Basic Game AI

IMGD 4000

What’s AI Part of a Game?

• Everything that isn’t graphics (sound) or networking... (says an AI professor ☺)
  – or physics (though sometimes lumped in)
  – 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 (other deck)
    • Behavior trees in UE4 (this deck)

Future Game AI?

• Take IMGD 4100
  – “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", e.g., pathfind to closer node)
    • Need to otherwise limit computation (e.g., threshold, time-slice pathfinding)

• 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. ;-)

With material from Ian Millington and John Funge, Artificial Intelligence for Games, Morgan Kaufmann, 2009 (Chapter 5)
How About AI Middleware (“AI Engines”)?

- Recent panel at GDC AI Summit: “Why so wary of AI 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 write their own AI from scratch (or reuse their own code)
  - Damian Isla, _Flame in the Flood_, custom procedural content generation
- So, we are going to look at coding details

AI Coding Theme (for Basic AI)

- Use *object-oriented* paradigm instead of...
- A tangle of *if-then-else* statements

Outline

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

First Basic AI Technique:

Decision Trees

See code at: https://github.com/idmillington/aicore
src/dectree.cpp and src/demos/05-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 (1 of 2)

```cpp
if visible? {
    if close? {
        attack;
    }
    else {
        if flank? {
            move;
        }
        else {
            attack;
        }
    }
}
else {
    if audible? {
        creep;
    }
}
```
Deciding How to Respond to an Enemy

Alternate form:

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

MODIFYING DECIDING HOW TO RESPOND TO AN ENEMY

Alternate form: Harder to see “depth”!

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

O-O Decision Trees (Pseudo-Code)

```java
class Node {
    def decide() {
        // return action/decision
    }
}

class Decision : Node {
    def getBranch() {
        // return a node
    }
    def decide() {
        return getBranch().decide();
    }
}

class Action : Node {
    def decide() {
        return this;
    }
}
```

Building an O-O Decision Tree

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

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
- Behavior Trees

Second Basic AI Technique:
(Hierarchical) Finite State Machines

Finite State Machines
- Often AI as agents: sense, think, then act
- But many different rules for agents
  - Ex: sensing, thinking and acting when fighting, running, exploring...
    - Can be difficult to keep rules consistent
- Try Finite State Machine
  - Natural correspondence between states and behaviors
  - Easy: to diagram, program, debug
- Formally:
  - Set of states
  - A starting state
  - An input vocabulary
  - A transition function that maps inputs and current state to next state

(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

class State
  def getOption():
  def getExitAction():
  class Transition
    def isTriggered():
    def getTargetState():

class StateMachine
  states
  initDataState
  currentState = initDataState
  def update():
    triggeredTransition = null
    for transition in currentState.getTransitions():
      if transition.isTriggered():
        triggeredTransition = transition
        break
    if triggeredTransition != null:
      targetState = triggeredTransition.getTargetState()
      actions = currentState.getExitAction()
      actions += targetState.getEntryAction()
      currentState = targetState
      return actions
    else:
      return currentState.getAction()

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)

Use decision tree for transitions in state machine

Outline

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

Hierarchical State Machines

• Why? Could be interruptions, want to return but not to start
  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 (e.g., may have trash or know where trash is).

Interruptions (e.g., Recharging)

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

12 states needed → doubled again!

Add Another Interruption (e.g., Baddies)

7 states needed (including composite) vs. 12

Cross-Hierarchy Transitions

• Why?
  — Suppose want robot to “top off” battery (even if it isn’t low) when it doesn’t see any trash

Cross-Hierarchy Transitions

no trash and less than 75% power

Hierarchical State Machine

• Leave any state in (composite) “clean” state when “low power”
• “clean” remembers internal state and continues when back from “recharge”

HFSM Implementation Sketch
Outline

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

What is a Behavior Tree?

- A model of plan execution
  - Switch between tasks in modular fashion
- Similar to HFSM, but block is task not state
- Early use for NPCs (Halo, Bioshock, Spore)
- Tree – notes are root, control flow, execution

“Behavior” in Behavior Tree

- Sense, Think, Act
- Repeat

Sense

- Generally rely on physics engine
- Usually very expensive
- Use infrequently

Think

- Decision logic
- Generally quite simple
- Design intensive

Act

- Action execution
- Often long running
- Can fail to complete
Behavior Tree with Memory

In UE4, the “Memory” is called “Blackboard”

UE4 Behavior Trees vs. Traditional

- UE4 Event Driven
  - Do not poll for changes, but listen for events that trigger changes
- UE4 “conditionals” not at leaf
  - Allows easier distinguish versus task
  - Allows them to be passive (event driven)
- UE4 simplifies parallel nodes (typically confusing)
  - Simple parallel for concurrent tasks
  - Services for periodic tasks


UE4 Behavior Tree - Root

- The starting execution node for the Behavior Tree.
- Every Behavior Tree has one.
- You cannot attach Decorators or Services to it.

UE4 Behavior Tree - Composite

- These are nodes that define the root of a branch and define the base rules for how that branch is executed.
- Sequence, Selector, Simple Parallel

Composite - Sequence

- Sequence Node execute their children from left to right, and will stop executing its children when one of their children fails.
  If a child fails, then the sequence fails.
**Composite - Selector**

- Selector Nodes execute their children from left to right, and will stop executing its children when one of their children succeeds. If a Selector's child succeed, the Selector succeeds.

**Composite – Simple Parallel**

- The Simple Parallel node allows a single main task node to be executed along side of a null tree. When the main task finishes, the setting in finish node dictates the secondary tree.

**Service**

- Decorators
  - These attach to Composite nodes, and will execute at their defined frequency. These are often used to make their and to update the Blackboard. These take the place of traditional Parallel nodes.

**Decorators**

- Also known as conditionals. These attach to another node and make decisions on whether or not a branch in the tree, or even single node, can be executed.

**Task**

- There are leaves of the tree, the nodes that “do” things.

**Blackboard (Memory)**

- A Blackboard is a simple place where data can be written and read for decision making purposes.
- A Blackboard can be used by a single AI team, shared by squad.
The Behavior Tree Quick Start Guide walks you through the process of creating a NavMesh, creating an AI Controller, creating a Character that will be controlled by that AI Controller, and creating all the parts necessary for a simple Behavior Tree.

**Resource Links**

- **HFSM from Millington and Funge**
  

- **FSM from IMGD 3000**
  
  - Slides
  
  - Header files
    [http://dragonfly.wpi.edu/include/classStateMachine.html](http://dragonfly.wpi.edu/include/classStateMachine.html)

- **UE4 Behavior Tree**
  
  - Difference between BT and DT
  
  - Quick Start
    [https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/](https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/)