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, aigamedev.com, www.aiwisdom.com, 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

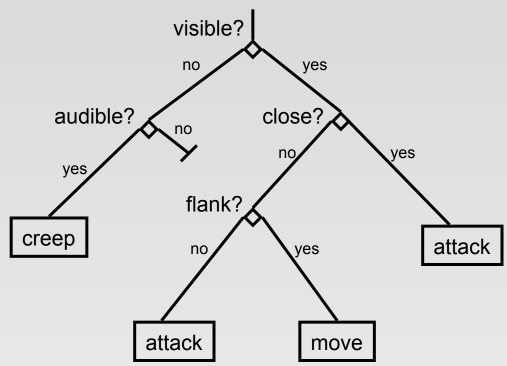
Reference: Millington, Section 5.2

Decision Trees

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

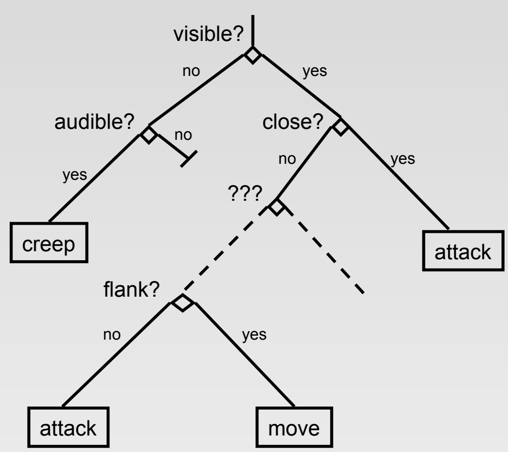
Deciding how to respond to an enemy

```
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;
}
```

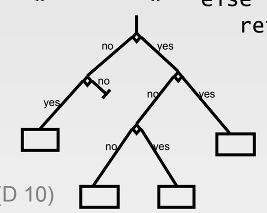


O-O Decision Trees (Pseudo-Code)

(see Millington, Section 5.2.3)

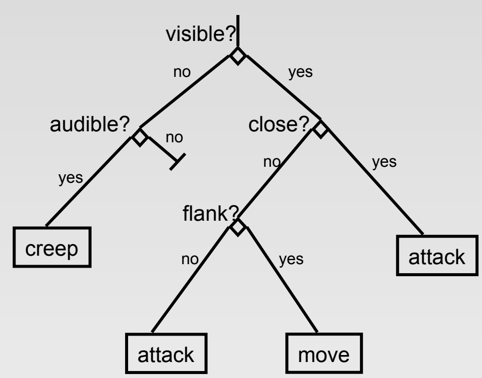
```
class Node
   def decide()

class Action : Node
   def decide()
      return this

class Decision : Node

   def getBranch()

   def decide()
      return getBranch().decide()
```

```
class Boolean : Decision
   yesNode
   noNode

class MinMax : Boolean
   minValue
   maxValue
   testValue

   def getBranch()
      if maxValue >= testValue >= minValue
         return yesNode
      else
         return noNode
```



Building an O-O Decision Tree

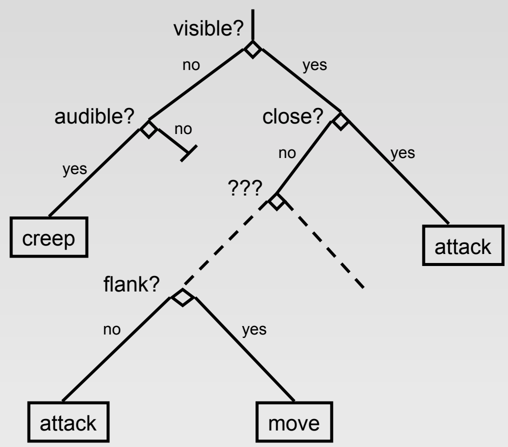
```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...

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
...
```



...or a graphical editor

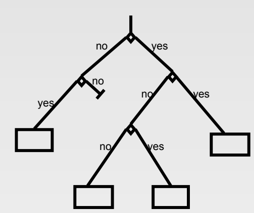
Modifying an O-O Decision Tree

```
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 (`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

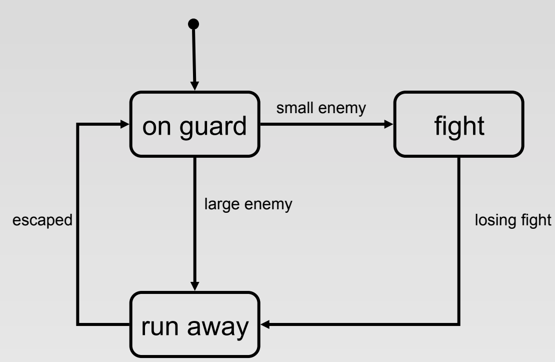


Next Basic AI Technique:

(Hierarchical) State Machines

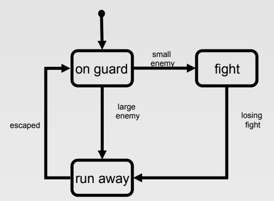
References: Buckland, Chapter 2
Millington, Section 5.3

State Machines



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

(see Millington, Section 5.3.3)

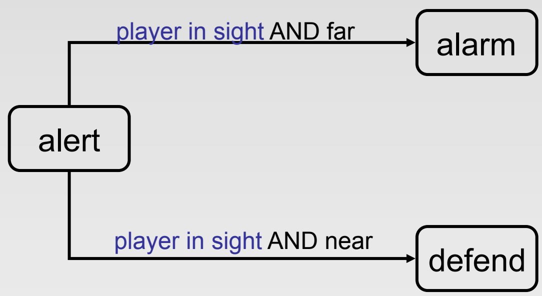
```
class State
   def getAction()
   def getEntryAction()
   def getExitAction()
   def getTransitions()

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
         targetState = triggeredTransition.getTargetState()
         actions = currentState.getExitAction()
         actions += triggeredTransition.getAction()
         actions += targetState.getEntryAction()
         currentState = targetState
         return actions
      else
         return currentState.getAction()
```

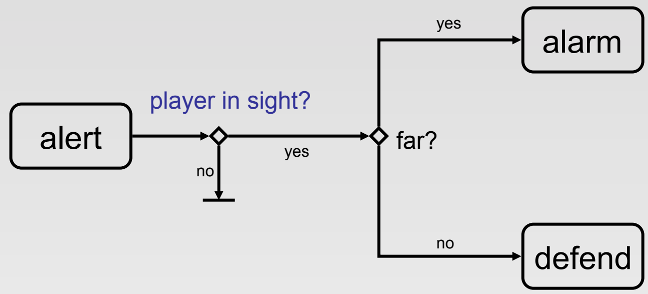
...add tracing

Combining Decision Trees & State Machines

- Why?
  - to avoid duplicating expensive tests

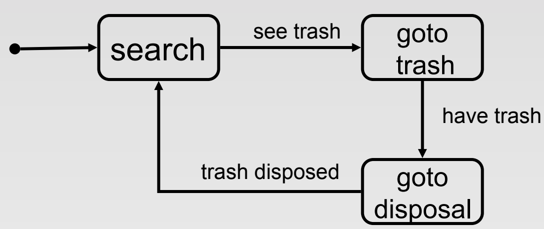


Combining Decision Trees & State Machines

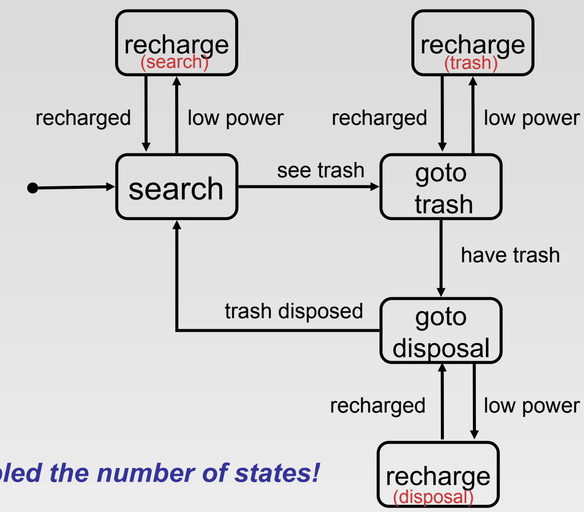


Hierarchical State Machines

- Why?

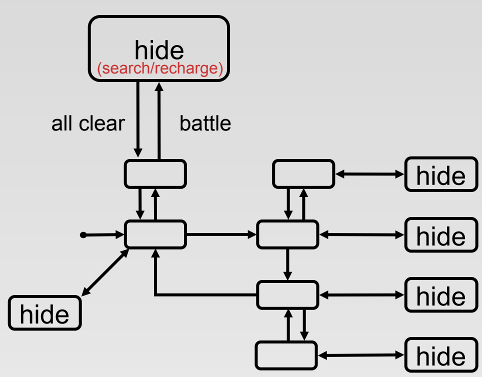


Interruptions (Alarms)



6 - doubled the number of states!

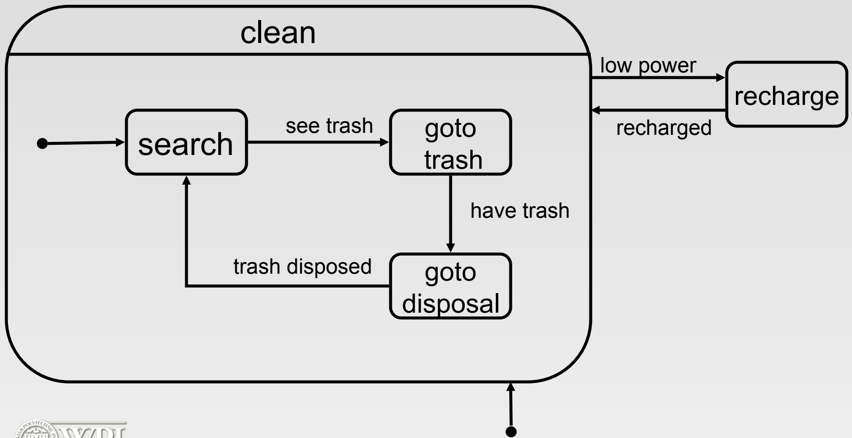
Add Another Interruption Type



12 - doubled the number of states again!

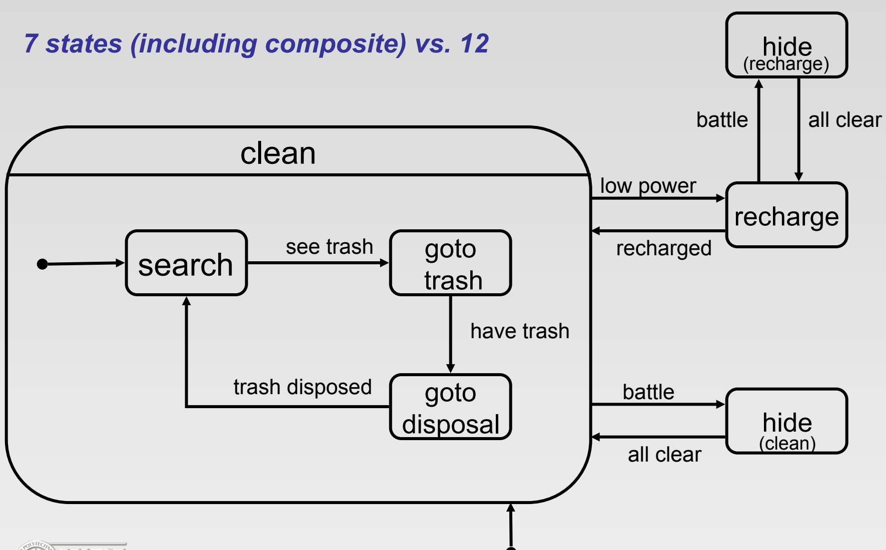
Hierarchical State Machine

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



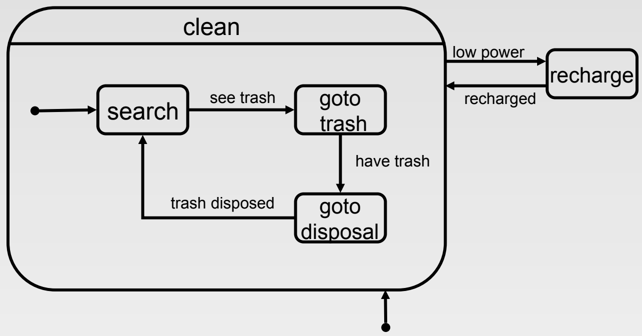
Add Another Interruption Type

7 states (including composite) vs. 12

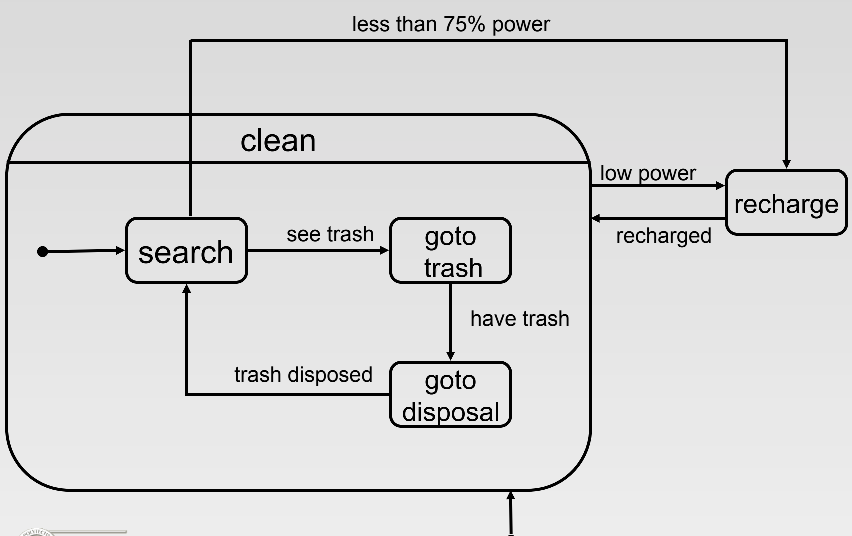


Cross-Hierarchy Transitions

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



Cross-Hierarchy Transitions



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)