09 AI in Games

Tvorba a dizajn počítačových hier

Návrh a vývoj počítačových hier

Motivation

- We require opponents/teammates in games
- Non-playing characters are usually required to perform some tasks that require AI
  - Following the player
  - Combat
  - Strategic thinking
  - …
- We require something that responds to user actions and imitates the behavior of human players
- Ideally, an AI should fool the players into thinking it is an actual human
  - To keep the immersion level high
  - Turing test

A little history

- Pac-man (1979) was one of the first games with character AI
- Very simple AI that decides at crossroads whether to follow the player, run from him, or take a random road
  - Used different states – scatter, chase…
- Each ghost has its personality – this determined behavior
  - New target tiles are determined based on personality at every crossroad
- It was still effective
- Randomness added a necessary factor – **unpredictable behavior**
- A completely predictable AI is usually easy to beat
  - After a few tries, the player has a detailed model of AI behavior

The Kind of AI in games

- Hacks
  - Games use a lot of hacks, not only in AI
  - Ad hoc solutions to specific problems
- Heuristics
  - Predictions that work most of the time (without guarantees)
- Algorithms
  - The true AI
  - Techniques that simulate behavior and are usually derived from how real people or animals make decisions and perform actions
- Machine Learning
  - Observe thousands of examples of player behavior
  - Derives its own algorithm on what to do

Hacks - “Game AI is not AI”

- Is the Pac-man example AI?
  - It’s just generating random numbers and performing one of three actions based on the result
  - It’s not an AI technique
- In Sims, a lot of actions are just pre-defined animation sequences
  - There is no actual AI going on
- We need to know what the right approach is
  - Simulating emotions with characters sometimes scratching their heads is very simple
  - However, maintaining and updating the emotional state of a character is overkill

Heuristics

- Approximate solutions to existing problems
- This is usually how the human mind solves problems
- E.g. lost keys => retrace your steps
- A simple heuristic just says how good an enemy's aim is
  - The lower the number, the smaller the chance they will hit you
- Common heuristics:
  - Most constrained
    - If we have two groups fighting, and one character in one group has a unique weapon that pierces through some unique armor, it should attack a character wearing that armor
  - Most difficult first
    - If you have resources to buy a strong unit, do it instead of getting several weak ones
  - Most promising first
    - Perform the action that will improve your chances the most (think chess AI)

Algorithms

- Some AI actions still require something more
  - Movement of characters
  - Decision making
  - Tactics or strategy
  - Analysis of game state and future game state
- “Academic” AI

Academic AI vs. Game AI

- Academic AI
  - Make the AI as smart as possible
  - Solve the problem as efficiently and precisely as possible
- Game AI
  - Make the player have fun
  - Provide interesting challenges for the player
  - React to the player
  - Be predictable enough for the player
  - Be believable enough to keep the illusion of a real being in control

Game State Analysis

- Process input data to simplify the decision process
- This is usually called *sensing* - *y*ou create senses for the AI
  - Vision
  - Hearing
  - Touch
  - Smell?
  - …

A 3-step process

1. **Sense** – what can I see/hear/feel
   - Some senses can cheat!

2. **Think** – what do I do based on what I am sensing
   - Process data from senses and decide what to do

3. **Act** - Perform actions I have decided to do
   - Walk to a destination
   - Attack someone
   - Use an item
   - …

An example: Sims

- When does a sim become hungry?
- What to do next, and what if the sim really has to go to the bathroom?
- These are several competing systems that are assigned weights
- Changing these weights alters the behavior of the sim
- Final decisions as to which action to perform are strongly affected by the weights of different systems
- When a sim is about to pass out of hunger, getting food becomes top priority
- The weight of the hunger system represents the desire to get food

AI types in games

- Hard-coded
  - Deterministic behavior – such as turning on lights in exact hours every day in a house when you are on vacation (to scare away thieves)
- Randomization
  - Throwing in randomized behavior
  - Not always exact times to turn the lights on/off
  - Add a random time offset every day, say in range +-60min (we could use normal distribution)
- Weighted randoms
  - Every possible next step is given a weight
  - When deciding what to do, we generate a random number in range [0, W], where W is the sum of all weights for all possibilities
  - Some of the possibilities will happen more often than others

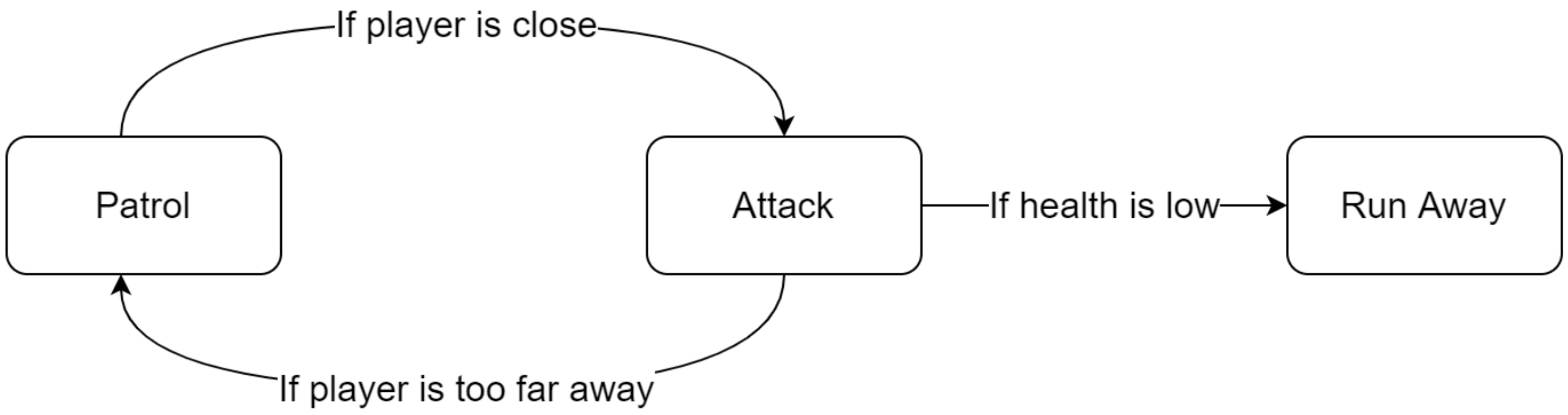
Weighted randoms example

- We have a creature that can perform 3 actions
  - Attack, Cast spell, Run away
- We assign weights to these actions, say 60%, 30%, 10%
- Very similar to the Sims example, but those weights change over time

```
X = RandomFromRange(0, 99);
if (X < 60)
    Attack();
else if (X < 90)
    CastSpell();
else RunAway();
```

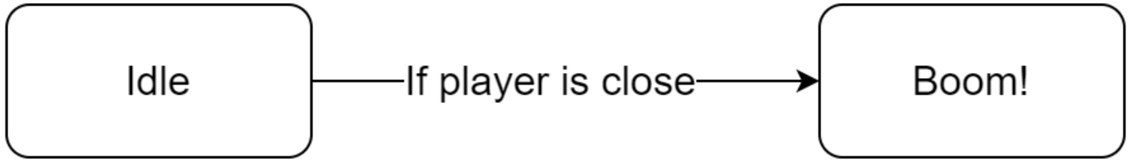
Finite state machines

- Very simple construct
- We have several states an entity can be in (sleeping, wary, attacking, running away…)
- We define rules to transition from one state to another
- There does not have to be a transition from each state into every other state
  - The transition running away -> sleeping does not make much sense
  - Running away -> wary -> sleeping makes much more sense
- We also define when these transitions from one state to another should happen



Finite state machines (2)

- Based on the current state of the entity, we perform a selected action
- If the guard is in the patrol state, they might walk through corridors along a pre-defined path
- If they are attacking, they might be moving continuously towards the player while shooting from a gun
- Once their health becomes low, they decide to run
- A proximity mine can use the same “proximity” check as the guard

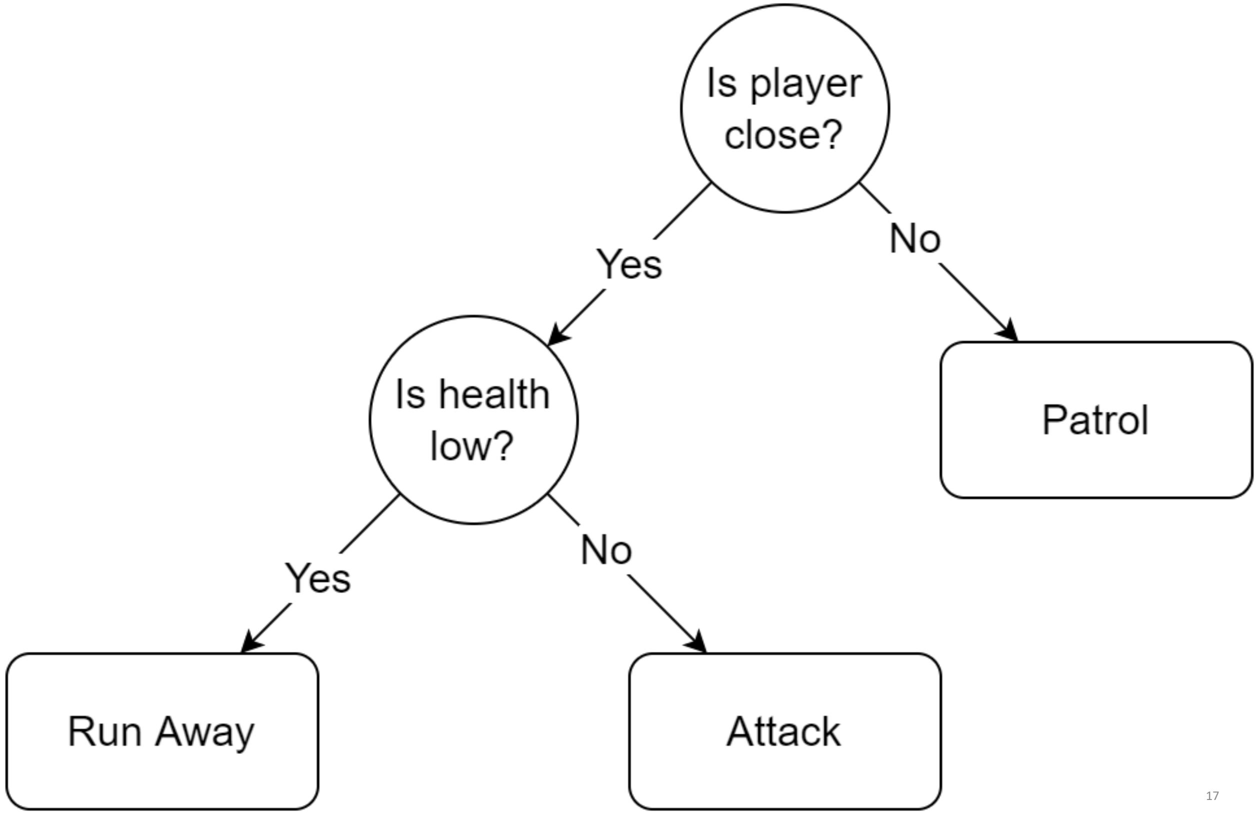


Finite state machines (3)

- The decision making is encapsulated in the transition rules
- The transition rules can incorporate a certain degree of randomization
  - Such as an enemy running away at less than 15-25% health
- This is called **reactive AI**, it always reacts to some game event
- The other type is **active AI**, which constantly seeks the best possible option
  - Such as a sim in Sims, or an AI opponent in Starcraft 2

Decision trees

- Decision trees are a simple way of representing decision making
- The inner nodes of a decision tree are decisions with only two possible answers: **Yes** or **No**
- Each leaf node is an action node
- Each node has two children
  - one for the Yes answer, one for the No answer
- We traverse the tree from the node
  - Evaluate each condition
  - Until we reach an action
  - Then perform the action



Decision trees (2)

- We apply an action every time a decision needs to be made
- Decision trees can be shared, as can be individual nodes
- Decision trees are built-in visual tools
  - The programmer has written code for decision nodes and action nodes
  - The designer connects these to build an actual tree
- With each decision we ignore a whole sub-tree
- This is relatively efficient even for hundreds of nodes
- The decision can actually take several frames to decide
  - We always save the node in which we pause the decision making
  - And then resume in the next frame
  - A bit risky since the situation can change abruptly, making the decision invalid

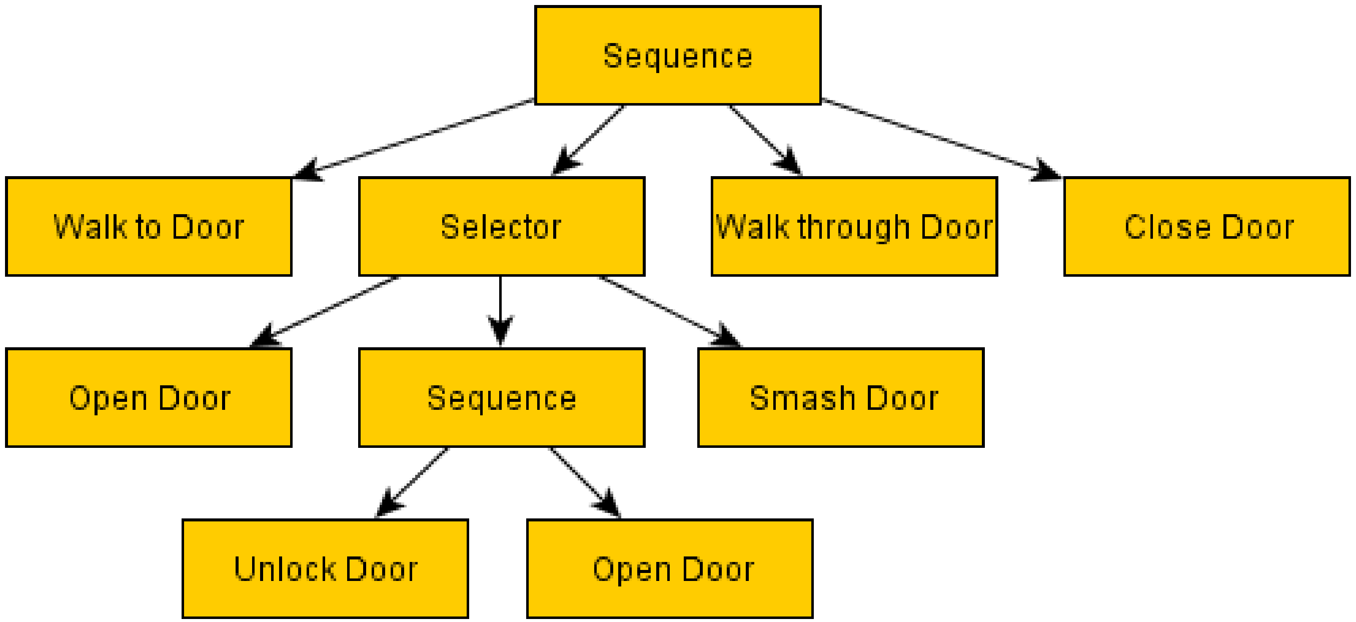
Decision trees (3)

- You can also have non-binary decision trees
- Such as a ProcessHealthNode that has 4 children based on how much health the character has
  - High health -> more aggressive
  - Low health -> seek cover/run

Behavior Trees

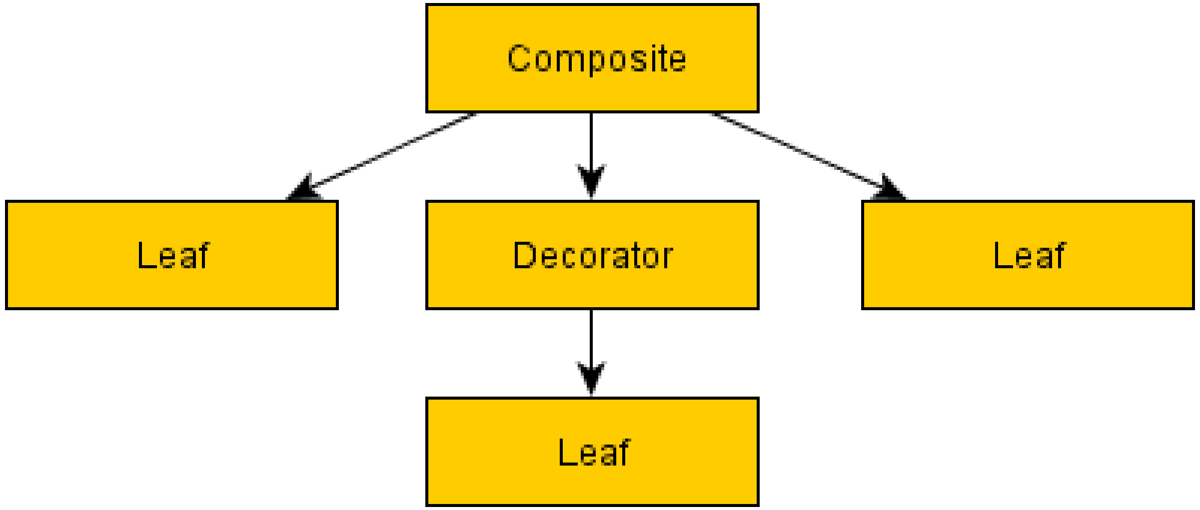
- Widely used in games
- It's a tree composed of nodes
- Each node can return a status to its parent
  - **Success** – the operation of this node finished successfully
  - **Failure** – the operation failed
  - **Running** – the operation is still running
- Nodes can have parameters
- Nodes can respond to context
  - Game state

- Lots of Unity plugins for BT
- Built-in support in Unreal



Behavior Trees (2)

- Leaf nodes represent actions
  - Running, Success, Failure
  - **Open a door**, **Run towards the player**
- Composite nodes
  - Encapsulate multiple children
  - Execute children in some order
  - Returns what is returned from children
  - **Sequence**, **Selector**
- Decorator nodes
  - Have a single child node
  - Transform result from the child, repeat, terminate
  - **Inverter**, **Repeater**



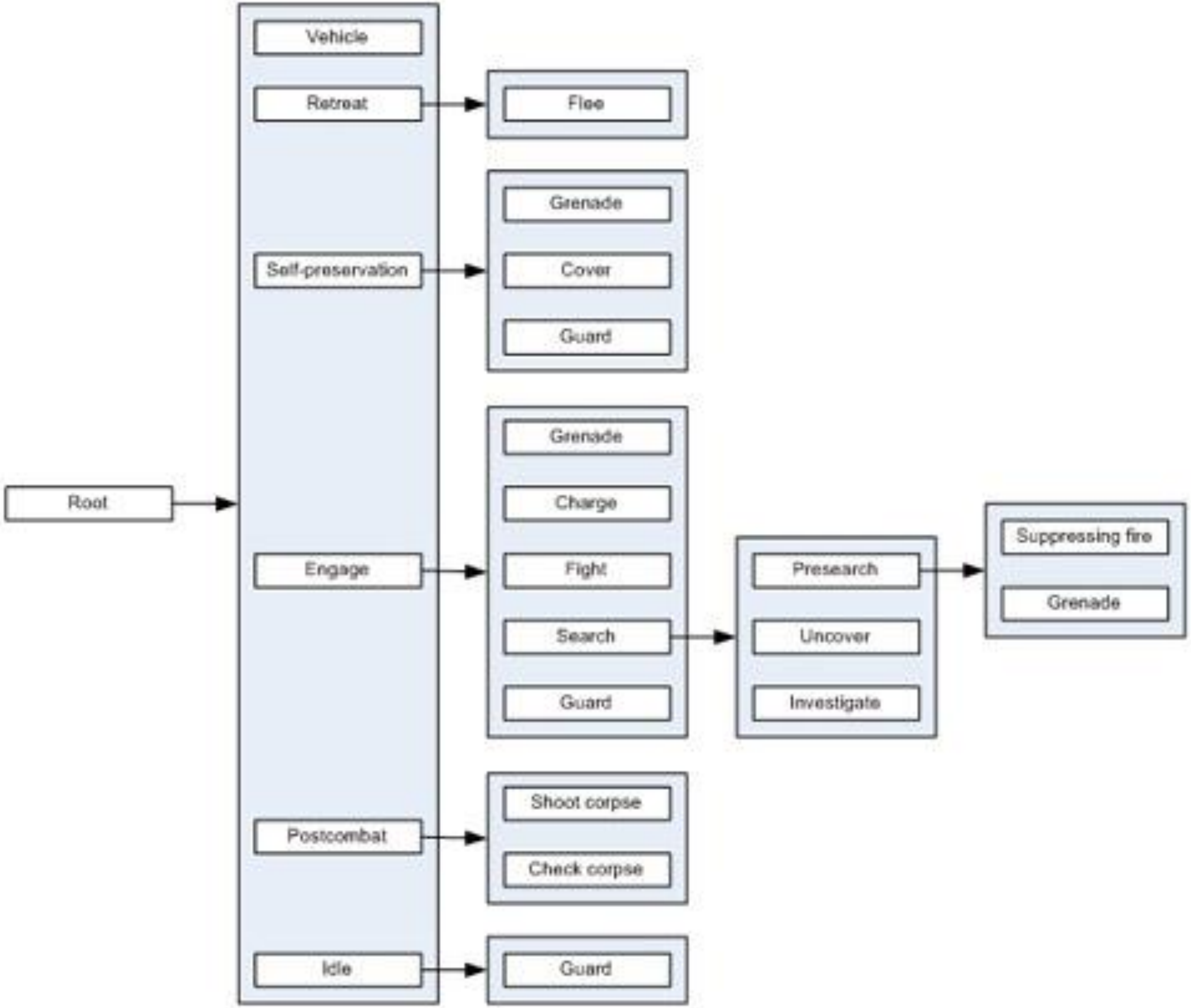
Behavior Trees – Actions

- Action **Walk**
  - Parameters
    - Character
    - Destination – location or another character

  - Success Reached destination
  - Failure Failed to reach destination (blocked/died/…)
  - Running On the way

- Init – called the first time the node is visited
- Process/Update – called every tick while the node is “running”

Halo 2



Root
Vehicle
Retreat
Self-preservation
Engage
Postcombat
Idle
Flee
Grenade
Cover
Guard
Grenade
Charge
Fight
Search
Guard
Shoot corpse
Check corpse
Guard
Presearch
Uncover
Investigate
Suppressing fire
Grenade

Fuzzy logic

- Decision trees work quite well, however it’s still not realistic enough
- Using absolute threshold values to decide
- There should be a range of values that allow for both decisions to happen
  - Such as the proximity test, sometimes we decide 5 meters is still too far, but sometimes we decide 7 meters are close enough
- The basic idea of fuzzy logic is that objects belong to multiple fuzzy sets by different amounts
  - A player partially behind cover can be in sets “in cover” as well as “exposed”, however we assign percentages for each set -> 60% in cover, 40% exposed

Fuzzy logic (2)

- The process of assigning the degrees of membership is called *fuzzification*
- In order to decide, we might have to *defuzzify* the membership degrees and give an exact result to which set we fully belong
- Simple fuzzification:
  - We can use cutoff values to give boundaries for fully belonging to a set
  - Proximity -> 2 sets “near” and “far”
  - 5 meters = near, 7 meters = far, between 5 and 7 meters, we can linearly interpolate

Fuzzy logic (3)

- Defuzzification is much harder
- From several degrees, we must choose the correct one
- Just generating a random number and considering which set is more likely to occur can work is some situations
- If we just take the set with the highest degree, we are likely ignoring the fuzziness at all
- If the result is just a number, it is much easier to defuzzify
  - An AI might be cautious, when combined with the fact that the player is behind cover, we generate a number that says how long the AI will take to aim
- For boolean values, we determine a cutoff and then compare it to the degree

Fuzzy logic (4)

- The real power comes from rapid AI prototyping
  - If (distance < 20 AND health > 1) then Attack()
  - If (player is close AND I am healthy) then Attack()
- We are using two fuzzy sets in the example
- We need to redefine the AND, OR and NOT operators for fuzzy sets
- AND -> R = min(A, B), where A and B are degrees of membership
- OR -> R = max(A, B)

Utility theory

- *“Utility theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states with higher utility.”*
- We take the current world state, think of what would happen if we performed some action
- What changes in the world state can be used to derive how much that agent improved its “happiness”
- Actions with the highest utility value are chosen and performed

Utility theory – examples

- Chess is perfect for executing the utility theory
  - If one action causes me to lose an important piece in the next move, this will probably have a low utility value
  - There are exceptions of course
  - Usually, you predict all possible outcomes in the next few steps and then choose the first step based on maximizing the utility value in your second step…
- A strategy game considers multiple things
  - Troop strength
  - Base/worker safety
  - Estimated enemy strength
  - Research level

Utility theory in practice

- We make a copy of the game state, perform the action, evaluate what happened
- Some actions might also take time to complete, the utility value is then utility-over-time instead (e.g. DPS)
- We usually can make localized decisions, meaning we do not always need the whole game state
  - In Sims, a sim usually cares only about himself
  - If the sim is hungry, eating will improve his happiness the most
  - So he goes to the kitchen
- Might require player prediction
- In FPS games, the agents have simple utility preferences
  - Agents will be preferring states where they continue to live
  - And prefer when the player will have low health as a result of their actions

Goal-oriented action planning (GOAP)

- Utility theory decides what an agent wants to do, not HOW to do it
- GOAP is working with *goals*, which are desirable world states that the agent wants to achieve through performing actions
- A goal could be to kill the player
  - Attacking the player is one action that could result in just that
- An agent has multiple goals, but usually only one active at a time
- Two-stage process:
  - Select the most relevant goal
  - Solve a goal by executing a sequence of actions
- The goal selection can be solved with decision trees, utility theory, …
- The second stage needs a special solution

https://www.gamedeveloper.com/programming/postmortem-AI-action-planning-on-Assassins-Creed-Odyssey-and-Immortals-Fenyx-Rising-

GOAP (2)

- Say a character is hungry
  - You have no food, so you need to create a plan to obtain food
  - Could be going into the woods to hunt animals, then extract the meat, cook it, and finally eat it
- Each action has a set of conditions it can satisfy, as well as a set of prerequisites that need to be satisfied
  - Eating food requires cooking food
  - Cooking food requires having raw food
  - Having raw food requires buying raw food
  - Buying food requires money
  - Money requires a job
- The algorithm walks back through these preconditions and identifies which actions need to be executed

GOAP (3)

- A sequence of goals might not exist
- There are lots of problems with world representation
  - Not only for GOAP
  - I desire a world state in which I am not hungry
  - I desire a world state in which the player is dead
  - We need to generate this world state with preconditions and effects
- Searching through possible actions is also a problem
- Search for a shortest (or least difficult) path in a graph of actions
  - There are many ways to solve a goal
- We always walk back from the desired state to the current state, trying to find a way that could work
- Quite advanced, but allows for very “intelligent” AI

Path-finding

- Not really an AI technique, more of a support technique for other AI
- Simply searching for the shortest path from point A to point B in a level
- We have nodes and edges
- Nodes describe points that the agent must be able to reach
- Nodes are connected by edges, which are just straight lines
- An agent may usually move along an edge to get from one node to another
- If you want to get to a neighboring node, you just rotate the agent and move him along the corresponding edge

Path-finding (2)

- However, moving along straight lines is highly unnatural
  - Except for robots maybe
- Nodes may be in a grid, resulting in not very smooth motion
- A few possibilities to avoid this:
  - Irregularly placed nodes
  - Allow each node to have a tolerance as to how close the agent must be to consider that he visited the node
  - Placing an interpolation curve (e.g. piecewise bezier curve) through the nodes

Path-finding (3)

- Edges may be unidirectional, bidirectional, even weighted
- Higher weight means a harder to pass route
- Weights could even be different for different types of agents
  - Flying units versus ground units, or units that can walk up cliffs
  - Results in different paths taken by different agents
- Weights can be dynamic
  - Building something on top of existing nodes sets the weight to infinity
  - Flying units might try to avoid guard towers, so the guard towers increase the weight of nearby edges

A* path-finding (aka. A-star)

- There are lots of algorithms that solve the path-finding problem
- A* is the most used one
- Relatively fast to compute
- Has lots of modifications

A* path-finding – the algorithm

- Each node of the graph is assigned 3 values
  - **goal** *g* – cost of the path up until this node
  - **heuristic** *h* – estimated cost from this node to the goal
  - **fitness** $f = g + h$ – the total estimated cost of the path passing through this node
- The magic happens by setting *h* to something meaningful
  - It can’t be greater than the actual cost of the remaining path
  - A simple algorithm sets the actual euclidean distance between the current and the end node as the value *h*
  - More accurate guess => the faster you find a path
- The algorithm maintains a set of nodes from which it can advance
  - The most promising has the lowest fitness
  - From this node, we explore the neighbors and calculate their fitness value

A* path-finding – the algorithm (2)

- Repeat until you get the goal node in your set
- If we run out of open nodes (we start with just one node), it means there is no path we could take

Path-finding – taking it a step further

- Another common technique is called a *navigation mesh*
- It is a simple mesh that describes **all** walkable terrain in the level
- Might be artist generated
- Much better is when it’s generated automatically
  - Might require some tweaking by artists or designers
- Triangles are nodes, edges are between neighboring triangles
- A* can be used, we just have to set the tolerance values based on the triangles

AI in Unity

- Very little AI support without plugins
  - Can use Unity's Animator for Finite State Machines
  - Can use Visual Scripting for Finite State Machines
- Has ML Agents package for reinforced learning
- Making your own is not that hard for simple games
- Lots of paid and a few free plugins
  - Behavior Designer
  - NodeCanvas
  - Apex Utility AI
  - Panda BT
- Simply coding it (no visual representation) is also OK
  - But think about configurability & the potential to modify it

AI in Unity

- Unity has built-in support for NavMesh Path-finding
  - Static NavMeshes
  - Dynamic obstacles and priorities
  - Rebuild NavMesh dynamically
  - Only for 3D
- For 2D and other uses, you can use A* Pathfinding Project
  - Has a free version & a paid one

References

- McShaffry, M., and Graham, D. *Game Coding Complete, Fourth Edition*. Course Technology PTR, 2012.
- Millington, Ian, and John D. Funge. *Artificial intelligence for games*. Burlington, MA: Morgan Kaufmann/Elsevier, 2009. Print.
- https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work
- https://www.gamedeveloper.com/programming/postmortem-AI-action-planning-on-Assassins-Creed-Odyssey-and-Immortals-Fenyx-Rising-