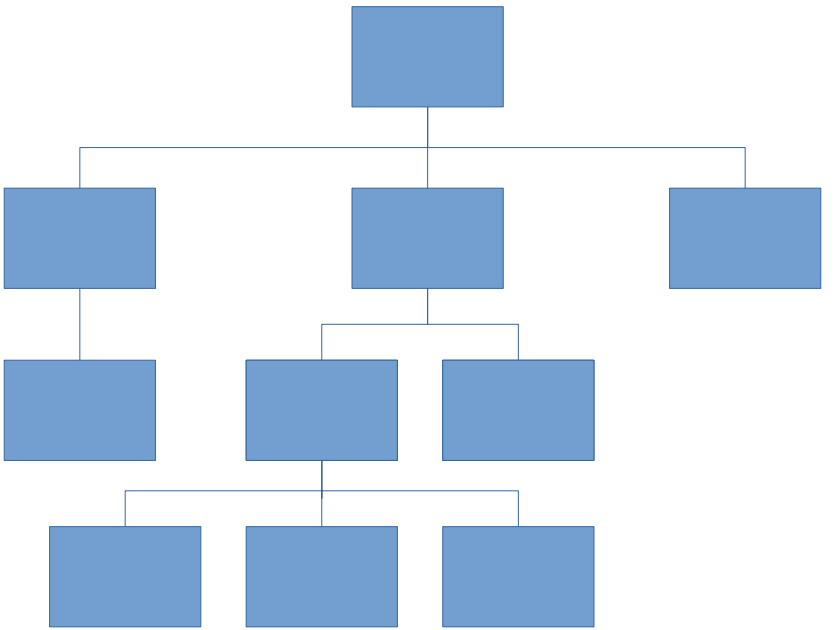
Games AI
Lecture 4.1
Behaviour Trees

- Behaviour Trees
  - Programming idiom for game AI
  - Task oriented, not state oriented
  - Modular, reusable behaviours
  - Build heirachy of increasingly complex beahviours

- Behaviour Trees
  - Heirarchical tree of nodes
  - Controls flow of behaviour
    - Intermediate nodes control flow of execution
    - Leaf nodes run code/perform actions



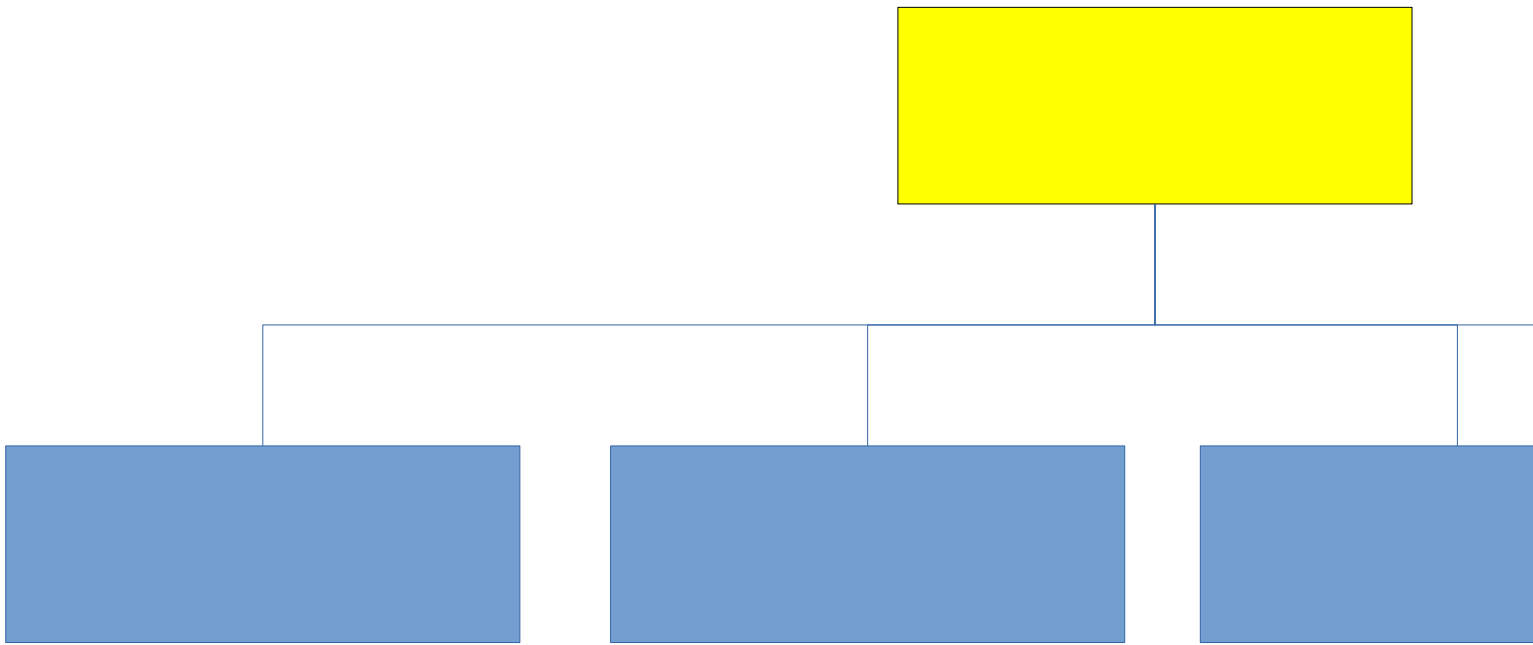
- Basic Behaviour Tree Flow
  - BT is traversed from the root node every frame*
  - Run each node as we encounter it
    - If node returns `running`, we don't go any further
  - Unless we hit a behaviour of higher priority to run (i.e. higher up the tree), we will find "active" node (where we were last frame), then continue normal execution

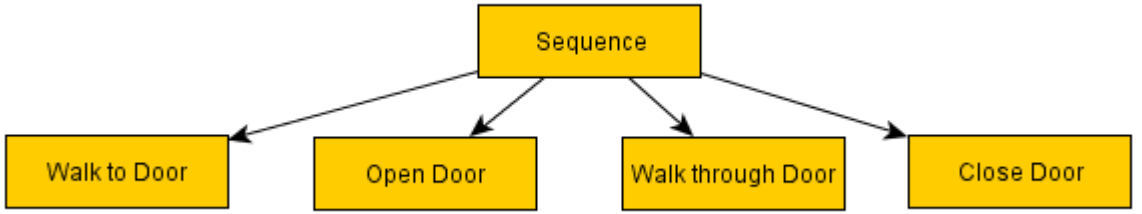
*not the only/best way to design a BT

- BT Flow
  - Any node will **return one of three statuses**:
    - `success, failure, running`
  - The `running` status permits actions that take multiple frames to complete, e.g. walk to door
  - Gives contract for what will be the case when node returns `success`
    - eg. walk to door – we will be at door

- Node types
  - **Action nodes** control the things an agent does
    - e.g. play animation, set variable, check variable
  - **Composite nodes** process children in some order, and at some point determine success or failure
  - **Decorator nodes** affect the processing or result of a single child node

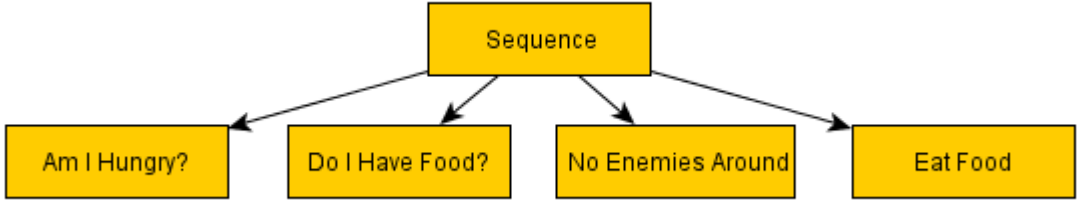
Composite Nodes

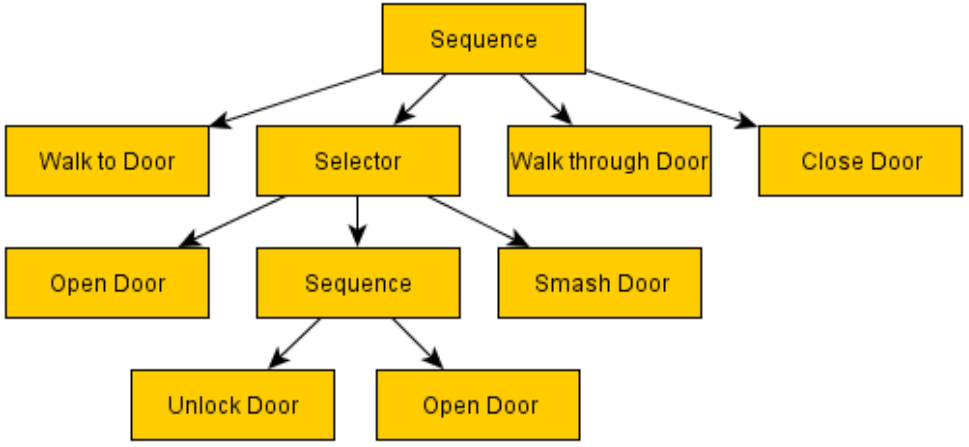




- Sequence Nodes
  - Tries each child in turn
  - Stops and returns `failure` when a child fails
  - Returns `success` if none fail
  - Used where you want to try and complete a whole sequence and only consider it a success if you get to the end without failure
    - e.g. open door. You can move to it pick the lock, but if it's wedged shut, you still return failure.
      - If you fail to move to the door, there is no point trying to pick the lock

- Other uses of sequences
  - Check variables
  - Behaves like &&, with short-circuiting



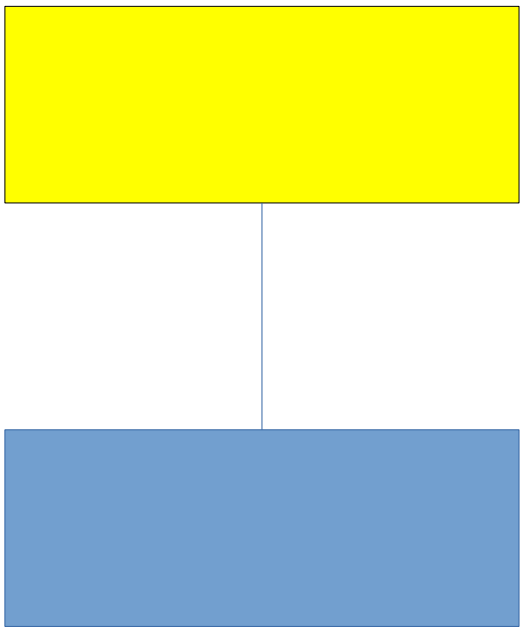


- Selector Nodes
  - Tries each child in turn
  - Stops and returns `success` when a child succeeds
  - Returns `failure` otherwise
  - Behaives like || with short-circuiting
  - eg. try to open door, try to unlock door and open it, then try to smash door.
    - If one fails, we keep trying the others
    - So long as one suceeds, the door is open

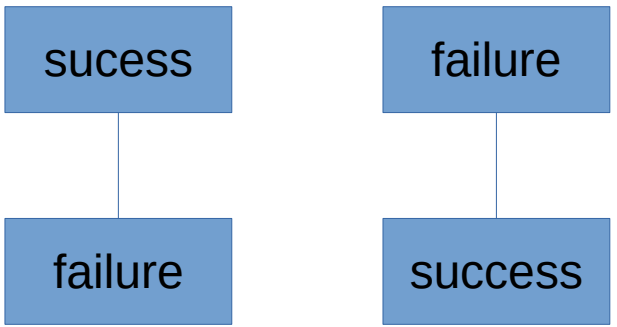
- Parallel
- Random Sequence

- And many more...

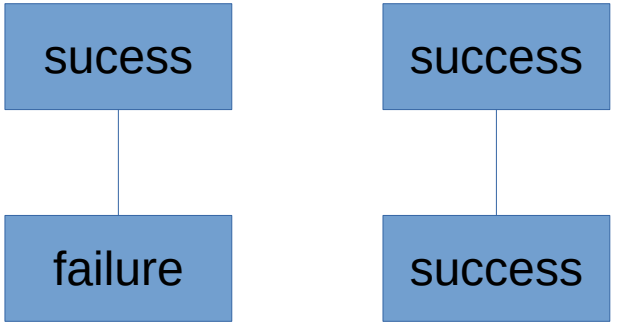
Decorator Nodes



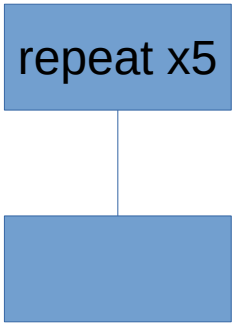
- Inverter
  - Flips result of child node



- Suceeder
  - Always return success
  - Use where failure is expected but you don’t want that failure to have its usual affect on the flow of execution

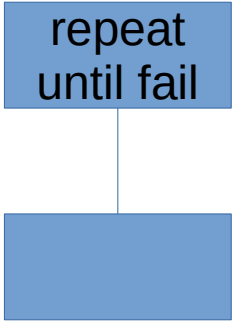


- Repeater
  - Reprocess child node when it returns a result
  - Creates an infinite loop
  - Can set a maximum number of repeats

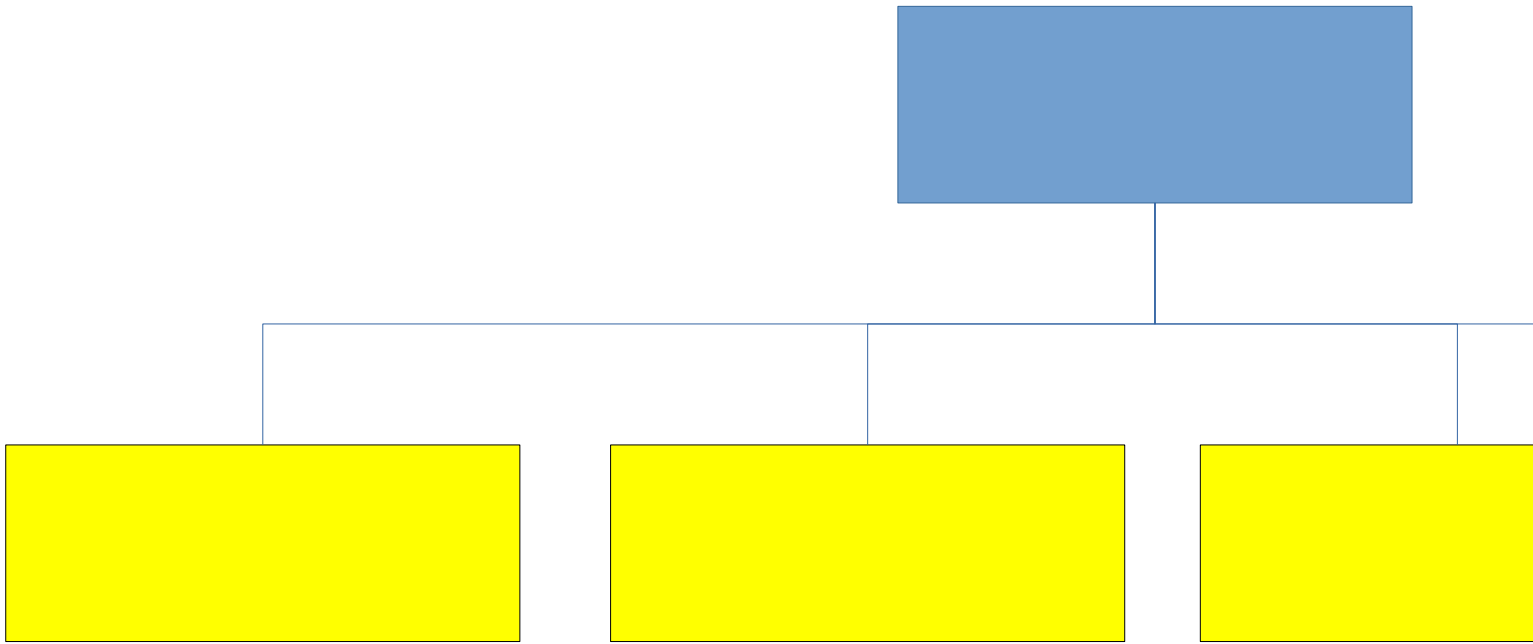


- Repeat until fail
  - Reprocess child node whenever it returns a success, but if it returns a failure, then stop

- And many more...



Action Nodes

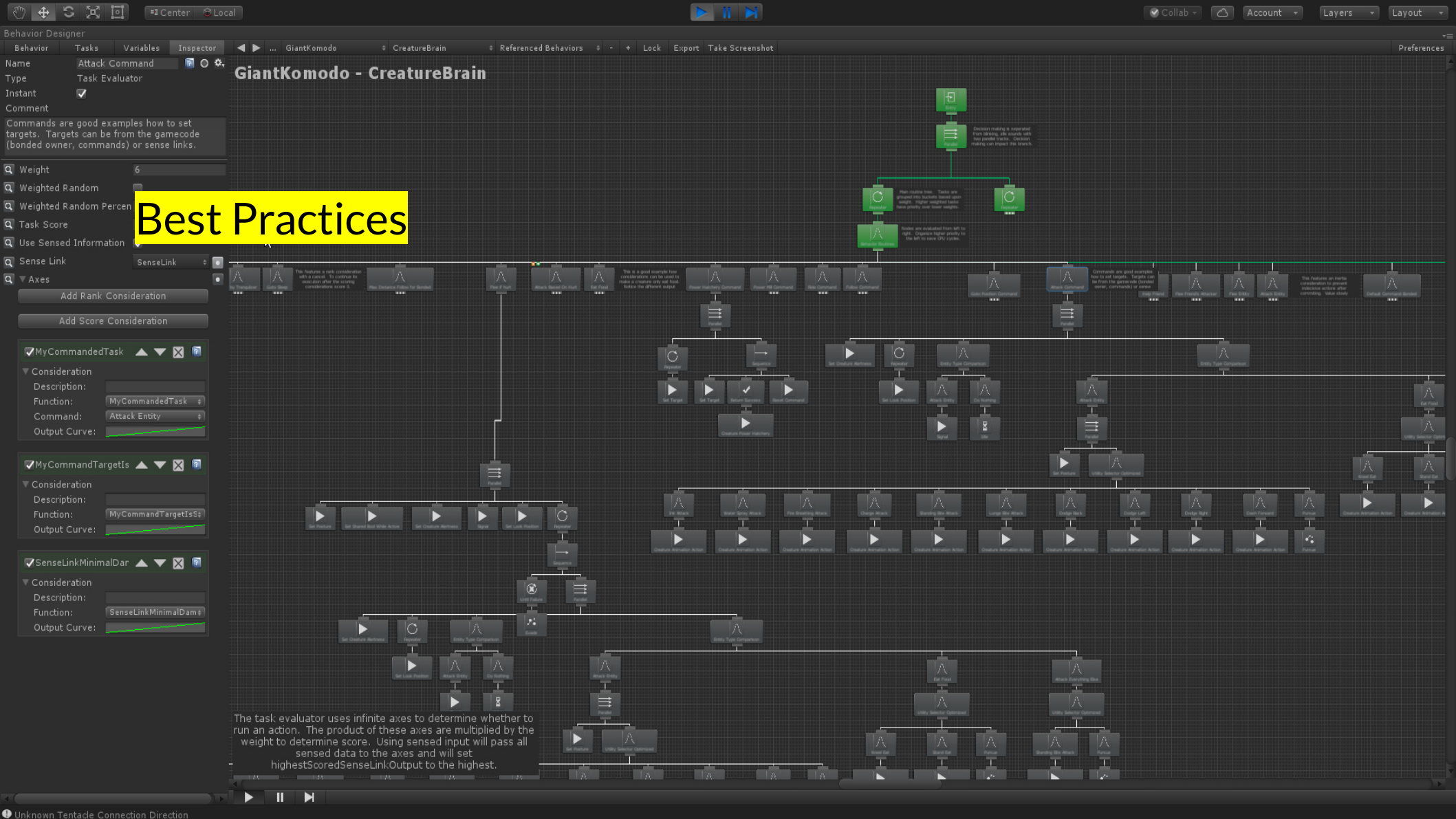


- Actions nodes
  - Change state of game world
    - Perform animation, play audio
    - Update character state
    - Set variable in context
      - eg. set WalkTarget to ClosestEnemy.Position
    - Run a pre-defined piece of code
      - e.g. pathfind to WalkTarget
  - Read/test variable e.g. Me.Health < Me.MaxHealth
  - Return `success` or `failure` (or `running`)

- Storing context
  - Nodes have shared context (i.e. database or blackboard) where they can update and access varibles
  - e.g. when a BT is called on an AI agent, a context is created which stores arbitrary variables in a dictionary
  - Action nodes can read/write to this dictionary, e.g.
    - set SpellAttackCooldown = 10 seconds
    - Calculate closest enemy and store as ClosestEnemy
    - Check if EnemyInSight is true

- Reference Node
  - Call another behaviour tree, passing context
  - Useful for splitting up different behaviour trees into separate files/resources for managing workflow

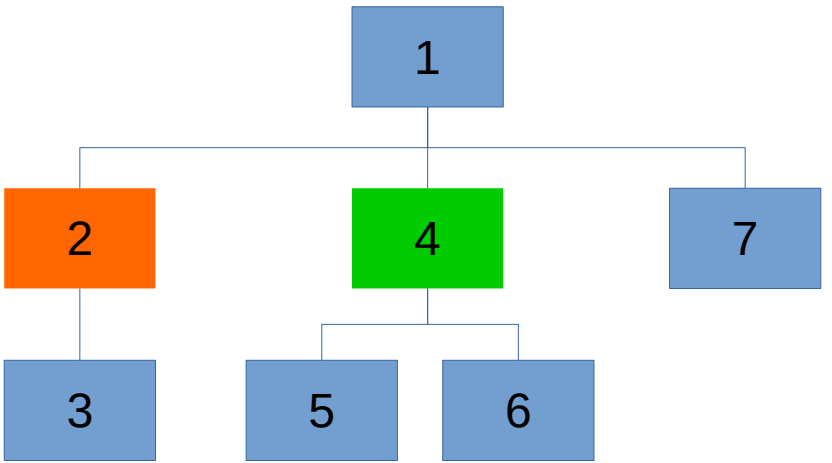
- Dynamic Node
  - Behaviour specified at runtime by attaching a behaviour tree in code
  - Useful for e.g. smart objects


Behavior Designer
Behavior
Tasks
Variables
Inspector
GiantKomodo
CreatureBrain
Referenced Behaviors
Lock
Export
Take Screenshot
Preferences
Name
Attack Command
Type
Task Evaluator
Instant
Comment
Commands are good examples how to set targets. Targets can be from the gamecode (bonded owner, commands) or sense links.
Weight
6
Weighted Random
Weighted Random Percen
Task Score
Use Sensed Information
Sense Link
SenseLink
Axes
Add Rank Consideration
Add Score Consideration
MyCommandedTask
Consideration
Description:
Function:
MyCommandedTask
Command:
Attack Entity
Output Curve:
MyCommandTargetIs
Consideration
Description:
Function:
MyCommandTargetIs
Output Curve:
SenseLinkMinimalDar
Consideration
Description:
Function:
SenseLinkMinimalDam
Output Curve:
GiantKomodo - CreatureBrain
Best Practices
The task evaluator uses infinite axes to determine whether to run an action. The product of these axes are multiplied by the weight to determine score. Using sensed input will pass all sensed data to the axes and will set highestScoredSenseLinkOutput to the highest.
Unknown Tentacle Connection Direction

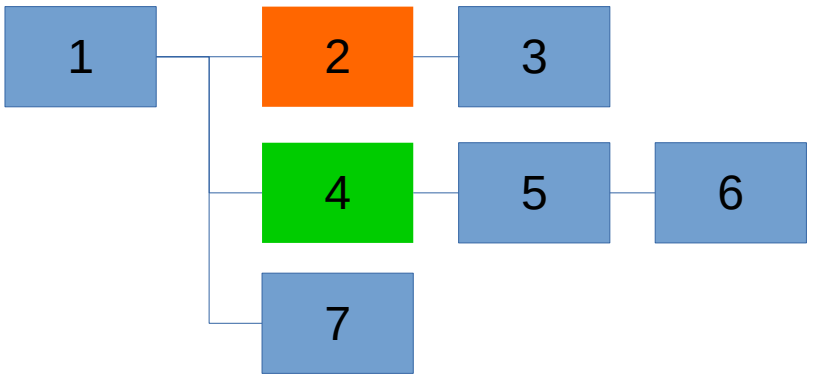
- Behaviour Trees
  - Good at describing individual behaviours
  - Definite execution of decisions
  - Bad at interruptions/transitions
  - Bad at representing cyclic behaviour
  - Get messy and hard to manage if allowed to grow too big

- Ideally decisions made should be local to “how to enact this behaviour”, not high level “what should I be doing”
- There are better ways for describing high-level decision logic, such as state machines
  - Combining approaches can keep your AI more managable

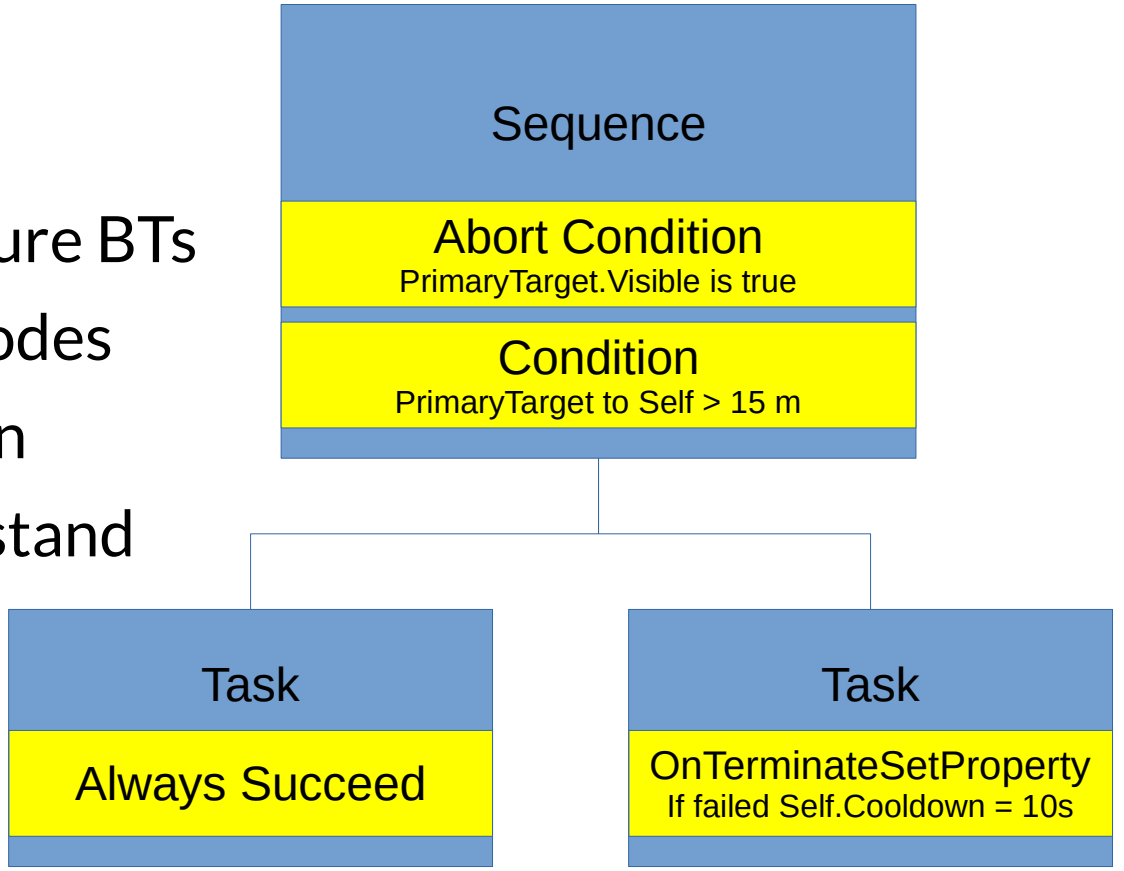
- Drawing a Behaviour tree
  - Usually top-down
  - Node priority top-down,
  - Execution order top-down



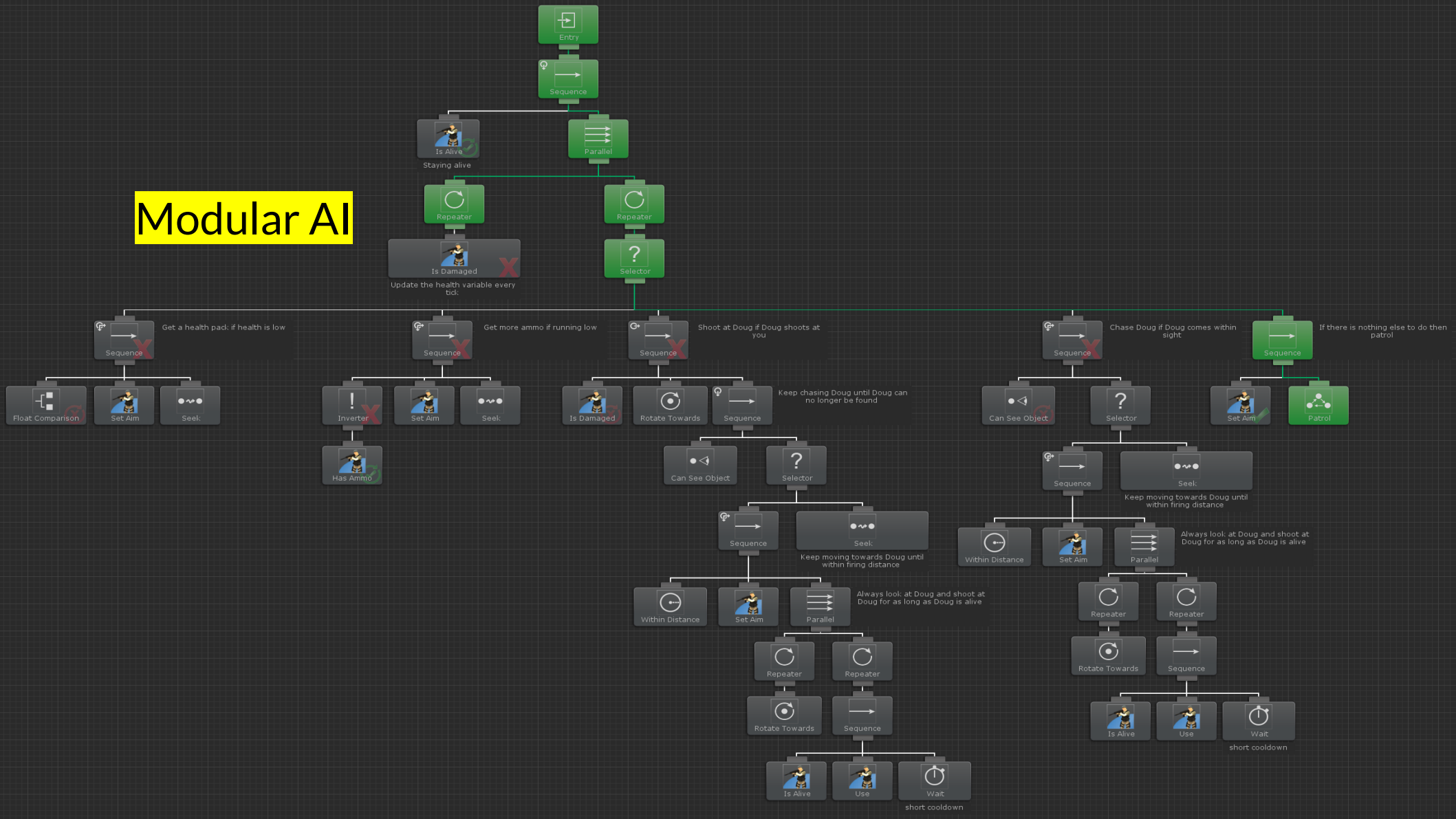
- Drawing a Behaviour tree
  - Can be nicer to draw in reading order
  - Node priority top-down,
  - Execution order left-right



- Node Decorators
  - Alternate way to structure BTs
  - Decorators as part of nodes
  - Compact representation
  - Easy to read and understand



Modular AI


Entry
Sequence
Is Alive
Staying alive
Parallel
Repeater
Repeater
Is Damaged
Update the health variable every tick:
Selector
Sequence
Get a health pack if health is low
Float Comparison
Set Aim
Seek:
Sequence
Get more ammo if running low
Inverter
Has Ammo
Set Aim
Seek:
Sequence
Shoot at Doug if Doug shoots at you
Is Damaged
Rotate Towards
Sequence
Keep chasing Doug until Doug can no longer be found
Can See Object
Selector
Sequence
Seek:
Keep moving towards Doug until within firing distance
Within Distance
Set Aim
Parallel
Always look at Doug and shoot at Doug for as long as Doug is alive
Repeater
Repeater
Rotate Towards
Sequence
Is Alive
Use
Wait
short cooldown
Sequence
Chase Doug if Doug comes within sight
Can See Object
Selector
Sequence
Seek:
Keep moving towards Doug until within firing distance
Within Distance
Set Aim
Parallel
Always look at Doug and shoot at Doug for as long as Doug is alive
Repeater
Repeater
Rotate Towards
Sequence
Is Alive
Use
Wait
short cooldown
Sequence
If there is nothing else to do then patrol
Set Aim
Patrol

- We’ve seen three ways of controlling behaviour:
  - State machines
  - Behaviour Trees
  - Utility-based AI
- These have their pros and cons, and can be used together
- How do we combine them?

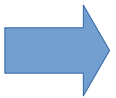
- “Modular AI” is a fancy term for the application of basic software engineering principles to AI development.
- Each AI module is
  - Small
  - Does one clearly defined task
- eg. “target selection” module decides on our primary target, for melee attack module, ranged attack module, flee module, …

- Modular encapsulation, as in OOP
- Have clearly defined interface for each module
- Some AI, e.g. behaviour trees, is primarily data driven
  - Define standard variables you use in your blackboard
- Have predictable behaviour (e.g. events in heirarhical state machines)

- Essential problem:
  - Connect output of one module to input of another
  - Ideally avoid any accidental complexity
- Loosely coupled modules
  - If remove module, then only impairs that behaviour
- Tightly coupled modules
  - Missing module causes compilation failures/runtime errors
- Special Cases
  - Special case code often breaks on re-use

- Work at right level of granularity
  - Small, but abstracting away from fine-grained implementation details
- Represent a single human concept
  - How far away is he?
  - How long have I been doing this
  - I want to move over there

- Conceptual abstractions
  - Considerations
  - Actions



- Modular Components
  - Distance Consideration
  - Move Action

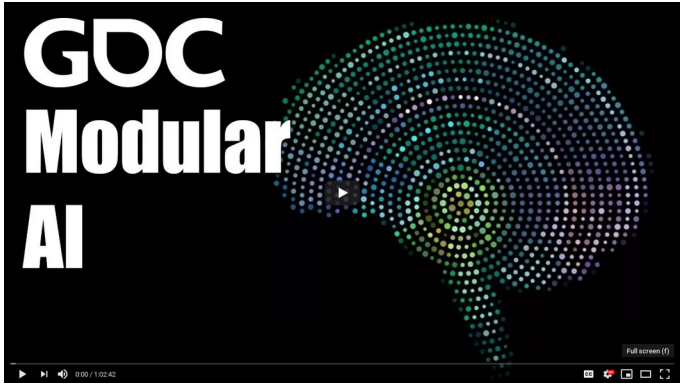
- Common Abstractions
  - Reasoners
    - Make decision
  - Considerations
    - e.g. Distance, Execution History, Picker
  - Actions
    - e.g. Moving, fire weapon, Subreasoner
  - Targets
    - e.g. Fixed position, named entity, controlled entity
  - Weight functions
    - Convert an input to weight values
  - Region
    - Repesents space with inside and outside

- Separation of Responsibility
  - Sensing
  - Deciding
  - Acting

- A lot of the challenge of implementing game AI is the architecture, rather than the algorithms
  - Games are big, complex projects
- In your assessment you are encouraged to combine multiple AI approaches
- You need to decide how best to do this
- How you architect your AI code is important. You will need to discuss this in your report.

Recommended “reading”

- *Behavior trees for AI: How they work* Chris Simpson. 2014 Gamasutra
  https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php
- *AI Arborist: Proper Cultivation and Care for Your Behavior Trees* Panel at GDC 2017 https://youtu.be/Qq_xX1JCreI
- *Nuts and Bolts: Modular AI From the Ground Up* Panel at GDC 2016 https://youtu.be/IvK0ZlNoxjw
- *The Behavior Tree Starter Kit*, Alex Champandard, Philip Dunstan in Game AI Pro Ch. 6 http://www.gameaipro.com/



Implementing BTs

- We'll need a class to represent our BT nodes
  - Subclasses for Composite nodes, Decorator nodes, Action nodes, ...
  - Keeps track of a list of child nodes
  - Has a `run()` method or similar that takes a context object and returns a status
- Use an enum to encode status (`Success`, `Failure`, `Running`)

- Action nodes
  - call `init()` method called first time it is run to do any setup
  - call `process()` method called every time it is run
- Overload these methods in subclass to define specific behaviour

- BTs are supposed to be data-driven - we don't want to manually specify them in code – so we'll need a way to create them easily from data
  - Specify tree structure in data object (e.g. JSON) and create tree using a Factory
  - Specify node type with an enum
  - Specify any arguments for the constructor

```
{
    type: BTNode.Sequence,
    children: [
        {
            type: BTNode.Inverter,
            args: [5],
            children: [
                { type:BTNodeType.Void }
            ]
        }
    ]
}
```

- Create a BehaviourTree class to manage the BT
  - Runs BT by calling run() on top level node and passing in context object
  - May want to keep track of currently running node to avoid having to traverse the tree each frame
    - (But need to re-check periodically in case a higher-priority action can run)
- Each agent may want their own BT

- Create a context or blackboard to share data within your BT (and with other parts of your AI)
  - Store data about the state of the world
  - e.g. sensors can process world data and provide agent-specific information tailored for easy decision making in your BT
    - Can I see enemy?
      - true/false
    - How injured am I?
      - not at all, a bit, lots, near death