Basic Game AI

Technical Game Development II

Professor Charles Rich
Computer Science Department
rich@wpi.edu

Definitions?

- What is artificial intelligence (AI)?
  - subfield of computer science?
  - subfield of cognitive science?
- What is “AI for Games”? 
  - versus “academic AI”?
  - arguments about “cheating”

In games, *everything* (including the AI) is in service of the *player’s* experience (“fun”)

What’s the AI part of a game?

- Everything that isn’t graphics (sound) or networking... 😊
  - or physics (though sometimes lumped in)
  - usually via the non-player characters
  - but sometimes operates more broadly, e.g.,
    - Civilization games
    - interactive storytelling

“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
    - decision trees  
      (Today)
    - (hierarchical) state machines  
      (Today)
    - scripting  
      (Monday)
    - minimax search  
      (Tuesday)
    - basic pathfinding (A*)  
      (IMGD 3000)

- **Advanced** game AI  
  (Later...)
  - used in practice, but in more sophisticated games
    - task/behavior trees in Halo 3
    - autonomous movement, steering
    - advanced pathfinding

Future Game AI?

- Take IMGD 400X next year (B) “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

- **Search vs. Non-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
    - 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

First Basic AI Technique:

**Decision Trees**

*Reference: Millington, Section 5.2*
Decision Trees

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

Deciding how to respond to an enemy

```java
if (visible) {
    if (close) {
        attack;
    } else {
        if (flank) {
            move;
        } else {
            attack;
        }
    }
} else {
    if (audible) {
        creep;
    }
}
```
Which would you rather modify?

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

OO Decision Trees (Pseudo-Code)

(see Millington, Section 5.2.3)

```python
class Node:
    def decide():
        return this
class Action:
    def decide():
        return this
class Decision:
    def getBranch():
    def decide():
        return getBranch().decide()
class Boolean:
    def getBranch():
    if maxValue >= testValue >= minValue
        return yesNode
    else
        return noNode
class MinMax:
    def getBranch():
    if maxValue >= testValue >= minValue
        return yesNode
    else
        return noNode
```
Building and Maintaining a Decision Tree

visible = decision[0] = new Boolean...
audible = decision[1] = new Boolean...
close = decision[2] = new MinMax...
flank = decision[3] = new Boolean...

attack = action[0] = new Move...
move = action[1] = new Move...
creep = action[2] = 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

Performance Issues

- 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
Next Basic AI Technique:

(Hierarchical) State Machines

References: Buckland, Chapter 2
Millington, Section 5.3
Hard-Coded Implementation

class Soldier

def update()
    if currentState = GUARD {
        if (small enemy)
            currentState = FIGHT
            startFighting
        if (big enemy)
            currentState = RUN_AWAY
            startRunningAway
    } else if currentState = FIGHT {
        if (losing fight) c
            currentState = RUN_AWAY
            startRunningAway
    } else if currentState = RUN_AWAY {
        if (escaped)
            currentState = GUARD
            startGuarding
    }

Hard-Coded State Machines

- Easy to write (at the start)
- Very efficient
- Notoriously hard to maintain (e.g., debug)
Cleaner & More Flexible Implementation

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

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

class StateMachine:
    states = {}
    initialState = None
    currentState = initialState

    def update(self):
        triggeredTransition = None
        for transition in self.currentState.getTransitions():
            if transition.isTriggered():
                triggeredTransition = transition
                break

        if triggeredTransition:
            targetState = triggeredTransition.getTargetState()
            actions = self.currentState.getExitAction()
            actions += triggeredTransition.getAction()
            actions += targetState.getEntryAction()
            self.currentState = targetState
        else:
            return self.currentState.getAction()

...add tracing
(see Millington, Section 5.3.3)
```

Combining Decision Trees & State Machines

- **Why?**
  - to avoid duplicating expensive tests

![Diagram]

```
player_in_sight AND far
  └── alarm

player_in_sight AND near
  └── defend

alert
```
Combining Decision Trees & State Machines

alert

player in sight?

no

yes

far?

yes

alarm

no

no

defend

Hierarchical State Machines

- Why?

search

see trash

goto trash

have trash

trash disposed

goto disposal

goto trash
Interruptions (Alarms)

Add Another Interruption Type

6 - doubled the number of states!

12 - doubled the number of states again!
Hierarchical State Machine

- leave any state in (composite) ‘clean’ state when ‘low power’
- ‘clean’ remembers internal state and continues when returned to via ‘recharged’

Add Another Interruption Type

7 states (including composite) vs. 12
**Cross-Hierarchy Transitions**

- **Why?**
  - Suppose we want robot to top-off battery if it doesn't see any trash.

Diagram:

- **clean**
  - Search: see trash
  - Goto trash: have trash
  - Trash disposed: Goto disposal
  - Low power: Recharge
  - Less than 75% power

- **Recharge**
  - Low power
  - Recharged
### Implementation Sketch

**class State**
- # stack of return states
  - def getStates() return [this]
- # recursive update
  - def update()
- # rest same as flat machine

**class Transition**
- # how deep this transition is
  - def getLevel()
- # rest same as flat machine

**struct UpdateResult** # returned from update
- transition
- level
- actions # same as flat machine

**class HierarchicalStateMachine**
- # same state variables as flat machine
- # complicated recursive algorithm
  - def update()

**class SubMachine : HierarchicalStateMachine. State**
- def getStates()
  - push this onto currentState.getStates()

*(see Millington, Section 5.3.9)*