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")

**Resources:** introduction to Buckland, [www.gameai.com](http://www.gameai.com), [aigamedev.com](http://aigamedev.com), [www.aiwisdom.com](http://www.aiwisdom.com), [www.ai4games.org](http://www.ai4games.org)
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
    - basic pathfinding (A*) *(IMGD 3000)*
    - decision trees *(today)*
    - (hierarchical) state machines *(today)*
    - scripting *(next week)*
    - minimax search *(next week)*

• **Advanced** game AI
  - used in practice, but in more sophisticated games
    - advanced pathfinding *(tomorrow)*
    - autonomous movement, steering *(next week)*
    - behavior trees (in Halo 3) *(after midterm)*

Future Game AI?

• Take IMGD 400X in 2011 (B) [alt yr course] “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

How about AI Middleware?

- Panel at GDC AI Summit: “Why so wary of middleware?”
- Only one panelist reported completely positive experience
  - Steve Gargolinski, Blue Fang (Zoo Tycoon, etc.)
  - Used Havok Behavior (with Physics)
- Most industry AI programmers still mostly write their own AI from scratch (or reuse their own code)
- So we are going to look at coding details
First Basic AI Technique:

Decision Trees

Reference: Millington, Section 5.2

- The 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;
    }
}

Which would you rather modify?

if (visible) {
    if (close) {
        attack;
    } else if (flank) {
        move;
    } else {
        attack;
    }
} else if (audible) {
    creep;
}
**O-O Decision Trees (Pseudo-Code)**

(see Millington, Section 5.2.3)

```python
class Node:
    def decide(self):
        return self

class Action(Node):
    def decide(self):
        return self
class Decision(Node):
    def getBranch(self):
        def decide()
            return getBranch().decide()
    class Boolean:
        yesNode
        noNode
    class MinMax(Boolean):
        minValue
        maxValue
        testValue
        def getBranch(self):
            if maxValue >= testValue >= minValue:
                return yesNode
            else:
                return noNode
```

![Decision Tree Diagram]

**Building an O-O Decision Tree**

```python
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...
attack = new Move...
moves = 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...

attack = new Move...
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

...```

Performance Issues

- individual node tests \( (\text{getBranch}) \) 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
enum State
  GUARD
  FIGHT
  RUN_AWAY
currentState

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)
      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 O-O Implementation

```python
class State:
    def __init__(self):
        pass

    def getAction(self):
        pass

    def getEntryAction(self):
        pass

    def getExitAction(self):
        pass

    def getTransitions(self):
        pass
class Transition:
    def __init__(self, source_state, target_state):
        pass

    def isTriggered(self):
        pass

    def getTargetState(self):
        pass

    def getAction(self):
        pass
class StateMachine:
    def __init__(self, initial_state):
        self.currentState = initial_state
        self.transitions = []

    def update(self):
        triggered_transition = None
        for transition in self.currentState.getTransitions():
            if transition.isTriggered():
                triggered_transition = transition
                break
        if triggered_transition:
            target_state = triggered_transition.getTargetState()
            actions = self.currentState.getExitAction()
            actions += triggered_transition.getAction()
            actions += target_state.getEntryAction()
            self.currentState = target_state
            return actions
        else:
            return self.currentState.getAction()
```

...add tracing

(see Millington, Section 5.3.3)

Combining Decision Trees & State Machines

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

```
player in sight AND far

player in sight AND near

alert

alarm

defend
```
Combining Decision Trees & State Machines

Hierarchical State Machines

- Why?

```
<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>see trash</td>
<td>trash</td>
</tr>
<tr>
<td>trash</td>
<td>have trash</td>
<td>disposal</td>
</tr>
</tbody>
</table>
```
Interruptions (Alarms)

6 - doubled the number of states!

Add Another Interruption Type

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

```
  clean

  search  —>  see trash  —>  goto trash  —>  have trash
                     |                       |
                     v                       v
trash disposed       trash disposed
                           |                       |
                           v                       v
                       goto disposal
```

- Cross-Hierarchy Transitions

```
  clean

  search  —>  see trash  —>  goto trash  —>  have trash
                     |                       |
                     v                       v
trash disposed       trash disposed
                           |                       |
                           v                       v
                       goto disposal
```

less than 75% power

```
  clean

  search  —>  see trash  —>  goto trash  —>  have trash
                     |                       |
                     v                       v
trash disposed       trash disposed
                           |                       |
                           v                       v
                       goto disposal
```

low power

```
  clean

  search  —>  see trash  —>  goto trash  —>  have trash
                     |                       |
                     v                       v
trash disposed       trash disposed
                           |                       |
                           v                       v
                       goto disposal
```

recharge

```
  clean

  search  —>  see trash  —>  goto trash  —>  have trash
                     |                       |
                     v                       v
trash disposed       trash disposed
                           |                       |
                           v                       v
                       goto disposal
```

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

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)