# Game AI I: Pathfinding

# Paths Overview

Given some representation of a space, how do we get from one point to another?

- The space could be abstract or physical
  - abstract spaces include narrative nodes in a murder mystery, or steps in a planning algorithm, or points in a search tree, or so many other things
- “Get” could mean traverse in the sense that we’re used to, or it could be some other transition function
  - taking an action, adding two numbers, consuming some resource, physically moving
- The validity of the path could depend on a number of things
  - obeying physical laws (doesn’t go through obstacles, etc.)
  - is below a total calculated cost (this could be distance, or difficulty, or otherwise)
  - goes through a certain set of points in the space

# Paths in Physical Spaces

We talk predominantly about paths through physical space.

1. Moving through physical space is something that we do *a lot*
2. It's also *soooooo* complicated.

- We walk around a lot, and being in the right place (at the right time (via the right path)) is a huge portion of looking intelligent.
- There are so many ways that physical space can screw us over.
- There is so much space (so computation on all of it can be really expensive).
- There is so much space¡¡¡¡¡

# Aside: Looking Intelligent

Game AI programming is about giving virtual characters the appearance of intelligence. But what do we mean by that? Here are some things that we look for when making characters look “smart.”

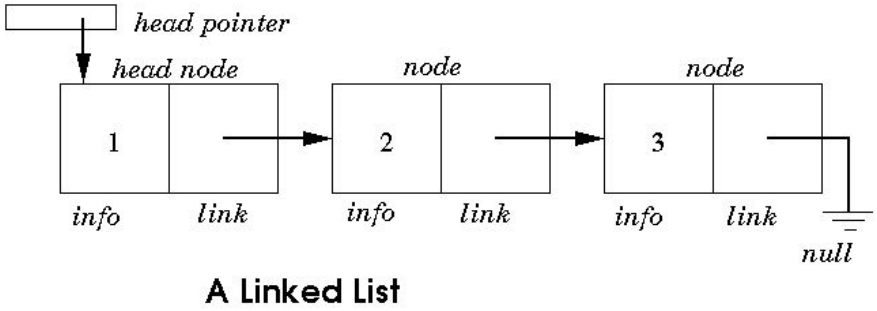
- Autonomy—characters shouldn’t need a “Wizard of Oz”
- Reaction—characters should respond to what’s going on around them
- Nondeterminism—characters should *never* do the exact same thing twice
- Cultural Authenticity—characters should represent the culture they embody
- Believability—characters should act like humans (for some value thereof)
- “Non-stupidity”—characters should be free of glitches that paint them as dumb

The first five elements of this list are taken from Kevin Dill’s excellent “A Game AI Approach to Autonomous Control of Virtual Characters,” accessed from https://course.ccs.neu.edu/cs5150f13/readings/dill_granny.pdf

# Space Representations

# Representations are Important

- The right representation can save (or add) a ton of work
- Data structures change your affordances as a designer/programmer
- Encodings are often more impactful than algorithms




A Linked List

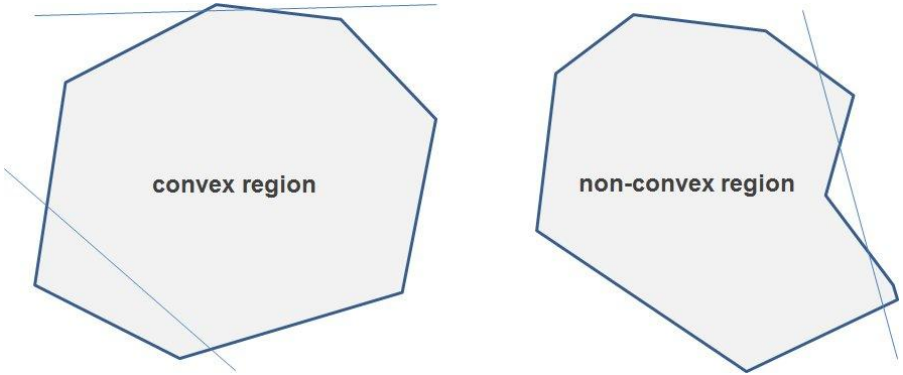
https://people.engr.ncsu.edu/efg/210/s99/Notes/LLdefs.gif
(Never seen “ground” represent NULL before)

# Convex Representations

- For any pair of points in a convex region, there is a straight line path between them
  - If you can get to the region, you can traverse it completely with no worries

A surefire way to represent things:

- Divide space into convex regions
- Make each region a graph node



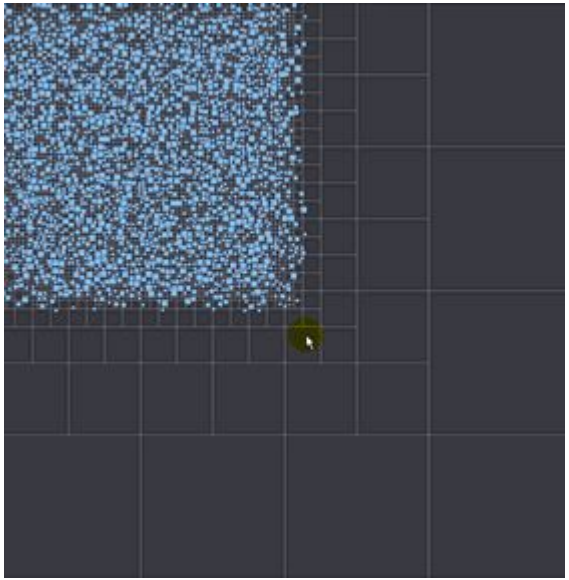

https://squared2020.files.wordpress.com/2015/11/convex-non-convex.jpg

# Tilemaps

- Turn the space into a series of convex divisions
  - Usually squares or hexagons
- A question of the size of the tiles relative to the units
  - Tiles >= unit size (e.g., chess)
  - Tiles << unit size (e.g., StarCraft)

# Quadtrees

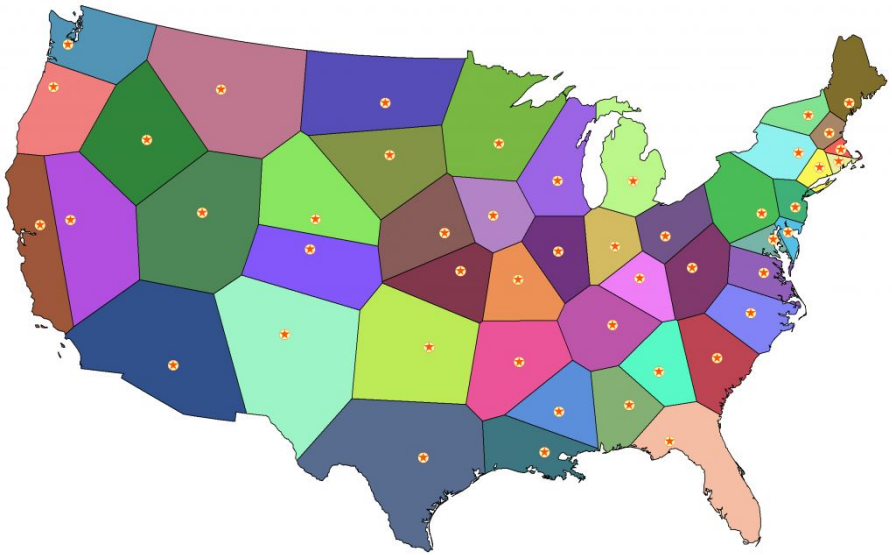
- Carve up space by emptiness
- Subdivide into segments that:
  - Contain nothing
  - Or are your  chosen smallest resolution
- Dynamic
- “Easy” to generate automatically



https://i.stack.imgur.com/6cQeQ.gif

# Voronoi Diagrams

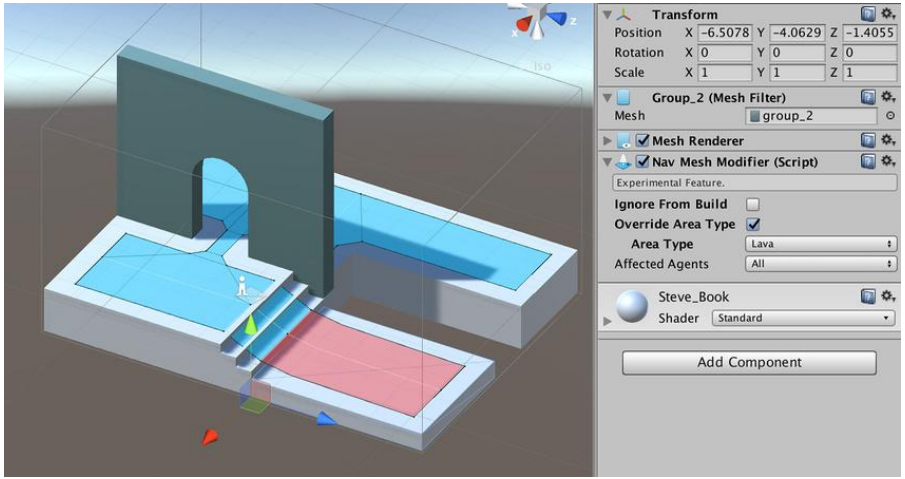
- Each vertex defines a region of all of the points which are closer to that vertex than any of the others
- Points in a region are by definition closer to its vertex than any other
- Regions are implicitly convex
- This is how zip codes work (fun fact)



https://www.thedataschool.co.uk/wp-content/uploads/2016/07/USAthen2-1024x655.png

# Navigation Meshes (Navmeshes)

- Pre-baked areas of traversability/non-traversability
  - Given things like radius, height, maximum step distance, etc.
- These are either annotated (á la waypoints) or they are generated algorithmically
  - Unity, for instance, offers Navmesh support



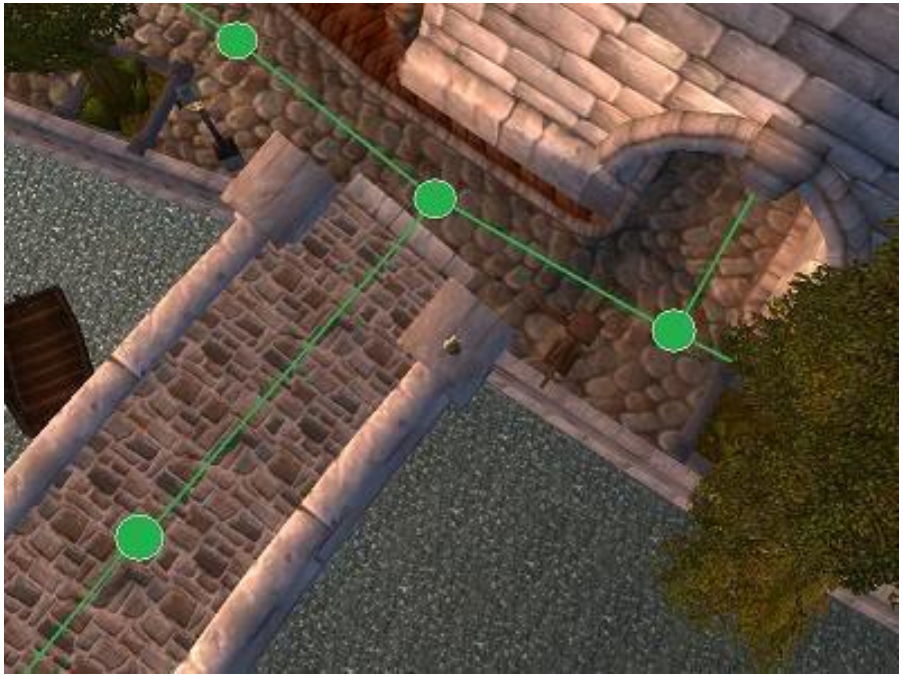

https://docs.unity3d.com/uploads/Main/class-NavMeshModifier-3.jpg

# “Odd” Representations

- The previous representations are common
- But they’re not the only option
- This is kind of a weird survey lecture anyway
- So here are two random additional representations

# Waypoints

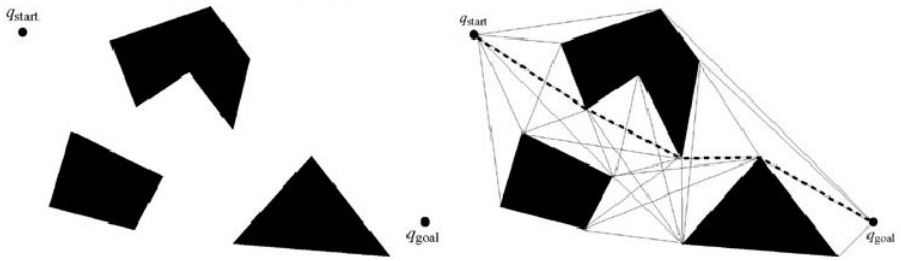
- Hand-annotated nodes in the space
- Agents find a path by:
  - Finding the nearest node, going there
  - Finding the node nearest their goal
  - Executing a pathfinding algorithm to get from point A to point B
  - Walking from the final node to their goal
- This is easy to code and to design



http://i252.photobucket.com/albums/hh9/PaulTozour/Stormwind_waypoints.jpg

# Visibility Graphs

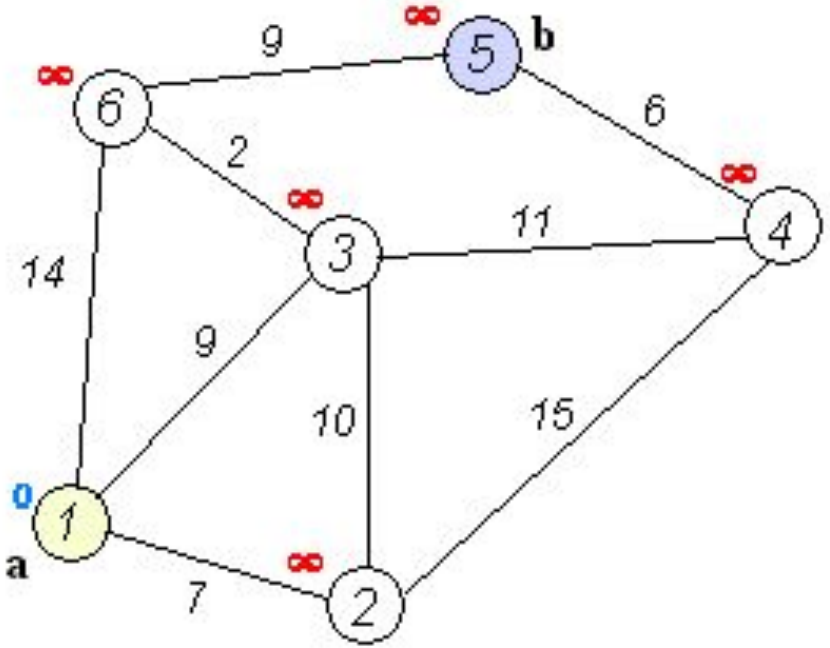
- The shortest path through polygonal obstacles will always skirt their corners
- Use the corners of the obstacles as waypoints
- These regions are also convex, but I’ve never seen these anywhere, so I put them in the “weird” section



# Dijkstra’s Algorithm and A*

# Dijkstra’s Algorithm

- Classic pathfinding algorithm!!!!
  - Wow!!!!
- Examines nodes in order of increasing distance from the start
- Guaranteed to find the shortest path (if it exists)




https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

# Dijkstra’s Complexity

If the shortest path has **n** tiles:

- Dijkstra’s has to search all tiles of distance **n - 1**
  - **$O(n^2)$** for a 2D grid
  - **$O(n^3)$** for a 3D grid
- Each tile requires a heap reordering
  - **O(log n)**
- So the total cost is
  - **$O(n^2 \log n)$** for a 2D grid
  - **$O(n^3 \log n)$** for a 3D grid
- ¡Super expensive!

# Problems with Dijkstra's

There's nothing we can do about the expense—this is why we try to use representations that chunk up space nicely.

- However, Dijkstra's does blindly search the space of possible nodes…
- …Going so far as to search nodes that are in the opposite direction of the goal
  - Which is something that you want to make sure *does happen*
  - But not in preference to searching nodes closer to the goal

# Informed (Heuristic) Search

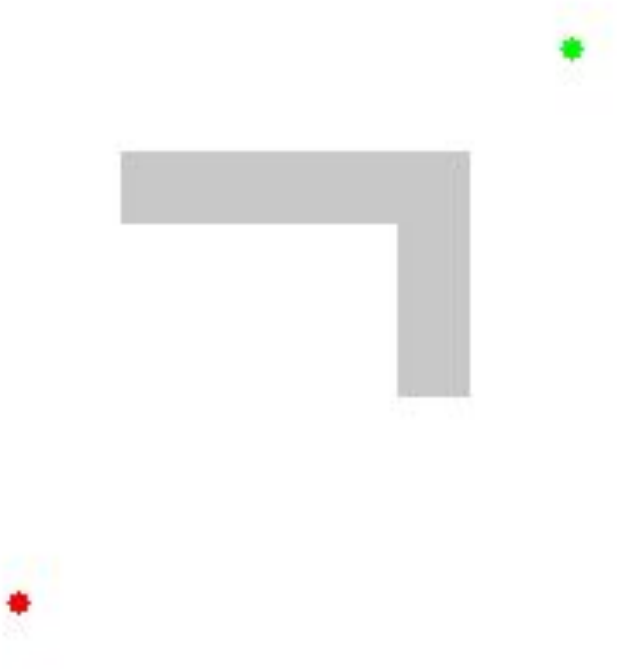
- “Common sense” to search nodes closer to the goal first
- Expected to work better than blind search
  - Sometimes worse, but…
- Rules of thumb that inform search are referred to as heuristics


https://dilbert.com/strip/2008-11-18

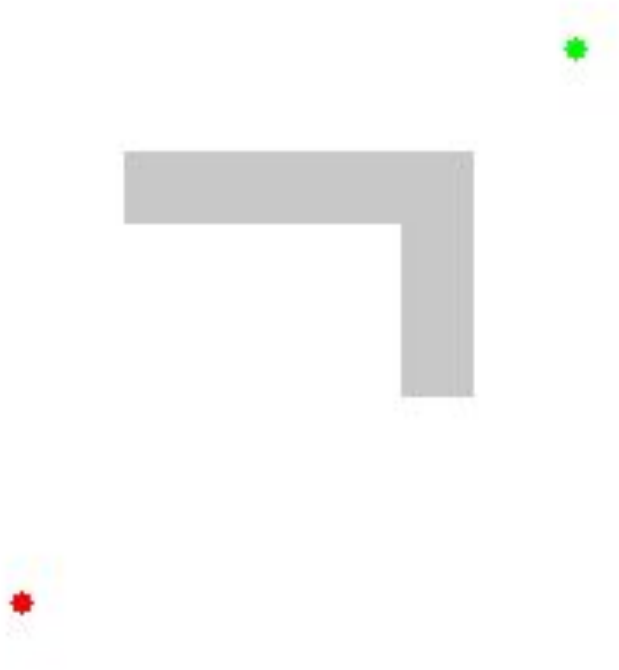
# Informed Search: A*

- Old-school (1968) algorithm designed for Shakey the Robot
- Usually uses euclidean distance from the goal as its heuristic
- Modification of Dijkstra's:
  - Sort the heap not just by distance from the start node, but also estimated distance from the goal
- Tries nodes nearer to the goal first
  - Backs up only when necessary

https://www.sri.com/work/timeline-innovation/images/shakey.jpg



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm



https://en.wikipedia.org/wiki/A*_search_algorithm

# Admissible Heuristics

The heuristic has to be **admissible**:

- **Non-negative**: $h(N) > 0$
- **Optimistic**: $h(N) <= cost_{NGoal}$

Common heuristics:

- **Zero (0)**
  - This is Dijkstra's
- **Euclidean Distance to the Goal**
  - A (physical) path can't be shorter
  - This gets used most of the time

etc.

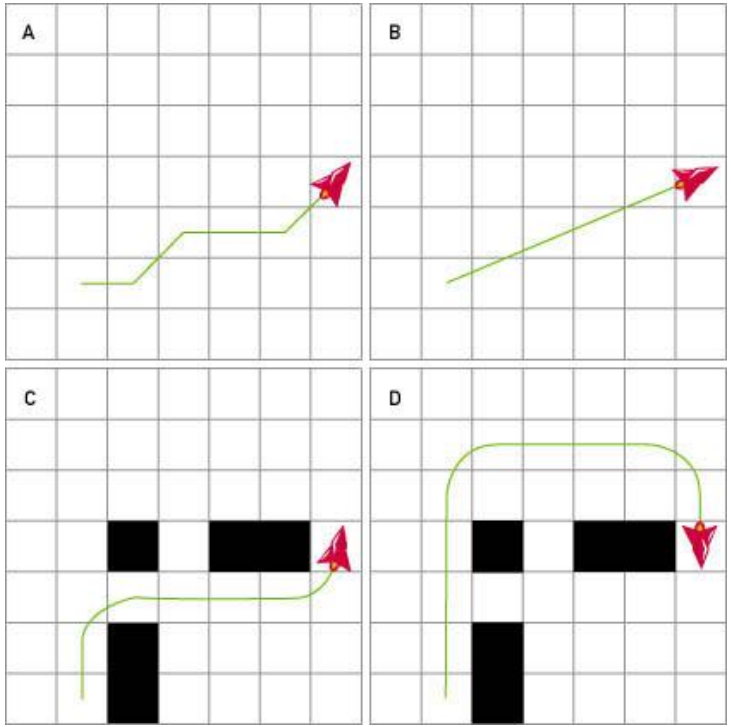
# General Path Troubles

- Generally pretty expensive :(
- Cost of failure is high
- Frustrating tradeoffs w.r.t. accuracy:
  - High resolution ➜ expensive
  - Low resolution ➜ janky
- Numerous edge cases which are hard to exhaustively test
- etc.

https://thumbs.gfycat.com/IcySelfreliantCreature-max-1mb.gif

# Fiddly Path Troubles

- Differently-sized agents
  - If you just use a representation that suits the smaller agents, you might have unexpected collisions
  - If you use one that suits the bigger ones, you might have awkward positioning
- Zig-zagging
- Smooth turning
- Turning radius



# Path Improvements

- Splining (spline interpolation)
  - Smooth out a series of points by finding curvers that fit among them
  - But watch that your new curve doesn't run into any obstacles!
- Rubber-banding
  - "Snap" path to nearby corners so that it looks more sane
  - Make sure to leave sufficient room that your agents aren't scraping the wall
- Linear Interpolation
  - The agent's goal is a weighted average of the next two waypoints

# Tell me your questions.