Basic Game AI

IMGD 4000

With material from: Ian Millington and John Funge. *Artificial Intelligence for Games*, Morgan Kaufmann, 2009. (Chapter 5)

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)

2

"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

3

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)*

4

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

5

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. ;-)

6

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/c05-dectree

Ian Millington and John Funge. *Artificial Intelligence for Games*, Morgan Kaufmann, 2009. (Chapter 5)

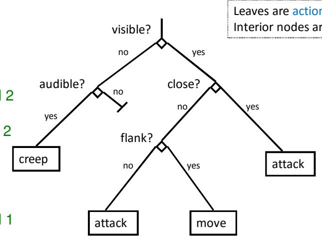
Decision Trees

- 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 0
  if audible? { // level 1
    creep;
  }
}
```

Leaves are actions
Interior nodes are decisions

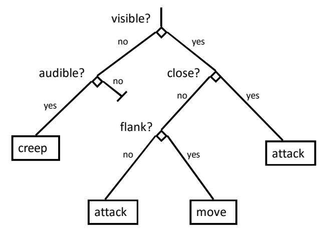


Typically binary
(if multiple choices, can be converted to binary)

Deciding How to Respond to an Enemy (2 of 2)

Alternate form.

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



Harder to see "depth"!

What if need to modify?
e.g., if *close*, only flank if ally near

???

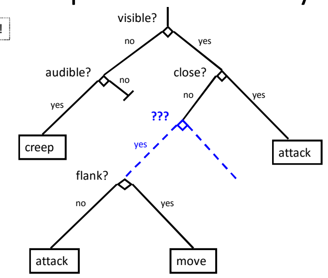
13

Modifying Deciding How to Respond to an Enemy

Alternate form. Harder to see "depth"!

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

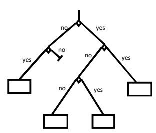
???



Modification restructures all below code! Code is *brittle*.
Solution? → Object-Oriented

14

O-O Decision Trees (Pseudo-Code)



```
class Node
  def decide() // return action/decision

class Decision : Node // interior
  def getBranch() // return a node
  def decide()
   return getBranch().decide()

class Action : Node // leaf
  def decide() return this
```

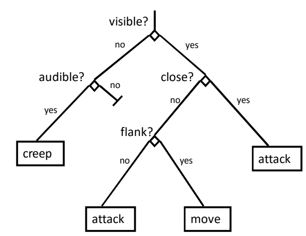
```
class Boolean : Decision // if yes/no
  yesNode
  noNode

class MinMax : Boolean // if range
  minValue
  maxValue
  testValue

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

```
// Define root as start of tree
Node *root

// Calls recursively until action
Action * action = root → decide()
action → doAction()
```

15

Building an O-O Decision Tree

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

attack = new Attack...
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
...
```

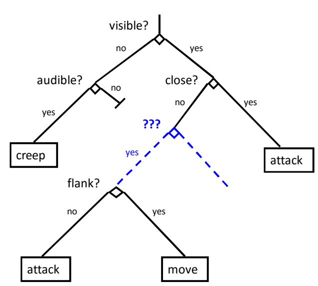


...or a graphical editor

16

Modifying an O-O Decision Tree

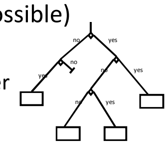
```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...
??? = new Boolean...

attack = new Action...
move = new Action...
creep = new Action...

visible.yesNode = close
visible.noNode = audible

audible.yesNode = creep

close.yesNode = attack
close.noNode = ???
???.yesNode = flank

flank.yesNode = move
flank.noNode = attack
...
```



17

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



18

Outline

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

Second Basic AI Technique:

(Hierarchical) Finite State Machines

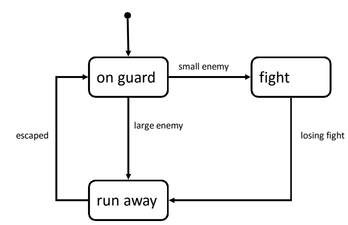
Finite State Machines

- 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

(Game example next slide)

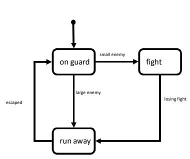
Finite State Machines

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



Hard-Coded Implementation

```
class Soldier

  enum State
    ON_GUARD
    FIGHT
    RUN_AWAY

  currentState
```



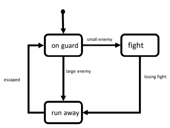

```
def update()
  if currentState == ON_GUARD {
    if small enemy {
      currentState = FIGHT
      start Fighting
    } else if big enemy {
      currentState = RUN_AWAY
      start RunningAway
    }
  } else if currentState == FIGHT {
    if losing fight {
      currentState = RUN_AWAY
      start 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 getAction()
  def getEntryAction()
  def getExitAction()
  def getTransitions()

class Transition
  def isTriggered()
  def getTargetState()
```

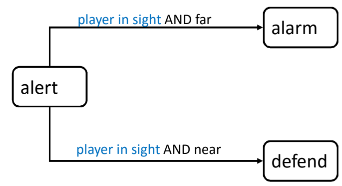



```
class StateMachine

 states
 initialState
 currentState = initialState

def update() // returns all actions needed this 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 // list of actions for transitions
  } else return currentState.getAction()  // action this state
```

25

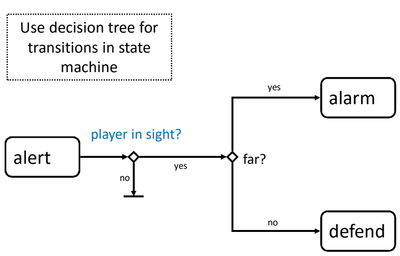
Combining Decision Trees & State Machines (1 of 2)

- Why?
  - to avoid duplicating expensive tests in state machine. e.g., assuming "player in sight" is expensive



26

Combining Decision Trees & State Machines (2 of 2)



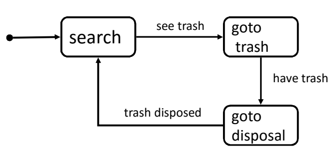
27

Outline

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

Hierarchical State Machines

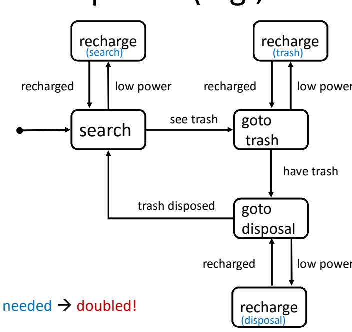
- Why? → Could be interruptions, want to return but not to start



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).

29

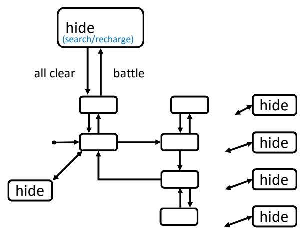
Interruptions (e.g., Recharging)



6 states needed → doubled!

30

Add Another Interruption (e.g., Baddies)

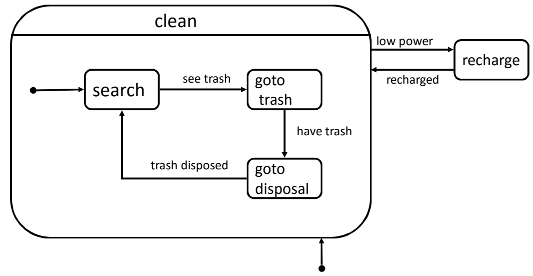


12 states needed → doubled again!

31

Hierarchical State Machine

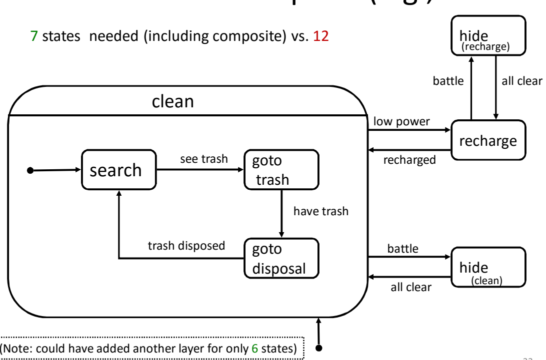
- Leave any state in (composite) "clean" state when "low power"
- "clean" remembers internal state and continues when back from "recharge"



32

Add Another Interruption (e.g., Baddies)

7 states needed (including composite) vs. 12

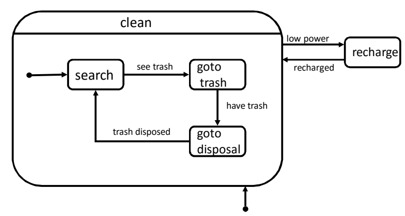


(Note: could have added another layer for only 6 states)

33

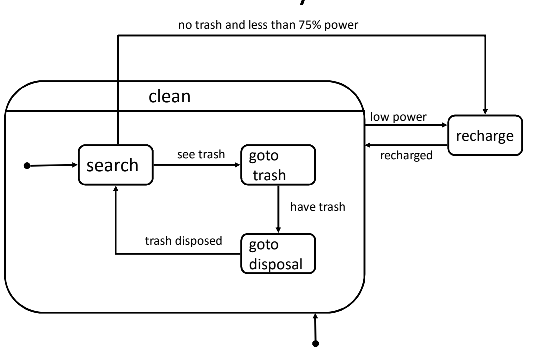
Cross-Hierarchy Transitions

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



34

Cross-Hierarchy Transitions



35

HFSM 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 full pseudo-code at http://web.cs.wpi.edu/~imgd4000/d16/slides/millington-hsm.pdf

36

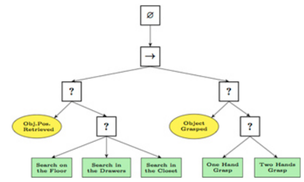
Outline

- Introduction (done)
- Decision Trees (done)
- Finite State Machines (FSM) (done)
- Hierarchical FSM (done)
- Behavior Trees (next)
  - In UE4

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

What is a Behavior Tree?

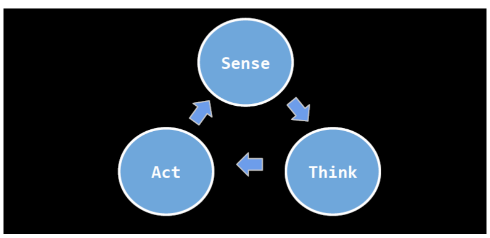
- A model of plan execution
  - Switch between tasks in modular fashion
- Similar to HFSM, but block is task not state
- Early use for NPCs (*Halo*, *Bioshock*, *Spore*)
- Tree – notes are *root*, *control flow*, *execution*



Search and grasp plan of two-armed robot

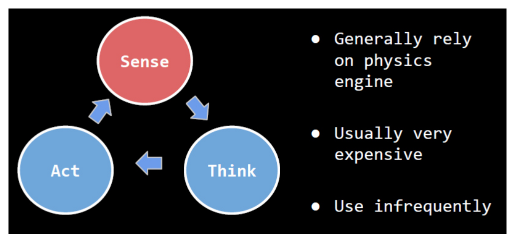
https://upload.wikimedia.org/wikipedia/commons/1/1b/BT_search_and_grasp.png

"Behavior" in Behavior Tree

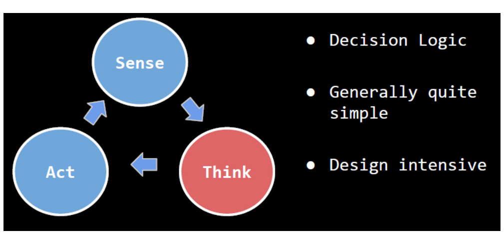


- Sense, Think, Act
  - Repeat

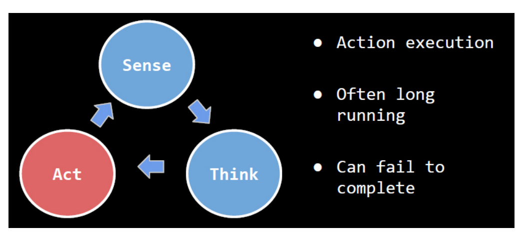
Sense



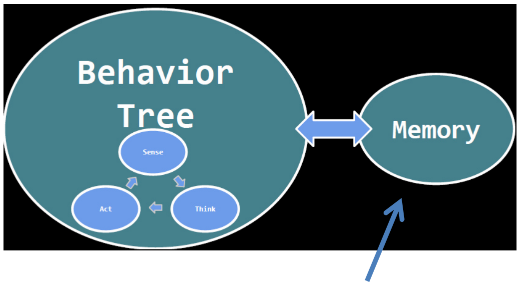
Think



Act



Behavior Tree with Memory



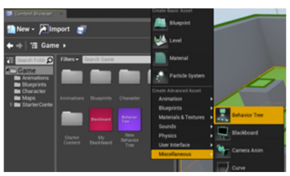
In UE4, the "Memory" is called "Blackboard"

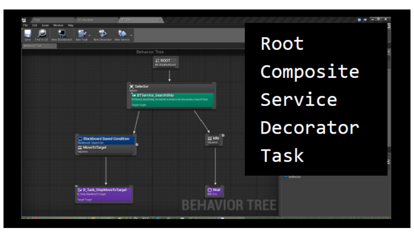
UE4 Behavior Trees vs. Traditional

- 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

https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html

UE4 Behavior Tree





(Describe each next)

UE4 Behavior Tree - Root

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

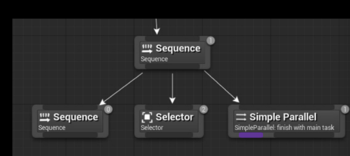
UE4 Behavior Tree - Composite

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

(Describe each next)

Composite - Sequence

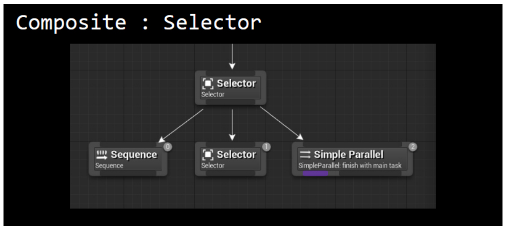
Composite : Sequence



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"

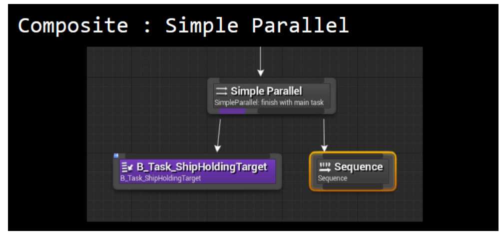
Composite - Selector



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.

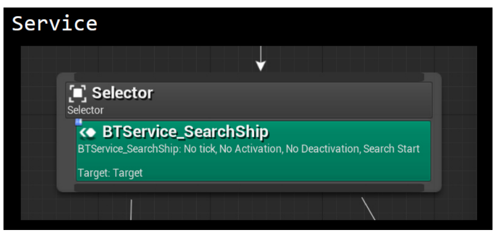
"Or"

Composite – Simple Parallel



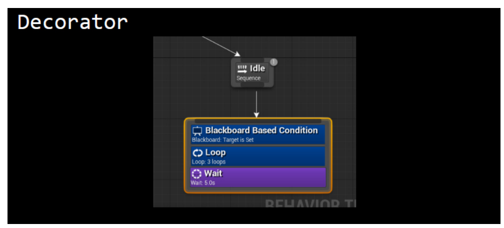
Composite : Simple Parallel
- The Simple Parallel node allows a single main task node to be executed along side of a full tree. When the main task finishes, the setting in Finish Mode dictates the secondary tree.

Service



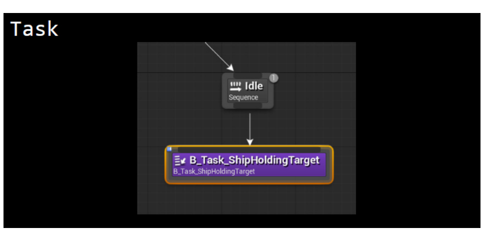
Service
- These attach to Composite nodes, and will execute at their defined frequency. These are often used to make checks and to update the Blackboard. These take the place of traditional Parallel nodes.

Decorators



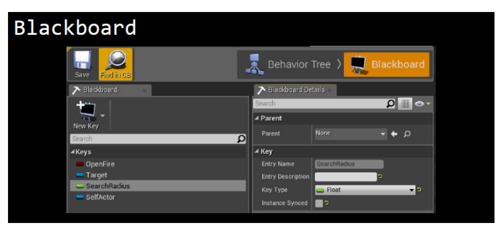
Decorator
- Also known as conditionals. These attach to another node and make decisions on whether or not a branch in the tree, or even single node, can be executed.

Task



Task
- Theres are leaves of the tree, the nodes that "do" things.

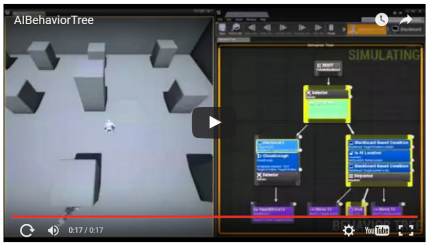
Blackboard (Memory)



Blackboard
- 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 pawn, shared by squad.

Blackboard : Why use?
- To make efficient event-driven behaviors
- To cache calculations
- As a scratch-pad for behaviors
- To centralize data

UE4 Behavior Tree Quick Start



https://youtu.be/q6vTg2roI6k

"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
    http://www.cs.wpi.edu/~imgd4000/d16/slides/imgd3000-fsm.pdf
  - Header files
    http://dragonfly.wpi.edu/include/classStateMachine.html
- UE4 Behavior Tree
  - Difference between BT and DT
    http://gamedev.stackexchange.com/questions/51693/decision-tree-vs-behavior-tree
  - Quick Start
    https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/