GAM250: Advanced Games Programming

8: AI

Learning outcomes

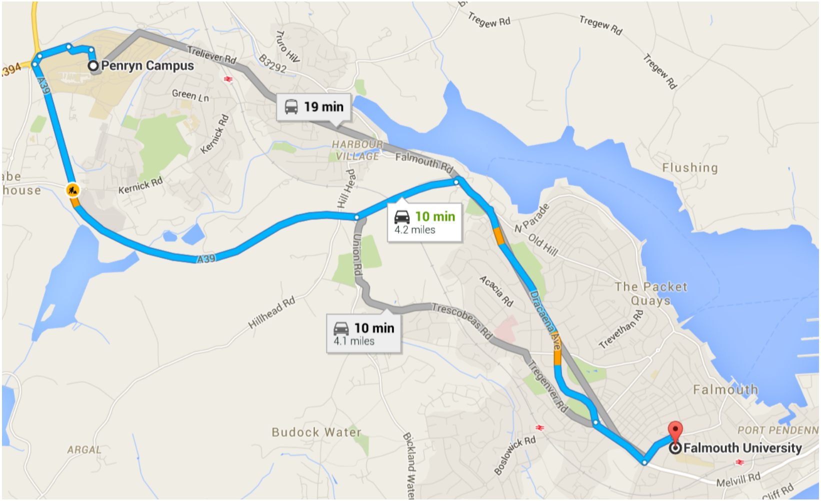
- **Understand** navigation in Video Games
- **Implement** Finite State Machines in Unity
- **Implement** Behaviour Trees in Unity

Pathfinding

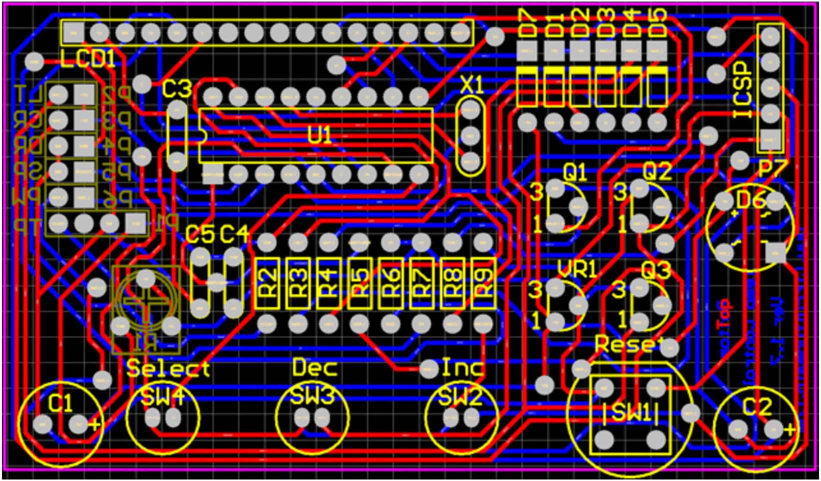
The problem

- We have a **graph**
  - **Nodes** (points)
  - **Edges** (lines between points, each with a **length**)
- E.g. a road map
  - Nodes = addresses
  - Edges = roads
- E.g. a tile-based 2D game
  - Nodes = grid squares
  - Edges = connections between adjacent squares
- Given two nodes A and B, find the **shortest path** from A to B
  - "Shortest" in terms of edge lengths — could be distance, time, fuel cost, ...

Applications of pathfinding



Applications of pathfinding



Applications of pathfinding

Many applications in game AI

- Non-player character AI
- Mouse-based movement (e.g. strategy games)
- Maze navigation
- Puzzle solving

Aside: data structures

- **Stack**: can **push** to the top and **pop** from the top
  - "Last in, first out"
- **Queue**: can **enqueue** to the back and **dequeue** to the front
  - "First in, first out"
- **Priority queue**: maintains its elements in **sorted** order
  - **Enqueue** automatically puts the element in the correct position according to its priority
  - **Dequeue** gives the highest priority element currently in the queue
  - Usually implemented as a **heap** or a **balanced tree**...
  - ... but implementations are available for all popular programming languages

Graph traversal

- **Depth-first** or **breadth-first**
- Recall: can be implemented with a **stack** or a **queue** respectively
- Inefficient — generally has to explore the **entire map**
- Finds a path, but probably not the **shortest**

Greedy search

- Always try to move **closer** to the goal
- Can be implemented with a **priority queue**
- Doesn't handle **dead ends** well
- Not guaranteed to find the **shortest** path

A* search

- Let $h(x)$ be an estimate of the distance from x to the goal
- Let $g(x)$ be the distance of the path found from the start to x
- Choose a node that minimises $g(x) + h(x)$
  - Contrast with greedy search, which just minimises $h(x)$

Properties of A* search

- A* is **guaranteed** to find the shortest path if the distance estimate $h(x)$ is **admissible**
- Essentially, **admissible** means it must be an **underestimate**
  - E.g. straight line Euclidean distance is clearly an underestimate for actual travel distance
- The more accurate $h(x)$ is, the more efficient the search
  - E.g. $h(x) = 0$ is admissible, but not very helpful
- $h(x)$ is a **heuristic**
  - In AI, a heuristic is an estimate based on human intuition
  - Heuristics are often used to prioritise search, i.e. explore the most promising options first

Tweaking A^*

- Can change how $g(x)$ is calculated
  - Increased movement cost for rough terrain, water, lava...
  - Penalty for changing direction
- Different $h(x)$ can lead to different paths (if there are multiple "shortest" paths)

String pulling

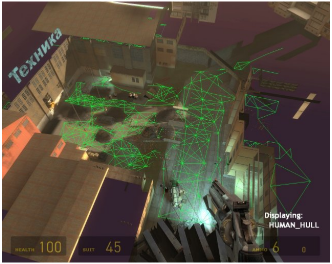
- Paths restricted to edges can look unnatural
- Intuition: visualise the path as a string, then pull both ends to make it taut
- Simple algorithm:
  - Found path is $p[0], p[1], \ldots, p[n]$
  - If the line from $p[i]$ to $p[i+2]$ is unobstructed, remove point $p[i+1]$
  - Repeat until there are no more points that can be removed

Navigation meshes

Pathfinding in videogames

- A* works on any **graph**
- But what if the game world is not a graph? E.g. complex 3D environments

Waypoint navigation

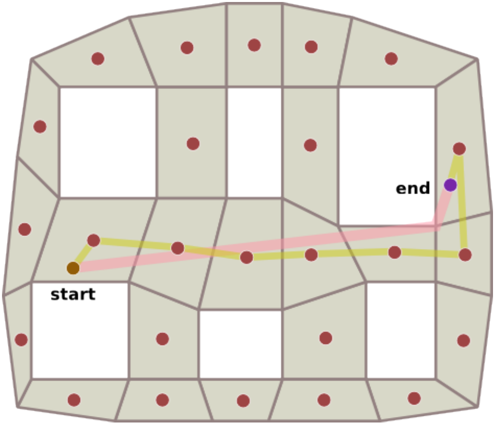


- Manually place graph nodes in the world
- Place them at key points, e.g. in doorways, around obstacles
- Works, but...
  - More work for level designers
  - Requires lots of testing and tweaking to get natural-looking results
  - No good for dynamic environments

Navigation meshes

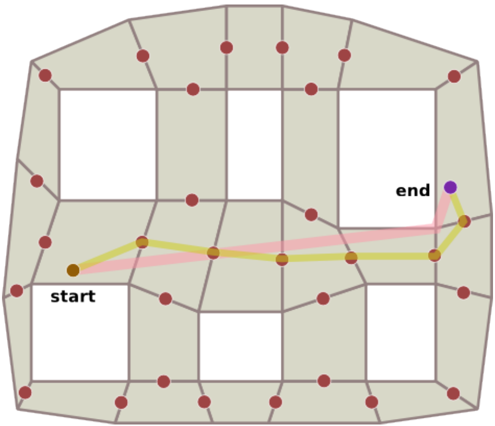
- Automatically generate navigation graph from level geometry
- Basic idea:
  - Filter level geometry to those polygons which are **passable** (i.e. floors, not walls/ceilings/obstacles)
  - Generate graph from polygons

Meshes to graphs



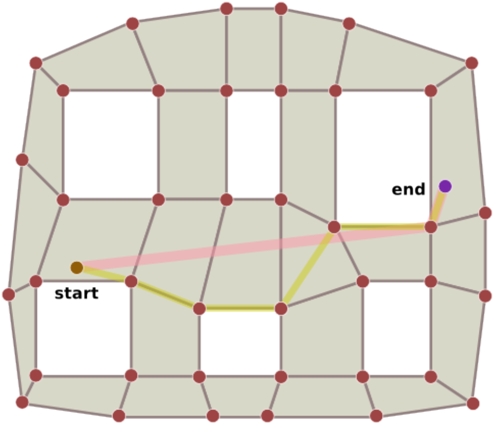
Centres of polygons

Meshes to graphs



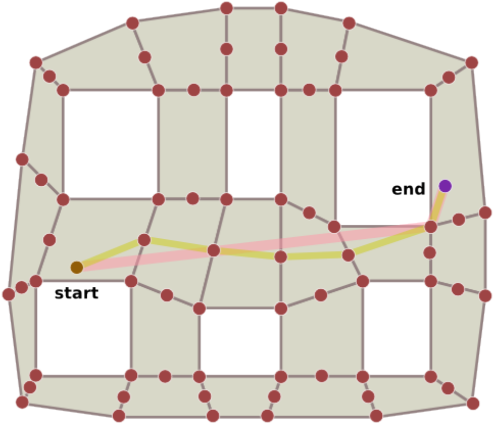
Centres of edges

Meshes to graphs



Vertices of polygons

Meshes to graphs



Hybrid approach: edges and vertices

Following the path

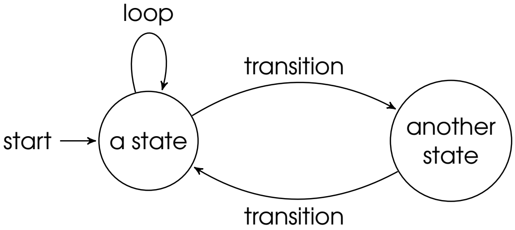
- **Funnelling**: like string pulling but for navigation meshes
  - `http://digestingduck.blogspot.co.uk/2010/03/simple-stupid-funnel-algorithm.html`
  - `http://jceipek.com/Olin-Coding-Tutorials/pathing.html`
- **Steering**: don't have your AI agent follow the path exactly, but instead try to stay close to it
- **Dynamic environments**: may need to re-run pathfinder if environment changes (e.g. movable obstacles, destructible terrain)

Finite state machines

Finite state machines

- A **finite state machine (FSM)** consists of:
  - A set of **states**; and
  - **Transitions** between states
- At any given time, the FSM is in a **single state**
- **Inputs** or **events** can cause the FSM to transition to a different state

State transition diagrams

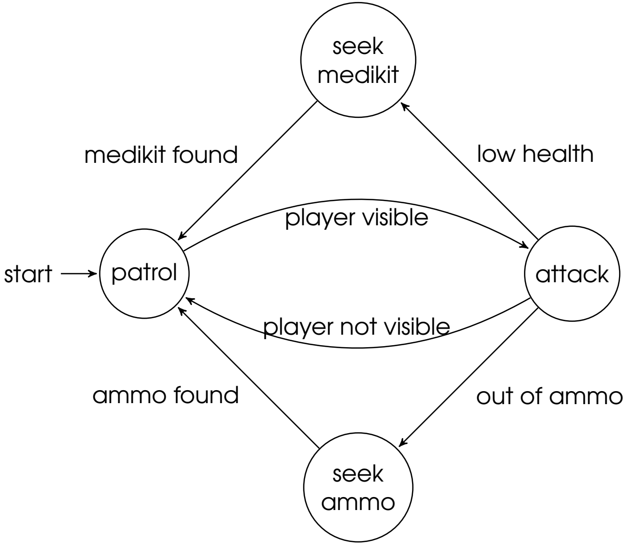


- FSMs are often drawn as **state transition diagrams**
- Reminiscent of **flowcharts** and certain types of **UML diagram**

FSMs for AI behaviour

The next slide shows a simple FSM for the following AI behaviour, for an enemy NPC in a shooter game:

- By default, patrol (e.g. along a preset route)
- If the player is spotted, attack them
- If the player is no longer visible, resume patrolling
- If you are low on health, run away and find a medikit. Then resume patrolling
- If you are low on ammo, run away and find ammo. Then resume patrolling


seek
medikit
medikit found
low health
player visible
start
patrol
attack
player not visible
ammo found
out of ammo
seek
ammo

Other uses of FSMs

As well as AI behaviours, FSMs may also be used for:

- UI menu systems
- Dialogue trees
- Token parsing
- ...

Beyond FSMs

Some topics for you to research, for when plain old FSMs aren't enough...

- Hierarchical FSMs
- Nested FSMs
- Stack-based FSMs
- Hierarchical task networks
- ...

Plus the topic we will be looking at today: **behaviour trees**

Behaviour Trees

Behaviour trees (BTs)

- A **hierarchical** model of decision making
- Allow **complex behaviours** to be built up from **simple components**
- Allow for **more complex** behaviours than FSMs
- First used in Halo 2 (2005), now used extensively
- Also used in robotics and other non-game AI applications

Using BTs

- Fairly easy to implement; plenty of resources online
- Some engines (e.g. Unreal) have BTs built in
- We will be using the free **Behaviour Machine** library for Unity

BT basics

- A BT is a **tree** of **nodes**
- On each game update (i.e. each frame), the root node is **ticked**
  - When a node is ticked, it might cause some or all of its **children** to tick as well
  - So ticks propagate down the tree from the root
- A ticked node returns one of three **statuses**:
  - Success
  - Running
  - Failure
- "Running" status allows nodes to represent operations that **last multiple frames**

Node types

- There are **three main types** of BT node
- **Leaf** nodes
  - No children
  - Represent actions (i.e. the AI agent actually doing something)
- **Decorator** nodes
  - One child
  - Modify the execution of the child
- **Composite** nodes
  - Control which of the children are executed on each tick

Leaf nodes

- Represent **atomic actions**
  - I.e. actions which can't sensibly be broken down into smaller actions
- E.g. walk to, crouch, attack, open door
- Status:
  - Success means "the action is done"
  - Failure means "the action cannot be done"
  - Running means "the action is still in progress"
- Leaf nodes can also be used to represent **conditions**
  - E.g. "is my health below 10%?"
  - Returns success for true, failure for false
- Leaf nodes often have **parameters** to allow for reuse in different situations

Leaf node example

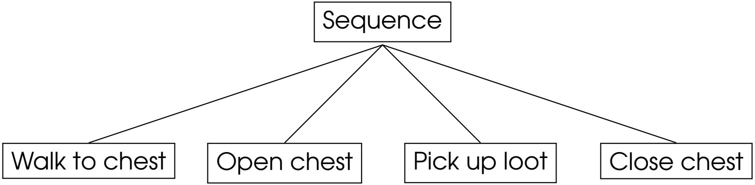
```
using UnityEngine;
using System.Collections;
using BehaviourMachine;

public class GoTo : ActionNode
{
    public GameObjectVar objectToMove;
    public Vector3Var target;
    public FloatVar speed;

    public override Status Update()
    {
        float distance = (objectToMove.Value.transform.position - target.Value). ↩
              magnitude;
        float step = speed.Value * Time.deltaTime;
        if (distance < step)
        {
            objectToMove.Value.transform.position = target.Value;
            return Status.Success;
        }
        else
        {
            objectToMove.Value.transform.position = Vector3.MoveTowards( ↩
                  objectToMove.Value.transform.position, target.Value, step);
            return Status.Running;
        }
    }
}
```

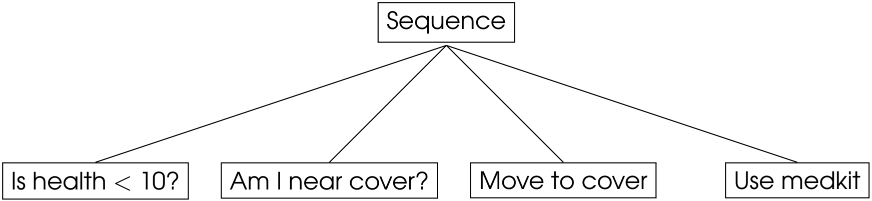
Composite nodes: sequence

- Run each child, in order
- If **any** child returns failure, stop and return failure
- If **all** children return success, stop and return success



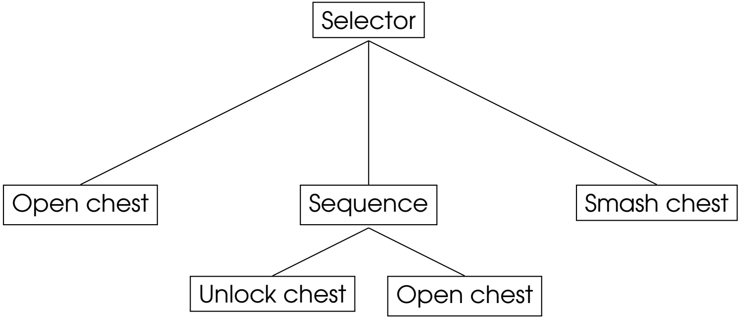
Sequence nodes and conditions

- A sequence node can be used like an `if (cond1 && cond2)` statement



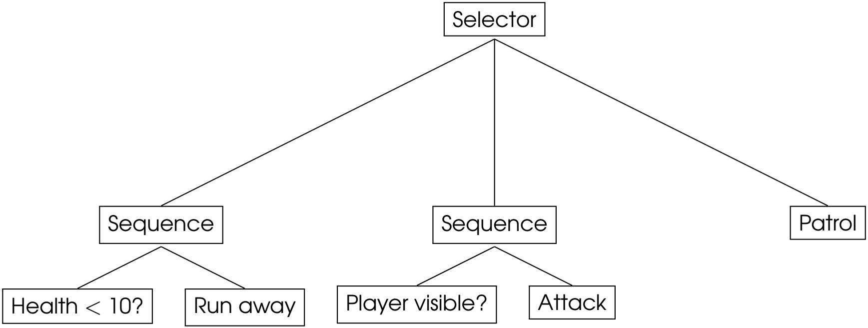
Composite nodes: selector

- Run each child, in order
- If a child returns failure, move onto the next one
- If **any** child returns success, stop and return success



Selectors and priority

- Order of selector children represents the **priority** of different alternatives



Sequence vs selector

- Sequence: perform a list of actions; if one of them fails then abandon the task
- Selector: try a list of alternatives; stop once you find one that works
- Sequence works like **`and`**, selector works like **`or`**

Other composite nodes

- Execute children in **random** order
- Execute children in **parallel**
- Most BT frameworks allow programmers to create custom composite nodes

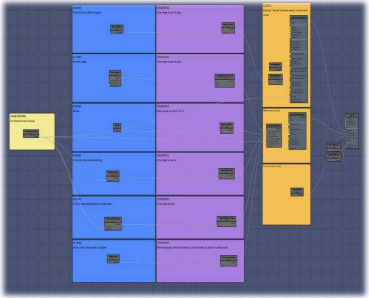
Decorator nodes

- **Inverter**: if child returns success then return failure, and vice versa
- **Repeater**: run the child a number of times, or forever
- Most BT frameworks allow programmers to create custom decorator nodes

Blackboard

- It is often useful to **share** data between nodes
- A **blackboard** (sometimes called a **data context**) allows this
- Blackboard defines **variables**, which can be **read** and **written** by nodes
- Blackboard can be **local** to the AI agent, **shared** between several agents, or **global** to all agents
- (Shared blackboards mean that your AI has "telepathy" — this may or may not be desirable!)

BTs in The Division



http://www.gdcvault.com/play/1023382/AI-Behavior-Editing-and-Debugging

Further Reading

- Game Programming Patterns - `http://gameprogrammingpatterns.com/contents.html`
- Game Programming Patterns in Unity - `http://www.habrador.com/tutorials/programming-patterns/`
- Unity Design Patterns - `https://github.com/Naphier/unity-design-patterns`