DM810

Computer Game Programming II: AI

Lecture 8
Decision Making

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Resume

- Hierarchical Pathfinding
- A* variants
- Decision Making
- Decision Trees

Outline

1. State Machine

2. Behavior Trees

Outline

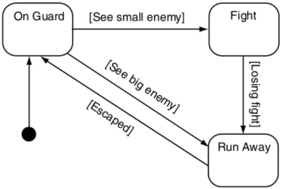
1. State Machine

2. Behavior Trees

Finite State Machines

An FSM is an algorithm used for parsing text, eg, tokenize the input code into symbols that can be interpreted by the compiler.

- States: actions or behaviors. Chars are in exactly one of them @ any time.
- Transitions: a set of associated conditions, if they are met the char changes state
- Initial state for the first frame the state machine is run



In a decision tree, the same set of decisions is always used, and any action can be reached through the tree.
In a state machine, only transitions from the current state are considered, so not every action can be reached.

General State Machines

- set of possible states
- current state
- set of transitions
- at each iteration (normally each frame), the state machine's update function is called.
- checks if any transition from the current state is triggered
- the first transition that is triggered is scheduled to fire (some actions related to transition are executed)

Implementation

```
class StateMachine:
  states # list of states for the machine
  initialState
  currentState = initialState
  def update(): # checks and applies
    triggeredTransition = None
    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()
```

```
class MyFSM:
  enum State:
    PATROL
    DEFEND
    SLEEP
  myState # holds current state

  # transition by polling (asking for information explicitly)
  def update():
    if myState == PATROL:
      if canSeePlayer(): myState = DEFEND # access to game state data
      if tired(): myState = SLEEP # access to game state data
    elif myState == DEFEND:
      if not canSeePlayer(): myState = PATROL
    elif myState == SLEEP:
      if not tired(): myState = PATROL

  # transition in an event-based approach (waiting to be told information)
  def notifyNoiseHeard(volume):
    if myState == SLEEP and volume > 10:
      myState = DEFEND

  def getAction():
    if myState == PATROL: return PatrolAction
    elif myState == DEFEND: return DefendAction
    elif myState == SLEEP: return SleepAction
```

State machines implemented like this can often get large and code unclear

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

```
class Transition:
  actions
  def getAction(): return actions
  targetState
  def getTargetState(): return targetState
  condition
  def isTriggered(): return condition.test()
```

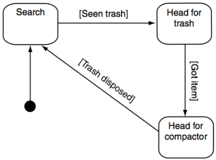
Often defined in a data file and read into the game at runtime.
Do not allow to compose questions easily.
Requires condition interface.

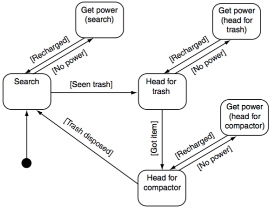
```
class Condition:
  def test()

class FloatCondition (Condition):
  minValue
  maxValue
  testValue # ptr to game data
  def test():
    return minValue <= testValue
          <= maxValue
```

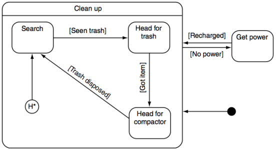
```
class AndCondition (Condition):
  conditionA
  conditionB
  def test():
    return conditionA.test() and conditionB.test()

class NotCondition (Condition):
  condition
  def test(): return not condition.test()

class OrCondition (Condition):
  conditionA
  conditionB
  def test():
    return conditionA.test() or conditionB.test()
```

Hierarchical State Machines

- Alarm mechanism: something that interrupts normal behavior to respond to something important.
- Representing this in a state machine leads to a doubling in the number of states.
- Instead: each alarm mechanism has its own state machine, along with the original behavior.



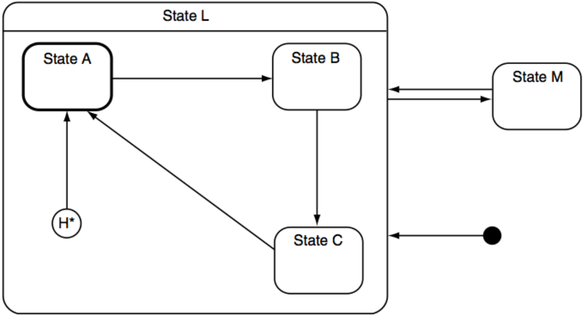




We can add transitions between layers of machines

Implementation

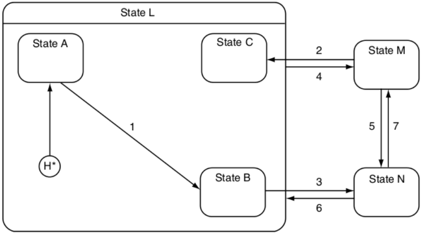
In a hierarchical state machine, each state can be a complete state machine in its own right $\rightsquigarrow$ recursive algorithm



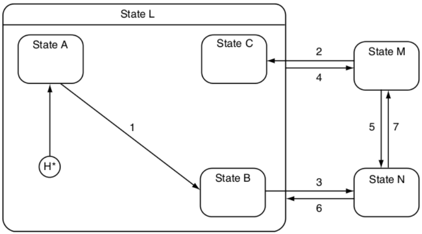
A triggered transition may be: (i) to another state at current level, (ii) to a state higher up, or (iii) to a lower state

Example

- start in State L
- from H* transition to A
  update = [L-active, A-entry]
  current State [L, A]



- top-level state machine no valid transitions
  state machine L: current state [A], triggered transition 1 ⇝ stay at current level, transition to B, update = [A-exit, 1-actions, B-entry]
  top-level state machine accepts and adds L-active. current State [L, B].
- top level machine: triggered transition 4
  transition to State M, update = [L-exit, 4-actions, M-entry].
  current State is [M]. (state machine L still keeps State B)
- top level machine: triggered transition 5
  transition to State N, update = [M-exit, 5-actions, N-entry]. current State N
- top level machine: triggered transition 6
  transitions to State L, update = [N-exit, 6-actions, L-entry].
  state machine L has current state still State [L, B] ⇝ no B-entry action



- top-level state machine no transition; State [L, B] triggered transition 3.
  top-level state machine no triggers state machine L: B, transition has one level up
  update: B-exit
  top-level machine: transition to State N; `update` += [L-exit, 3-actions, N-entry]
- State N $\rightarrow$ transition 7 $\rightarrow$ State M
  ...
- top level machine: triggered transition 2. top-level state machine: transition down $\rightsquigarrow$ updateDown. state machine L: `update` = C-enter
  top-level state machine changes from State M to State L, `update` += [M-exit, L-entry, 2-actions]

Implementation

```
class HSMBase:
  struct UpdateResult:
  actions
  transition
  level
  def getAction(): return []
  def update():
    UpdateResult result
    result.actions = getAction()
    result.transition = None
    result.level = 0
    return result
  def getStates()

class State (HSMBase):
  def getStates():
    return [this]
  def getAction()
  def getEntryAction()
  def getExitAction()
  def getTransitions()

class Transition:
  def getLevel()
  def isTriggered()
  def getTargetState()
  def getAction()
```

```
class HierarchicalStateMachine (HSMBase):
  states # List of states at this level
  initialState # when no current state
  currentState = initialState
  def getStates():
    if currentState: return currentState.getStates()
    else: return []
  def update(): ...
  def updateDown(state, level): ...

class SubMachineState (State,HierarchicStateMachine):
  def getAction(): return State::getAction()
  def update(): return HierarchicalStateMachine::
        update()
  def getStates():
    if currentState:
      return [this] + currentState.getStates()
    else:
      return [this]
```

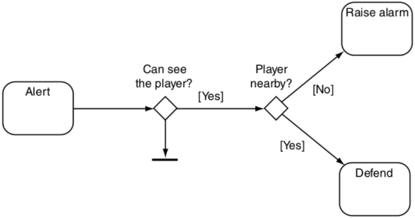
```
class HierarchicalStateMachine (HSMBase):
  states # List of states at this level
  initialState # when no current state
  currentState = initialState
  def getStates():
    if currentState: return currentState.getStates()
    else: return []
  def update():
    if not currentState:
      currentState = initialState
      return currentState.getEntryAction()
    triggeredTransition = None
    for transition in currentState.getTransitions():
      if transition.isTriggered():
        triggeredTransition = transition
        break
    if triggeredTransition:
      result = UpdateResult()
      result.actions = []
      result.transition = triggeredTransition
      result.level = triggeredTransition.getLevel()
    else:
      result = currentState.update() # rcrs.
```

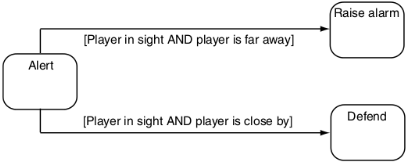
```
if result.transition:
  if result.level == 0: # Its on this level: honor it
    targetState = result.transition.getTargetState()
    result.actions += currentState.getExitAction()
    result.actions += result.transition.getAction()
    result.actions += targetState.getEntryAction()
    currentState = targetState
    result.actions += getAction()
    result.transition = None # so nobody else does it
  else if result.level > 0: # it is for a higher level
    result.actions += currentState.getExitAction()
    currentState = None
    result.level -= 1
  else: # It needs to be passed down
    targetState = result.transition.getTargetState()
    targetMachine = targetState.parent
    result.actions += result.transition.getAction()
    result.actions += targetMachine.updateDown(targetState,-result.level) #
          recursion
    result.transition = None # so nobody else does it
else: # no transition
  result.action += getAction()
return result
```

```
def updateDown(state, level):
  if level > 0: # continue recursing
    actions = parent.updateDown(this, level-1)
  else: actions = []
  if currentState:
    actions += currentState.getExitAction()
  currentState = state # move to the new state
  actions += state.getEntryAction()
  return actions
```

Combining DT and SM

Decision trees can be used to implement more complex transitions





Outline

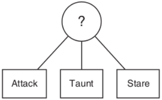
1. State Machine

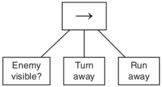
2. Behavior Trees

Behavior Trees

- synthesis of: Hierarchical State Machines, Scheduling, Planning, and Action Execution.
- state: task composed of sub-trees
- tasks are Conditions, Actions, Composites
- tasks return `true, false, error, need more time`
- Actions: animation, character movement, change the internal state of the character, play audio samples, engage the player in dialog, pathfinding.
- Conditions are logical conditions
- behavior trees are coupled with a graphical user interface (GUI) to edit the trees.

- Both Conditions and Actions sit at the leaf nodes of the tree. Branches are made up of Composite nodes.
- Composites: two main types: Selector and Sequence
- Both run each of their child behaviors in turn and decide whether to continue through its children or to stop according to the returned value.
- Selector returns immediately with a success when one of its children succeeds. As long as children are failing, it keeps on trying. If no children left, returns failure. (used to choose the first of a set of possible actions that is successful) Eg: a character wanting to reach safety.

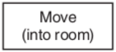


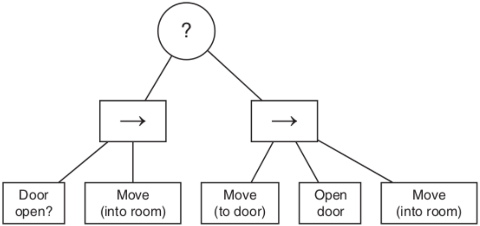


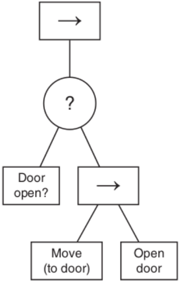
- Sequence returns immediately with a failure when one of its children fails. As long as children are succeeding, it keeps on trying. If no children left, returns success. (series of tasks that need to be undertaken)

Developing Behaviour Trees

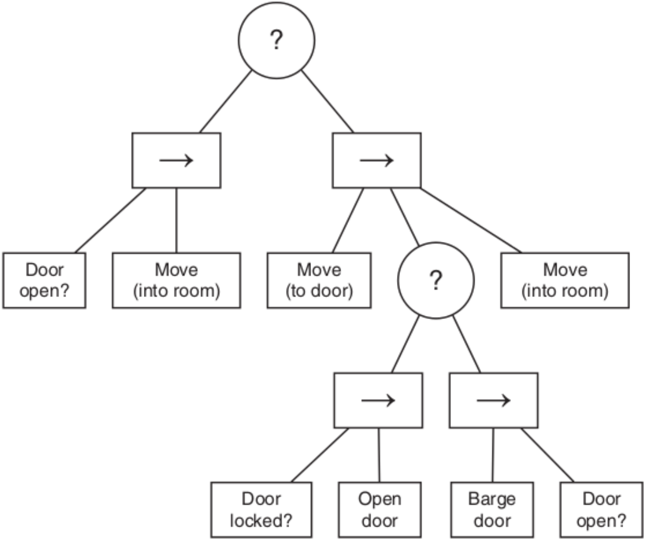
- get something very simple to work initially







Condition task in a Sequence acts like an IF-statement.
If the Sequence is placed within a Selector, then it acts like an IF-ELSE-statement


?
→
→
Door open?
Move (into room)
Move (to door)
?
Move (into room)
→
→
Door locked?
Open door
Barge door
Door open?

- behaviour trees implement a sort of reactive planning. Selectors allow the character to try things, and fall back to other behaviors if they fail. (look ahead only via actions)

- depth-first search

- could be written as state machines or decision trees but more complicated

Implementation


```
class Task:
  children
  def run() # true/false


class Selector (Task):
  def run():
    for c in children:
      if c.run():
        return True
    return False


class Sequence (Task):
  def run():
    for c in children:
      if not c.run():
        return False
    return True
```

```
class EnemyNear (Task):
  def run():
    if distanceToEnemy < 10:
      return True
    return False


class PlayAnimation (Task):
  animation_id
  speed
  def Attack(animation_id, loop=False,
        speed=1.0):
    this.animation = animation
    this.speed = speed
    def run():
      if animationEngine.ready(): #
           resource checking
        animationEngine.play(animation,
              speed)
        return True
      return False
```

Non-Deterministic Composite Tasks

- In some cases, always trying the same things in the same order can lead to predictable AIs.
- Selectors: eg, if altrnative ways to enter the door, no relevant the order
- Sequences: eg, collect components, no relevant the order "partial-order" constraints in the AI literature. Some
- parts may be strictly ordered, and others can be processed in any order.

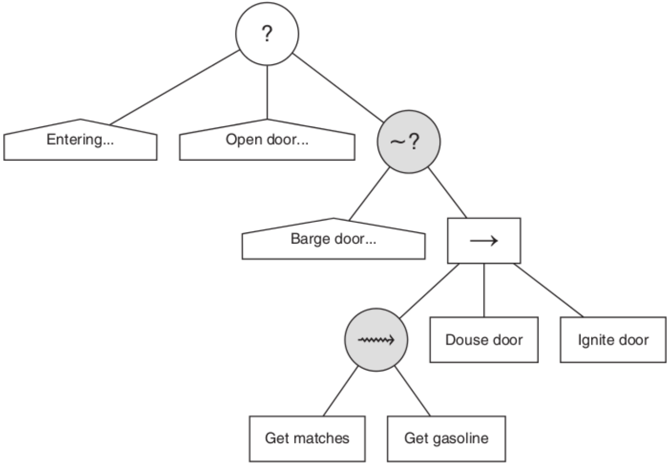
```
class NonDeterministicSelector (Task):
  children
  def run():
    shuffled = random.shuffle(children)
    for child in shuffled:
      if child.run(): break
    return result
```

```
class NonDeterministicSequence (Task):
  children
  def run():
    shuffled = random.shuffle(children)
    for child in shuffled:
      if not child.run(): break
    return result
```

Shuffle

by Richard Durstenfeld in 1964 in *Communications of the ACM*, volume 7, issue 7, as "Algorithm 235: Random permutation", and by Donald E. Knuth in volume 2 of his book *The Art of Computer Programming* as "Algorithm P" but originally by Fisher and Yates.

```
def shuffle(original):
  list = original.copy()
  n = list.length
  while n > 1:
    k = random.integer_less_than(n)
    n--;
    tmp = list[k], list[k] = list[n], list[n] = tmp
  return list
```


?
Entering...
Open door...
~?
Barge door...
→
Douse door
Ignite door
Get matches
Get gasoline