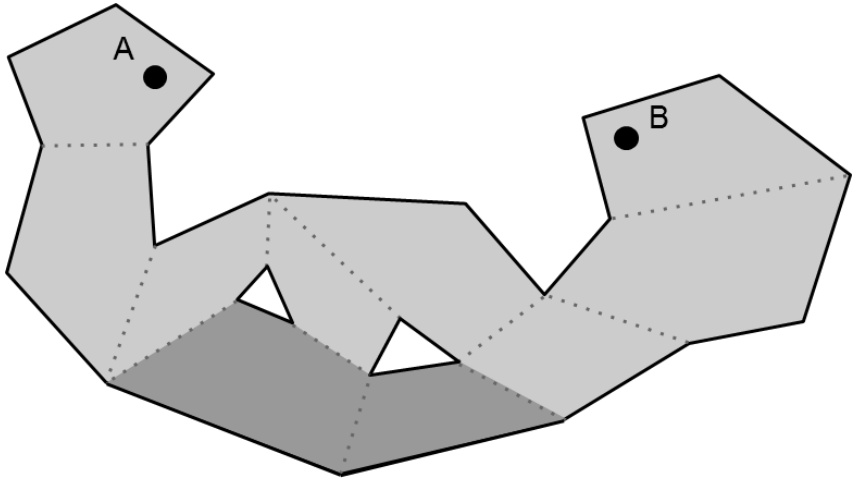


Graph search

- 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 (mud or steep surfaces may have higher cost)
  - Uses a heuristic to arrive at an answer faster

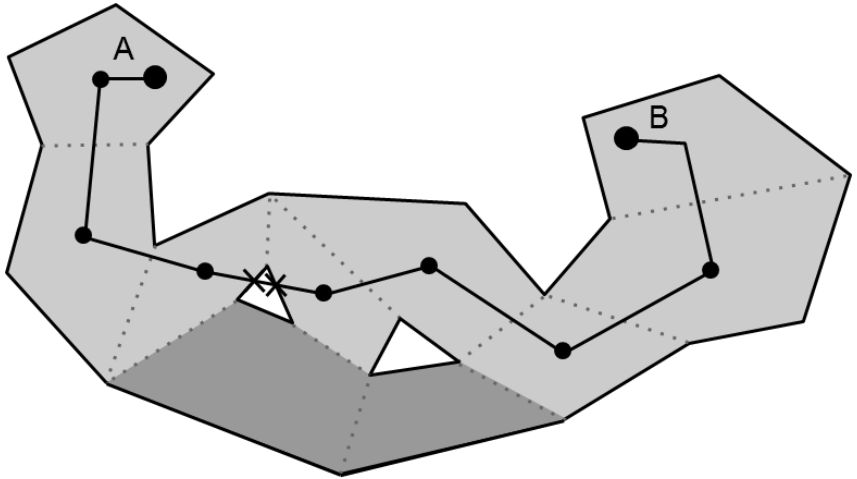
Path generation: problem statement

- Given a list of polygons (output of a graph search)
  - The light polygons
- Construct the shortest path for the agent
  - Where a path is a sequence of connected segments
  - The path must lie entirely in the list of polygons



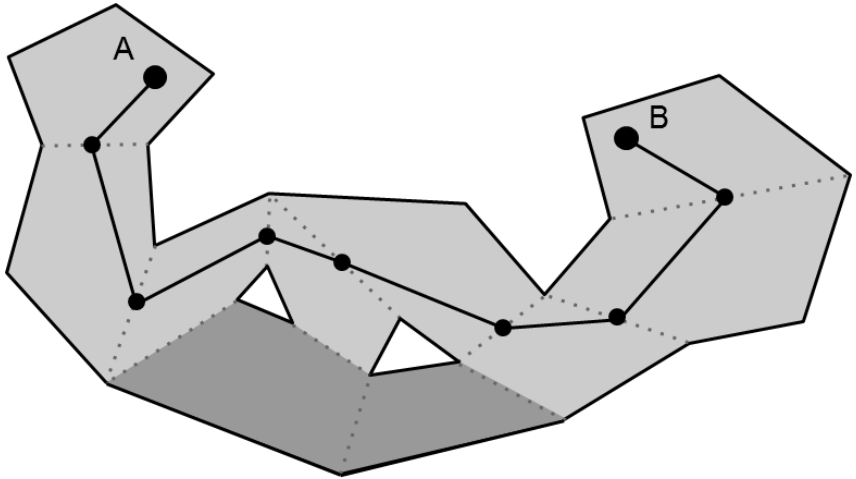
Path generation: first attempt

- Can we just connect polygon centers?
- No: the path might not even be within the polygons
- Polygons' convexity only tells us that any 2 points in a single polygon can be connected with a segment



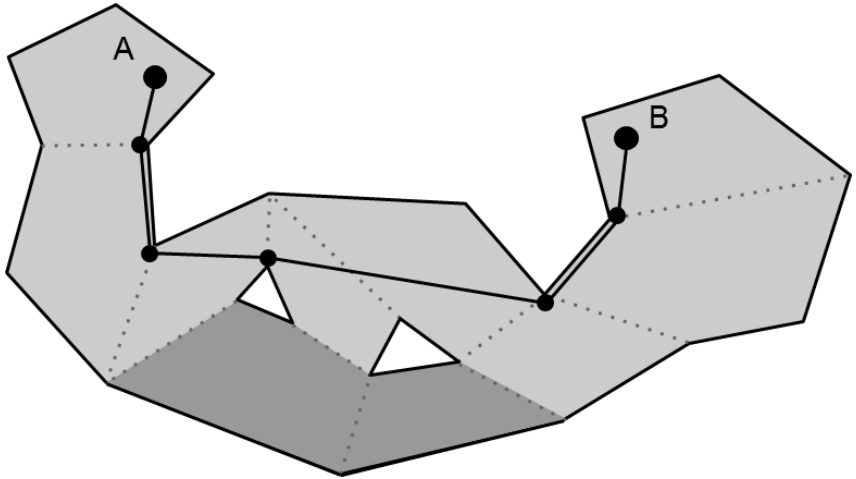
Path generation: second attempt

- Can we just connect centers of polygon sides?
- This always produces a valid path (within the polygons)
- But not always the optimal path (zig-zagging)
- This is just a waypoint graph!



Path generation: third attempt

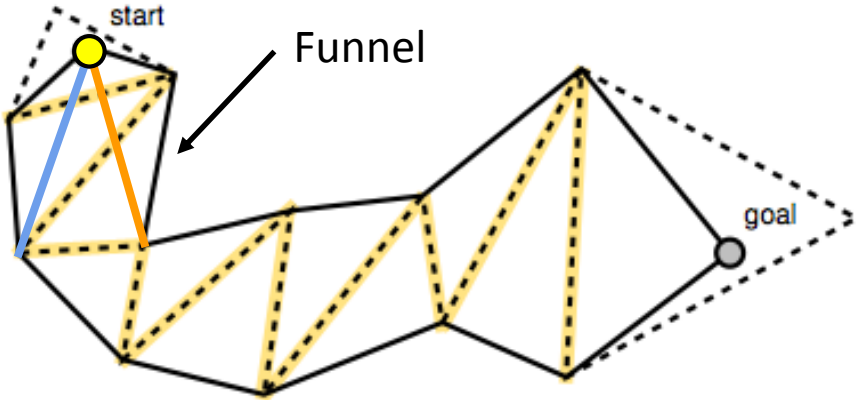
- The Funnel algorithm finds the optimal path
- Hugs corners
- Is like “pulling a string” that connects A and B until it is taut



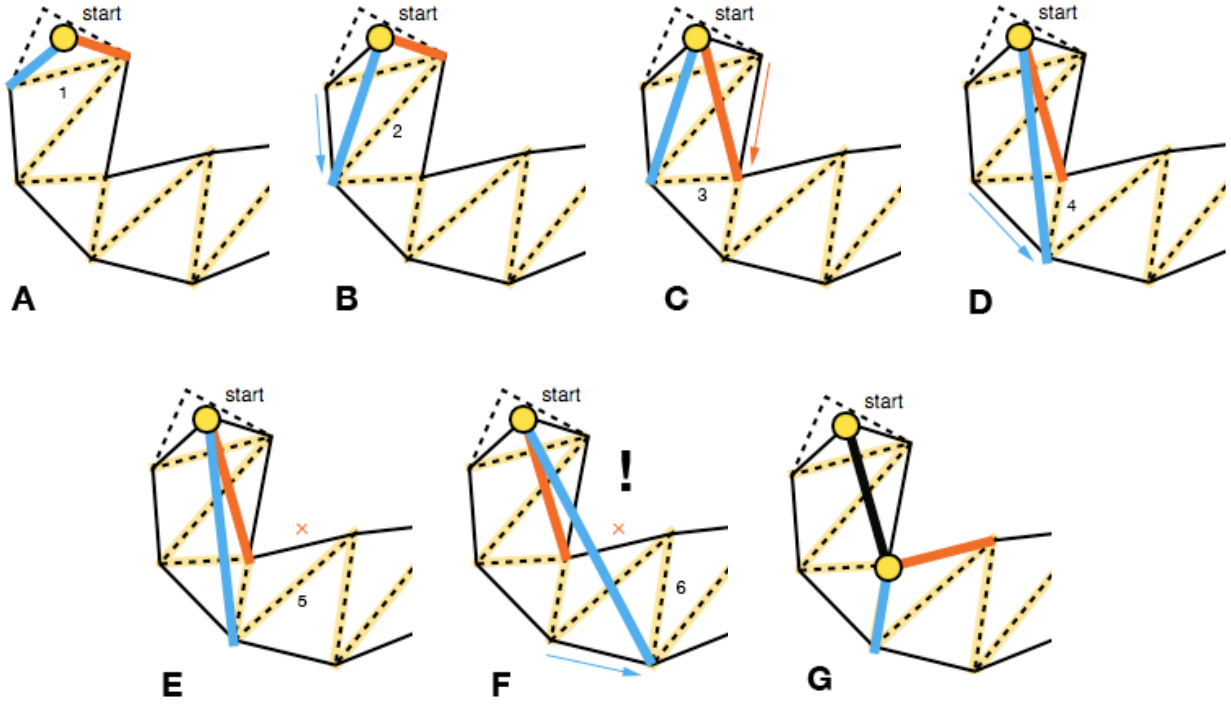

LECTURE 6
Simple Stupid™!
Pathfinding
Funnel Algorithm
Obstacle Avoidance
Tips for Platformer 3

Funnel algorithm

- 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



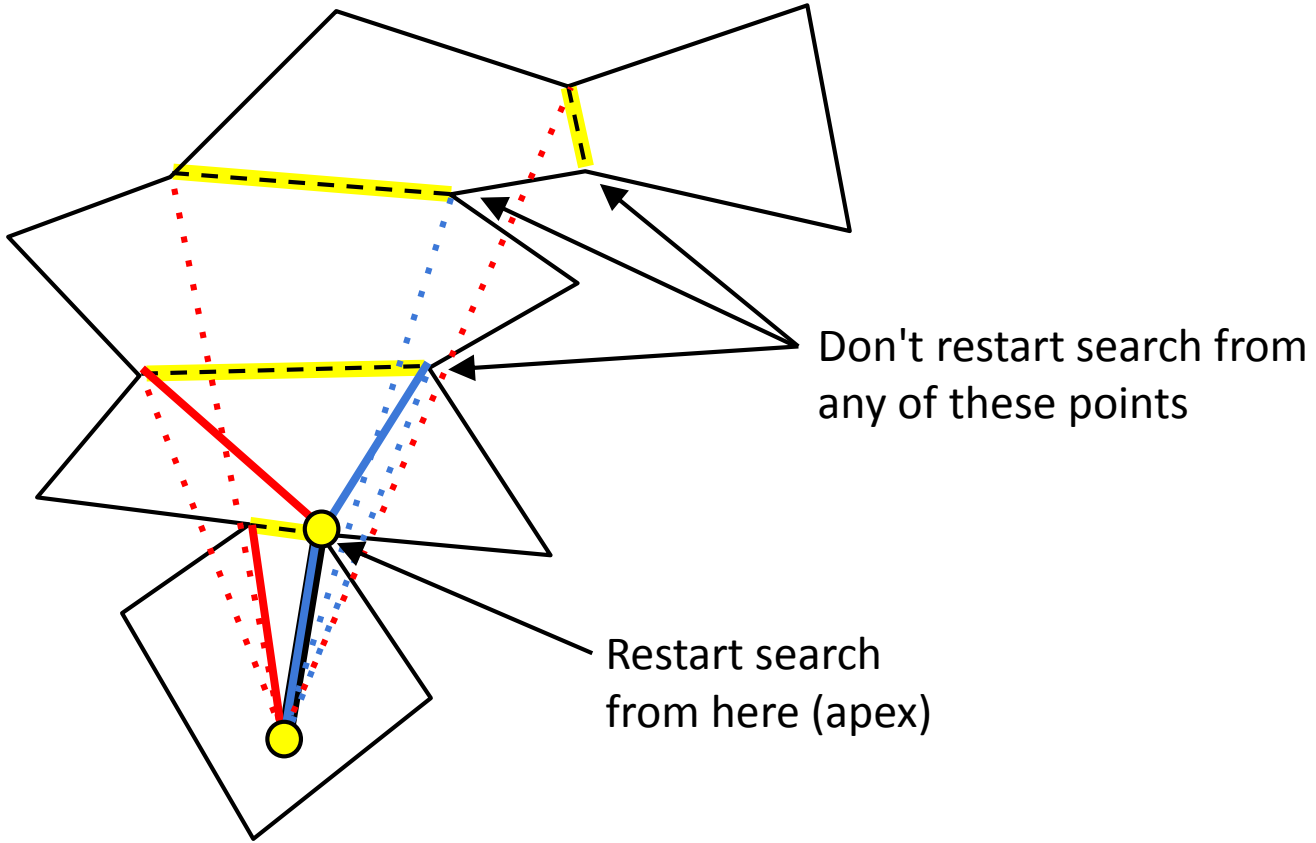
Funnel Algorithm

- 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 previous slide)
    - If past the right side of the funnel, turn a corner (E-G in previous slide)
      - Add right point to path
      - Set apex point to right point
      - Restart at next portal
  - Try to move right point to right vertex of next portal
    - Similar to left point

Edge cases

- Zero-length funnel side (portals that share a vertex)
  - Always use left*0.99+right*0.01 for the left and left*0.01+right*0.99 for the right (shrinks portal slightly)
  - Use actual vertices for left and right points
- End iteration of the funnel algorithm
  - End point is portal of size 0, need to check for potential restart like other portals

Funnel algorithm example




LECTURE 6
Watch out for that tree!
Pathfinding
Funnel Algorithm
Obstacle Avoidance
Tips for Platformer 3

Steering

- There are many different ways for an entity to move towards a point
- Moving in straight lines towards each destination gives a robotic look
- Many alternatives exist: which to 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 there are no collisions, moving and colliding is as simple as using the destination and steering to move
- Collisions can cause a variety of issues
  - May need to re-plan path if a collision is impeding movement
  - Can detect getting stuck if the entity stays in roughly the same spot for a few seconds

Obstacle avoidance

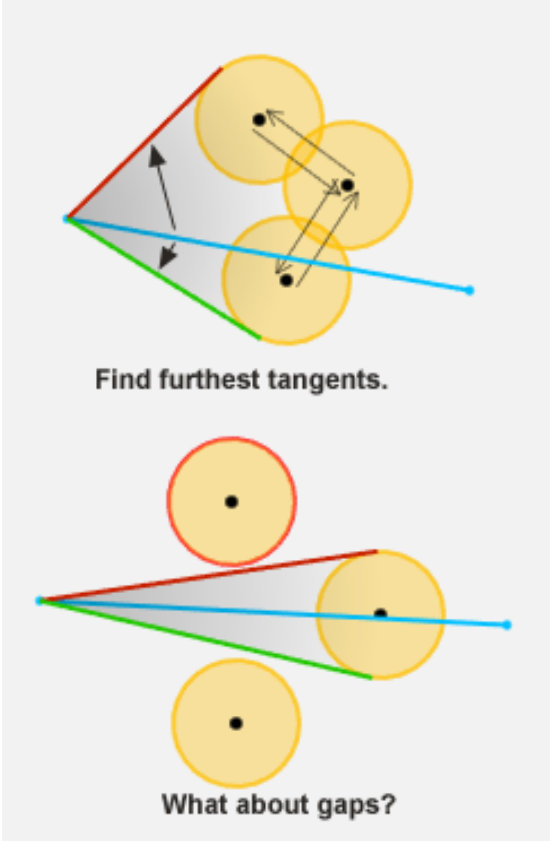
- Static obstacles can be avoided by generating the right navigation mesh
- Dynamic obstacles are trickier
- Baseline approach for dynamic obstacles
  - Use raycast or sweep test to determine if in obstacle is in the way
  - Apply steering force away from obstacle
  - Adjust force based on distance to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

Dynamic obstacle avoidance

- If we consider each obstacle individually, this is purely local avoidance
  - Can easily get stuck in local minima
  - Remember, this step is added on top of global path planning
- We need an 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

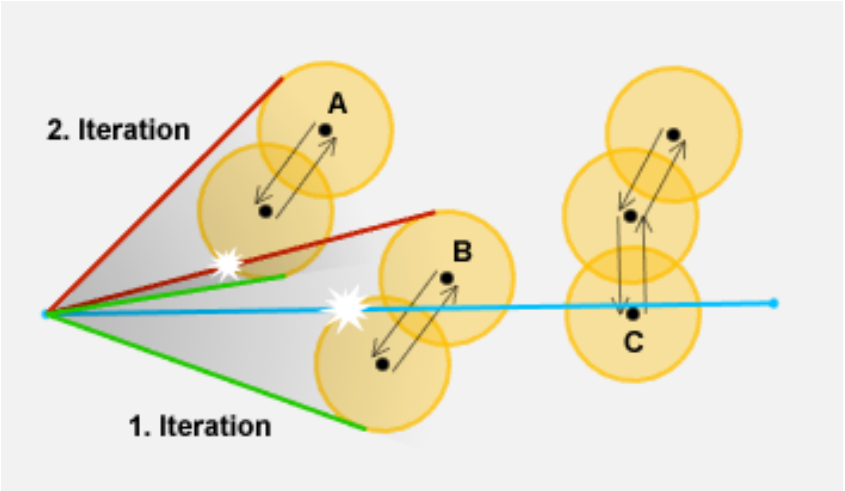
Very Temporary Obstacle Avoidance

- 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



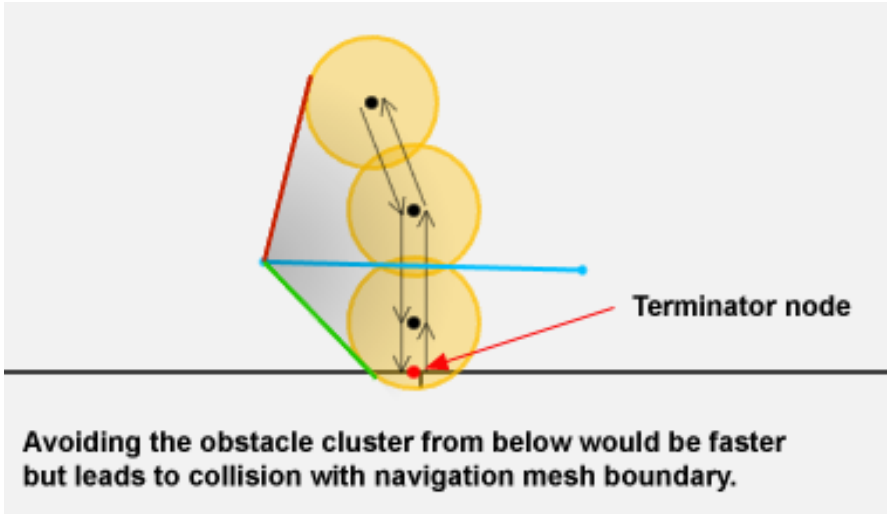
Very Temporary Obstacle Avoidance

- 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



Very Temporary Obstacle Avoidance

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



Very Temporary Obstacle Avoidance

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
- Detour: navigation mesh pathfinding

Case study: Recast and Detour

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/


LECTURE 6
Better than Apple Maps!
Pathfinding
Funnel Algorithm
Obstacle Avoidance
Tips for Platformer 3

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*, though this is optional
- Generate a path using the funnel algorithm
  - Pretend polygons are 2D in the horizontal plane
  - Collision response will handle the vertical position
- Local obstacle avoidance is not required

Platformer: week 3

- In your handin:
  - 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

- Reference types
  - Like pointers, but can't be reassigned
  - Better than value types (no copying)
  - Better than pointers (safer, more limited)
  - Good for knows-about relationships, especially back-references
- Syntax:

```
World& w = player.getWorld();
```

References example

```
class Player {
	// doesn't own, knows about
	World &w;

	Player(World &w) : w(w)
	{
		// modification is ok
		w.clearEntities();
	}

	World &getWorld() {
		return w;
	}

	void setWorld(World &w2) {
		// BAD – copies w2 into w1!
		w = w2;
	}
}
```

```
// Good for helper functions
void clamp(Vector3 &v,
		const Vector3 &min,
		const Vector3 &max)
{
	for (int i=0; i < 3; ++i) {
		v[i] = fmax(min[i], fmin(max[i], v[i]));
	}
}

// Good for out parameters
bool raycast(float &outT, Vector3 &outP,
		const World &world, const Ray &ray)
{
	// ...
}
```

Weeklies!