Basic Game AI

IMGD 4000

What’s the AI Part of a Game?

• Everything that isn’t graphics (sound) or networking... (says an AI professor 😊)
  – or physics (though sometimes lumped in)
  – or usually via non-player characters
  – but sometimes operates more broadly, e.g.,
    • Civilization-style games (sophisticated simulations)
    • interactive storytelling (drama control)

“Levels” of Game AI

• Basic
  – Decision-making techniques commonly used in almost all games

• Advanced
  – Used in practice, but in more sophisticated games

• Future
  – Not yet used, but explored in research

This course

• Basic game AI
  – Decision-making techniques commonly used in almost all games
    • Basic pathfinding (A*) (IMGD 3000)
    • Decision trees (this deck)
    • (Hierarchical) state machines (this deck)

• Advanced game AI
  – Used in practice, but in more sophisticated games
    • Advanced pathfinding (later deck)
    • Behavior trees in Halo 3 (maybe after that)
**Future Game AI?**

- Take IMGD 4100 in 2016
  - “AI for Interactive Media and Games”
    - Fuzzy logic
    - More goal-driven agent behavior

- Take CS 4341 “Artificial Intelligence”
  - Machine learning
  - Planning

---

**Two Fundamental Types of AI Algorithms**

- **Non-Search vs. Search**
  - **Non-Search**: amount of computation is predictable
    - e.g., decision trees, state machines
  - **Search**: upper bound depends on size of search space (often large)
    - e.g., minimax, planning
    - scary for real-time games (or need ways to “short-circuit”)
    - need to otherwise limit computation (e.g., threshold)

- Where’s the “knowledge”?
  - **Non-Search**: in the code logic (or external tables)
  - **Search**: in state evaluation and search order functions

- Which one is better? Whichever has better knowledge. ;-)
Outline

- Introduction (done)
- Decision Trees (next)
- Finite State Machines (FSM)
- Hierarchical FSM

First Basic AI Technique:

Decision Trees

See code at: https://github.com/idmillington/aicore
src/dectree.cpp and src/demos/c05-dectree

Ian Millington and John Funge. Artificial Intelligence for Games, Morgan Kaufmann, 2009. (Chapter 5)

Decision Trees

- Most basic of the basic AI techniques
- Easy to implement
- Fast execution
- Simple to understand

Deciding How to Respond to an Enemy

```
if visible? {
  if close? {
    attack;
  } else {
    if flank? {
      move;
    } else {
      attack;
    }
  }
} else {
  if audible? {
    creep;
  }
}
```

Leaves are actions
Interior nodes are decisions
Modifying Deciding How to Respond to an Enemy

```java
if visible? {
    if close? {
        attack;
    } else if flank? {
        move;
    } else {
        attack;
    }
} else if audible? {
    creep;
}
```

Alternate form. Harder to see "depth"! Modification in either form tough!

Building an O-O Decision Tree

```java
visible = new Boolean... audible = new Boolean... close = new MinMax... flank = new Boolean... attack = new Attack... move = new Move... creep = new Creep...
visible.yesNode = close visible.noNode = audible audible.yesNode = creep close.yesNode = attack close.noNode = flank flank.yesNode = move flank.noNode = attack
```

...or a graphical editor

Modifying an O-O Decision Tree

```java
visible = new Boolean... audible = new Boolean... close = new MinMax... flank = new Boolean... ?? = new Boolean... attack = new Action... move = new Action... creep = new Action...
visible.yesNode = close visible.noNode = audible audible.yesNode = creep close.yesNode = attack close.noNode = ?? flank.yesNode = move flank.noNode = attack
```

...or a graphical editor

O-O Decision Trees (Pseudo-Code)

```java
class Node {
    def decide() { return action }
}
class Decision : Node {
    def decide() { return node }
}
class MinMax : Boolean {
    def decide() { return node }
}
class Action : Node {
    def decide() { return this }
}
class Boolean {
    def decide() { return yesNode }
}
class MinMax : Boolean {
    def decide() { return node }
}
class Decision : Node {
    def decide() { return node }
}
class Action : Node {
    def decide() { return this }
}
class Boolean {
    def decide() { return yesNode }
}
```
Decision Tree Performance

- Individual node tests (getBranch) typically constant time (and fast)
- Worst case behavior depends on depth of tree
  - longest path from root to action
- Roughly “balance” tree (when possible)
  - not too deep, not too wide
  - make commonly used paths shorter
  - put most expensive decisions late

Outline

- Introduction (done)
- Decision Trees (done)
- Finite State Machines (FSM) (next)
- Hierarchical FSM

Second Basic AI Technique:
(Hierarchical) Finite State Machines

Finite State Machines
Hard-Coded Implementation

```python
class Soldier
    enum State
        ON_GUARD
        FIGHT
        RUN_AWAY
    currentState

    def update()
        if currentState == ON_GUARD
            if small enemy
                currentState = FIGHT
                start Fighting
            else if big enemy
                currentState = RUN_AWAY
                start RunningAway
            else if currentState == FIGHT
                if losing fight
                    currentState = RUN_AWAY
                    start RunningAway
                else if currentState == RUN_AWAY
                    if escaped
                        currentState = ON_GUARD
                        start Guarding
```

Hard-Coded State Machines

- Easy to write (at the start)
- Very efficient
- Notoriously hard to maintain (e.g., modify and debug)

Cleaner & More Flexible O-O Implementation

```python
class State
    def getAction()
    def getEntryAction()
    def getExitAction()
    def getTransitions()

class Transition
    def isTriggered()
    def getTargetState()
    def getAction()

class StateMachine
    states
    initialState
    currentState = initialState

    def update()
        triggeredTransition = null
        for transition in currentState.getTransitions()
            if transition.isTriggered()
                triggeredTransition = transition
                break
        if triggeredTransition != null
            actions = currentState.getExitAction()
            actions += triggeredTransition.getAction()
            actions += triggeredTargetState.getEntryAction()
            currentState = triggeredTargetState
            return actions // list of actions for transitions
        else return currentState.getAction() // action this state
```

Combining Decision Trees & State Machines (1 of 2)

- Why?
  - To avoid duplicating expensive tests in state machine. E.g., assuming "player in sight" is expensive
Combining Decision Trees & State Machines (2 of 2)

Outline

- Introduction  
  (done)
- Decision Trees  
  (done)
- Finite State Machines (FSM)  
  (done)
- Hierarchical FSM  
  (next)

Hierarchical State Machines

- Why? → Could be interruptions

E.g., robot can run out of power in any state. Needs to recharge when out of power. When charged, needs to return to previous state.

Interruptions (e.g., Recharging)

6 states needed → doubled!
Add Another Interruption (e.g., Baddies)

Hierarchical State Machine

Cross-Hierarchy Transitions
Cross-Hierarchy Transitions

resource Links

- HFSM from Millington and Funge
- FSM from IMGD 3000
  – Slides
  – Header files
  http://dragonfly.wpi.edu/include/classStateMachine.html
- UE4 Behavior Tree
  – Decision tree + HSM = Behavior tree
  – Cannot use for your state-based AI!