Behaviour Trees – Part 2

Extending Behaviour Trees

Behaviour Tree Recap

- Behaviour Trees are a method of breaking down A.I. in to discrete behaviours
  - Modular
  - Allows **AND** and **OR** conditional logic

- Branch nodes are **Composites** of child behaviours, and leaf nodes are either **Actions** or **Conditions**
  - **Composites** add the **AND** and **OR** logic
  - **Conditions** ask a question and return **Success** or **Failure**
  - **Actions** perform the A.I. behaviours, such as “Move a step” or “Attack”

Behaviour Tree Recap

- **Actions** and **Conditions** should be quick
  - Ideally perform their method and return as soon as possible
  - Actions should be broken down in to the smallest possible discrete action
    - Perform a single task, rather than many
    - i.e. *"Step forward"* action, *"Play animation"* action, not *"Step forward and play animation X while playing sound Y and particle effect Z"*

- But not all behaviours can execute instantly, and not all are interruptable
  - Sometimes the agent needs to wait until a behaviour has completed
  - For example, "climbing on to a ladder" might need to wait until the animation ends before it can make other decisions

Pending Behaviours

- It is possible that a behaviour runs longer than a single frame
  - We still want the behaviour to return or else it will block our program

- We can add this functionality to our Behaviour Trees by adding a 3rd return type
  - **Running**, or **Pending**

- **Actions** and **Conditions** could return this new type if they have not finished their task
  - **Composites** need to be changed to allow **Pending**

```
enum BehaviourResult
   Success
   Failure
   Pending

class Behaviour

   func execute(Agent agent)
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Pending Behaviours

- Composites need to keep track of any child behaviour that returns Pending
  - The **Composite** should also return **Pending** straight away without executing any remaining child behaviours
  - When a **Composite** is executed that had a **Pending** child behaviour last time it begins execution at the **Pending** child

- Pending propagates up the tree to the root so that next time it is executed it branches down to the Pending Action or Condition

```
class Composite : Behaviour
   list childBehaviours
   Behaviour pendingChild : null

   func execute(Agent agent) = 0
```

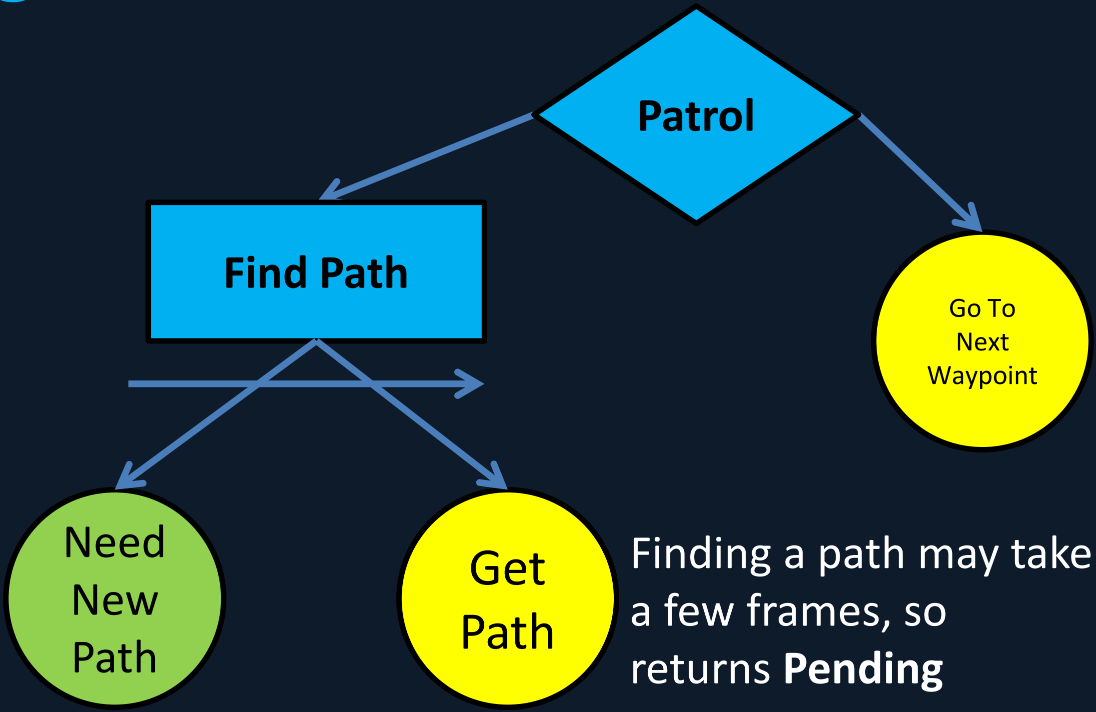
```
class Selector : Composite

   func execute(Agent agent)
      child = pendingChild
      pendingChild = null

      if child is null
         child = childBehaviours.first

      while child <= childBehaviours.last
         result = child.execute(agent)
         if result is Success
            return Success
         if result is Failure
            child = next child
         if result is Pending
            pendingChild = child
            return Pending
      return Failure
```

Pending Behaviours



Will path once path is found

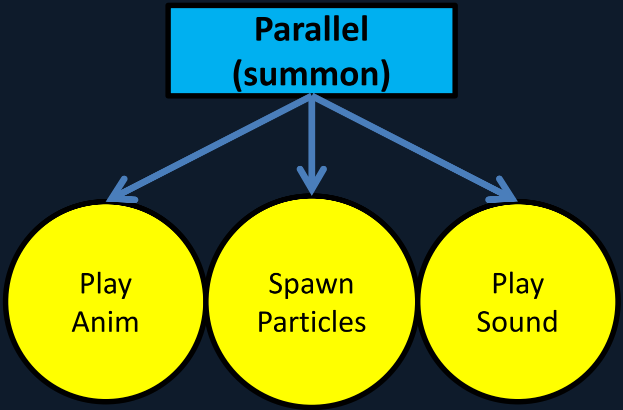
Finding a path may take a few frames, so returns **Pending**

Composite Behaviours

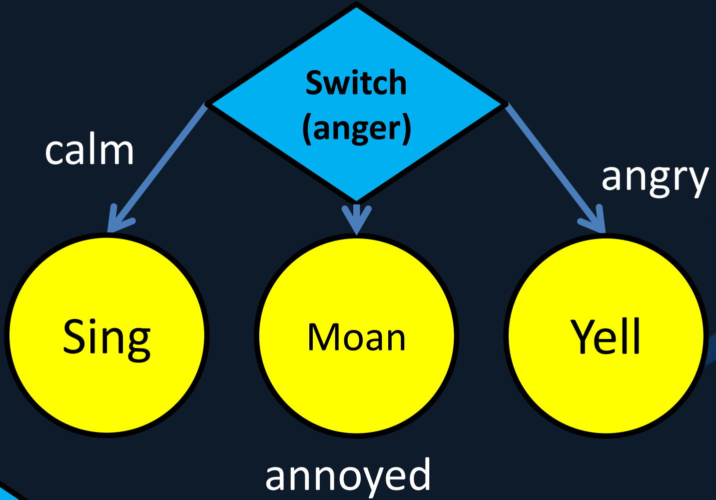
- We have previously discussed the **Sequence** and **Selector** Composite Behaviours
  - Control the logic and flow of the tree

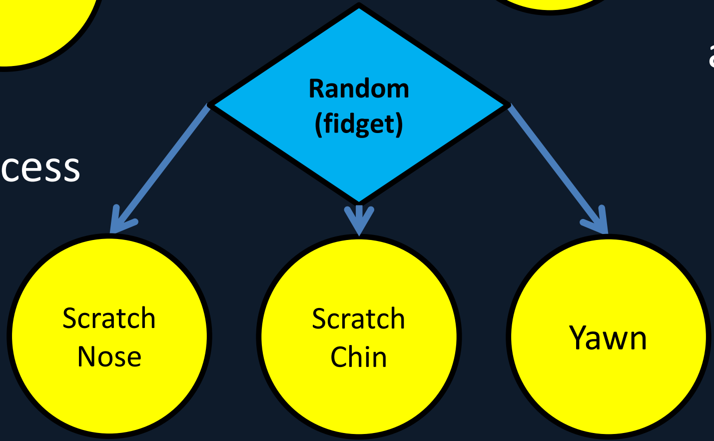
- But there are more types of Composites available to us
  - **Selectors** could select their child behaviours at random rather than ordered, for a **RandomSelector**
  - A **Switch** Composite could select a child behaviour based on a switch
  - A **Parallel** Composite is a composite that can execute all of its child behaviours at the same time
    - Child behaviours don't have to rely on each other or a set order
      - A "**PlayAnimation**" action doesn't have a reliance on "**PlaySound**" so both could execute at the same time
    - Usually requires multi-threading

Composite Behaviours



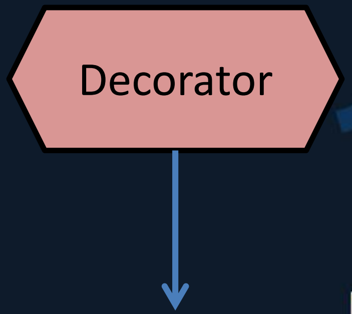
All execute at once
Parallel simply returns Success





Decorator Behaviours

- Another common type of Behaviour is a **Decorator**
- A **Decorator** is a type of **Composite** Behaviour that usually only has one child
  - But it can contain more than one if desired
- A Decorator simply modifies the tree's logic in some way
  - For example, a Decorator could be setup to act as a logical **NOT** operator, reversing the result of its child behaviour



Decorator Behaviours

- Decorators are useful for modifying the tree logic
  - A Decorator could be setup to only allow its child behaviour to execute 5 times
    - After the 5th time it always returns Failure
  - A Decorator could have a timeout, requiring a certain amount of time to elapse before the child behaviour can execute again
    - Always returns Failure if the timeout is still running
- They can also be useful for debugging purposes
  - The Decorator could log any time it executes its child behaviour

```
class NotDecorator : Behaviour
   Behaviour child

   func execute(Agent agent)
      result = child.execute(agent)
      switch (result)
         case Success: return Failure
         case Failure: return Success
```

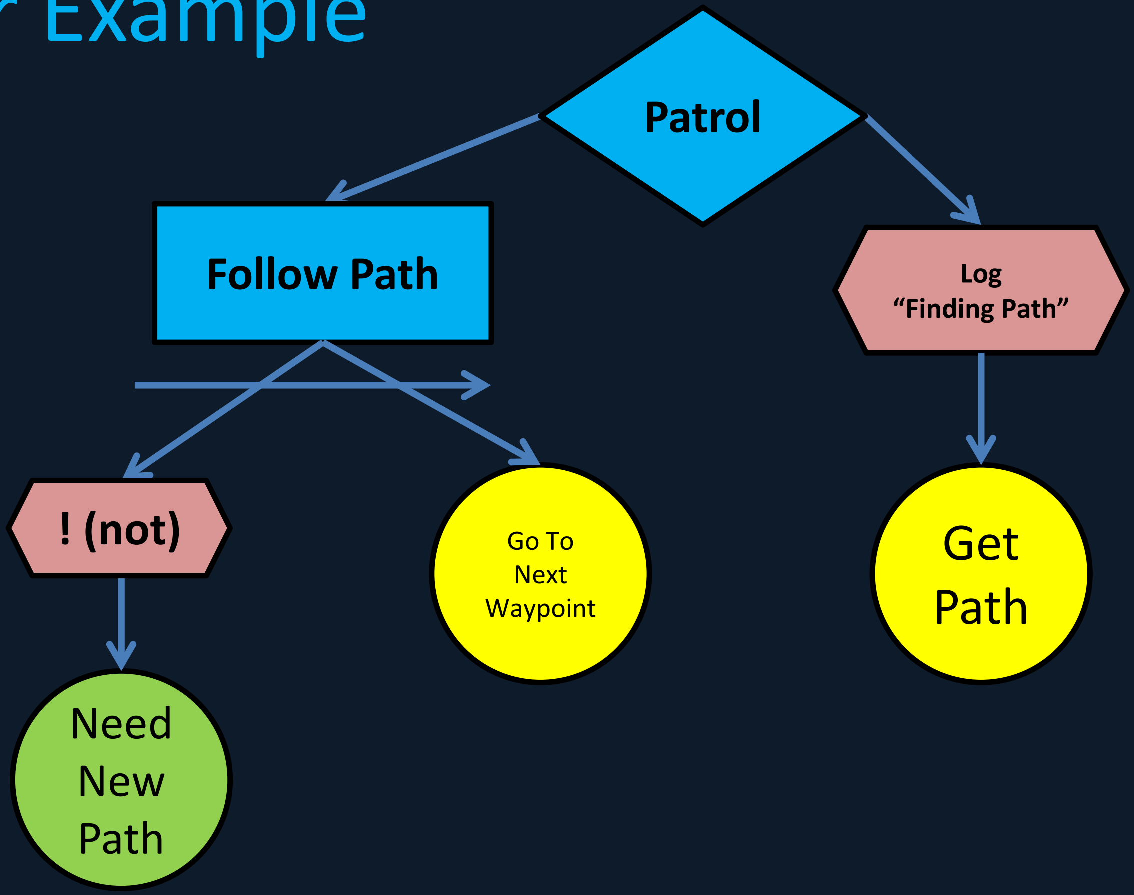
```
class TimeoutDecorator : Behaviour
   Behaviour child
   float timeout, duration

   func execute(Agent agent)
      timeout -= deltaTime
      if timeout > 0
         return Failure
      timeout = duration
      return child.execute(agent)
```

```
class LogDecorator : Behaviour
   Behaviour child
   string message

   func execute(Agent agent)
      print message
      return child.execute(agent)
```

Decorator Example

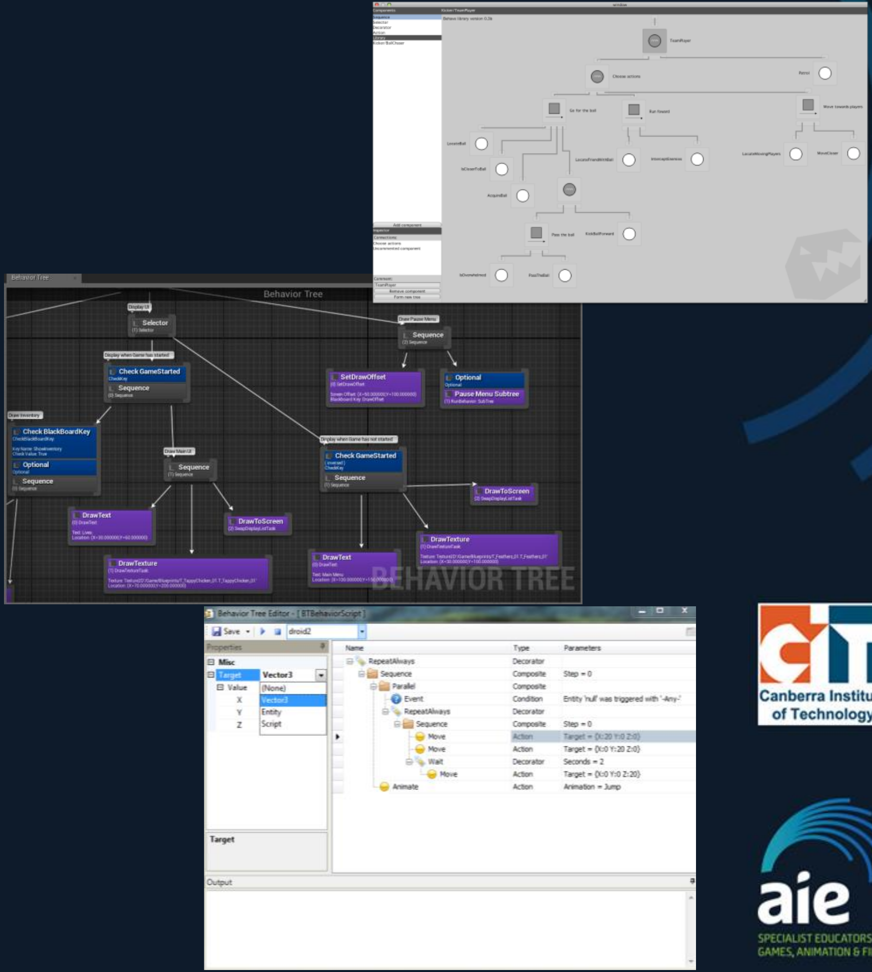


Behaviour Tree Tools

- One benefit of Behaviour Trees is the logic
  - Easy to understand for artists and designers
- A common occurrence in studios is the creation of a Behaviour Tree editing tool
  - This way designers can design the game and A.I. logic without needing a programmer, in the way that they like
  - The game is developed quicker
  - Data driven design allows fast iterative design

Behaviour Tree Tools

- Various tools exist in common game engines
  - Unity3D plugin “Behave”
  - Unreal Engine 4 Behavior Tree
- You can also create your own
  - Simple tree editors can be written with C#
  - XML can be used as a file format



Summary

- Behaviour Trees are extremely powerful and extensible
  - Custom behaviours and complex logic can be easy to implement
- Starting with 2 simple Composites, **Selector** and **Sequence**, we can construct A.I. for a vast range of agents