DM810

Computer Game Programming II: AI

Lecture 9
Decision Making

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Behavior Trees
2. Fuzzy Logic

Outline

1. Behavior Trees

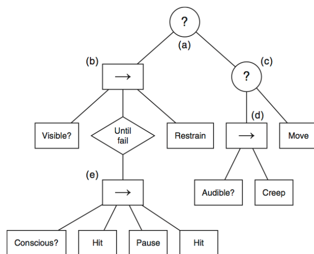
2. Fuzzy Logic

Decorators

- The decorator pattern is a class that wraps another class, modifying its behavior (from object-oriented software engineering).
- Composite that has one single child task and modifies its behavior in some way.

Like filters that:

- limit the number of times a task can be run (eg, does not insist with some action)
- keep running a task until it fails
- negation



Combination:

```
ex = Selector(
          Sequence(
                 Visible,
                 UntilFail(Sequence(Conscious,Hit,Pause,Hit)),
                 Restrain
                 ),
          Selector(Sequence(Audible,Creep),Move)
         )
```

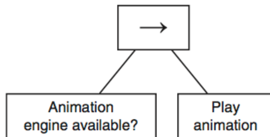
Resource Limitation

Limitations on resources: Examples:

- animation engine can only play one animation on each part of the skeleton at any time.
- no more than one audio sample per character at a time

Hence we need to check availability of resource:

1. By hard-coding the test in the behavior
2. By creating a Condition task to perform the test and using a Sequence
3. By using a Decorator to guard the resource



Which solution is this?
What is the problem with it?

Decorators solution

- Semaphores are a mechanism for ensuring that a limited resource is not over subscribed.
- can cope with resources that aren't limited to one single user at a time.
- before using the resource, a piece of code must ask the semaphore if it can acquire it.
- when the code is done it should notify the semaphore that it can be released
- Most programming languages have libraries for semaphores removing the need to deal with low-level operating system primitives for locking.
- The Decorator returns its failure status code when it cannot acquire the semaphore. A select task higher up the tree will find a different action

Independent from the resource (animation engine, a health-station, or a pathfinding pool):

```
class Semaphore:
    def Semaphore(maximum_users)
    def acquire()
    def release()
```

```
class SemaphoreGuard (Decorator):
    semaphore
    def SemaphoreGuard(semaphore):
        this.semaphore = semaphore
    def run():
        if semaphore.acquire()
            result = child.run()
            semaphore.release()
            return result
        else:
            return False
```

Concurrency, Timing

- So far we assumed only one task runs at a time.
- How do behavior trees work with respect to subsequent frames? how will it know what to do? Should we restart from the top of the tree every time?

1. Concurrency: each behavior tree is running in its own thread. An Action can take seconds to carry out: the thread just sleeps while it is happening and wakes again to return True back to whatever task was above it in the tree.
2. Since it can be highly wasteful to run lots of threads at the same time even on multi-core machines we need also cooperative multitasking and scheduling algorithms

Parallel tasks

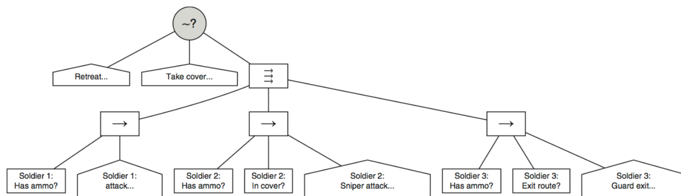
- Parallel task similar to the Sequence task.
- it has a set of child tasks, and runs them until one of them fails, in which case it return failure.
  If all child tasks complete successfully, the Parallel task returns with success.
- children are run simultaneously
- when one thread fail all other threads are terminated.
- tasks need to be able to receive a termination message and clean up themselves after termination

```
class Parallel (Task):
  children
  runningChildren
  result
  def run():
    result = undefined
    for child in children:
      thread = new Thread()
      thread.start(runChild, child)
    while result == undefined:
      sleep()
    return result
```

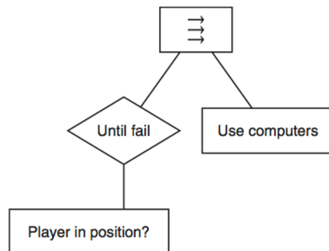
```
def runChild(child):
  runningChildren.add(child)
  returned = child.run()
  runningChildren.remove(child)
  if returned == False:
    terminate() # terminate all
    result = False
  else if runningChildren.length == 0:
    result = True

def terminate():
  for child in runningChildren:
    child.terminate()
```

parallel tasks, application example

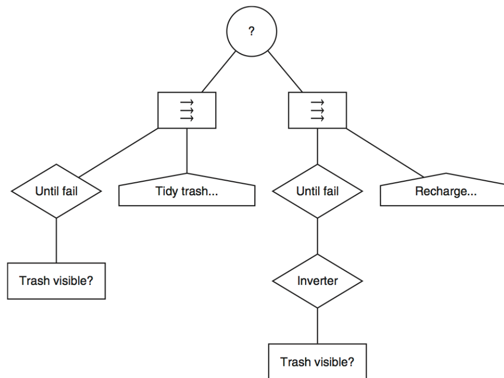


Using Parallel blocks to make sure that Conditions hold



Using Parallel blocks to make sure that Conditions hold

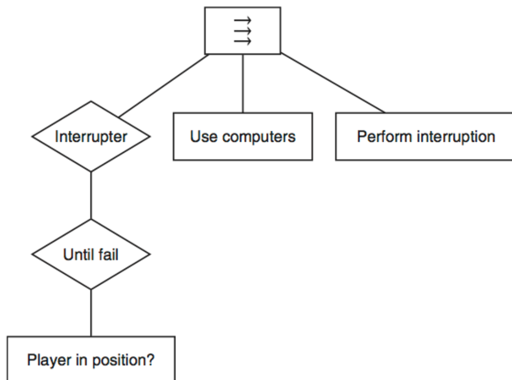
achieve state machines and ability to switch tasks when events occur



- but unnatural: events cause a change of action, rather than the lack of the event allows the lack of a change of action.
- state-based behaviors are hard to model!
  eg: a character who needs to respond to external events
  eg: interrupting a patrol route to go into hiding or to raise an alarm

Inter-behavior communication

Interrupter decorators



We want data to pass between behavior trees, but we dont want data into tasks as parameters to their run method. Else each task needs to know what arguments its child tasks take and how to find these data.

Data

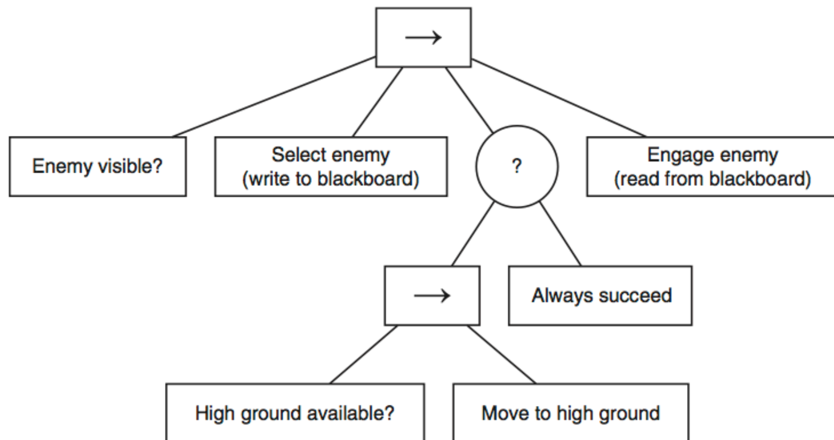
- Decouple the data that behaviors need from the tasks themselves.
- `blackboard`: external data store for all the data that the behavior tree needs. Tasks can then query the `blackboard` for data.
- we can write tasks that are still independent of one another but can communicate when needed.
- we can still handle data privacy by introducing hierarchies of `blackboards`. Tasks that are roots of Subtrees create their own `blackboards`
- tasks can communicate by writing and reading from the `blackboard` rather than calling methods.

Implementation Issues

A. Construction
   Levels of abstraction

   1. classes abstract concepts about how to achieve some task we saw in pseudo-code.
   2. instances of these classes arranged in a behavior tree.
   3. the behavior tree needs is instantiated for a particular character at a particular time.

   Use a cloning operation to instantiate trees for characters.
   use behavior tree as an "archetype"; Any time we need an instance of that behavior tree we take a copy of the archetype and use the copy; each task has a clone method that makes a copy of itself.

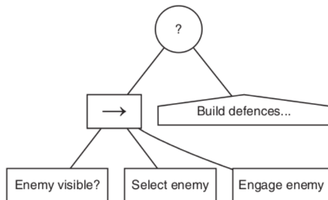


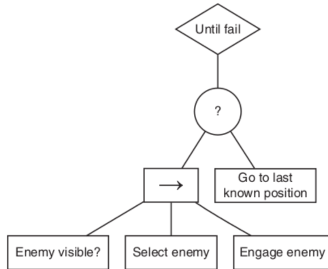
The tasks should be written so that, if the blackboard had no target, then the task fails, and the behavior tree can look for something else to do.

B. Reuse behavior trees for multiple characters

```
Enemy Character (goon):
model = ‘‘enemy34.model’’
texture = ‘‘enemy34-urban.tex’’
weapon = pistol-4
behavior = goon-behavior
```

C. use sub-trees multiple times in different contexts





store partial sub-trees in the behavior tree library

Outline

1. Behavior Trees

2. Fuzzy Logic

Fuzzy Logic

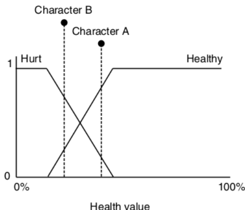
- So far decisions based on `true` and `false`.
- Fuzzy logic includes a range of degrees for decisions.
- It is popular in games to represent any kind of uncertainty **but** any method based on probability theory (ie, probabilistic graphical models) is always better: in any kind of betting game, any player who is not basing their decisions on probability theory can expect to eventually lose his money. Flaws in any other theory of uncertainty, besides probability theory, can potentially be exploited by an opponent.

Fuzzy sets

- predicates: sentence about the world are true or false
- classical sets: either belongs to the set or not
- fuzzy sets: everything can partially belong to the set according to a numeric value called degree of membership.
  sets without sharp boundaries
- in fuzzy logic predicate have a value.
- degree of membership: $[0, .., 255] \subset \mathbf{N}$ or $[0, 1] \subset \mathbf{R}$
- everything can be a member of multiple sets at same time

Fuzzification

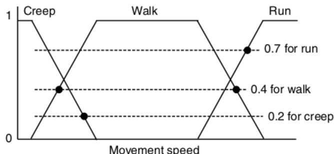
- fuzzification: turning regular data into degrees of membership
- Eg: membership function: a function that maps the input value (hit points) to a degree of membership, for each fuzzy set.



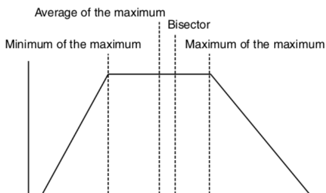
- Infinite number of different membership functions
- their values don't need to add up to 1, although in most cases it is convenient if they do.
- for Boolean values and enumerations: pre-determined membership values for each relevant set.

Defuzzification

- Defuzzification: turning a set of membership values into a single output value.



1. Highest Membership: Eg: 0.7 $\rightsquigarrow$ run + precomputed value:

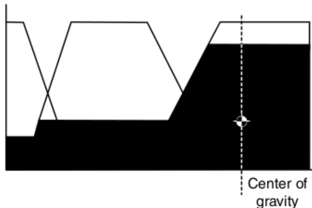


$\rightsquigarrow$ 0 creep, 0 walk, 1 run $\equiv$ 0.33 creep, 0.33 walk, 0.34 run.

2. Blending Based on Membership:
   eg: 0.33 creep, 0.33 walk, 0.34 run $\mapsto$ (0.33$\times$ characteristic creep speed) + (0.33$\times$ characteristic walk speed) + (0.34$\times$ characteristic run speed).

- blend of minima: Smallest of Maximum, or Left of Maximum (LM).
  blend of the maxima: Largest of Maximum (LM), Right of Maximum.
  blend of the average values: Mean of Maximum (MoM).
- allows predefined values
- often the preferred approach because quick to calculate

3. Center of Gravity: crop functions, and calculate center of mass (requires integration)



IEEE version doesn't crop each function before calculating its center of gravity and so can be precomputed ⇝ blending

Defuzzification to a Boolean Value: cut-off value

Defuzzification to an Enumerated Value:
if ordered: rangification and then just look at where the value falls
if not ordered: one fuzzy set for each alternative and the one with the highest value gets assigned.

Fuzzy Logic Reasoning

It might be partially raining (membership of 0.5) and slightly cold (membership of 0.2).

What is the value of the compound statement such as "it is raining AND cold"?

AND				$m_{(A \text{ AND } B)} = \min\{m_A, m_B\}$
OR				$m_{(A \text{ OR } B)} = \max\{m_A, m_B\}$
NOT				$m_{\text{NOT } (A)} = 1 - m_A$
XOR	NOT(B)	AND	A	$m_{(A \text{ XOR } B)} = \min\{m_A, 1 - m_B\}$
	NOT(A)	AND	B	$m_{(A \text{ OR } B)} = \min\{1 - m_A, m_B\}$

(corresponds to first order logic when $m_A = \{0, 1\}$ and $m_B = \{0, 1\}$)

It may be reasonable but also not: if $T(Tall(Marco)) = 0.4$ then $T(Tall(Marco) \wedge \neg(Tall(Marco)) = 0.4$

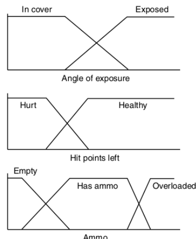
Fuzzy Rules

from known membership of certain fuzzy sets
to membership values for other fuzzy sets:

$$m_{\text{(should brake)}} = \min\{m_{\text{(close to the corner)}}, m_{\text{(traveling fast)}}\}$$

Fuzzy Control – Block Format Rules

- also used to build industrial controllers that take action based on a set of inputs
- it can be used to determine if transitions in a state machine should fire, or the activity at the states
- based on AND rules



input 1 state AND ... AND input n state THEN out state

Hurt AND In cover AND Empty THEN brake
Healthy AND Exposed AND Overloaded THEN accelerate
...

needs rules for each combination of inputs.
Here 12 rules (2 x 2 x 3).

- For each rule calculate degree of membership for output state taking the minimum degree of membership for input states in that rule (AND).
- Output: maximum output from any of the applicable rules.

Example

with only two inputs:

corner-entry AND going-fast THEN brake
corner-exit AND going-fast THEN accelerate
corner-entry AND going-slow THEN accelerate
corner-exit AND going-slow THEN accelerate

We might have the following degrees of membership:

Corner-entry: 0.1
Corner-exit: 0.9
Going-fast: 0.4
Going-slow: 0.6

Then the results from each rule are

$$\begin{aligned} \text{Brake} &= \min(0.1, 0.4) = 0.1 \\ \text{Accelerate} &= \min(0.9, 0.4) = 0.4 \\ \text{Accelerate} &= \min(0.1, 0.6) = 0.1 \\ \text{Accelerate} &= \min(0.9, 0.6) = 0.6 \end{aligned}$$

then take max per state (0.1;0.6), and defuzzificay or keep numerical value.

Complexity? How can we speed up the computation procedure?

Fuzzy Control – Combs

Alternative fuzzy control method based on rules of the form:

a AND b ENTAILS c

a AND b ENTAILS c $\equiv$ (a ENTAILS c) OR (b ENTAILS c) $\equiv$ IF a THEN c
 IF b THEN c

IF a_1 AND . . . AND a_n THEN c

IF a_1 THEN c
.
.
.
IF a_n THEN c.

From having rules involving all possible combinations of states to a simple set of rules with only one state in the IF-clause and one in the THEN-clause

but

IF corner-entry AND going-fast THEN brake
IF corner-exit AND going-fast THEN accelerate

IF corner-entry THEN brake
IF going-fast THEN brake
IF corner-exit THEN accelerate
IF going-fast THEN accelerate

This is an inconsistent set of rules $\rightsquigarrow$ one rule can be decomposed, more than one rule cannot

One has to take care of defining only consistent rules
more restrictive but less heavy

corner-entry AND going-fast THEN brake
corner-exit AND going-fast THEN accelerate
corner-entry AND going-slow THEN accelerate
corner-exit AND going-slow THEN accelerate

could be expressed as:

corner-entry THEN brake
corner-exit THEN accelerate
going-fast THEN brake
going-slow THEN accelerate

With inputs of:

Corner-entry: 0.1
Corner-exit: 0.9
Going-fast: 0.4
Going-slow: 0.6

the block format rules give us results of:

Brake = 0.1
Accelerate = 0.6

while Combs method gives us:

Brake = 0.4
Accelerate = 0.9