Game AI Overview

Introduction

- History
- Overview / Categorize
- Agent Based Modeling
  - Sense-> Think->Act
- FSM in biological simulation (separate slides)
  - Hybrid Controllers
  - Simple Perceptual Schemas
- Discussion: Examples
- Resources (Homework, read)

What is Artificial Intelligence

- The term Artificial Intelligence (AI) was coined by John McCarthy in 1956
  - "The science and engineering of making intelligent machines."
- AI Origin, even than that (of-course)!
  - Greek Mythology:
    - Talos of Crete (Giant Bronze Man)
    - Galatea (Ivory Statue)
  - Fiction: Robot – 1921 Karel Patek
    - Asimov, Three laws of robotics
    - Hal – Space Odyssey

AI in Games

- Game AI less complicated than AI taught in machine learning classes or robotics
  - Self awareness
  - World is more limited
  - Physics is more limited
  - Less constraints, 'less intelligent'
- More 'artificial' than 'intelligent' (Donald Kehoe)

AI in Game

- Pong
  - **Predictive Logic**: how the computer moves paddle
    - Predicts ball location then moves paddle there
- Pacman
  - **Rule Based** (hard coded) ghosts
    - Always turn left
    - Always turns right
    - Random
    - Turn towards player

Scripted AI

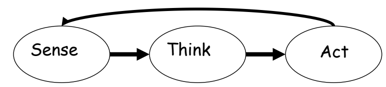
- Enemy units in the game are designed to follow a scripted pattern.
- Either move back and forth in a given location or attack a player **if nearby** (perception)
- Became a staple technique for AI design.

More Complex and Traditional AI

- Behavior Models
  - Agent Model (Focus)

Game Agents

- Game Agents, Examples:
  - Enemy
  - Ally
  - Neutral
- Loops through : Sense-Think-Act Cycle



Sensing

- How the agent perceives its environment
  - Simple check the position of the player entity
  - Identify covers, paths, area of conflict
  - Hearing, sight, smell, touch (pain) …
    - Sight (limited)
      - Ray tracing

Thinking

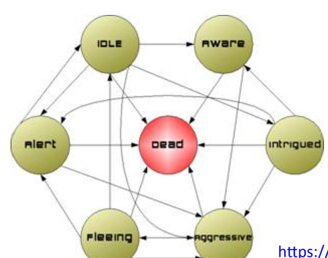
- **Decision making**, deciding what it needs to do as a result of what it senses (and possible, what 'state;' it is in) Coming UP!
- **Planning – more complex thinking.**
  - **Path planning**
- **Range: Reactive to Deliberative**

Acting

- After thinking Actuate the Action!

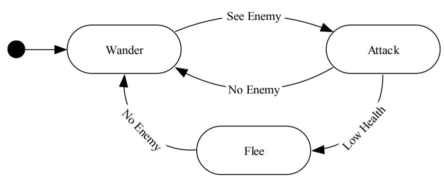
More Complex Agent

- Behavior depends on the state they are in
- Representation: Finite State Machine



https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1

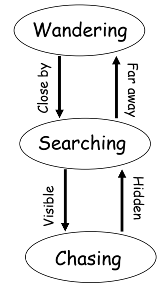
Finite State Machine



- Abstract model of computation
- Formally:
  - Set of states
  - A starting state
  - An input vocabulary
  - A transition function that maps inputs and the current state to a next state

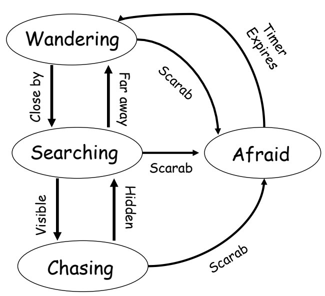
Egyptian Tomb Finite state Machine

- Mummies! Behavior
  - Spend all of eternity w*andering* in tomb
  - When player is close, ***search***
  - When see player, ***chase***
- Make separate states
  - Define behavior in each state
    - Wander – move slowly, randomly
    - Search – move faster, in lines
    - Chasing – direct to player
- Define transitions
  - Close is 100 meters (smell/sense)
  - Visible is line of sight



Can Extend FSM easily

- Ex: Add magical scarab (amulet)
- When player gets scarab, Mummy is afraid. Runs.
- Behavior
  - Move away from player fast
- Transition
  - When player gets scarab
  - When timer expires
- Can have sub-states
  - Same transitions, but different actions
    - i.e.,- range attack versus melee attack



How to Implement

- Hard Coded
  - Switch Statement

Finite-State Machine: Hardcoded FSM

```
void Step(int *state) { // call by reference since state can change
    switch(state) {

        case 0:  // Wander
            Wander();
            if( SeeEnemy() )    { *state = 1; }
            break;

        case 1:  // Attack
            Attack();
            if( LowOnHealth() ) { *state = 2; }
            if( NoEnemy() )     { *state = 0; }
            break;

        case 2:  // Flee
            Flee();
            if( NoEnemy() )     { *state = 0; }
            break;
    }
}
```

Finite-State Machine: Object Oriented withPattern Matching *parameters*

```
void AgentFSM
{
   State( STATE_Wander )
       Wander();
       if( SeeEnemy() ) { setState( STATE_Attack ) }

   State( STATE_ATTACK )
       Attack();
       if ( LowOnHealth ) { setState( STATE_Flee ) }

   .
   .
   .

 }
```

- AD Hoc Code
- Inefficient
  - Check variables frequently

Better

- Object Oriented
- Transitions are events

Embellishments

- Adaptive AI
  - Memory
- Prediction
- Path Planning, Tomorrow

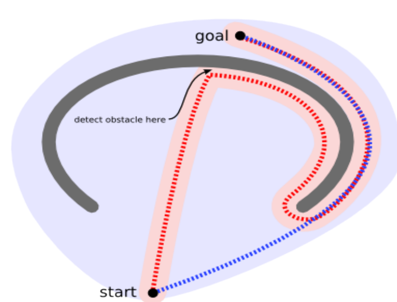
Resources

- https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1 (there are 4 parts, read the first 3)
- http://www.policyalmanac.org/games/aStarTutorial.htm (you will implement this visualization as project 3)
- http://www-cs-students.stanford.edu/~amitp/gameprog.html (great resources for game AI)

Path Planning

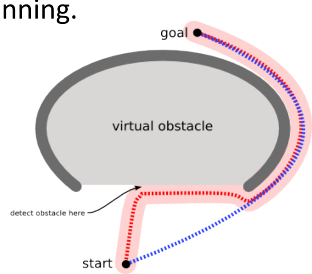
- Problem: How to navigate from point A to point B in real time. Possible a 3D terrain.
  - We will start with a 2D terrain.
- What about if we ignore the problem:

No Path Planning bad Sensors



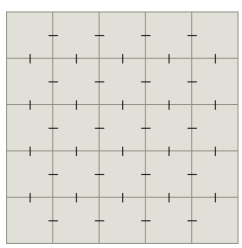
With Better Sensors (Red)

- Blue Planning.



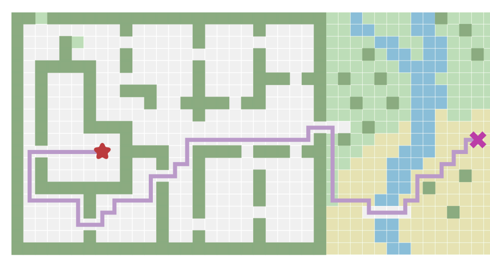
- Watch AI Navigation Bloopers:
  - http://www.youtube.com/watch?v=lw9G-8gL5o0

Environment Assumptions



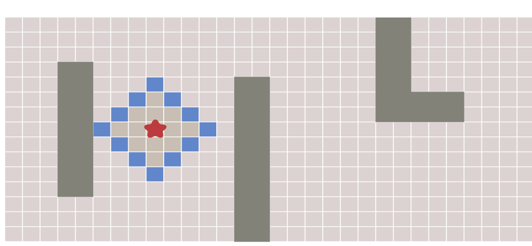
- 2D Grid

Problem Statement



- Point A (star) to Point B (x) : Shortest amount of steps or fastest time

Explore the Environment



- Frontier Expands
- Stops at walls

http://www.redblobgames.com/pathfinding/a-star/introduction.html

Common Theme: Frontier Implementation

- Pick and remove a location from frontier
- Mark location as “done processing”
- Expand my looking at its unprocessed neighbors and add to frontier

```
frontier = Queue()
frontier.put(start)
visited = {}
visited[start] = True

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in visited:
         frontier.put(next)
         visited[next] = True
```

Shortest Path: Breath First

- We got the visiting part, now how do we find the shortest path?
  - Solution: Keep track :
    1. where we came from, and later compute
    2. the distance traveled so far

```
frontier = Queue()
frontier.put(start)
visited = {}
visited[start] = True

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in visited:
         frontier.put(next)
         visited[next] = True
```

```
frontier = Queue()
frontier.put(start)
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```

Measure path links

- Start at Goal and traverse where it 'came from'
  - Shortest path

Embellishments: Make if more efficient

- All Paths from one location **to all others**
  - **Early exit: Stop expanding once frontier covers goal**

- http://www.redblobgames.com/pathfinding/a-star/introduction.html

Movement cost not enough

- Some movements may be more expensive than other to move through
  - Use a new heuristics
  - Add to frontier if cost is less.

- We: Board
- Th: Board. Sketch out the algorithm.

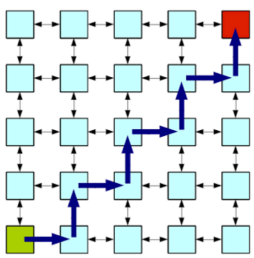
Summary from Board

- A Star favor neighbors with smallest F value.
  - F = H + G
- Breath First Search
  - Explore all neighbors, typically using a simple queue that explores neighbors first in first out (FIFO).
- Best First Search: H
  - Favor neighbors that have shortest distance to goal.
- Dijskstra: G
  - Favor neighbors that are closest to starting point (smallest G).

```
ef a_star_search(graph, start, goal):
   openList = PriorityQueue() // A Star / Dijskastra / Best First
   openList.put(start, 0)
   came_from = {}
   cost_so_far = {}
   came_from[start] = None
   cost_so_far[start] = 0

   while not openList.empty():
       // process node with low F cost (dequeue priority queue)
       current = openList.get()

       // dropped from open list, added it to closed list
       // is the a goal node in closed list?
       if current == goal: break

       // compute neighbors new values.
       for next in graph.neighbors(current):
           new_cost = cost_so_far[current] + graph.cost(current, next)
           if next not in cost_so_far or new_cost < cost_so_far[next]:
               cost_so_far[next] = new_cost
               priority = new_cost + heuristic(goal, next)   // Dijsktra use new cost
               openList.put(next, priority)
               // each node needs to points to its parent.
               came_from[next] = current

   return came_from, cost_so_far
```

Revisit Representing of grids as graphs

- Grid to Node Example



Pathfinding on a grid of nodes from point A (green) to point B (red).

- Dijkstra node on board.

Hackathon tomorrow.

- Hackathon tomorrow will be doing node based algorithms on ‘paper’ but you will need to covert it to digital text.
  - Best First, Breath First, Dijkstra, A*
- You will also draw a FSM of some game entity, in the same vain as the mummy FSM.