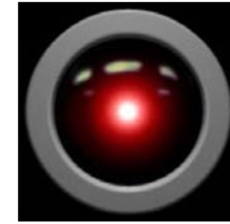
Game AI

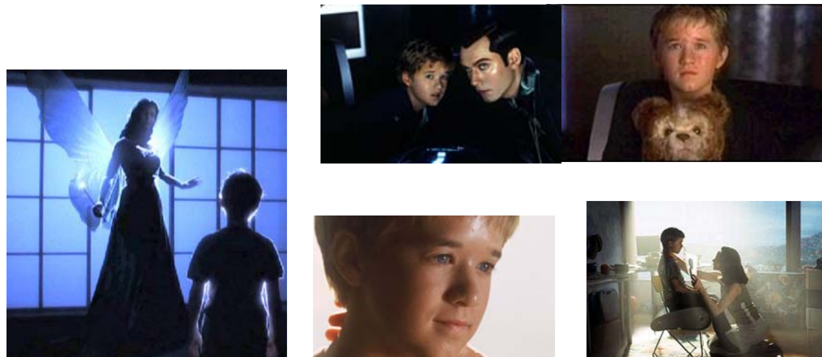
305900
Fall 2010
11/8/2010
Kyoung Shin Park

HAL



HAL (Heuristic ALgorithmic) in Standley Kubrick's film "2001: A Space Odyssey" (1968)

David



Monica "turns on" David's "love" function in which he virtually morphs from a robot to a boy

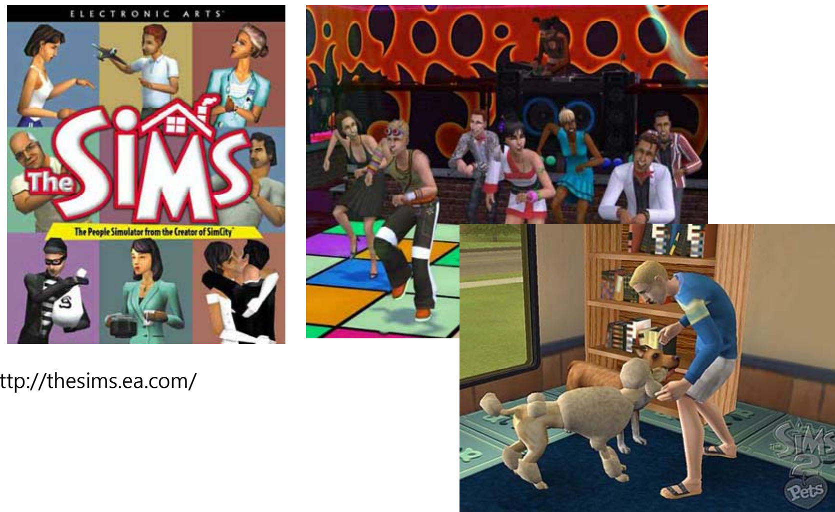
The *mecha* David in Steven Spielberg's film "A.I.: Artificial Intelligence" (2001)

Interactive Drama - Façade

"Façade takes character to a new depth.. Trying to push the boundaries of both gaming and AI" – Newsweek, Oct, 17, 2005
http://www.interactivestory.net/

Emotional Rescue: Are interactive soap operas the future of gaming? http://www.cbc.ca/arts/media/facade.html

The Sims



http://thesims.ea.com/

Game AI

- The computer always has perfect knowledge of the entire virtual world. When you press a button the computer knows instantaneously that you have pressed it. That means in a fighting game, it can always deflect every blow you dish out and dodge every bullet!
- The noble goal is to require that the AI system follow the same rules and constraints as the human so that it plays fairly.
  - E.g. in a strategy game the human can only see parts of a map that they have explored.
  - In practice this is still very difficult and many AI systems still seem "dumb" in certain circumstances. E.g. Jedi Academy game, NPCs often fall to their deaths.
  - To get around this the AI system "cheats" a little but not so much that it is obvious to the player.
- Game AI is largely an unsolved problem.
- NPCs still follow a limited set of rules based on FSMs and rule-based systems. They cannot do more than what is expected of them. Especially evident in RPGs which have a limited set of responses to user input. You cannot have a long and open ended conversation with the computer.

Game AI

- Game AI History
  - The first video games developed in the 1960s and early 1970s, like Spacewar, Pong, Gotcha, were games implemented on discrete logic, and strictly based on the competition of two players, with AI.
  - Use a simple FSM in 1970s era, like Pac-Man, Space Invaders.
  - Focus on high-fidelity graphics and hardware in 1980s era.
  - The emergence of new game genres in the mid 1990s promoted the use of formal AI tools like FSM.
    - Half-Life (1998) featured enemies that worked together to look for the player. They also lobbed grenades from behind cover, ran and hid from the player when injured and patrolled areas without breaking the pattern.
    - Halo (2001) featured AI that could use vehicles and some basic team actions. The AI could recognize threats such as grenades and incoming vehicles and move out of danger accordingly.

Game AI

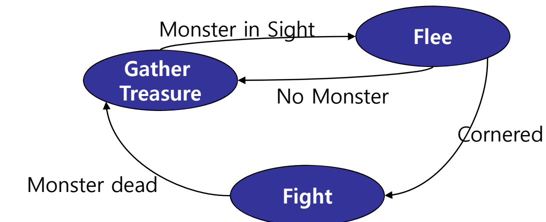
- Finite State Machine
- Trigger System
- Production System
- Search
- Planning System
- Multi-agent System
- Robot
- Artificial Life
- Evolutionary Algorithm
- Flocking
- Neural Network
- Fuzzy Logic
- Path Finding: A* Algorithm
- Scripting

Mainstream AI techniques that have been applied to Games

- Finite State Machines
  - FSMs are one of the most commonly used programming structures for games.
    - Set of states (S)
    - Input vocabulary (I)
    - Transitional function T(s, i)
  - Each state represents some desired behavior.
  - The transition function T resides across all states.
  - Accepting states (those that require more input) are considered to be the end of execution for an FSM.
  - Input to the FSM continues as long as the game continues.

Mainstream AI techniques that have been applied to Games

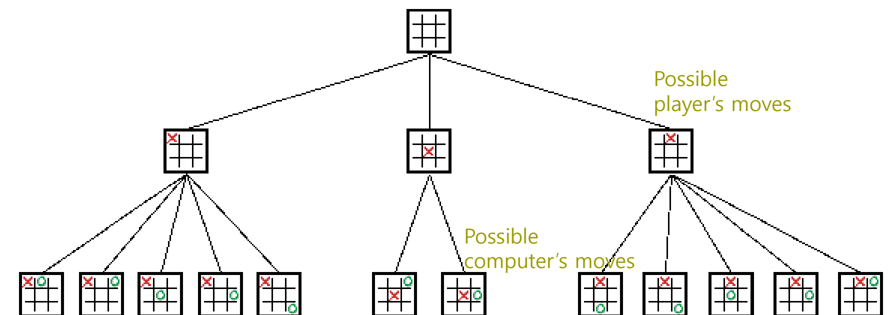
- Finite State Machines
  - Character AI can be modeled as a sequence of mental states.
  - World events can force a change in state.
  - The mental model is easy to grasp, even for non-programmers.
  - In Quake (FPS Game)
    - Bot has the states like FindArmor, FindHealth, SeekCover, RunAway, etc.
    - Missile has the states like Move, TouchObject, Die, etc.



Mainstream AI techniques that have been applied to Games

- Trigger Systems
  - Use if/then rules (if condition, then response)
  - Simple for designers/players to understand and create
- Production Systems
  - Essentially a set of if-then-else rules that match a set of input & generate a new set of "input" that may cause other rules to fire-hence the system is capable of performing inferencing.
  - Useful in adventure games when a combination of inputs and states can infer something about a situation.
- Search
  - Permuting all possible states of a game board to attempt to predict outcome.
  - Used in chess and pathfinding.

E.g. Search : Tic-Tac-Toe Search Tree (with symmetric moves removed)



Alternate between permuting all the players moves & all the computers moves. Estimate a "fitness" at each level so the computer can decide the "best" move to make up to a specified "lookahead".

Mainstream AI techniques that have been applied to Games

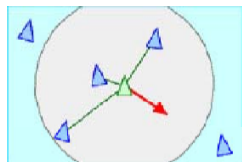
- Planning Systems
  - Used to determine the best set of tasks needed to go from a start state to an end state.
  - Useful for organizing a collection of troops for a unified goal.
- Multi-agent Systems
  - Studies how emergent behavior can arise from the interaction of a number of cooperating agents with limited knowledge
  - Agents are essentially individual FSMs with sensors.
  - NPCs are essentially modeled as Agents.
- Robotics
  - Very similar to problems in game AI - except much more difficult- inputs are from the real world where anything can happen.
  - Key is to understand WHAT is the minimum amount of information in the world you need to consider in order for your robot to function as expected.

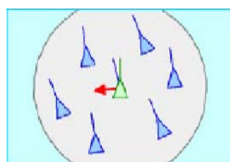
Mainstream AI techniques that have been applied to Games

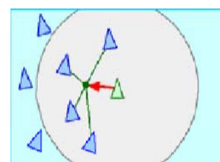
- Artificial Life/A-Life
  - Multi-agent systems that attempt to use living systems to AI agents.
  - Used in games like SimCity and SimAnt.
- Evolutionary Algorithm
  - Includes Genetic Algorithm, Evolutionary Computing
  - Encodes characteristics of a system as genes and uses "mutation" and "crossover" in conjunction with a "fitness function" to help evolve to a more "perfect system".
  - Can be used to evolve A-life creatures to make them "smarter."
  - E.g. Game may have many settings for the AI, but interaction between settings makes it hard to find an optimal combination.

Mainstream AI techniques that have been applied to Games

- Flocking
  - Simulates flocking of birds or schooling fish
  - Used to represent the movement of group of characters (mimics military formations) in the game, like Unreal Tournament, Half-Life, etc.
  - Three classic rules
    - Separation: avoid local flockmates
    - Alignment: steer toward average heading
    - Cohesion: steer toward average position
    - Avoidance


Separation


Alignment


Cohesion

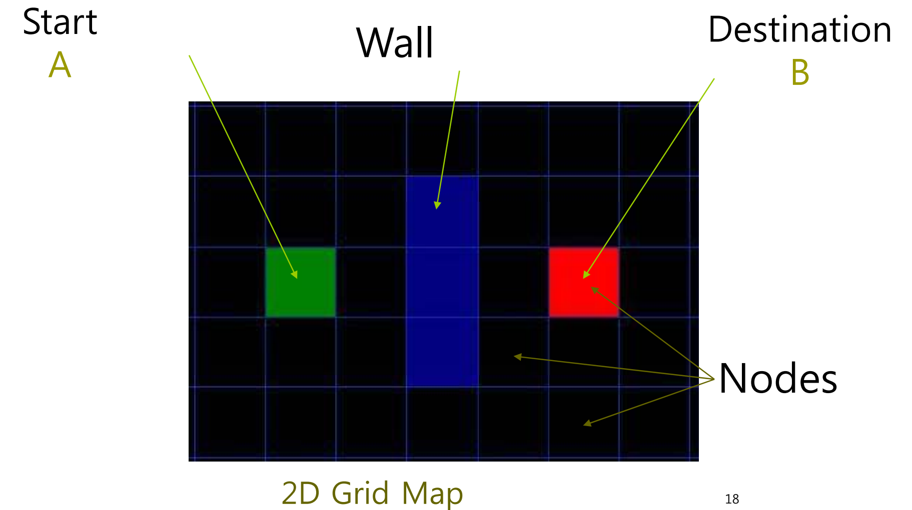
Mainstream AI techniques that have been applied to Games

- Fuzzy Logic
  - Use real numbers to determine if a situation belongs to 1 set or another. Kind of like if-statements with probabilities attached to them.
  - Used in The Sims to determine what the action of an agent might be, or could be used in RPGs to determine if something should attack or run away.
  - E.g., NPC's emotion, flocking algorithm, fuzzy-based FSM
- Neural Networks
  - Complex non-linear functions that relate one or more inputs to an output
  - Must be trained with numerous examples
    - Training is computationally expensive making them unsuited for in-game learning
    - Training can take place before game ships

Pathfinding

- Pathfinding
  - NPCs have to figure out how to navigate from point A to point B in a 3D terrain, in real time.
  - What is the most efficient way?

E.g. Given a terrain map as follows:

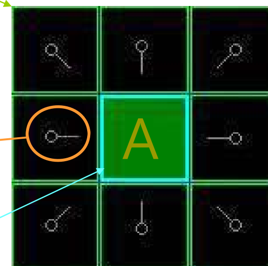


2D Grid Map

A Well Known Algorithm in AI called A*

Main idea: Search for the shortest path, starting at point A and checking the adjacent squares, and generally searching outward until we find our target.

1. Add A to the "TODO list". The **TODO List** is a list of squares that need to be checked out.
2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the TODO list, too. For each of these squares, indicate A as its "parent square".
3. Remove the starting square A from your TODO list, and add it to a "DONE list" of squares. The **DONE List** is a list of squares you've already explored.

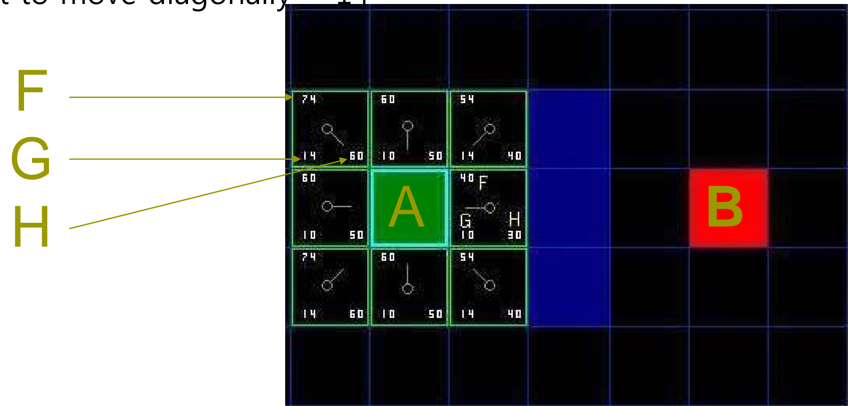


Now choose one of the TODO list items to explore next- but which one?

- Path Scoring
  - Compute the "cost" of traveling to each of the squares adjacent to A
  - **F = G + H**
  - F means Fitness
  - G is the movement cost to go from A to the current square. [G means Goal]
  - H is the estimated cost to go from the current square to the destination (B). [H means Heuristic]
  - E.g. Heuristic in this case is "Manhattan Distance" Distance to walk horizontally & vertically to reach a destination- like when you walk around city blocks (hence the name Manhattan Distance).

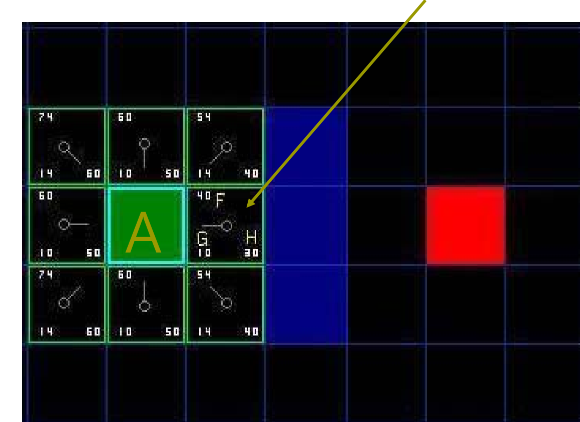
Calculate F,G,H at each of the TODO List items

Assume in this e.g.
Cost to move horizontally/vertically = 10
Cost to move diagonally = 14

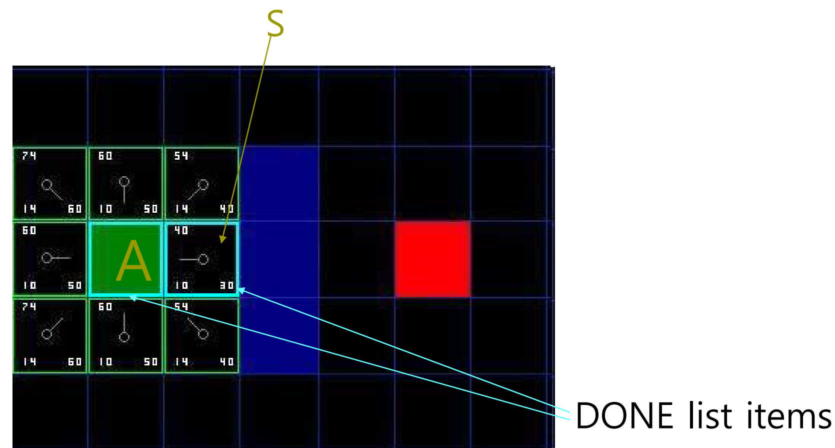


Select the node (S) with the smallest F from the TODO List

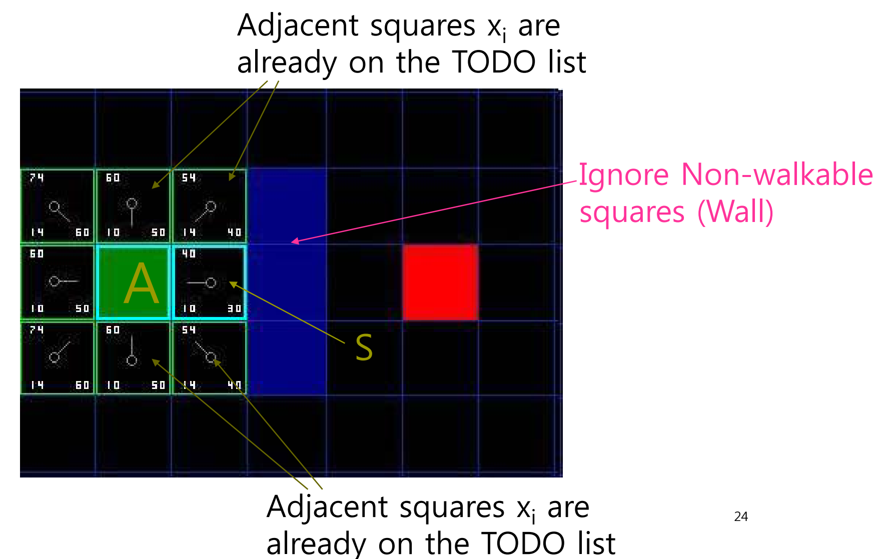
This square has lowest F=40. Call this the selected square, S.



Select S, Remove it from TODO List, Add it to DONE List

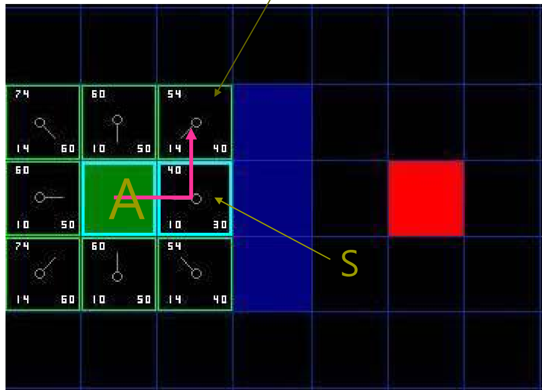


Check adjacent squares from S, and add to TODO List unless they are already on the list.



So look at each of the Xi that are already on the TODO list...

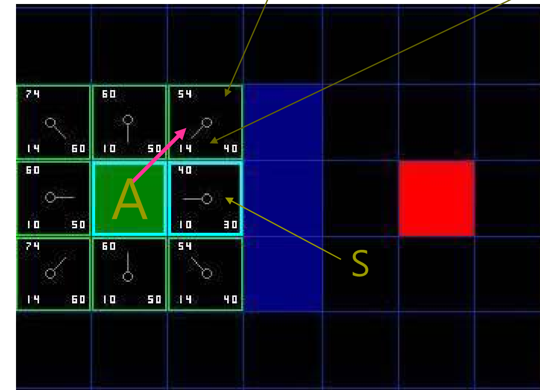
Calculate G for path S to here (G=10+10=20)



...and see if the current path (A→S→Xi) is better than the previous ones (A→Xi)

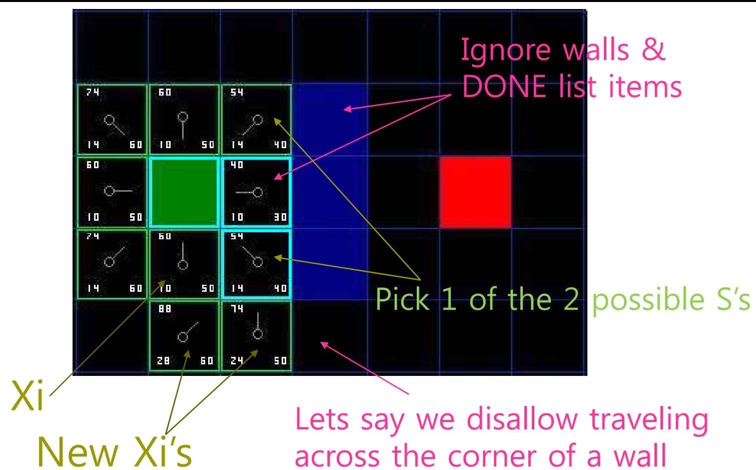
G for path S to here (G=10+10=20)

Compare against previous path that was already on the TODO list: (G=14)

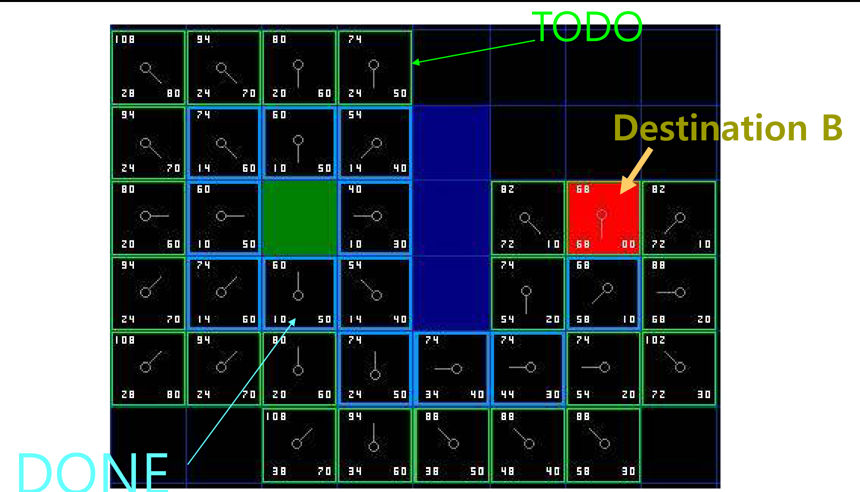


Answer for this case is: A→S→Xi is more costly than A→Xi, so leave the TODO list alone- ie ignore the path A→S→Xi but keep A→Xi on the TODO list so that you can continue searching from there later.

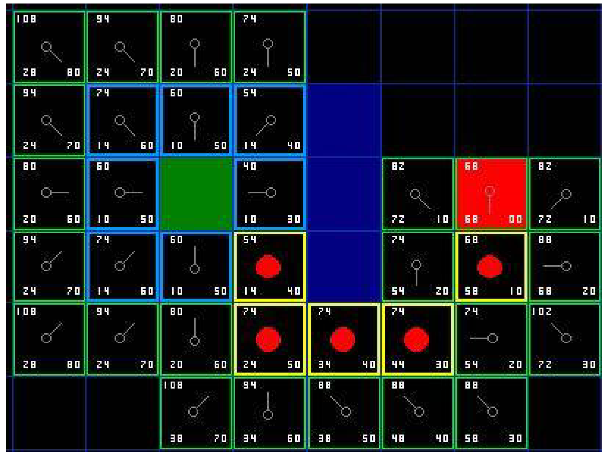
Now repeat the algorithm: Select node (S) with the smallest F in the TODO List, add to DONE List, and examine its adjacent squares (Xi). Add new Xi if not on TODO list.



Repeat until Destination B is reached.



Compute the final path by following the parents from B.

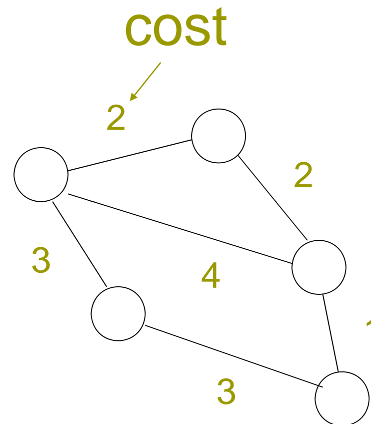


The Whole A* Algorithm Review at your Leisure

1. Add the starting square to the TODO list.
2. Repeat the following:
   a. Look for the lowest F cost square on the TODO list. We refer to this as the current square.
   b. Switch it to the DONE list.
   c. For each of the 8 squares adjacent to this current square …
      - If it is not walkable or if it is on the DONE list, ignore it.Otherwise do the following.
      - If it isn't on the TODO list, add it to the TODO list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
      - If it is on the TODO list already, check to see if this path to that square is better,using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your TODO list sorted by F score,you may need to resort the list to account for the change.
   d. Stop when you:
      - Add the target square to the TODO list, in which case the path has been found, or
      - Fail to find the target square, and the TODO list is empty. In this case, there is no path.
3. Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Note: Maps need not necessarily be Gridded.

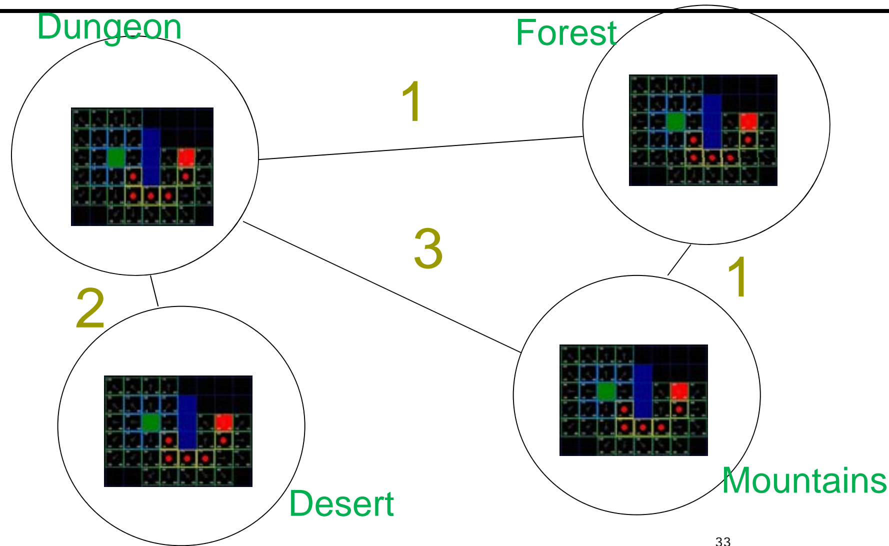
- They may consist of waypoints (e.g. between rooms in a large dungeon)
- Navigational Maps – convex polygonal representation of terrain.
- Each convex polygon can be considered a node in A*.



A* Mods

- Maintaining the TODO List
  - Best to keep it sorted if search space is expected to be large to minimize having to traverse the whole todo list looking for the smallest F.
  - A linked list that you update by inserting and removing from the list is an easy way.
- What if there are mobile units that you also have to navigate around? (like other tanks)
  - Use different code to navigate around the mobile unit (like choose any trajectory that is clear) then compute a whole new path to the final destination.
- Variable Terrain Costs
  - G can take into account cost of walking through swamps etc. or through areas where the AI has lost many troops. AI can keep track of an **Influence Map** that maps out areas where they have lost troops so that they can avoid traversing it by increasing cost in G calculation.
- Handling Large Terrains
  - Hierarchies of maps – Tiered A* Pathfinding
  - Pre-computed paths

Tiered A* Pathfinding



Pre-Computed Paths

	Dungeon1	Dungeon2	Dungeon3
Dungeon1	x	Lowest cost path from D1→D2	Lowest cost path from D1→3
Dungeon2	Reverse Path maybe symmetric, maybe not	x	Lowest cost path from D2→D3
Dungeon3			x

Complementary Navigational Strategies

- Digital scent that fade away with time
  - When user walks about the space, they leave a 3D vector path whose tail segments fades with time
  - NPCs can determine if they are close enough to a node of the tail to want to follow it
  - Good if the user runs far away and NPC needs a way to find the player without having to go through A*
- Prevent NPCs from falling to their death
  - NPCs are essentially agents. Agents have sensors that they can use to detect environment around them- like walls, floors etc
  - Can also place "attractors" and "repulsors" in the environment to attract or repel agents. E.g. can be used to avoid death by lava
- Pre-defined patrol routes
  - When agent is in idle-mode, it can follow a pre-defined patrol path

Possible Agent Navigation AI

- If idle, perform 1 of N pre-defined patrol routes
- If you are beyond a certain distance from AI turn off AI's proximity sensors
- With proximity sensors:
  - If detect you/enemy, jump to "attack" state and move toward you
  - If you fire on it, jump to "evade" state. Agent may move at an angle perpendicular to your weapons fire
  - If you are beyond a certain range, where navigation may become tricky, follow user's scent.
  - If agent loses scent but still wants to pursue, cheat by computing A* path to your location
  - Else resume patrol routes- problem is- how do you get back to the old patrol routes??? Agent can keep its own scent so it can follow it back to previous patrol route
- In all cases environmental sensors should always be on to make sure agent does not accidentally walk on lava or fall into an abyss or run through other agents
- Play games in 'cheat mode' to observe how agents behave in a "real" game.

Teaching the Computer to Aim

- Dead reckoning
  - predicting the position of an entity (often the game player) at a given moment in time based on the entity's current position, velocity and acceleration
  - By dead reckoning, AI can determine where to shoot to increase its odds of a hit
  - Extremely important in First Person Shooters, 3D dog fight games
  - But remember to also randomize the aim slightly so that the computer doesn't look like it's a perfect marksman

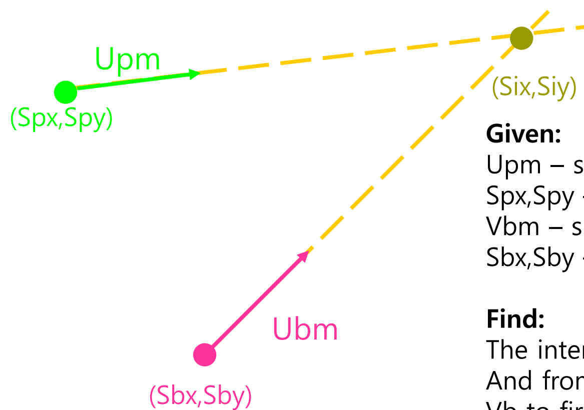
Calculating Dead Reckoning

- 2 equations of motion you should know from physics :
  - $v = v_0 + at$
  - $\mathbf{x}(t) = \mathbf{x}_0 + vt$

$$\mathbf{x}(t) = \int_0^t (v_0 + at)dt = \mathbf{x}_0 + v_0 t + \frac{1}{2} a\, t^2$$

- 2 methods for calculating dead reckoning
  - The precise method
  - The approximate method

Precise Method (Assume a 2D case)



Given:
Upm – speed of player
Spx,Spy – position of player
Vbm – speed of AI's bullet
Sbx,Sby – position of AI's bullet

Find:
The intersection (Six,Siy)
And from that determine the vector Vb to fire the bullet.

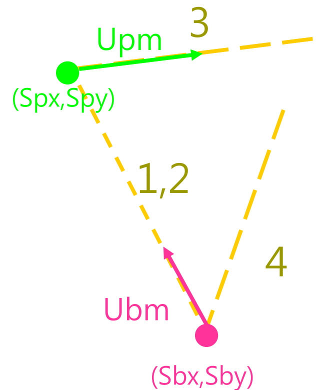
Note: User and bullet may also have an acceleration- but for simplicity we will assume the acceleration is zero.

Precise Method (Assume a 2D case)

- Note that Ubm (ie speed of the bullet) is a magnitude
- The velocity of the bullet has both an x and y component. This is what determines the direction to fire the bullet.
- Ubm = sqrt(Vbx^2+Vby^2)
- Basically you solve for t (the time of the intersection) at (Six,Siy).
- In practice this is not really used or necessary because players tend to be erratic. Instead and approximation is used.

Approximate Method

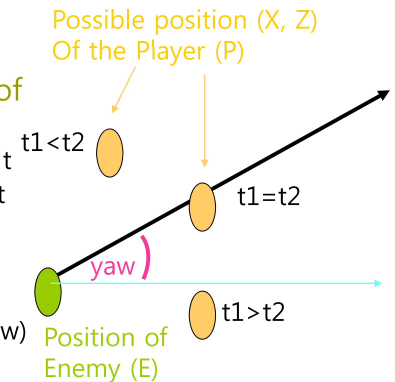
Given:
Upm – speed of player
Spx,Spy – position of player
Ubm – speed of AI's bullet
Sbx,Sby – position of AI's bullet



1. Calculate distance between player and AI's bullet.
2. Use $s=ut+(at^2)/2$ to solve for time (t) it would take for the bullet to travel to the player given the speed of the bullet (ie Upm).
3. Use Upx, Upy, Spx, Spy, t, and same equation to calculate where the player might be in time t.
4. Fire the bullet at that location.

Normally you need the enemy to turn to face the player before it is allowed to shoot.

- Use 2D Hemiplane Test to test which side Player is on
- Given the parametric equation of line:
  - P.x= position of E.x + cos(E.yaw) * t
  - P.z= position of E.z + sin(E.yaw) * t
- Solving for t in both cases:
  - t1= (P.x - position of E.x) / cos(E.yaw)
  - t2= (P.z - position of E.z) / sin(E.yaw)
- If t1=t2 then player is dead-ahead of enemy
- If t1 > t2 then player is on one side of the line
- If t1 < t2 then player is on the other side



But if you really want the exact ANGLE you can calculate it as follows

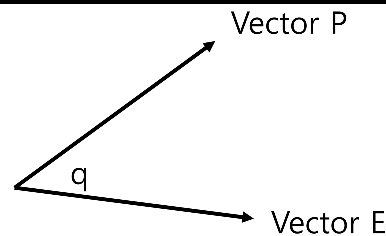
Magnitude

Dot product

$$E \cdot P = \|E\|\|P\|\cos\theta$$

$$\theta = \text{acos}\left(\frac{E \cdot P}{\|E\|\|P\|}\right)$$

$$\theta = \text{acos}(E \cdot P), \textit{where } E, P \textit{ are unit vectors}$$



Note, this does not tell you if an enemy is on your left or on your right- Hence, use previous slide to determine that.

Scripting

- Scripting specifies game data or logic outside of the game's source language, like Python, Lua, XML, Unreal Script, GameMonkey, AndelScript, etc
- Scripting influence spectrum
  - Level 0: Everything hardcoded
  - Level 1: Data in files specify states/locations
  - Level 2: Scripted cut-scenes (non-interactive)
  - Level 3: Lightweight logic, like trigger system
  - Level 4: Heavy logic in scripts
  - Level 5: Everything coded in scripts

References

- A* Pathfinding for Beginners - Patrick Lester
  http://www.policyalmanac.org/games/aStarTutorial.htm
- AI Game Programming Wisdom – Steve Rabin, Charles River Media.
- Programming Game AI by Example – Mat Buckland