

# Game Playing Techniques

CE811 Game Artificial Intelligence

---

Michael Fairbank

11th October 2021

University of Essex

# Overview

## Making Decisions

- Aiming for simplicity

- Decision Trees

- Finite State Machines

- Behaviour Trees

- Goal Oriented Behaviour

## Knowledge Representation

- Blackboards for Intra-Agent Coordination

- Blackboards for Inter-Agent Coordination

- Canonical Blackboard Architecture

## Wrap-up

# Outline

## Making Decisions

- Aiming for simplicity

- Decision Trees

- Finite State Machines

- Behaviour Trees

- Goal Oriented Behaviour

## Knowledge Representation

- Blackboards for Intra-Agent Coordination

- Blackboards for Inter-Agent Coordination

- Canonical Blackboard Architecture

- Wrap-up

# Outline

## Making Decisions

- Aiming for simplicity

- Decision Trees

- Finite State Machines

- Behaviour Trees

- Goal Oriented Behaviour

## Knowledge Representation

- Blackboards for Intra-Agent Coordination

- Blackboards for Inter-Agent Coordination

- Canonical Blackboard Architecture

## Wrap-up

# The basic concept

Given some observations about the world, what's the best thing to do?

- State observations can be everything (chess, ie. fully observable) or limited (poker, or an obscured view in a first-person shooter game; ie. “partially observable”)
- And what does “best” mean in this context?

- We could use complex decision making algorithms
- ...or we could fake it.
- Guess which one games often do...

# Simplistic can work well

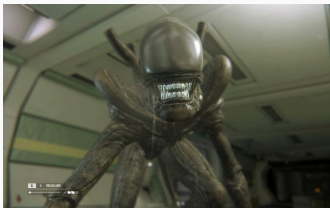


Figure 1: **Alien Isolation** is an example of a game which uses behaviour trees.

- Alien Isolation makes use of behaviour trees to control the alien
- Initial reviewers first thought the NPCs were showing incredible levels of intelligence
- Parts of the tree are turned on and off depending on what happens
- Providing the AI isn't actively doing something stupid, you can get away with a lot.

- See Chapter 2 of Millington and Funge textbook, the “Complexity Fallacy” and “Hacks”:
  - “A simple random number generator applied judiciously can produce a lot of believability”



# Why use simple systems?

- Simple systems are cheap to execute
  - Note that games have lots of non-AI tasks also competing for CPU time
- Simple systems give the designers a lot of control
- ... but they are very time consuming to construct
- They can get stuck if a player does something unexpected

# Basic Implementations



Figure 2: Controllers for characters are often very simplistic ...

# Outline

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up

# Decision Trees

A decision tree maps knowledge to decisions.

- In games, they can be used to allow an NPC to choose an “intelligent” action, based on their knowledge of the current game state.
- The decision trees are usually made by the designer/programmer; but Decision Trees can be learnt from experience too (next lecture)

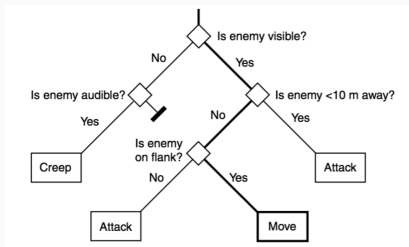


Figure 3: A simple decision tree for dealing with enemies

# Decision Trees

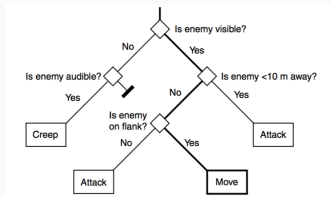


Figure 4: A simple decision tree for dealing with enemies

- The tree is constructed of decision nodes and leaf nodes
- Decisions are usually simple
  - Checking a single value, or a single condition
- Can be given access to more complex operations
  - Pathfinding, distances, visibility checks
- The leaf nodes represent actions to perform
  - The same action can be placed on different branches

# Decision Trees

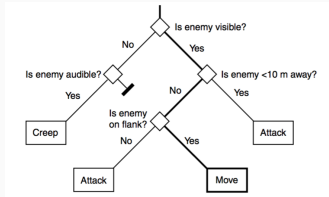


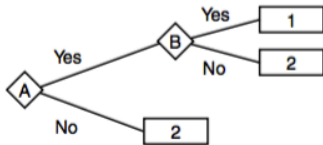
Figure 5: A simple decision tree for dealing with enemies

- Each decision needs at least two children - it's not a decision if you've only got one option!
- Decisions are made based on internal knowledge of the agent
- Could also query a centralised manager for information

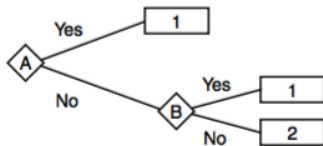
# Decision Trees: and/or

You can combine decisions to make AND and OR clauses:

If A AND B then action 1, otherwise action 2

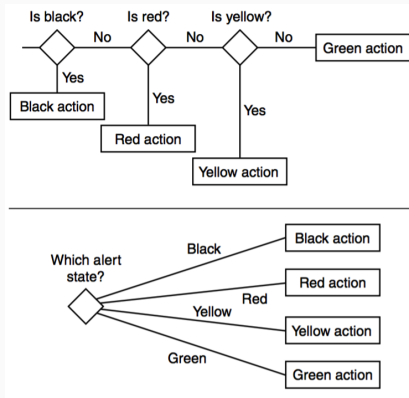


If A OR B then action 1, otherwise action 2



# Decision Trees: Binary vs N-ary

You can have more than 2 branches for a node, (aka n-ary):



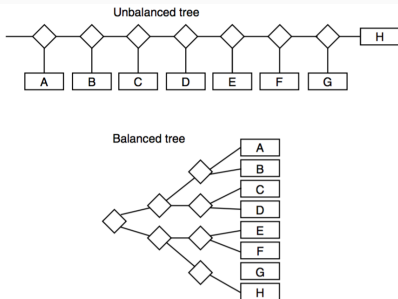
- These two diagrams are equivalent

- Fewer comparisons (in the lower diagram) are potentially more efficient than lots of comparisons
- In practice, the efficiency gain is lost at the code level
  - The compiler might internally change a switch statement into a series of binary ifs
- Binary trees are more common and can be easily optimized



# Decision Trees: Balancing

You can reduce the number of comparisons by making sure the tree is balanced.

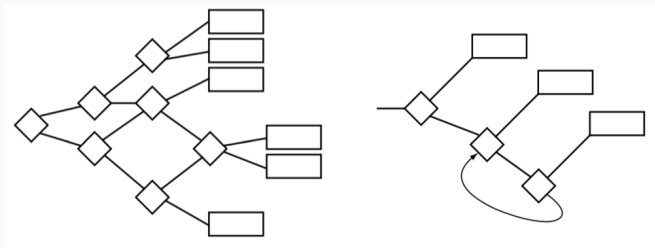


- A balanced tree can reach a leaf node in  $O(\log_2(n))$  time
- A totally unbalanced tree can reach a leaf node in  $O(n)$  time

1. Balanced trees has roughly the same number of nodes on each branch
2. The balanced tree only needs to check a small subset of the nodes
3. The unbalanced tree needs to check against every node
4. This means that in the unbalanced case the tree is growing proportionally to the number of decisions

# Decision Trees: Graphs

You can create a Directed Acyclic Graph (DAG) by allowing certain nodes in the tree to be reached by different branches:

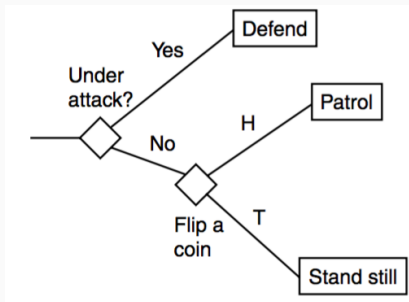


The left one is a valid DAG. The right one isn't - why?

1. Allowing nodes to link to a parent nodes creates cycles
2. Cycles mean that the code can produce infinite loops - which we don't want
3. The QA department will not be happy if you do...

# Decision Trees: Random Decisions

You can create a Random Decision Tree by introducing a random element into the decision making process.



What happens if we execute this every frame?

1. The agent will flip between partroling and standing still at random
2. This might still be useful to create varience in the agent's behaviour
3. Again, might want to avoid the QA department...

# Maintaining state

This flickering behaviour motivates what we need to try next:  
Recording the state an agent is in.

- This leads to “state machines”.

# Outline

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up



- State machines turn up everywhere in computer science
- Why would games be any different?
- Until behaviour trees became more mainstream, they were used extensively in game AI.
- State machines are commonly called “Finite State Machines” (FSMs) in Game AI.
  - FSM can mean something different in other areas of CS.

1. They're still used fairly extensively for animations

# Finite State Machines

- In a state machine (FSM) each character occupies a state.
- While in that state it will continue carrying out the same action.
- States are linked by transitions with conditions.

# Finite State Machines

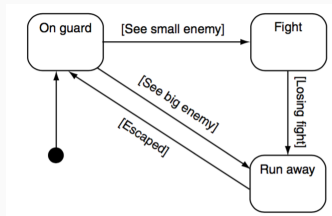


Figure 6: A simple FSM for a guard

A FSM consists of:

- An initial state (where you start)
- A (finite) set of states, while in the same state you keep doing the same thing.
- A (finite) set of transitions that control how you move from one state into another

1. In general, decisions are deterministic - you don't have a choice when you see a big enemy, you run away
2. Transitions between states are governed by conditions that can be triggered by external events
3. Very well studied topic with decades of research behind them

# A FSM for a guard

Example FSM created for game Phoneline Cincinatti 2, by Essex 2021 graduate, George Carpenter:



Figure 7: Guards reacting to line-of-sight to player

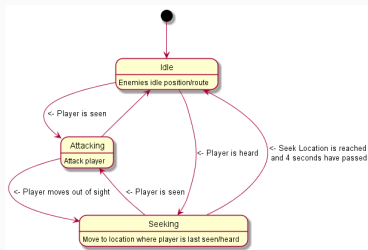


Figure 8: State diagram

See the [Gameplay video](#) on YouTube (Warning: Gore)

# A FSM for a guard: Phoneline Cincinatti 2



```
if (CollisionMaths.lineOfSightClear(player.position.cpy(), mob.position.cpy().add(mob.body.head.getPosition())))  
    if (canReachPlayer()) {  
        currentState=Following;  
    } else {  
        currentState=MovingTo;  
    }  
    if (currentState==MovingTo) {  
        pathfinder.moveTo(mob, player.position.cpy());  
    }  
} else {  
    if (currentState == MovingTo || currentState==Following) {  
        currentState=Tracking;  
        pathfinder.track(mob, player.position.cpy());  
    } else if (currentState==Tracking && !pathfinder.isTracking()) {  
        currentState=Returning;  
    } else if (currentState==Returning && !pathfinder.isReturning()) {  
        currentState=Patrolling;  
    }  
}
```

# A possible problem

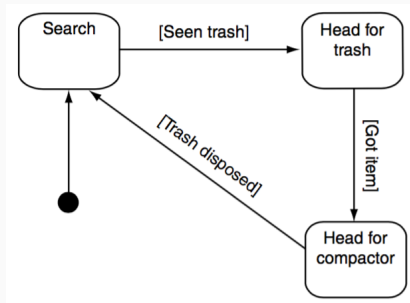


Figure 9: A simple rubbish robot

- FSMs are fine when all of the states share the same context
- Under Normal Conditions the robot wanders around cleaning up
- But what happens when the robot starts to run out of power?





Figure 10: A simple rubbish robot (also, cat taxi)

# A possible solution

- We could add states representing recharging to every state
- These can't be the same state because we want it to go back to whatever it was doing before

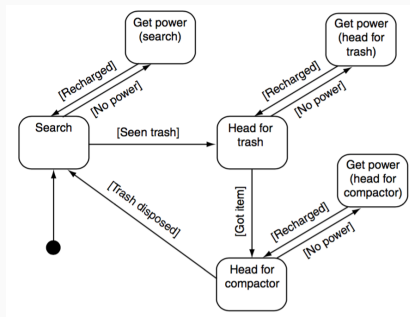


Figure 11: A not-so-simple rubbish robot

- By adding new states to compensate for when we run out of power we can solve the power problem!
- But we end up doubling the number of states
- But at least this solves the problem right?

## Another possible problem

- The problem is essentially one of memory - we want to remember where we came from and go back to that state
- Any new context switch will cause us to need even more states.
- For games think shooting, running, swimming, talking, defending
- The solution doesn't scale well.

# Hierarchical State Machines

Instead, we can remember where we came from, grouping states into a higher-level state.

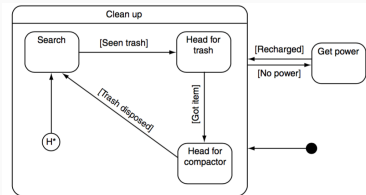


Figure 12: A HFSM for the robot

- There is a high-level starting state (black dot)
- Each high-level state has a starting state ( $H^*$ )
- Transitions happen normally within each high-level state
- low-level states can trigger transfers to another high-level state.

1. Helps to slow down the growth of states
2. There isn't a need to know how the other state works internally

# FSM/DT Hybrids

You can combine FSMs and Decision Trees into a single structure:

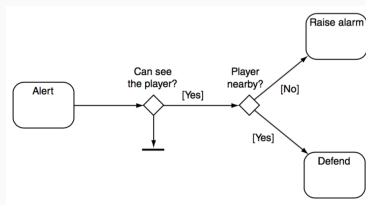


Figure 13: A FSM/DT hybrid

- Transitions are replaced with decision trees
- Actions from the decision trees are FSM states (e.g. “Raise Alarm”)

# Outline

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up

Behaviour trees have become state-of-the-art for game AI in recent years.

- One of the first popular games to use them was Halo 2
- Behaviour Trees can be built using tools which means that they can be more accessible to non-programmers.



# Behaviour Trees: Tasks

- The main building block of a behaviour tree is a task
- A task is an activity that returns a value.
  - Simplest case: returns success or failure
  - Can also return a more complex outcome (enumeration, range)
    - E.g. look up the value of a variable
    - Execute an animation (returns success or failure)
- Tasks should be broken down into smaller complete (independent) actions.

# Behaviour Trees: Task Composability

- Tasks are composed into sub-trees to perform more complex actions.
  - These complex actions can themselves be composed into higher-level behaviours.
- It is this composability that gives decision trees their power.
- Even more powerful when combined with a GUI editor
  - allowing designers to rapidly create complex AI behaviour

# Tasks Types

A basic behaviour tree consists of 3 types of Task:

Conditions: test some property of the game

- Do we have at least 40 health?
- Is player is nearby?

Actions: alter the state of the game; they usually succeed

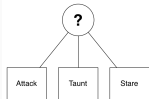
- Break down the door
- Heal the player

Composites: collections of child tasks (conditions, actions, composites)

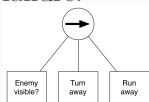
# Composite Tasks

Composite tasks can be divided into two types:

**Selector:** Runs children in order, and returns success as soon as one child behaviour succeeds. If all fail then the selector node returns failure.



**Sequence:** Runs children in order, and returns success only if all child nodes have succeeded. As soon as one child fails, the sequence aborts, and the Sequence node returns failure.



Question



What do these correspond to in terms of logical operations?

1. selector is or, sequence is and

# Order of Actions

- Actions are always executed in the order they are defined
  - We can use this to define priorities
  - If an action depends on another a fixed order is necessary
  - But this makes our characters predictable (and possibly boring)

# Order of Actions

- Partial Ordering gives us the best of both worlds - we can mix fixed and random ordering
- To do this we'll need two new composites:
  - Random selector 
  - Random sequence 
- These are just like “Selector” and “Sequence” (from the previous slide), but now the child nodes are chosen in a random order, instead of left-to-right order.

1. If you have tasks A, B and C that are independent you can add more variety by allowing them to be executed in any order
2. you can see from this that the design of behaviours does not only have to consider functionality (i.e., getting the job done) but also gameplay experience.



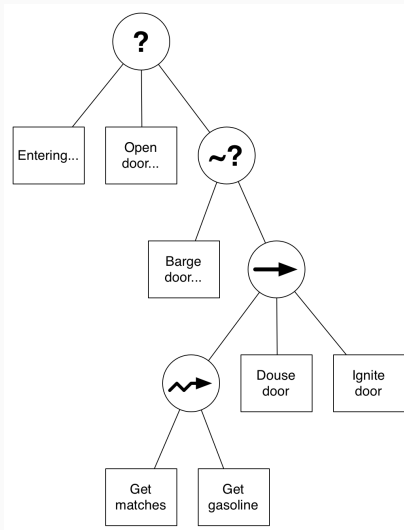
# Partial Ordering Example

Random selector :

Chooses tasks in random order, until one of them has succeeded. Returns failure only if all fail.

Random sequence :

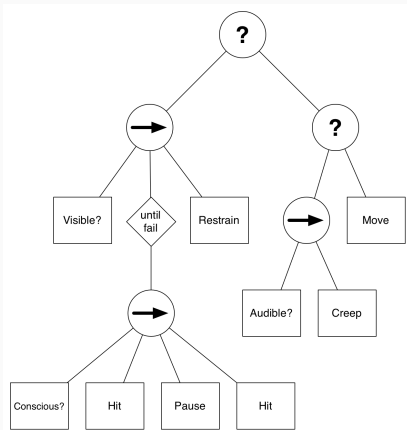
Chooses tasks in random order, until all of them have succeeded. Returns failure as soon as one fails.




# Decorators

- A decorator is a task that has a single child node, and modifies the behaviour of that child node in some way.
  - E.g. It may make the child node run multiple times
  - Or it may limit the number of times the child node can run.
- You can use them as filters; to dictate if and how a task can run
- For instance, we can repeatedly execute a task until some goal has been achieved.
- Or, you could use them to invert the return type of an action.  
(What logical operand does this correspond to?)

# Decorators



- Here the decorator is 
  - It runs the child node repeatedly.
    - Until the child node fails.
    - and then returns “success”
- Hence a visible enemy is hit repeatedly, until unconscious; and then tied up!

# Decorators: Resource Management

- Often in games we have constraints on resources.
  - E.g. a character can utter only one sound at a time
  - A character's hand can only make one pre-recorded motion at a time
  - CPU resources might limit the number of path-finding algorithms concurrently running.
- There are a number of possible ways we could address these constraints:
  - Hard-Code limits in our AI system
  - Use conditions and Sequence nodes
  - Use decorators to guard other nodes

# Decorators: Resource Management


- Using decorators is a **general** approach that allows the designer to not **care** about implementation details of resource management.
- The decorator implements a **semaphore**<sup>2</sup>
  - A semaphore works by keeping a tally of the number of resources there are available
  - 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.”
- the Decorator returns a failure code if it fails to acquire the semaphore.

---

<sup>2</sup>[https://en.wikipedia.org/wiki/Semaphore\\_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))

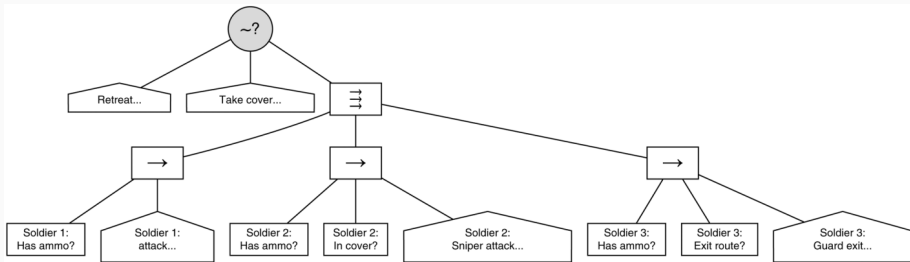
# Concurrency

- The above behaviour trees often include actions that may take many animation frames to complete
  - E.g. battering the enemy to unconsciousness
- Hence we need to be able to run them concurrently with the game animation
  - And concurrently with other behaviour trees that are running.
- We could execute each behaviour tree in its own thread
- We need to take the time needed to execute an action into account.
  - Wait for a thread to finish executing (ie, sleep)
  - Timeout tasks that take too long
  - Don't execute an action for some period of time after completion

- Concurrency can also be used for another composite task:
  - Parallel task 
  - Same as sequence task, but runs all tasks in parallel.
  - Terminates all tasks if one of them fails (and returns failure).
  - Returns success if all of them complete with success.

# Parallel

- Parallel has multiple uses
  - Execute multiple actions simultaneously such as falling and shouting
  - Control a group of characters: combine individual and group behaviours



3

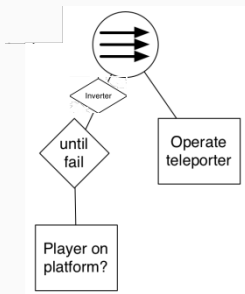
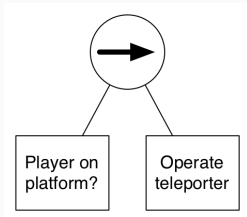
---

<sup>3</sup>Image from Millington and Funge



- We can also use parallel for continuous condition checking

4



Q: What is the difference between these two trees?

---

<sup>4</sup>In this example, the “Inverter” decorator flips the output of the Until Fail decorator from Success to Fail.

1. Tree on the left will only check that the player is on the platform once before trying to teleport them
2. Tree on the right will keep checking the platform, even while in the middle of teleporting the player. As soon as the player ceases to be on the platform, then the Until Fail will return “success”, and hence the inverter will return “fail”, and hence parallel node will terminate all child nodes, and hence teleportation will be aborted mid-way through.

# Blackboards

- We often need shared data between nodes in our tree.
  - e.g. “Go to door 27”, followed by “Open door number 27”
  - Here the data that needs passing is “door 27”
- The best approach is to decouple the data from the actions...
- (If we didn’t decouple the data here (i.e. the door number) then we’d need one sub-tree creating for every door in the game!)
  - To accomplish this, nodes have access to a shared data store known as a blackboard (more later)
  - It’s used to store arbitrary data between nodes
  - Often, sub-trees will have their own blackboards

- Behaviour trees, if well designed are quite modular
- It makes sense to allow trees (or parts of trees) to be reused
- (Especially if this is a large game you are making)
- This can be implemented in a number of ways:
  - Cloning
  - A library of behaviour trees
  - Storing and referencing trees in a key-value store (ie, a map)

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up

# Goal Directed Behaviour

- Up to this point we've talked about hard-coded behaviours
- Wouldn't it be better if our characters could figure out what to do for themselves?
- To do this, we'll need to give them Goals

# Key Concepts

Some important concepts for goal-driven agents:

goals: The character can have one or multiple goals (motives) to accomplish. Not all of them need to be active at the same time, and they most likely will have different priorities.

insistence: How important this goal is to the agent

actions: The things we can do to try and accomplish our goals

utility: How much effect does an action have on achieving our goal.

# An example

Our agent is hungry.

goal: minimise hunger.

insistence: goes up with time having not eaten

actions: eat (direct), pre-heat oven (indirect)



# Simple Selection

Choose the action which has the highest utility for the highest insistence goal.

Goal: Eat (insistence: 4)

Goal: Sleep (insistence: 3)

Action	Sleep	Eat
Get Food	0	-3
Get Snack	0	-2
Sleep bed	-4	0
Sleep sofa	-2	0

- The agent would clearly choose:
  - The goal with the highest insistence: “Eat”
  - The action with the best utility for addressing that chosen insistence: “Get food”
- Simple and fast approach which gives reasonable results

# Simple Selection: side effects

Problems start to occur when actions have side-effects...

Goals:

- Eat (insistence: 4)
- Bathroom (insistence: 3)
- Dealing with the most urgent action may interfere with other goals
- Related problem: **Sussman Anomaly**

Action	Eat	Bathroom
Drink Soda	-2	+3
Use bathroom	0	-4

It's better to use Overall Utility

# Overall Utility

- The concept of Discontent can be used to address this problem.
- The objective is to minimise discontent.
- Calculate the discontent from the resulting state:
  - Sum the insistence of all goals
  - or better, sum the squares of the goals
- In our example:
  - After ‘drink’, we would have insistences:  
Eat = 2, Bathroom = 6, so discontent (sum: 8, squared: 40)
  - After ‘visit bathroom’, we would have insistences:  
Eat = 4, Bathroom = 0, so discontent (sum: 4, squared: 16)
- So we should pick ‘visit bathroom’

- We also need to take into account the amount of time it takes to do something
- Be careful, dealing with durative actions can make things complicated...
- There are few different things we could try
  - Preference for longer or shorter actions
  - Incorporate into the discontent value
  - Incorporate into the utility value
- there are multiple ways in which timing can affect decision making:
  - The time taken to complete an action
  - The time needed to start an action (ie, walking to the location)
  - The rate at which insistence changes

# Timing Example

- Goals
  - Eat (Insistence: 4, +4 per hour).
  - Bathroom (Insistence: 3, +2 per hour).
- Eat Snack
  - (Utility: Eat - 2; Time: 15 minutes).
  - If taken: Eat = 2, Bathroom = 3.5, Discontentment (sq): 21.25
- Eat Main Meal
  - (Utility: Eat - 4; Time: 1 hour)
  - If taken: Eat = 0, Bathroom = 5, Discontentment (sq): 25
- Visit Bathroom
  - (Utility: Bathroom - 4; Time: 15 minutes)
  - If taken: Eat = 5, Bathroom = 0, Discontentment (sq): 25

# Planning

How can we maximise Utility by choosing actions, when choosing the first action might make more actions become available?

- Example: opening the door requires that you unlock the door
- This means Goal-Oriented Behaviour is potentially much more complicated than our initial simple examples.
- We need to consider how actions can be chained together in order, and imagine what will happen several steps into the future.
- We're starting to stray into the area of planning ...

Planning is a topic in AI which involves consideration of many possible combinations of future actions

- and choosing the combination which best produces a desired outcome.

- There has not been much attempt to include classical planning techniques into games.
- There are two fairly large successes:
  - Goal Oriented Action Planning (GOAP)
  - Hierarchical Task Networks (HTN)

- Goal-Oriented Action Planning (GOAP) is an attempt to bring old-school planning into games
- GOAP generates successive states and looks for ones which meet our goal
- Take the sequence of actions which minimises discontent
- Advantages: It's simple, and it works
- Caveats:
  - We still don't know When actions become available
  - Most games have very large numbers of possible actions (ie, branching factor) - far too many to fully expand.



- Availability of actions
  - We can use a simulation of the world to see what becomes available during the execution (ie, a model).
  - We would then know what is possible and when
- Number of actions
  - We use an algorithm to identify promising solutions
  - e.g., MCTS,
  - A\*,
  - Depth First Search

# Games using GOAP i

- F.E.A.R. (X360/PS3/PC) - Monolith Productions/VU Games, 2005
- Condemned: Criminal Origins (X360/PC) - Monolith Productions, 2005
- S.T.A.L.K.E.R.: Shadow of Chernobyl (PC) - GSC Game World, 2007
- Mushroom Men: The Spore Wars (Wii) - Red Fly Studio, 2008
- Ghostbusters (Wii) - Red Fly Studio, 2008
- Silent Hill: Homecoming (X360/PS3) - Double Helix Games, 2008
- Fallout 3 (X360/PS3/PC) - Bethesda Softworks, 2008
- Empire: Total War (PC) - Creative Assembly/SEGA, 2009
- F.E.A.R. 2: Project Origin (X360/PS3/PC) - Monolith Production- s/Warner Bros, 2009

- Demigod (PC) - Gas Powered Games/Stardock, 2009
- Just Cause 2 (PC/X360/PS3) - Avalanche Studios/Eidos, 2010
- Transformers: War for Cybertron (PC/X360/PS3) - Activision, 2010
- Trapped Dead (PC) - Headup Games, 2011

# Outline

## Making Decisions

- Aiming for simplicity

- Decision Trees

- Finite State Machines

- Behaviour Trees

- Goal Oriented Behaviour

## Knowledge Representation

- Blackboards for Intra-Agent Coordination

- Blackboards for Inter-Agent Coordination

- Canonical Blackboard Architecture

- Wrap-up

# Knowledge Representation

Knowledge representation: Representing information about the world in a form that an agent can utilize it to solve complex tasks.

We represent data:

- Symbolic: Specific values (speed=15.0, temperature=30, name="John")
- Fuzzy: Fuzzy values (speed='fast', temperature='high')

This data might refer to:

- the world: pathfinding (navigational mesh), positioning systems, line of sight information.
- internal: facts that the agent knows about the world and other agents.

There are multiple ways of representing this information:

- Simple: arrays, dictionaries / hash maps.
- Working memory.
- Blackboards.

# Working Memories

A working memory is memory responsible for holding temporary information while it's processed. It stores a set of facts that are true to the agent.

It contains:

Long-term memory: long term facts (I was attacked by this player, this agent is my friend).

Short-term memory: volatile facts (there's an enemy behind that door, there's no food in the fridge).

Working memories function as an internal model of the world, which allows the agent to draw beliefs (which might be or not accurate with respect to the real world).

# Outline

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up

# Blackboards for Intra-Agent Coordination

- Blackboard: A publicly read/writeable information display.
- The blackboard is a tool that helps the exchange of static-typed data between agents and components.
- It's a list of logical assertions upon which other components operate.
- The blackboard can distinguish its contents by grouping them according to the nature of the data: facts, goals or beliefs.



# Advantages and Disadvantages

## Advantages:

- Provide a simple overview to the agent's state.
- Possible place to store agent's global state variables.
- It's modular.

## Disadvantages:

- Scales poorly for multiple agents.
- Centralized data.
- Hard to keep track of its usage: what's modified by whom and when?

# Blackboards for Intra-Agent Coordination

- Blackboards are very useful systems for task coordination.
- It is simple to implement and it provides a shared space to break a problem into subproblems and tackle it incrementally.

5

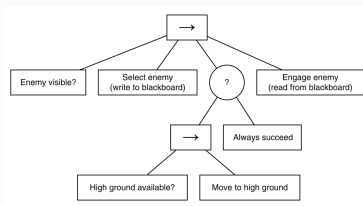


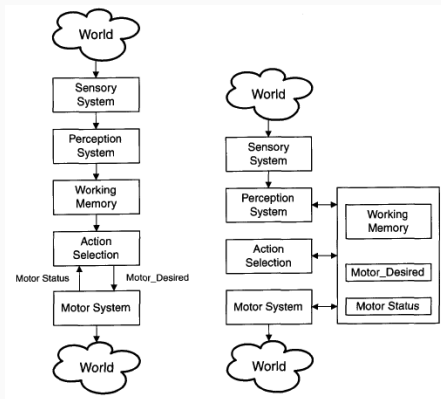
Figure 5.36: A behavior tree communicating via blackboard

---

<sup>5</sup>Image from Millington and Funge

# Blackboards for Intra-Agent Coordination

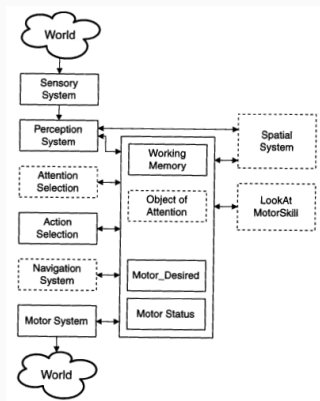
Blackboards can be used for communication between the different components within the same agent:



Equivalent systems, with and without a blackboard. What advantage has the blackboard version over the other one?

# Blackboards for Intra-Agent Coordination

The main advantage is its flexibility: adding new components becomes easier because it doesn't modify the previous architecture. The other systems don't get affected by the additions: the data flow is not disrupted.



# Outline

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up

# Blackboards for Inter-Agent Coordination i

- A single blackboard could be shared by a group of NPCs, to coordinate their activity as a team.
- E.g. The team of NPCs use a blackboard to hold some shared objective information:
  - “Our current mission is: destroy the bridge”

## Making Decisions

Aiming for simplicity

Decision Trees

Finite State Machines

Behaviour Trees

Goal Oriented Behaviour

## Knowledge Representation

Blackboards for Intra-Agent Coordination

Blackboards for Inter-Agent Coordination

Canonical Blackboard Architecture

## Wrap-up

Blackboards have a history in AI research. The Canonical Blackboard Architecture consists of three components:

- The Blackboard
- Knowledge Sources (KSs): Set of components that are able to operate on the information held by the blackboard.
- Arbiter: In case of conflicts, the Arbiter decides which KSs should operate. Different strategies are possible, and they may take into account the relevance of each KS, and their contribution towards goals or sub-goals.

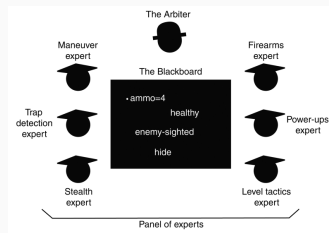
In our previous examples, the KSs were the components in the behaviour tree, or different agents talking to each other.



# Canonical Blackboard Architecture

In another usage, we can form an advanced expert system by combining several simpler “experts”

- In this usage the experts write to the blackboard. They write information, and issue priorities.
- An arbiter decides which expert to listen to next (in case of conflicting advice).



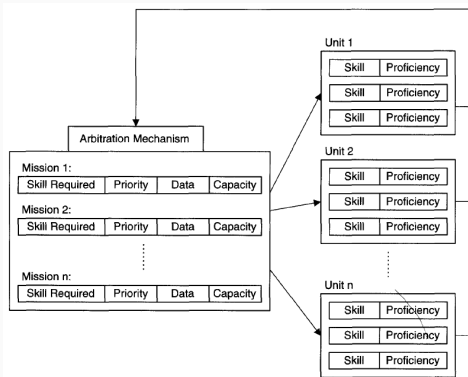
6

---

<sup>6</sup>Image from Millington and Funge, Ch. 5

# Blackboards for Inter-Agent Coordination i

Multiple configurations are possible. Example:



Each unit is given a set of skills (KS's action) and proficiencies at them, which can be executed on demand. The results of most actions are new postings on the blackboard, or modifications to the existing ones.

The blackboard contents are open missions. The arbitration mechanism decides which unit takes care of which mission, which requires a skill name, priority level, capacity and data (that the unit will interpret to carry out the mission). Details about how the missions are performed are not relevant.

# Outline

## Making Decisions

- Aiming for simplicity

- Decision Trees

- Finite State Machines

- Behaviour Trees

- Goal Oriented Behaviour

## Knowledge Representation

- Blackboards for Intra-Agent Coordination

- Blackboards for Inter-Agent Coordination

- Canonical Blackboard Architecture

## Wrap-up

# Lecture Quizzes

- Check the lecture quizzes on Moodle.
  - Just 1 this week, on today's lecture.
- These are to help you absorb the lecture information more easily
- Remember, they contribute to the course total mark. Each quiz has a deadline of 9 days after its corresponding lecture date.
- Email me about bits you are stuck on/ any bugs.

# This week's Lab (Next Monday morning)

- We will be programming with The Resistance python framework
- To make this lab run smoothly, in advance of this lab, please go through the “Familiarisation” sections (1-1.2) of the [CE811 Moodle instructions for Labs 1-3](#).
- This includes:
  1. Learning the game rules (watch the youtube “tabletop” example game, 20mins),
  2. Learning about the python framework (watch the tutorial video, 20mins) before this week’s lab.
  3. Setting up your Python3 environment on your home computer
  4. Checking you can run the competition engine at home, e.g. with the command “py competition.py 1000 bots/beginners.py”

# Acknowledgements

Acknowledgements to the previous lecturers of this module:

- Joseph Walton-Rivers
- Diego Perez-Liebana

Plus, most of the materials of this lecture are based on:

- Ian Millington and John Funge, Artificial Intelligence for Games, Ed. Morgan Kaufmann (2009). Chapter 5.
- Alex Champandard's web/course on Game AI
  - Alex Champandard created The Resistance Python framework we use in our labs
- Damian Isla and Bruce Blumberg, Blackboard Architectures (AI Game Programming Wisdom, Rabin 2002).

# Further Reading

From the Ian Millington and John Funge, *Artificial Intelligence for Games*, textbook (2nd Edition, 2009), we have:

- Section 5.2: Decision Trees
- Section 5.3: Finite State Machines
- Section 5.4: Behaviour Trees
- Section 5.7: Goal-Oriented Behaviour
- Section 5.9: Blackboard Architectures



Questions?