Game Intelligence

Perception – detection
(objects in world)

Knowledge (memory)
genetic – "at creation"
learned, adaption

Emotions (motivation, bias)
fearful, aggressive, reckless
"likes/dislikes", phobias

Planning
goal(s) directed
React: S → R
Reinforcement: habit, strength
Concepts:

Communication
language: hear/understand, speak
gesture: facial, hand

7 examples of AI in games
Game Intelligence

GI (game – artificial intelligence) usually relates to controlling an NPC.

Navigation: chase/evade, pattern movement, flocking, path finding,


Game state \(\xrightarrow{\text{updates}}\) Game Intelligence Controller / Simulator \(\xrightarrow{\text{Update}(...)}\) actions

Renderer \(\xrightarrow{\text{information}}\) Game Intelligence Controller / Simulator \(\xrightarrow{\text{draw}}\) Controller
Controllers

Expert System – brute force controller knows everything about the "game".
? size of KB (knowledge base)

Searching controllers: explicit goals -- search for action

Reactive controllers: State $\Rightarrow$ Action
? as game becomes complex state determination becomes difficult
? scalability

Remembering controllers: memory changes game state
next state $\Leftarrow$ past states
? probabilistic decisions

Learning controllers:
? machine learning
? reinforcement (habit, strength)
Simulators

Actions: Controller selects actions, simulator applies actions to game state

representation – duration, strength, visual effects,

Animations – actions from non character objects? walk / run …

physics – gravity, velocity, inertia, friction, steering forces ...

Time

Fixed Step – sampling continuous time
simulator polls controllers at fixed intervals.

(controller poll order continually shuffled / randomized)
probability of performing action-this-step

Next Event – eventList ordered by timeOfEvent
simulator selects next schedule event

actions "schedule" future events
startAction, endAction events
FSM

Machine has predefined states and transitions between states.

current state \( \rightarrow \) next state

Controller is
- switch stmt
- nested if-elses
- production system graph (w/ many states)

FSM can be deterministic or probabilistic in state determination.

AI Ants (Bourg, OReilly, 2004), see AntFSM.cs example
Delgate implementation

C# Delegate ADT wrapper for a method.

```csharp
public delegate void ActionDelegate (string actionStr);
```

Production class encapsulates boolean condition, probability and ActionDelegate (action)

FSM class defines methods to wrap in ActionDelegates

```csharp
public static void forageAction (string actionStr) {
    Console.Write(actionStr);
}
```

```csharp
public ActionDelegate forage = new ActionDelegate(forageAction);
```

FSNode setProductions() adds productions to fsm

```csharp
production.Add(new Production( 0.50, 
    fsm.getState(FSMTYPE.FORAGE), fsm.forage, "foraging" ));
```
Consider a simple game of tag – 2 NPCs "Finder" and "Hider"
NPCs can't see through walls, can see 6 * BoundingSphere, "FOV 180"
FrontWall, RightWall, LeftWall, SeeFinder, SeeHider
NPCs actions: step (forward 10 pixels), turn right (40°), turn left (40°),
turnToHider + step, turnFromFinder + step (reverse direction)
Reacting

Perceive $\rightarrow$ State $\rightarrow$ Action

```
gameState = {perceptions}
perceptions = {wallFront, wallLeft, wallRight, seeHider}
```

Perception can set game state

Perception can have uncertainty (noise – probabilistic results)

Simplest controller – Action does not modify game state (no adaption or learning – no simulator)

Deterministic reactive controller (predictable $\rightarrow$ poor game play)

collection of state-action pairs (production rules)
search for current state – perform action (usually few states)

Probabilistic reactive controller (unpredictable $\rightarrow$ stupid game play)

random controller: all states equally likely

$$\text{Probability of Action}_i = \frac{1}{\text{NumberOfActions}}$$

state is the probability of the action
Production System (PS) for Finder

`setGameState()` sets values, wall detection w/in ‘n’ steps

GameStates (T || F) : FrontWall (FW), RightWall (RW), LeftWall (LW)
- SeeHider  can’t see thru walls

Actions: step (10), right (40°), left (40°), `turnToHider`

Production ← if state do action

“Wall” productions, evaluate in sequence (do first selected action) for Finder

<table>
<thead>
<tr>
<th>State (True)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeeHider</td>
<td><code>turnToHider</code></td>
</tr>
<tr>
<td>FW, LW</td>
<td>right</td>
</tr>
<tr>
<td>FW, RW</td>
<td>left</td>
</tr>
<tr>
<td>FW</td>
<td>left / 2</td>
</tr>
<tr>
<td>true</td>
<td>step</td>
</tr>
</tbody>
</table>

```java
while (!done) {
    updateGameState();
    if (state_0) action_0
    else if (state_i) action_i
    ...
    else action_n  // (state_n)
}
```

not the “smartest” production system…

Hider’s complement,
- SeeFinder   turn 180°
- FW          right / 2  half turn
### Probabilistic Productions for Finder

<table>
<thead>
<tr>
<th>Prob</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>SeeHider</td>
<td>turnToHider</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, LW</td>
<td>right</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, RW</td>
<td>left</td>
</tr>
<tr>
<td>0.50</td>
<td>FW</td>
<td>left / 2</td>
</tr>
<tr>
<td>1.0</td>
<td>FW</td>
<td>right / 2</td>
</tr>
</tbody>
</table>
| 0.01 | true (explore) | left | Hider's “Explore” productions similar:
| 0.01 | true (explore) | right  |
| 1.0  | true (explore) | step |

```
while (! done) {
    setGameState()
    if ( P_i > Random.Sample() && state_i ) action_i
    ...
}
```
Hider evolves to survive ....

<table>
<thead>
<tr>
<th>Prob</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>SeeFinder</td>
<td>turnAwayFrom</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, LW</td>
<td>right</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, RW</td>
<td>left</td>
</tr>
<tr>
<td>0.50</td>
<td>FW</td>
<td>left / 2</td>
</tr>
<tr>
<td>1.0</td>
<td>FW</td>
<td>right / 2</td>
</tr>
<tr>
<td>0.01</td>
<td>true, see behind</td>
<td>right * 3</td>
</tr>
<tr>
<td>0.01</td>
<td>true, see behind</td>
<td>left * 3</td>
</tr>
<tr>
<td>0.01</td>
<td>true, explore</td>
<td>left</td>
</tr>
<tr>
<td>0.01</td>
<td>true, explore</td>
<td>right</td>
</tr>
<tr>
<td>1.0</td>
<td>true, explore</td>
<td>step</td>
</tr>
</tbody>
</table>

“Look behind” productions
Finder has an advantage, it continues to see Hider to pursue, but Hider can't see Finder when it evades.

Hider could evolve.
Hider could try to see behind.
Hider could see farther than Finder.
Hider could have an eye to look behind.

Which is better?
Experimental Evaluation

Three NPC conditions (Walls, Explore, Look Behind) were run 10 times and the number of Updates for each run was measured. There was an overall significant difference in number of updates for the 3 conditions; Anova $F(2, 27) = 4.366, p < 0.023$.

Walls and Explore not different.
Wall Vs Explore
\[ p < 0.610 \]

Look Behind > Walls and Explore.
Wall Vs Look Behind
\[ p < 0.010 \]
Explore Vs Look Behind
\[ p < 0.032 \]
Decision Trees

Design tool to create productions for implementing FSM
Non – leaf decisions in tree partition exclusive states.
Tree traversal is a path to an action
Leafs are actions.
PSs and User NPC scripting

Production systems work well w/ end-user simulation scripting.

Consider a class that encapsulates a production.

- state is a reference to Agent's collection of known boolean variables
- action is “pointer to method”

  C# ADT delegate encapsulates methods
  
  eval() returns true if all members of state are true, else it returns false.
  exec() calls action delegate.

List<Production> script
  contains productions that are created by user evaluated every Update during simulation.

```
Update {
  perceive() // sets the Agent's states
  for (Production p in Productions)
    if (p.eval()) { exec(); break; }
```

Production
- state : List<bool>
- action : delegate
  + eval() : bool
  + exec() : void
Strategy building dialog

Non-programmers (players) can "script" NPC behavior using GUI based selection of state variables that are added to state and action.

User scripting can write “script files” read in when simulation begins or can be part of simulation’s configuration / initialization sequence.
Pac-Man detection ➔ action

Pac-Man (PM) eats 244 dots in the maze, to proceed to the next round. Play game.

Four ghost try to capture and eat PM.

There are four large, flashing energizers dots. When PM eats them he can eat the ghosts.

Ghost states:

- Chase: ghosts chase PM
- Blinky (red): chases from behind
- Pinky: blocks by getting in front
- Inky (blue): unpredictable
- Clyde (orange): unpredictable
- Scatter: ghost briefly return to their corners
- Frightened PM: is energized, ghosts random move

Ghosts can only turn to side path or move forward in chase mode. Direction can reverse when state changes.
Ghost path finding

Four ghosts use the same pathfinding logic.

Conceptually, the game screen is a regular grid of 28 by 36 tiles. Tiles are either legal space or dead space.

Legal space is shown as the gray-colored tiles.

Ghosts are on and travel between legal space tiles with their current direction.

Distances are measured in tiles.

Collisions occur when PM and ghost occupy the same tile at the same time.

In chase or scatter, ghosts have target tiles. Chase ghost's target depends on PM tile, scatter ghost's target is home corner.
Look ahead

Ghost look ahead one move. When it moves to a tile it examines the tile ahead and determines how it move.

Red ghost moving right to left moves to tile A. Looks ahead to tile B and can only move forward (left exit)

B has 4 exits (right, left, up, down)
Up and down are dead tiles, right would reverse the ghost

Consider the intersection diagram. At A red ghost has Target (green) and three exits.

Ghost determines distance of three exit tiles from Target and chooses to move right.

With ties exits are preferred: up, left, down, right.
Ghost chase targets

Blinky uses PM's tile as his target.

Pinky targets an offset four tiles away from PM in the direction PM is moving.

Inky has most complex targeting scheme. He uses PM's current tile/orientation and Blinky's current tile to calculate his final target.

Establish an intermediate offset two tiles in front of PM. Draw a vector from the center of the Blinky's current tile to the center of the offset tile. Double the vector length by extending it out just as far again beyond the offset tile.
Clyde's targeting logic is based on his proximity to PM. (represented by the green target tile).

First calculate the Euclidean distance between current tile and PM's tile.

If distance $\Rightarrow 8$ tiles target PM like Blinky. else use scatter mode target (move away from PM).

Clyde's behavior can oscillate.

Simple rules of detection $\Rightarrow$ state $\Rightarrow$ action generate interesting play.

From J. Pittman's *The Pac-Man Dossier.*
Remembering

(Perceive & Memory) \(\Rightarrow\) State \(\Rightarrow\) Action

\[
\text{Game state} = \{ \text{perceptions} \} + \{ \text{memories} \}
\]

\[
\text{memories} = \{ \text{fn}(\text{moves}) \} \]

Action can change the game state – change values of a remembered attribute.

Memory states (values) valid? How often are values validated.
- forgetting – probability of recall decreases w/ time?

Memory state is a strategy (explore, hide), an emotion (rage, fear), ...

Memory state value transitions – FSMs
- gameState determines allowable changes of current memory value.

Production rules can change memory values.

Probablistic "rules" can change memory values.
Memory && Forgetting

Memory of perceptions can be stored w/in NPC or if using navigation graph in the graph.

Consider a TagWorld with a "sense" memory for past positions stored in a regular navigation node. NPCs see walls and turn, but only "sense" other's with memory (to evade or pursue).

- Hider's memory (blue nodes ± 2) < Finder's memory (red nodes ± 3)
- Hider's memory persistence (2) < Finder's memory persistence (6)
- Hider's detection of Finder (± 3) > Finder's detection of Hider (± 2)
  (White nodes have sense memories for both)

MemoryGraph's Update(...)

- Hider leaves less memory traces than Finder,
- Finder's memory lasts longer
  - adds 2 to all Hider ± 2 surrounding memory nodes
  - adds 6 to all Finder ± 3 surrounding memory nodes
- Memory trace degrades with time
  - All memory node's w/ trace strength > 0 are decremented by 1
### Finder's productions:

<table>
<thead>
<tr>
<th>Prob</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>FW, LW</td>
<td>right</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, RW</td>
<td>left</td>
</tr>
<tr>
<td>0.50</td>
<td>FW</td>
<td>left / 2</td>
</tr>
<tr>
<td>1.0</td>
<td>FW</td>
<td>right / 2</td>
</tr>
<tr>
<td>1.0</td>
<td>Sense Hider</td>
<td>turnToNode</td>
</tr>
<tr>
<td>1.0</td>
<td>true (<em>explore</em>)</td>
<td>step</td>
</tr>
</tbody>
</table>

**Priority is Walls before pursuit of Hider**

If Hider sensed: `turnToNode` rotates to the node with the maximum sense value within sense range.

Hider sense, Finder turn away.
Learning

Simulated learning – all NPC knowledge is known "in game" but revealed (used) as game time progresses – NPC appears to learn.

Learning    NPC decides action based on state, action has a reward and a utility.

Utility = immediate reward (+, 0, -) + long term reward (heuristic)

utility evaluated at time of action
utility of action can be modified by later actions (steps)

S-R learning (simple learning) – Utility sum of past
State whose action has highest reward are chosen next

Learning could:
re-order a Production System
change probability (or recalling) memory states
change costs (utility) of way points in path finding

GDC 2017 tutorial, Deep Learning for Games
<table>
<thead>
<tr>
<th>Utility</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>SeeFinder</td>
<td>turnAwayFrom</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, LW</td>
<td>right</td>
</tr>
<tr>
<td>1.0</td>
<td>FW, RW</td>
<td>left</td>
</tr>
<tr>
<td>0.50</td>
<td>FW</td>
<td>left / 2</td>
</tr>
<tr>
<td>1.0</td>
<td>FW</td>
<td>right / 2</td>
</tr>
<tr>
<td>0.5 – 0.1</td>
<td>true, see behind</td>
<td>right * 3, step</td>
</tr>
<tr>
<td>0.5 – 0.1</td>
<td>true, see behind</td>
<td>left * 3, step</td>
</tr>
<tr>
<td>0.05</td>
<td>true, explore</td>
<td>left</td>
</tr>
<tr>
<td>0.05</td>
<td>true, explore</td>
<td>right</td>
</tr>
<tr>
<td>1</td>
<td>True, explore</td>
<td>step</td>
</tr>
</tbody>
</table>

Hider's productions constant and variable utility (has a range, initially at minimum)

Constant utility responds to necessary decisions

W/ no constant state, all variable utility values are evaluated and highest is taken (tie?)

Utility of step \(i\) evaluated at step \(i+1\)

Variable utility evaluation

if on step \(i+1\) if Finder is seen, step’s variable utility is set to 0.5
else last step’s variable utility is decremented (- 0.1).
Memory & Recall

David Lu's 2013 CSUN MS thesis conducted an experiment comparing Memory Structure and Percent Recall for a NPC.

Task: A Unity3D "flat terrain" of a 588 by 588 with 11 by 11 "nodes" evenly spaced (78 units between nodes). An "ant" is randomly placed on the terrain with 100 units of food. There are 16 piles of 100 food units randomly placed on the 121 nodes. The ant begins a random walk exploration of the terrain. When it encounters food and it is not hungry is remembers where the food is (not how much food). For each second the ant moves it uses 1 of its food units. When its food level goes below 50 it becomes hungry and uses its memory to select a food node to go to. When hungry and at a node with food it will either consume to reach its 100 food units limit or until there is no more food. If its food unit count is above 50 it is no longer hungry and begins exploring. This continues until the ant dies.

How should the ant’s memory be structured?
Experimental Design

Experimental Variables (4 by 3 or 12 experimental conditions)

Memory Structure: random, FIFO, LIFO, closest

Percent Recall: 33%, 66%, 100%

2 Dependent Measurements

distance traveled (Unity3D units)
food remaining

Memory is an ArrayList. Only 6 items in memory can be accessed.

random select 6 by random
FIFO the 6 oldest
LIFO the most recent 6

closest the six closest to ant

After the "working memory" is determined, retrieval of memory is attempted. With 66% recall items are tested until one has retrieved.

With 66% retrieval a memory should gotten w/in 3 tries, but may never be retrieved.

With 100% the first attempt is a retrieval.
Results

230 "samples" of experimental condition were obtained (10 to 33)

Memory Structure
Distance travel was significantly different ($F = 7.299, p < .001$).
Food eaten was significantly different ($F = 10.612, p < .001$).
random and closest lived longer (distance, food), FIFO died first

Percent Recall
Food eaten was significantly different ($F = 3.071, p < 0.05$).
66% recall had more food eaten

Distance Travelled
Food Eaten
Food Eaten

33% 66% 100%
Simulations = Controller + Events + Actions

Simulation systems more applicable to graphical scientific visualization than games. Civilization, Sims, are “simulation” games.

System has a sense of time (now, a variable) mechanism to advance time, and a representation of events to do at a future time (an Action).

Now is the current “simulation time” during simulation

Fixed step interval. Update(...) and GameTime is a fixed step simulation.

\[ \text{Next time} \leftarrow \text{now} + \text{updateTime} \]

Next event scheduling advances now to the time of the next scheduled event.

\[ \text{now} \leftarrow \text{event.scheduledTime} \]

A simulation has a controller to hold and execute scheduled events. Schedule events have an action – a delegate method to execute.
AgentSim an ant simulation

AgentSim

Class

- Methods
  - AgentSim
  - Main

SimDelegates

Class

- Fields
  - dead
  - forage
  - goHome
  - thirsty

- Methods
  - deadAction
  - forageAction
  - goHomeAction
  - thirstyAction

ActionDelegate

Delegate

agent

SimEvent

Class

- Fields
  - action
  - agent
  - scheduledTime

- Properties
  - CompareTo
  - SimAgent

- Methods
  - ScheduledTime
  - doAction

Simulation

Static Class

- Fields
  - deadCount
  - events
  - minute
  - now
  - rand
  - second

- Properties
  - DeadCount
  - Events
  - Now

- Methods
  - eval
  - initialize
  - sample
  - sampleNow
  - schedule

AgentSim console example distro
**Events**

```csharp
public class SimEvent : IComparable<SimEvent> {
    long scheduledTime;  // when to execute
    Agent agent;         // object to execute action
    SimDelegates.ActionDelegate action;  // what to execute
}

public SimEvent(Agent anAgent, long eventTime,
                 SimDelegates.ActionDelegate anAction) {...}

public int CompareTo(SimEvent other) {  // for sorting
    if (scheduledTime < other.ScheduledTime) return -1;
    else if (scheduledTime == other.ScheduledTime)
        return 0;
    else return 1;
}

public void doAction() { action(agent); }  // execution
```
Actions

Actions are methods executed when scheduled. C# implementation would be a Delegate.

```csharp
public class SimDelegates {
    public delegate void ActionDelegate (Agent agent);
    public static void forageAction (Agent agent) {
        if (Simulation.sample(0.50))
            Simulation.schedule(agent,
                Simulation.sampleNow(0, 40), forage);
        else if (Simulation.sample(0.10))
            Simulation.schedule(agent,
                Simulation.sampleNow(0, 5), dead);
        else Simulation.schedule(agent,
            Simulation.sampleNow(0, 10), goHome);
    }

    public static void goHomeAction(Agent agent) {
        ...
    }
    public static void deadAction(Agent agent) {
        ...
    }
    public static void thirstyAction(Agent agent) {
        ...
    }
```
static public ActionDelegate forage =
    new ActionDelegate(forageAction);
static public ActionDelegate goHome =
    new ActionDelegate(goHomeAction);
static public ActionDelegate dead =
    new ActionDelegate(deadAction);
static public ActionDelegate thirsty =
    new ActionDelegate(thirstyAction);
public static class Simulation {
    public static long now; // current "simulation time"
    public static List<SimEvent> events; // scheduled events
    public static Random rand; // Random number generator

    static public bool Events {
        get { if (events.Count != 0) return true;
            else return false; }
    }

    public static void initialize(int nAgents) {
        now = 0;
        long scheduleAt;
        rand = new Random();
        Agent Agent;
        events = new List<SimEvent>();
        // schedule initial events here
        for(int i = 0; i < nAgents; i++) {
            Agent = new Agent(i);
            scheduleAt = rand.Next(1 * second, 3 * minute);
            schedule(Agent, scheduleAt, SimDelegates.forage);
        }
        while (Events) eval(); // process event list
    }
public static void schedule(Agent Agent, long eventTime, SimDelegates.ActionDelegate action) {
    SimEvent simEvent = new SimEvent(Agent, now + eventTime, action);
    events.Add(simEvent);
    events.Sort(); // min to max scheduledTime
}

public static void eval() {
    SimEvent simEvent;
    while (Events) {
        simEvent = events.First<SimEvent>();
        events.RemoveAt(0);
        now = simEvent.ScheduledTime;
        simEvent.doAction();
    }
}
example

C:\> AntSim

Agent 0 will exist and forage at 7531
Agent 1 will exist and forage at 5006
...
Agent 18 will exist and forage at 233
Agent 19 will exist and forage at 2691

Now = 0, 20 Agents created with times to start foraging

time = 233 :: agent 18 : Foraging

time = 518 :: agent 18 : going home

time = 842 :: agent 14 : Foraging

time = 1792 :: agent 14 : going home
...

time = 167891351 :: agent 4 : going home

Now = 531776362, 20 Agents died