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. Sometimes pathfinding
    • 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. ;-)
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 in UE4

First Basic AI Technique: Decision Trees

See code at: [https://github.com/idmillington/aicore](https://github.com/idmillington/aicore)
- src/dectree.cpp
- src/demos/DS/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)

- If visible? (level 0)
  - If close? (level 1)
    - attack;
  - else (level 1)
    - if flank? (level 2)
      - move;
    - else (level 2)
      - attack;
  - else (level 1)
    - if audible? (level 0)
      - creep;
    - else (level 0)
      - attack;

Leaves are actions
Interior nodes are decisions
Typically binary
(If multiple choices, can be converted to binary)
Deciding How to Respond to an Enemy

(2 of 2)

```plaintext
if visible? { // level 0
  if close? { // level 1
    attack;
  } else if audible? { // level 0 & 1
    creep;
  } else { // level 0 & 1
    attack;
  }
}
else if flank? { // level 1 & 2
  move;
}
else { // level 0 & 1
  attack;
}
```

Alternate form.  Harder to see “depth”!

```plaintext
if visible? { // level 0
  if close? { // level 1
    attack;
  } else if audible? { // level 0 & 1
    creep;
  } else { // level 0 & 1
    attack;
  }
}
else if flank? { // level 1 & 2
  move;
}
else { // level 0 & 1
  attack;
}
```

Building an O-O Decision Tree

```plaintext
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = 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 = ???
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:
  - A set of states
  - A starting state
  - An input vocabulary
  - A transition function that maps inputs and current state to next state

Finite State Machines

- Acting done states
- Sensing done by conditionals
- Thinking done by transitions

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 __init__(self):
        self.initialState = initial_state
        self.current_state = self.initialState

def getExitAction() -> List[Action]:
    if triggered_transition:
        self.current_state = triggered_transition.target_state
    return self.current_state.exit_actions

def getEntryAction() -> List[Action]:
    return self.current_state.entry_actions

def getTransitions() -> List[Transition]:
    return self.current_state.transitions

class Transition:
    def isTriggered():
        for transition in self.getTransitions():
            if transition.isTriggered():
                return transition
        return None

    def getNextState():
        return self.target_state

class StateMachine:
    def __init__(self, states: List[State], initialState: State):
        self.states = states
        self.currentState = initialState

    def update(self) -> List[Action]:
        triggeredTransition = None
        for transition in self.currentState.getTransitions():
            if transition.isTriggered():
                triggeredTransition = transition
                break
        if triggeredTransition:
            self.targetState = triggered_transition.target_state
            return self.currentState.getExitAction() +
                    self.targetState.getEntryAction()
        else:
            return self.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.

```
player in sight AND far
  → alarm

player in sight AND near
  → alert

player in sight AND close
  → defend
```

Combining Decision Trees & State Machines (2 of 2)

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

```
search
    goto trash
        have trash
        trash disposed
    see trash

trash disposed
    goto disposal
```

Interruptions (e.g., Recharging)

```
recharge
    low power
    recharged
    low power

search
    see trash
    goto trash
    have trash
    trash disposed

trash disposed
    goto disposal
    recharge
```

---

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

Hierarchical State Machine

- Leave any state in (composite) "clean" state when "low power"
- "clean" remembers internal state and continues when back from "recharge"

Cross-Hierarchy Transitions

- Why?
  - Suppose want robot to "top off" battery (even if it isn't low) when it doesn't see any trash

HFSM Implementation Sketch

```
class State:
    # stack of return states
    def __init__(self):
        self.level = 0
        self.state = None

    def update(self):
        # rest same as flat machine

class Transition:
    # how deep this transition is
    def __init__(self, level):
        self.level = level

    def update(self):
        # rest same as flat machine

class SubMachine(State):
    def __init__(self):
        self.level = 0

    def update(self):
        # rest same as flat machine

    def getStates(self):
        return [self]

class HierarchicalStateMachine:
    # same state variables as flat machine
    # complicated recursive algorithm
    def __init__(self):
        self.level = 0

    def update(self):
        # rest same as flat machine
```

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

Search and grasp plan of two-armed robot

https://upload.wikimedia.org/wikipedia/commons/1/1b/BT_search_and_grasp.png

“Behavior” in Behavior Tree

https://www.slideshare.net/JaeWanPark2/behavior-tree-in-unreal-engine-4

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

Search and grasp plan of two-armed robot

https://upload.wikimedia.org/wikipedia/commons/1/1b/BT_search_and_grasp.png
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.

“And”
UE4 Behavior Tree Quick Start

“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
  
  http://web.cs.wpi.edu/~imgd4000/d16/slides/millington-hsm.pdf

- FSM from IMGD 3000
  
  Slides
  
  
  Header files
  
  http://dragonfly.wpi.edu/include/classStateMachine.html

- UE4 Behavior Tree
  
  Quick Start
  
  https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/