345 Ludic Computing

Lecture 11

Behaviour Trees

Simon Colton & Alison Pease
Computational Creativity Group
Department of Computing
Imperial College London

ccg.doc.ic.ac.uk

http://www.youtube.com/watch?v=N04gXtW2xEM

Controlling NPCs

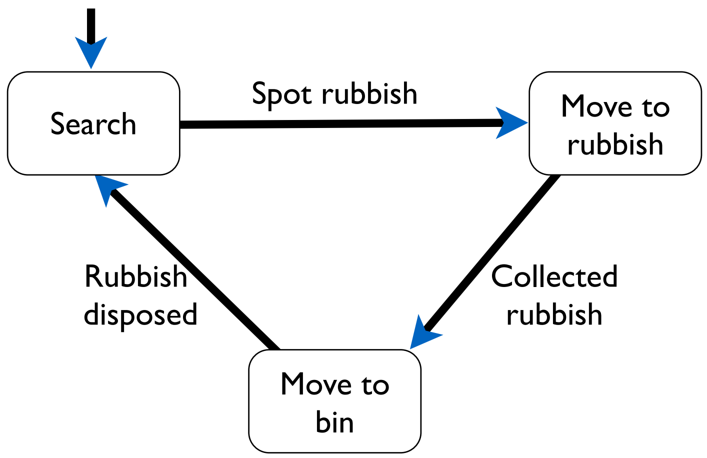
- NPC AI can be critical to a game: often the reason players can "see behind the curtain"
- Range of solutions suitable for different contexts
  - Finite State Machines, e.g. Quake
  - Hierarchical Concurrent State Machines, e.g. Left for Dead
  - AI planners, e.g. F.E.A.R.
  - Behaviour Trees, e.g. Halo 2
  - + many others
- Multi-level solutions are common (e.g. global, agent, animation)

This Lecture

- Brief look at state machines
- Behaviour trees
  - Basic concepts
  - Non-determinism and concurrency
  - Decorators
  - Behaviour blackboards

NPC State Machines

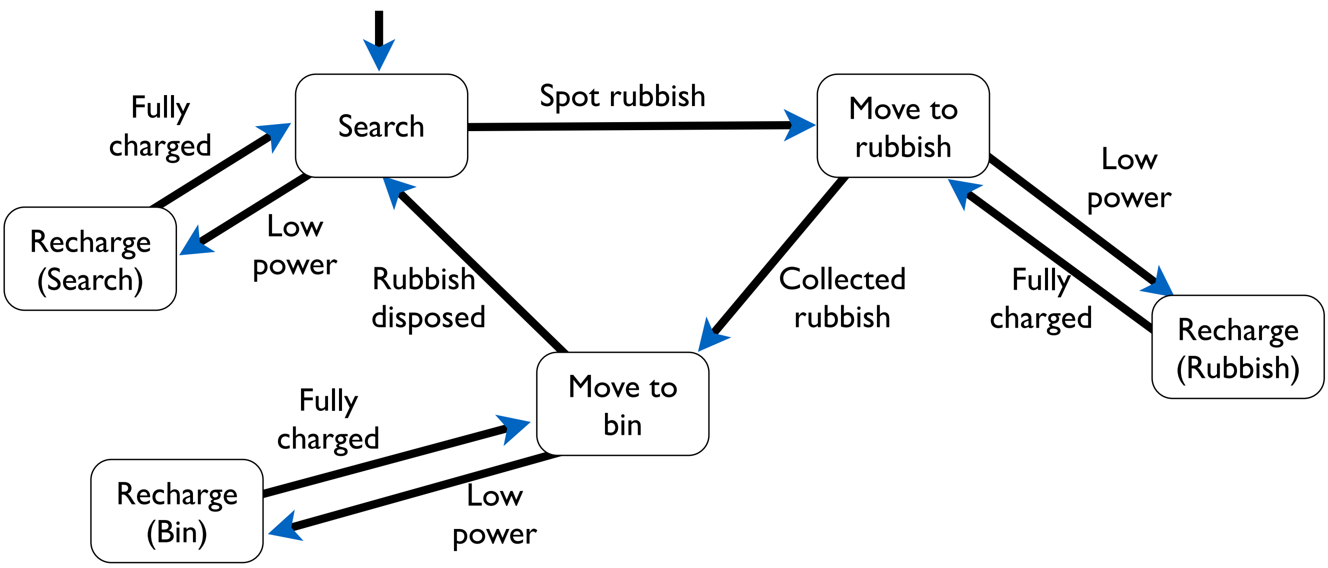
Example: Cleaner Robot



Events trigger transitions between states

NPC State Machines

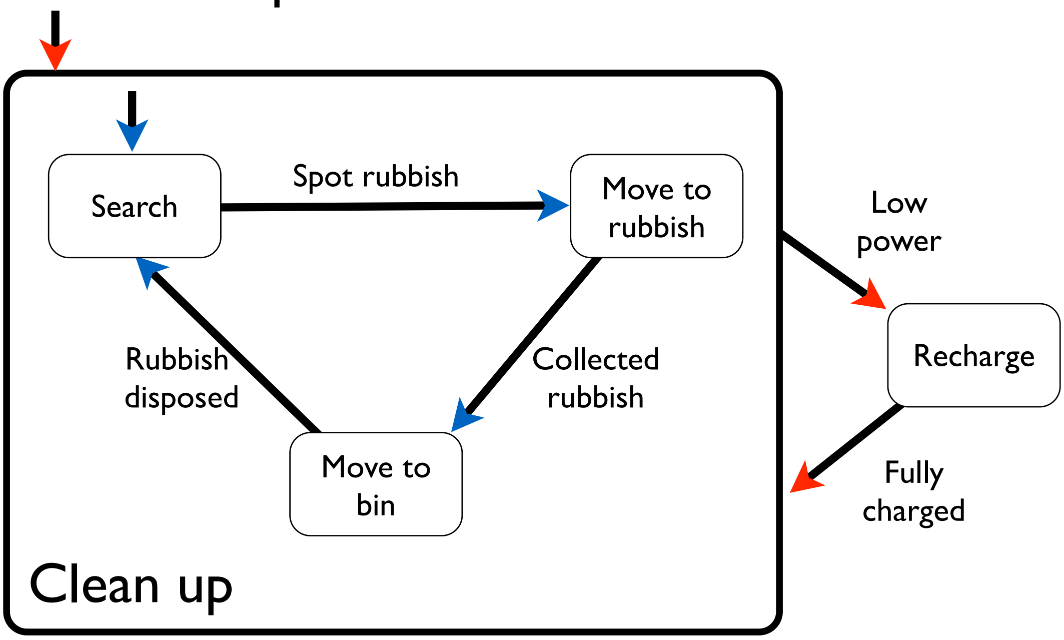
Example: Cleaner Robot



Behaviour code quickly becomes unmanageable

NPC State Machines

Example: Cleaner Robot



Hierarchical machines handle complexity better

NPC State Machines

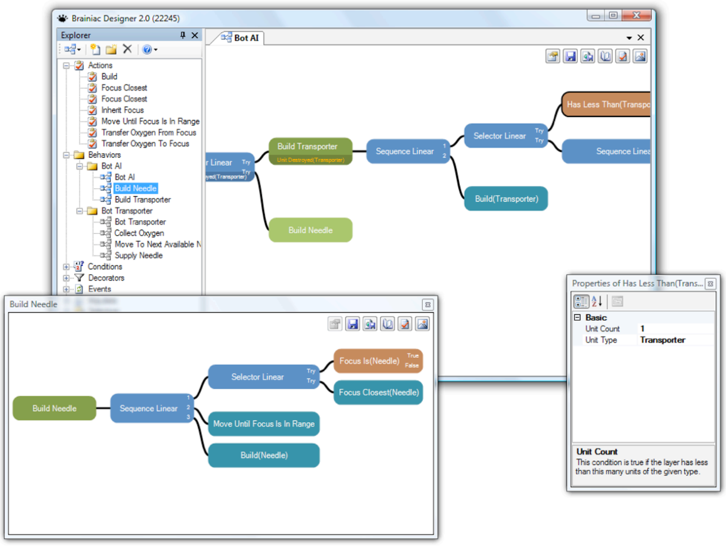
- Hierarchical machines help tame "transition spaghetti" for complex behaviours
  - But can still be hard to maintain/reuse
  - State/event-oriented, hard to design goal-oriented behaviour
- Still an important tool in game AI, especially combined with other AI techniques
- Adding concurrency and non-determinism can increase sophistication/realism

Behaviour Trees

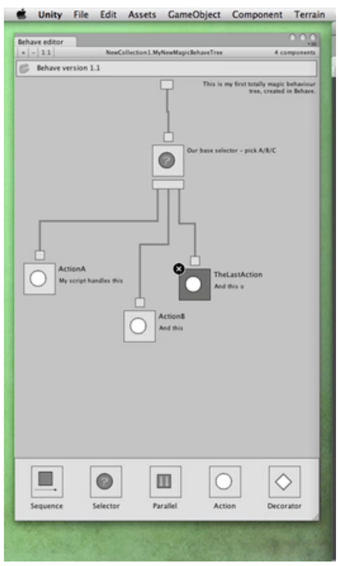
- Programming idiom for Game AI
  - Task-oriented rather than state-oriented
  - Modular, reusable behaviours
  - Can easily be built up into hierarchies of increasingly complex behaviours
- Used in *Halo 2* (2004), *Spore* (2008), *Grand Theft Auto* series + many others

Behaviour Trees

Enable designers (often non-programmers) to easily manage NPC behaviour with a GUI editor



Brainiac editor



Behave for Unity

Basic Node Types

- Leaf nodes
  - Conditions, e.g. "is the player visible?"
  - Actions, e.g. "attack the player"
- Composite nodes (with 2 or more subtrees)
  - Selector ("a or else b")
  - Sequence ("a and then b")
- When a node is executed, it passes `succeed` or `fail` back up to its parent node

Basic Node Types
Conditions

- Test some property of the game world
  - Proximity
  - Line of sight
  - Object state
  - Character state, e.g. health > 50
- The test returns `succeed` or `fail`

Basic Node Types

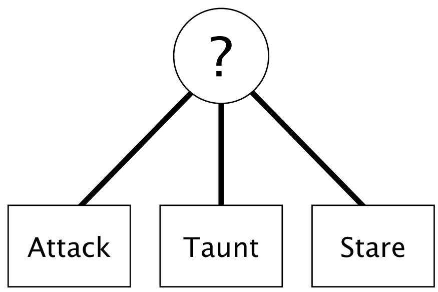
Actions

- Change the state of the game world
  - Perform animations or play audio
  - Character state, e.g. resting increases health
  - Engage the player
- Can use specialised code, e.g. pathfinding
- Normally will `succeed`, but can `fail`
  - Best to catch the failure cases with a conditional

Basic Node Types

Selectors

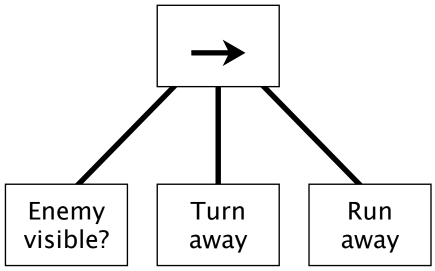
- Tries each child tree in turn
- Stops and returns `success` when a child succeeds
- Returns `fail` if none succeed



Basic Node Types

Sequences

- Tries each child tree in turn
- Stops and returns `fail` when a child fails
- Returns `success` if none fail



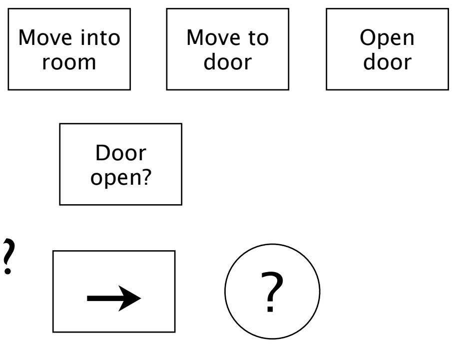
Example

Entering a Room

If the door is open, then move into the room. If the door is not open, then move to the door, open the door, and move into the room.

Ask yourself:

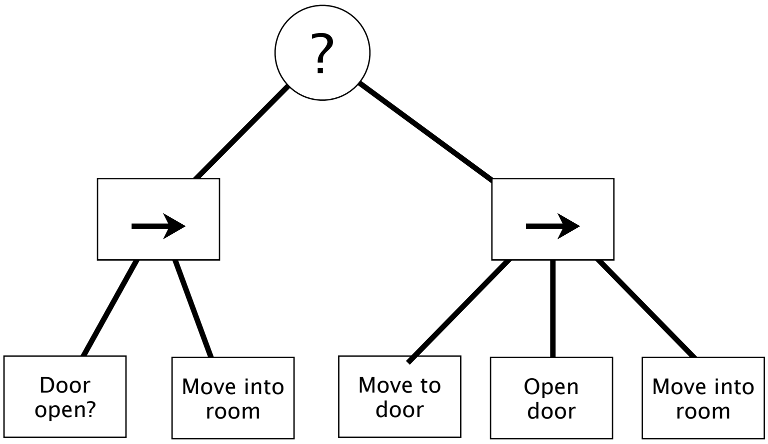
- What are the actions?
- What are the conditions?
- How are these combined?



Example

Entering a Room

If the door is open, then move into the room. If the door is not open, then move to the door, open the door, and move into the room.



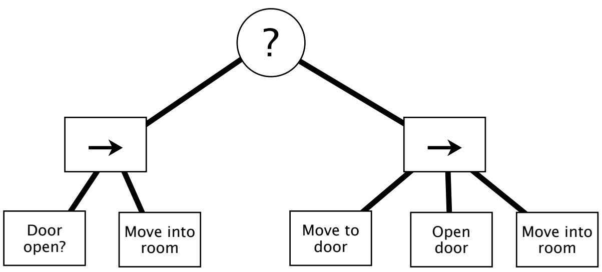
Example

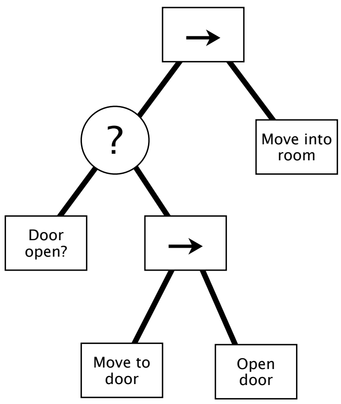
Entering a Room

```
if door.isOpen()
    move_to(room)
else
    move_to(door)
    open(door)
    move_to(room)
```

Refactor

```
if !door.isOpen()
    move_to(door)
    open(door)
move_to(room)
```

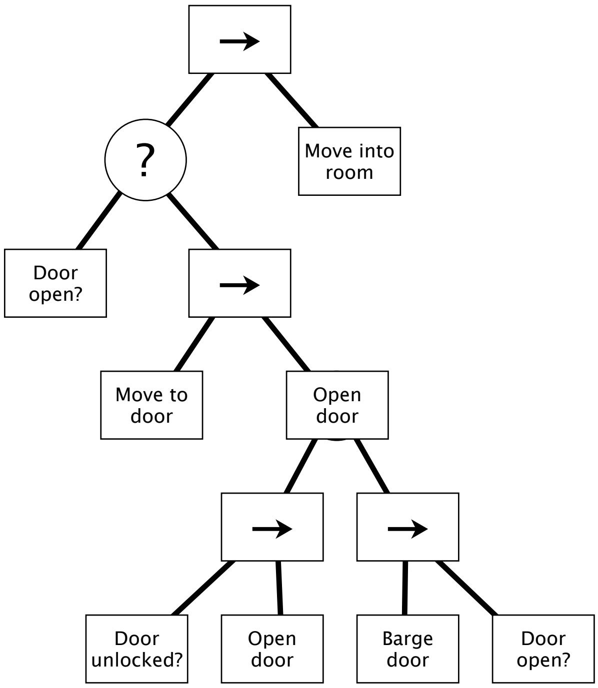




Example

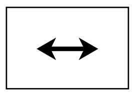
Entering a Locked Room

If the door is open, move into the room. If the door is not open, check whether it is locked. If it is not locked, then open the door and move into the room. If the door is locked, then barge the door open, and move into the room

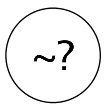


Non-Determinism

- We don't want NPCs to be predictable
- *ND-Selector*: choose between children in random order

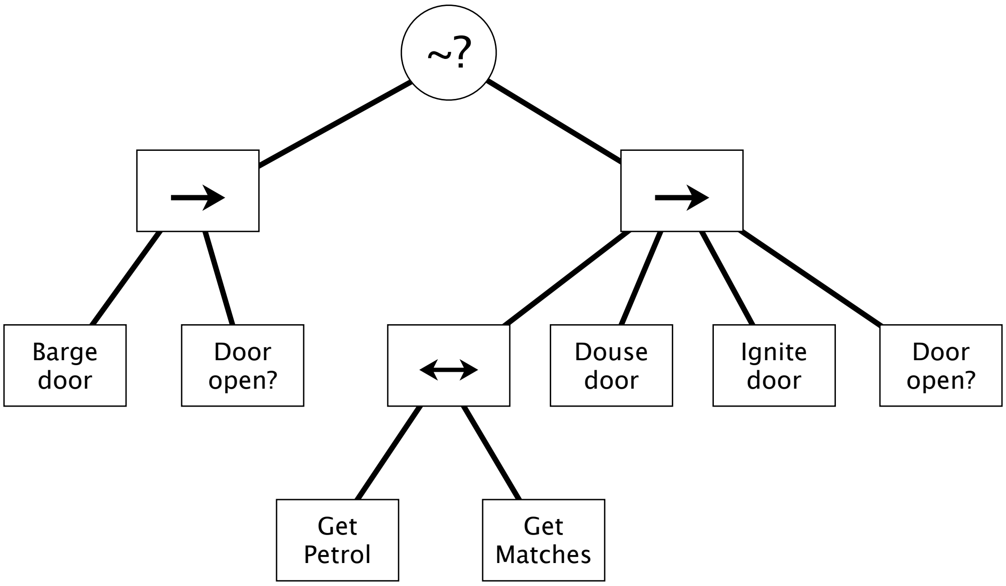


- *ND-Sequence*: carry out children in random order



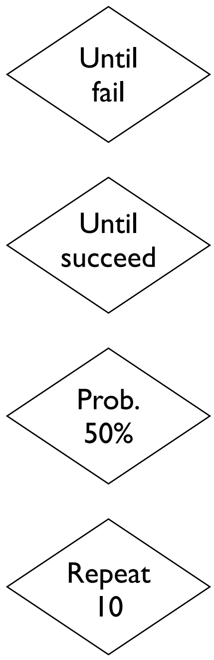
Non-Determinism

Example: Destroy the Door



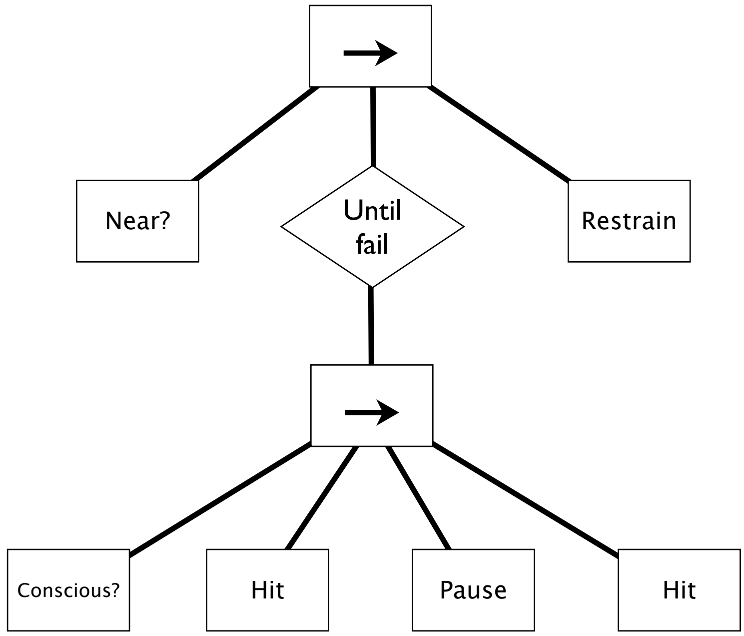
Decorators

- Nodes which modify a single child
- Many possible modifications...
  - Carry out child until it succeeds or fails
  - Timer
  - Execute with probability
  - Repeat N times
  - Invert success and failure
  - ...



Decorators

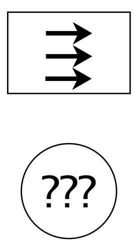
Until Fail Example



Concurrency

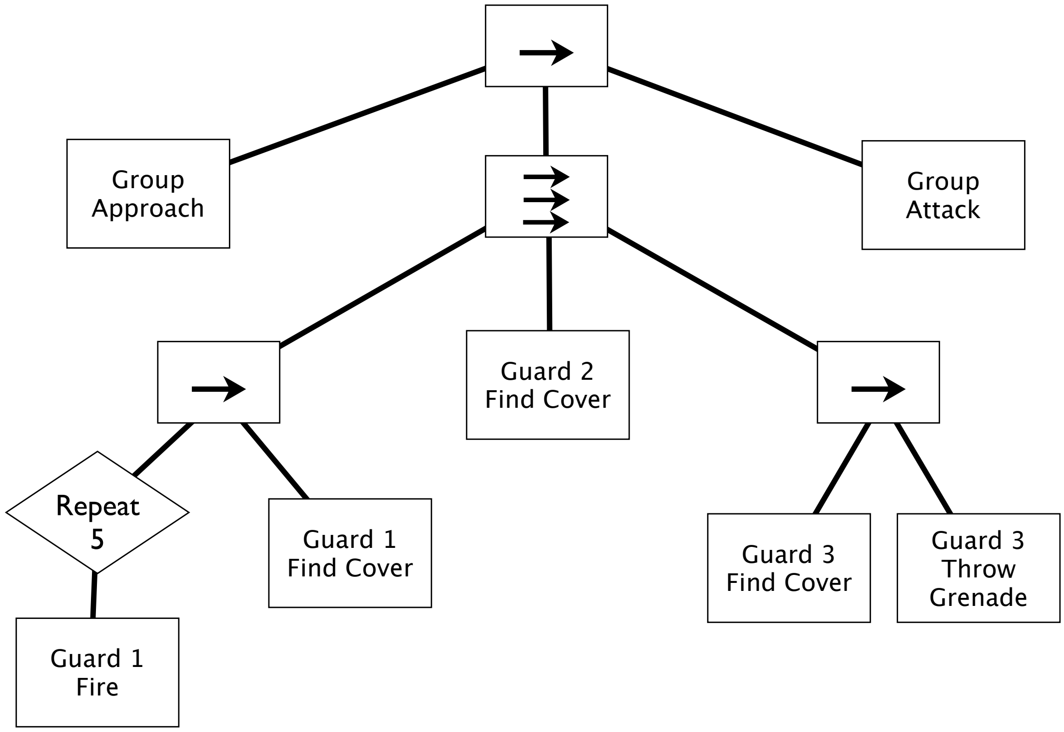
Parallel Composite Nodes

- Behaviours often need be performed concurrently
- Adds flexibility and realism (talking & walking)
- *Parallel sequence*: try all children concurrently and succeed when all have succeeded. When one fails, request other children terminate, then fail
- *Parallel selector*: try all children concurrently and succeed when one has succeeds. Fail when all failed
- We have to worry about thread efficiency and safety issues. In particular, in-game actions which conflict



Concurrency

Group AI Example



Concurrency

Interrupters

- Concurrent behaviours may go on indefinitely
- But we may want to stop some/all of these in response to the state of just one of them
- Wrap each concurrent child in *interrupter* decorator
- This is referenced by an *interrupt* action, which can requests the interrupter terminates a child's behaviour

Concurrency

Game Resources

- Concurrent behaviours may be want to use the same limited resources
  - **Computing** resources, e.g. too many NPCs requiring pathfinding solutions will overload the processor
  - **In-game** resources, e.g. too many NPCs docking at the same health station will look very odd
  - **Multimedia** resources, e.g., too many noises at the same time
- Can add conditions to check current usage to each task
  - Relies on designer to include relevant checks, difficult to manage for large trees

Concurrency

Semaphore Guards

- A more elegant solution is to employ **semaphores**
- These keep a tally of the number of resources available and which tasks are using them
- Each behaviour must ask a semaphore whether it can use the given resource
  - Wrap behaviour in *semaphore guard* decorator
  - Returns `failure` when the semaphore says no, in which case another behaviour can be tried

Using Data

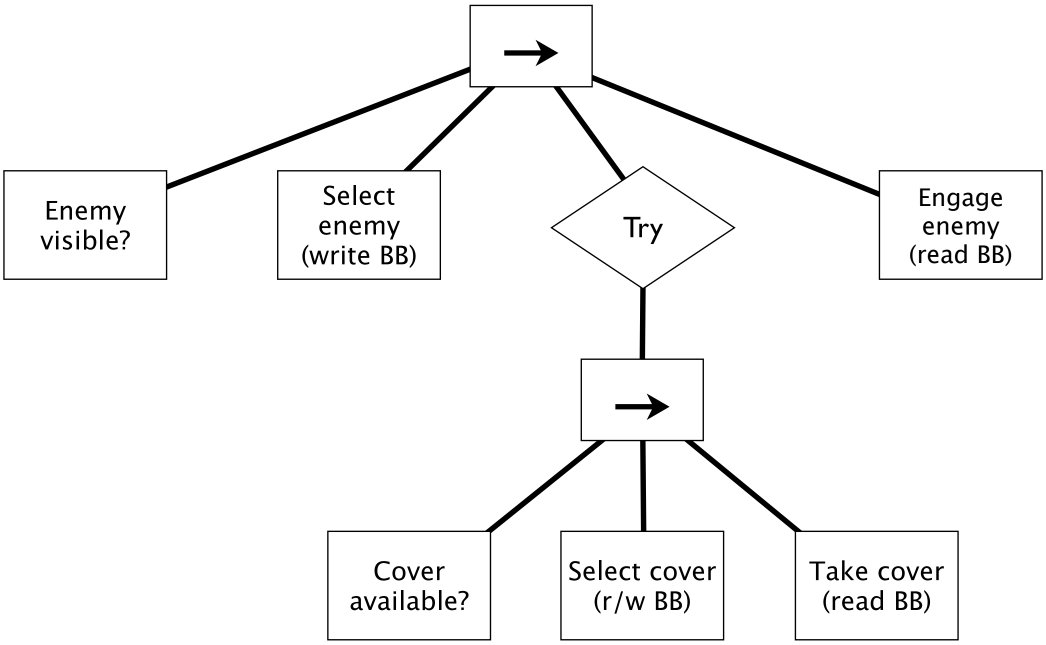
- Behaviours need to retain and exchange information
- Details about the choices made (e.g. target this enemy) and information discovered so far (e.g. good cover spots)
- Idea is that the overall behaviour remains the same, but is modified to fit the given data
- Want to avoid behaviours with huge number of parameters, as this breaks the clean programming interface
  - Although small number of typed parameters is fine

Behaviour Blackboards

- **Blackboard architectures** are standard AI technique for avoiding excessive parameterisation
- Each task has the ability to write to the blackboard and to read from it anything that has been written by other tasks
- Design choices...
  - Global blackboard *and/or* per-behaviour blackboards *and/or* blackboards spawned by sub-behaviours
  - Which blackboard takes priority? One possible solution: passing the blackboards down the tree, look at the most specific blackboard

Behaviour Blackboards

Example: Storing Selected Enemy



Reusing Behaviours

- One of the benefits of behaviour trees is reuse of (sub)behaviours with very little extra coding
- Maintain library of trees/subtrees
- Can dynamically instantiate subtrees to control agent if/when behaviours are required
  - Saves memory, e.g. memory constrained platforms, or when you have 1000s of NPCs

Limitations

- Difficult (but not impossible) to build behaviour trees which are quickly reactive
  - Dynamic nature of games mean one behaviour has to be aborted midway-through in favour of another one
  - Can get around this with interrupter decorators, but that is cumbersome
- Possible solution: combine with a state machine approach, with each state having one or more behaviour trees

Summary

- Behaviour trees are a conceptually simple and powerful solution to programming game agents
  - Accessible to designers
- More sophisticated behaviours with decorators, non-determinism, concurrency, blackboards
- Currently popular in commercial game AI — though many other techniques are used (state machines, planners, priority systems, sensory systems, ...)
  - Custom hybrid approaches often used

Reading & Resources

(Optional)

- Ian Millington & John Funge (2009) *AI for Games*, 2nd edition. **Pages 334-371 on BTs**
- Damian Isla (2005) *Handling Complexity in the Halo 2 AI*, GDC 2005. [http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml]
- http://aigamedev.com