Minor Skilled: Unit AI

Rodrigo Sanchez 436703

Saxion University of Applied Sciences

Contents

Introduction

Without a doubt, artificial intelligence is one of the most useful tools in game development since its inception. Games like Pong and Pacman would find themselves stale or pointless without AI. Fast forwarding a couple decades, AI is still a core feature in multiple game genres, with various degrees of complexity. To adjust to the needs of bigger and more complex games, multiple techniques have been developed to help handle the complexity and improve the quality of AI.

The purpose of this minor is to explore some of these techniques, analyze their usability and implement them for empirical study. It is pertinent to mention that the focus of the minor is Unit AI, referring to the logic that governs individual units or groups of units; as opposed to strategic or global AI, which may make decisions on a global basis or command an artificial player as can be found in grand strategy games. The concept of unit AI tends to be connected with the local perception an entity has of the space that surrounds it, how it interprets an environment and acts upon it.

The main topics covered in this project are steering behaviors, pathfinding, and decision-making systems. The document first tackles steering behaviors and how they can be used to recreate group movement behaviors; they also offer great utility as a local avoidance layer of logic. Next is pathfinding, in which I delve into the construction of navigation meshes, as well as useful tools and techniques to execute the A* pathfinding algorithm over them. Lastly, I visited different decision-making systems currently used in game development, compared and tested them to determine their usability in different situations.

The structure of this document

Minor Skilled is supposed to follow the structure of these four phases: Analysis, Design, Production and Quality Assurance. However, since I focused on various small topics, I went through smaller cycles of all of these phases per technique of study. Thus it makes more sense to divide this report into sections per topic, instead of per phase. Also of particular note is that, because some topics were mostly focused on research rather than on actual production, subsections will only be present where appropriate.

Goals and motives

Motivation

For as long as I have been creating games, I have only ever developed rather rudimentary AI. Having faced some of the complications that come with managing a rapidly expanding AI, I have been motivated to expand my knowledge and skill in the topic to be able to create better games. I knew these techniques would prove very useful in the future since so many games implemented them. Learning about them would enable me to become a better developer.

Learning Goals

My goals are centered on extending my toolset in AI programming and thus becoming a better and more prepared game developer. With this in mind, I looked into three sections that form part of AI development that piqued my interest and with which I've had zero or limited experience with:

- Steering behaviors.
    - Understand and be capable of applying knowledge of steering behaviors; specifically as a local avoidance system and group behavior patterns.
    - Create a small library of behaviors that can be reused in future projects.
- Pathfinding and Navigation Meshes.
    - Improve my competence with the A* pathfinding algorithm and expand on its features. I would like to enhance and optimize it.
    - Learn how navigation meshes are created and their benefits over other types of spatial navigation datasets.
- Decision Making Systems.
    - Educate myself on contemporary logic handling systems that are commonly used across the games industry. Compare their benefits and disadvantages to be able to decide which pattern is appropriate for a given challenge.
    - Gain experience and competence in these systems for possible future implementations.

Expectations

A desirable outcome would present a tangible final product that represents my learning, such as a simulation test environment or a small game where all implemented techniques are observable, easy to analyze and play around with. This would demonstrate that I am capable of implementing and working with them in future projects and serve as proof of my proficiency in game AI. Additionally, I would like to improve and maintain my level of self-discipline.

Steering Behaviors

Originally designed in 1986 by Craig Reynolds, who experimented with creating computer simulations of birds flocking, steering behaviors offer a way of moving characters in organic, natural looking ways. Reynolds named his creations '*boids*', a play on birds, while, on the space of game AI these may be referred to as "autonomous agents", described by Buckland (2005):

> An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future. (p. 85)

This is to say that once an agent is set in motion, it doesn't require help to perform within the game world. The reason why this is so great is that steering behaviors offer an elegant and fluid solution to dealing with dynamic obstacles; this includes other characters. A great example of these behaviors can be found in Starcraft 2, where units can move in formations, execute swarm-like behavior and avoid getting in each other's way.

The movement of an autonomous agent can be broken down into 3 layers:

- **Action Selection**: Decides the course of action (Destination to go to).
- **Steering**: Calculates the trajectory, direction and speed.
- **Locomotion**: This is the method of travel of the agent. It differentiates the movement of a tank, fish, or camel. This can be separate from Steering.

Design

My intent was to recreate some of the most common steering behaviors, and build an extendable library of them that can be reused in future projects. Since steering is usually combined with action selection (pathfinding), this would also tie well with my other topics of research. Consequently, I made a list of behaviors to develop:

- Flee
- Arrive
- Pursuit
- Evade
- Wander
- Hide
- Flocking
  - Separation
  - Alignment
  - Cohesion
- Path following
- Leader following

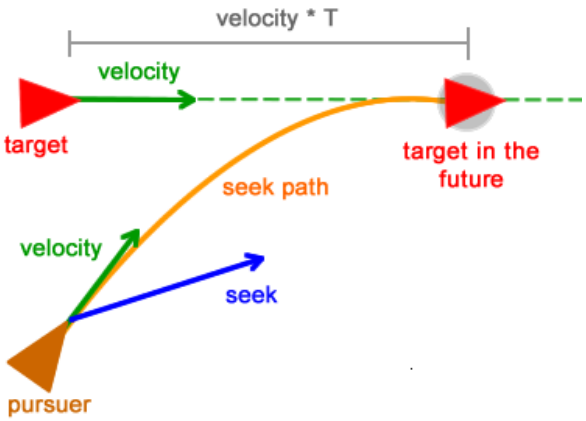


Figure 1. Depicting the Pursuit behavior (Bevilacqua, 2012).

Production

Steering basically consists of vector math and combining forces. The flocking behavior, for example, is the resulting effect of the Separation, Cohesion and Alignment forces combined together. As a demo, I created a contained area with obstacles, where the user can increase or decrease the number of boids active at a time. I also added a second type of agent that would serve for the rest to react to in behaviors like Pursue, Evade and Hide.

The main feature of the demo is toggling behaviors on and off through an Interactive UI that can be shown or hidden to the side of the screen. Steering is not mutually exclusive in most cases, and so the user can mix and match to create new, interesting behavior. I also enabled debugging tools to visualize the forces actuating over each agent.

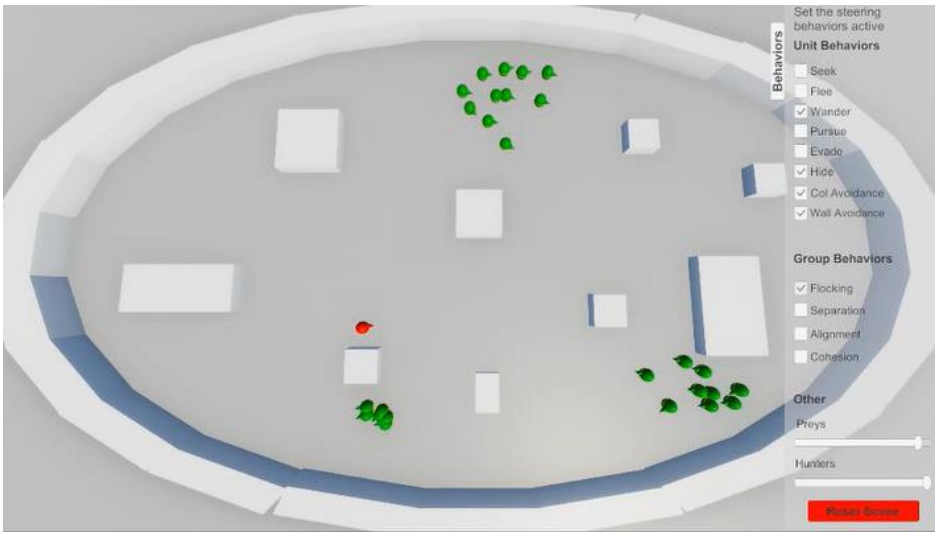


Figure 2. The side panel shows the functionality the demo offers to the user.

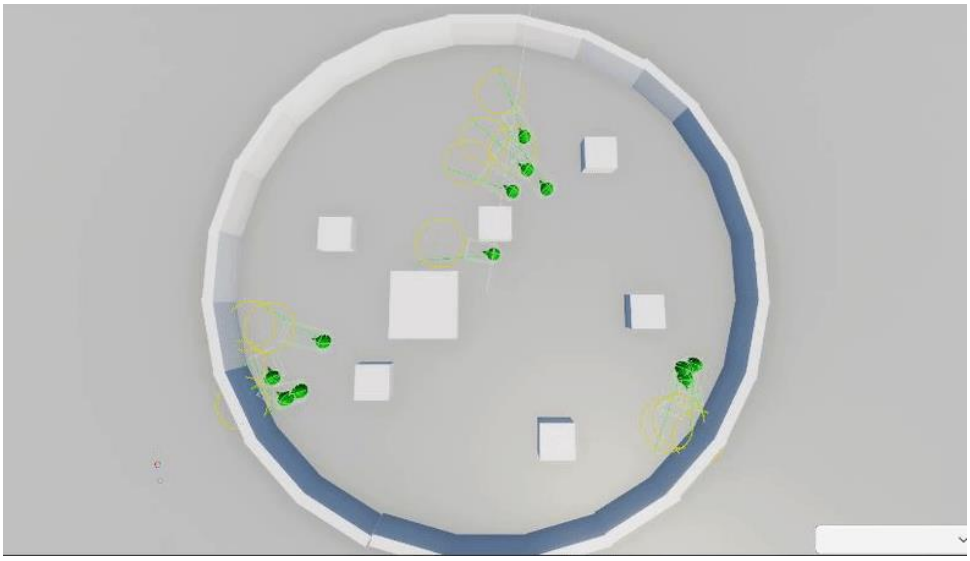


Figure 3. Debug of forces visualized when the Wander and Flocking behaviors are active.

Implementation

I used Unity for creating my prototypes. However, I wanted to be able to use this library of behaviors outside of Unity, and ideally be able to plug them into any environment or project with minimal or no adaptations needed.

With this in mind, I avoided using Unity's physics system with rigid bodies. Instead, I created an ISteerable interface, requiring any class implementing it to provide a position, velocity and mass values. Boid is a base class from which agents with specific logic can inherit; it implements the ISteerable interface and extends MonoBehaviour, this is my only tie to Unity. It also holds a static container with references to all existing boids.

The core of my library is a SteeringManager. The manager holds methods for all steering behaviors, exposes variables that affect their execution (such as a Flee radius), and calculates the resulting steering force. The resulting steering force is added to the velocity as acceleration, and the velocity is truncated to a maximum value. To get the steering force, all steering forces for each behavior have to be combined. This particular implementation does that through a "weighted truncated running sum" as explained by Buckland (2005, p. 120). In it, a running total equivalent to the added forces is kept. Each time a new force is added in a frame, the running total is compared against the maximum steering force, if the total surpasses the threshold, the force will be ignored this frame. The weighted sum allows to determine how much effect a behavior can have on the overall system, however tweaking these weights can become tedious.

Evaluation

I am very satisfied with how this turned out. I managed to implement and showcase all behaviors I set as a goal and constructed them in a way that will let me reuse them in the future. I understand the concept of steering, how to use and expand on it.

I would like to improve the way the steering behaviors are combined, as currently some get overpowered by others. A sum with prioritized dithering is a common way to improve this, however, Fray (2013) proposed a better system of concatenating forces. Fray's "context steering behaviors" solve some common issues that steering usually runs into, like equal opposing forces. Context behaviors consist of danger and interest maps where the forces are stored before processing them as shown in *Figure 4*. Moreover, for convenience the architecture holds all behaviors inside the SteeringManager; it would be better to separate them into their own, self-contained classes. Additionally, I would like to improve the accuracy of some behaviors like Wander and Collision avoidance, which often showed small mistakes in their trajectories.

Finally, it would be a trivial task to add more steering behaviors to my already existing library, thus I would like to eventually make it more complete. One of such is Queuing.

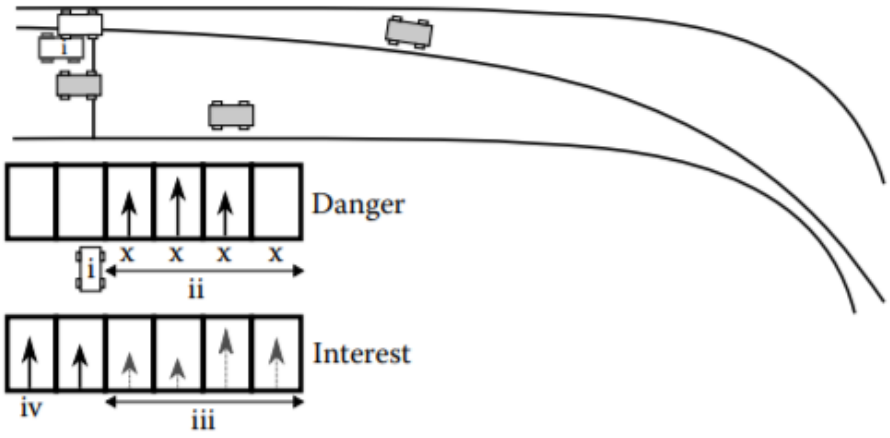


Figure 4. Processing the contextual maps of a final racing line offset (Fray, 2013, p. 191).

Pathfinding

Characters need to find their way through the polygon soup that is level design. Nothing quite breaks the immersion of a game like a character or monster repeatedly banging their face against a wall. Pathfinding techniques have improved over the years, both in pathing algorithms like A*, and in the space representation they operate on, such as grid and node graphs. These elements vary in accuracy and efficiency, but the right tool for the job is a phrase that certainly applies to them.

Needless to say, pathfinding is a core element of AI development and improving my competence with it would prove to be a valuable asset for any future projects. Moreover it ties really well with my previous topic, steering behaviors, as these tend to control the local trajectory while pathfinding takes care of choosing the destination of travel. They go hand in hand, and thus, it makes sense for me to tackle them one after the other.

I would like to preface this section by clarifying that there was no product developed for it. As I progressed my extensive research on these topics I came to realize the scope of them. I even tried to delve into and understand a couple exiting open source frameworks, however this was just too ambitious and it would have been better suited for a minor project of its own. Regardless, a lot of time was sunk into this topic, and what I offer here is a thorough look into navigation mesh generation and some techniques to improve pathfinding over navigation meshes.

Navigation Mesh

Pathfinding directly over terrain data is too expensive, instead, our search space representation is based on simpler data structures such as grid graphs, point graphs and navigation meshes. Navigation meshes (Navmesh for short) represent the accessibility of surfaces or spaces as a collection of interconnected convex polygons. They present multiple advantages:

Precise movement. A navmesh offers precise free movement for agents along the accessible surface area while other graphs only allow more fixed states that need extra logic added to get to the desired position.

Low memory footprint. Expanding grid or point graphs over large scale levels becomes exponentially costlier if we want to maintain enough granularity to have precise enough movement. For this, most big name games have switched to using navmeshes, as polygons can cover any area and are much more efficient on big maps.

Region accessibility. Similar to other search space representations, navmeshes provide information on which surfaces are accessible, the interconnectivity between different surfaces (polygons) and the ability to define regions with costs.

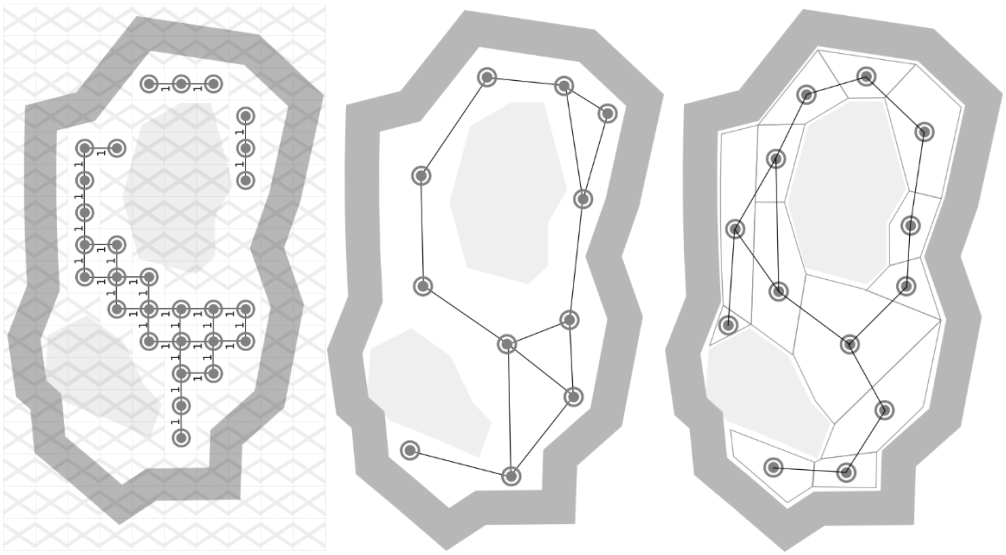


Figure 5. A grid graph, a point graph and a navigation mesh, respectively (Ceipek, n.d.)

Generating a Navmesh

Navigation meshes are typically built during development and stored as data as part of the level file. Dynamic navmeshes can be used to recalculate obstacles moving in the level at runtime, however, this is a fairly complex and expensive process. Building can be done manually, by having a designer create the polygons in a 3D program and placing them manually on the level; or, it can be automatically generated by taking the spatial geometry data (vertices and edges) and going through various algorithms in order to transform it into a much more simplified mesh of polygons with navigation data.

There is more than one way to generate a navmesh, Surface Merging and Voxelization are two popular methods of doing it. This document, however, will focus on the Voxelization approach, as it has the most information available online and easy to find. Before continuing, here is a small description of the surface merging process taken from Magnusson's (n.d.) webpage on navmesh generation:

> **Surface Merging**
>
> Take all the surfaces that are walkable and simplify them as far as possible by merging them with each other to convex polygons. Cull away the really small nodes and then take all things that block out the navmesh. Subdivide all polygons it covers, remove the ones below them and then merge the navmesh polygons again.

Voxelization is a technique where a 3D model is converted into a set of voxels that represents the model. Voxels are like 3D pixels, so boxes. The problem becomes how to turn the voxels back into polygons.

Mikko Mononen is the creator of *Recast*, an open source navmesh generation library. Recast is robust and well optimized, many companies implement it or have adapted it for their purposes, including Epic Game's Unreal Engine, Unity and Insomniac Games, developers of the Resistance series. As the information for understanding and utilizing Recast is abundant, the process for automatically generating a navmesh explained here will be based on it. Mononen (2009) listed the steps in the generation, taking as input an arbitrary polygon soup with triangles marked as walkable:

1. **Voxelize the polygons**

   This process converts the given 3D mesh into Voxels, hence the name. There are many viable algorithms for voxelization, like using octrees or rasterization of triangles. Recast uses the latter.

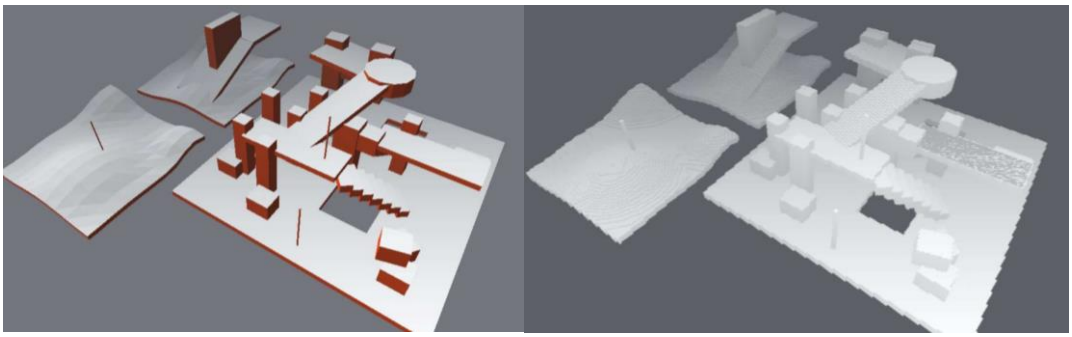


Figure 6. An arbitrary 3D mesh transformed into its voxel representation (Mononen, 2009).

2. **Build navigable space from solid voxels**

   Here the areas where characters cannot stand are filtered out (too steep slopes or low places). The agent radius is not yet taken into account. This navigable space could be already used as pathfinding data.

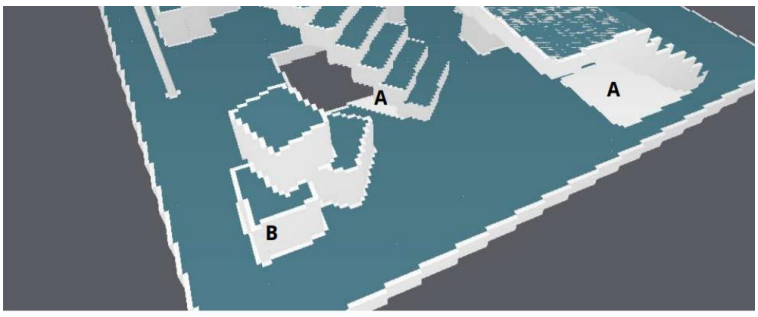


Figure 7. The non-walkable areas are filtered of the process (Mononen, 2009).

3. **Build watershed partitioning and filter out unwanted regions**

   This is a space partitioning method that simulates a water level rising that finds catchment basins before expanding. Each basin is filled with a new ID for creating the regions. Some extra filtering can be done to remove small unconnected regions and merge smaller regions together. If desired, the walkable surface can be eroded further to take into account the agent's radius. The resulting partitioning is a set of non-overlapping simple regions, and it can also be used as basis for generating a point graph by placing a node in the center of each region and making connections between them.

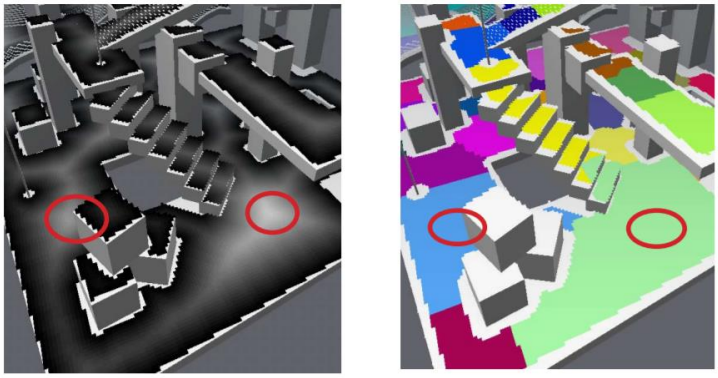


Figure 8. The watershed algorithm and space partitioning that stems from it (Mononen, 2009).

4. **Trace and simplify region contours**

   Start out of a point from a region edge cell and trace around the boundaries of the regions. At each step of the process, the cell corner points are stored to form the polygons, as well as the neighbor region ID. Recast uses the Ramer-Douglas-Peucker algorithm to further simplify the contours. This consists of taking the vertices at the region edges that form a simple polygon, and subdividing it based on the vertices of the region with maximum error (distance) of this polygon. This process continues until a certain error criteria is met.

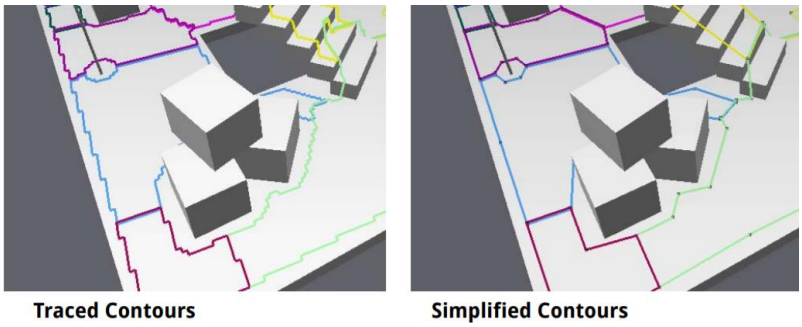


Figure 9. The Ramer-Douglas-Peucker algorithm executed to simplify contour (Mononen, 2009).

5. **Triangulate the region polygons and build triangle connectivity**

   Having a set of simple polygons, any triangulation algorithm can be used. A Delaunay triangulation is a very popular way to achieve this due to its efficacy (for example, Starcraft 2 uses a constrained version of this algorithm). When the triangles are generated they are combined and edge connectivity is established by looking up identical vertices.

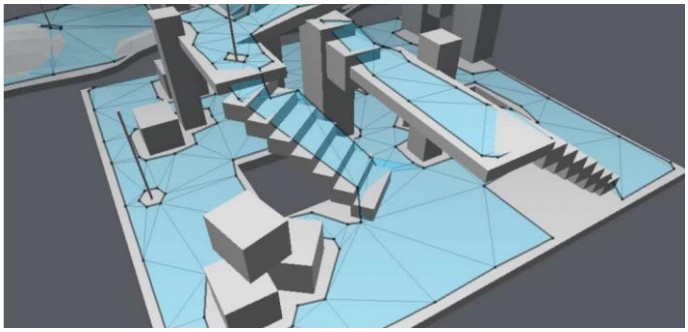


Figure 10. Finished navmesh post triangulation and edge connectivity (Mononen, 2009).

Although slightly modified, this is the same process used in some of the relatively newer titles from Insomniac Games Inc. as explained by Sambavaram (2011) in his GDC talk, *Navigation in Insomniac Engine*.

It is important to mention that there is no need to use a triangulated mesh, any type of convex polygons works for a navmesh; depending on the use case, one or the other may be preferred. Triangulation, however, is fast, convenient and offers some nice functionality in using the triangle vertices and edges as pathfinding data.

Dynamic obstacles

Having moving objects in a game means that agents need to adapt to their location in order to avoid phasing through them, or getting paths that land them stuck in a previously accessible area. Avoiding other agents and small obstacles that do not form part of the navmesh can be done with local collision avoidance methods. Steering behaviors commonly lie as an extra layer on top of A* pathfinding to avoid recalculating the navmesh at runtime and still get good, natural looking results.

Modifying a navigation mesh at runtime comprises of cutting a hole and plugging the newly data back into, or on top the existing navmesh. This is a very useful feature if a game needs to adapt to falling buildings, dropping boxes and other chunks of 3D data being thrown around or modified at runtime. However, it is an expensive process and depending on implementation, the whole navigation mesh may need to be recalculated to adapt to the new state of the world.

Robust systems often offer this feature, taking care of minimizing the time where the whole navmesh needs to be recalculated entirely; this usually entails isolating the region to be modified and reinserting it to the navmesh afterward or caching the data to return to the previous state, which avoids a recalculation in its entirety. Another method is to cache all the dynamic areas and have them associated to static areas, yet this can run into some memory juggling issues, like whenever a dynamic area has to be inserted or removed, all the other cached areas need to be removed from the cache and added again.

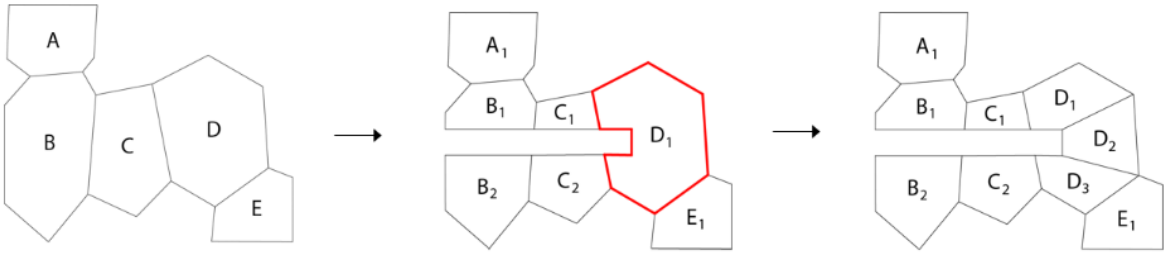


Figure 11. Illustrating how a dynamic obstacle is carved (Miles, 2006).

Pathfinding over a Navmesh

Navigation meshes allow for multiple techniques to improve the accuracy and efficiency of pathfinding.

Navigational raycasts are the first example, they can be very fast (100x faster than a collision probe). If the origin of the ray stems from the creature, we know which area the creature is in, and we can just test if there is an intersection between the ray and one of the edges of the area. If we hit an obstacle or wall edge, we're done. If we find a connection edge, we go into the next area and repeat the process. We could also limit the max areas the ray considers in the strange case where there is a lot of ground in a straight line. Navigational raycasts also allow us to very easily detect drop-offs.

Object localization is an important part of pathfinding and can be expensive if we iterate through every polygon. Commonly we want to know what vertex or edge the unit is closest to, we can also limit the search if we know what region the unit is in. An efficient technique used in Startcraft2, as Anhalt (2011) describes, is a "Cached jump-and-walk" point location system. In it, a sparse, regular grid is laid over the triangulation. Starting from the cached triangle the unit is in, the mesh is walked towards the destination triangle, and after finding it it's reinserted into the navmesh.

For traversing between disconnected polygons we can use Navigation Links, these have a start and end, placed as a connection between the disjointed areas. They indicate that an agent can traverse through it and if it's bidirectional or unidirectional. Links can have custom information stored in them, like the type of agents that can traverse through it, the type of link (jump, drop), and the type of animation to play while traversing it. These features are all used by Insomniac Games to control the navigation of their agents over complex geometry (Sambavaram, 2011). Similar to how links can have data tags associated to them, areas in the navmesh can be tagged with terrain type, as well as obstacle data and transition flags, like being able to vault or cover.

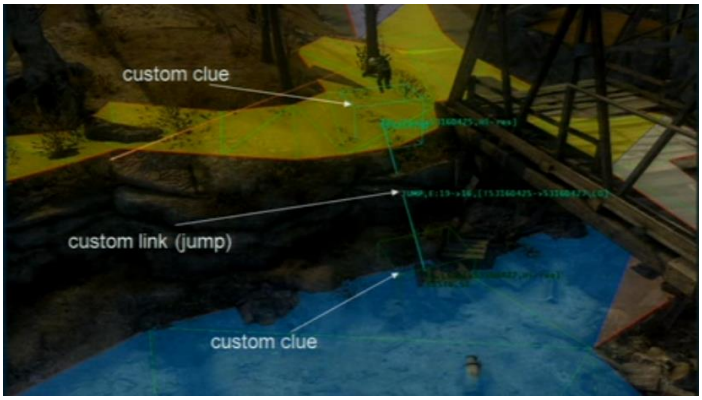


Figure 12. Custom links in the Insomniac Engine (Sambavaram, 2011).

Pathfinding

Pathfinding over a navmesh is commonly done using the A* algorithm and smoothing is applied after to get more natural movement. Hierarchical Pathfinding can be used to optimize the search speed in large scale maps; multiple games combine the use of Navmeshes with grids, or bigger navmeshes with different granularity (level of detail) that interlace on top of each other. In this way A* can be run on the simpler graphs before doing so on the more detailed maps, which makes search much faster. Another trick is to do Incremental Path Planning by allowing the lookup to travel only up to a number of cells on a single cycle, to break up the processing needed on a single frame.

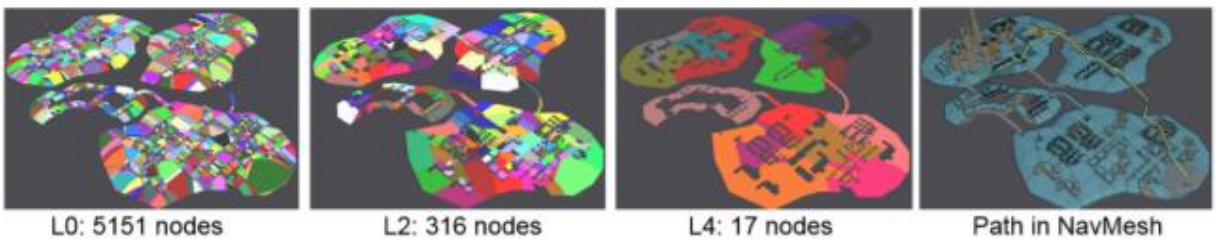


Figure 13. Hierarchical pathfinding with granularity levels (Pelechano & Fuentes, 2016).

Another standard technique found in most games to improve the efficacy of A* is the Funnel algorithm. It is used to find the shortest path between two points by looking for the next corner along the visible path. It does this by establishing "portals", constituted by the connecting edge of two areas of a navmesh. Taking the agent's angle of sight, the two vertices in the closest portal are evaluated for visibility and stored; this repeated with the next portals in the path until the visibility is broken. Finally, another useful technique that can be done before calling A* is checking a straight-line raycast to the destination point against any obstacles. This can avoid calculating A* at all, given that if a straight-line path can be taken, there is no further need for calculations.

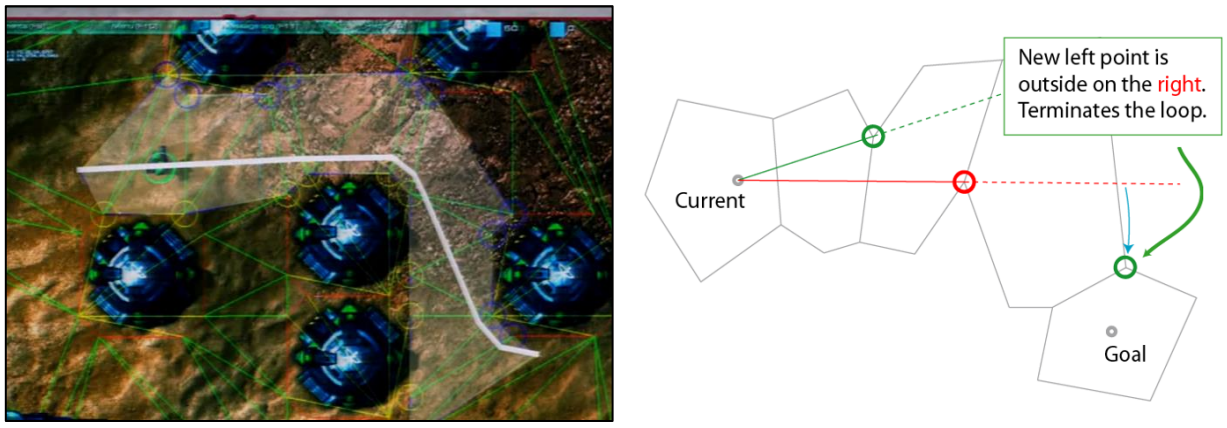


Figure 14. The funnel algorithm in a debug view of Startcraft 2 (Anhalt, 2011), and an image of the process (Miles, 2006).

Timeline

Initially, the plan was to craft my own navigation mesh, implement all the tools and techniques that would enhance my pathfinding algorithms and the move on to working on A*. I would use the A* algorithm developed during the Game AI course and expand on it.

Subsequently, when I realized the heft of the task at hand, the plan became to use a third party tool to provide the data for generating a navmesh, instead of starting from scratch. I strived away from Unity's navmesh because the data I needed was hidden from my access. Instead I turned to A* pathfinding project by Aaron Granberg, a framework developed for Unity that offered much more freedom, and was also an adaptation of Mikko Mononen's Recast.

After spending some days looking into this tool's source code, and struggling to understand and strip away the bits I wanted to use for my purposes, I gave up. It was simply too much for me to handle in a time where I could not handle much. The new plan was to just use Granberg's tool entirely, since it already implemented all features I wanted to develop myself. Sadly, due to time constraints I had to scrap this too, and move on to the next section of my minor.

Evaluation

Learning about navigation meshes and how they are created has been a thrilling experience. I originally wanted to focus more on pathfinding algorithms rather than the space they operated in. The small detour I took to understand these spaces better, however, quickly became a behemoth of information that my information-hungry self was dragged by. I do regret not focusing more on the A* pathfinding algorithm, as it is more likely that I will work with it in the future than creating navigation meshes. Objectively, this also took too much time from my minor and I have ended up with no physical or observable product from it, only research papers. That being said, I am satisfied with the research and I plan to put it in practice in my spare time, since I still find it an interesting subject.

Decision Making Systems

Handling the complex logic of agents can become a daunting task to build, and that quickly surpasses the level of management a bunch of simple conditional statements can offer. This has led to the creation of various systems or patterns to aid programmers write, extend and debug AI agents. While there is no universal name to call this group, they are hereby referred to as Decision-Making Systems. They can be considered the brain of an agent, for they handle top level logic if it, if their next action will be moving to a door or taking a seat to drink some wine.

Theory

AI is made out of logical statements, an AI that does not make decisions, simple as they may be like moving left to right, is not and should not be called “intelligence”. Given the fundamental nature of AI to take decisions, Decision-Making systems are an essential part of their development. This led me to choose them as the final topic to look into during this minor; my experience with them limited itself to state machines, and I knew other industry standard systems like behavior trees existed. It was my goal to educate myself in their theory, construction and use. While there are more than the ones mentioned in this document, this minor is focused on Finite State Machines, Behavior Trees and Utility AIs, comparing their usability and implementing testable prototypes.

Finite State Machine (FSM)

The most basic of AI architectures, state machines certainly add more structure to a bunch of otherwise disjointed rules mapped over the codebase. The most basic element of a FSM is a state; everything the agent needs to know about what it is doing is contained in the code for the state that it is in. The other part of a FSM is the logic for what to do next, whether it is switching to a different state or staying in the current one. Finite state machines receive their name for their properties of only being able to be in one state at a time, and the total number of states an agent can have is finite.

How a state switches to another is achieved through transitions, they can be anything from a timer to complex conditional logic that respond to the game environment. Transitions need to be specifically coded for all states that can transit to another. This is also the intrinsic problem state machines have, as the number states increase, the number of possible transitions between them does so as well – at an alarming rate (thirty transitions at six states). The real issue with this is that for every new state added to the mix, every single other state needs to be updated with appropriate transitions to make it accessible. This quickly becomes hard to keep track of and difficult to manage. Additionally, the transition logic needs to be in the same place as the state logic.

State machines are quite useful. They're easy to understand and are simple to learn and create. However, an agent can only be in one state at a time, they are difficult to extend due to the needed transition logic, and state nodes are coupled tightly to all other nodes, making it relatively easy to break the AI due to the fragile structure.

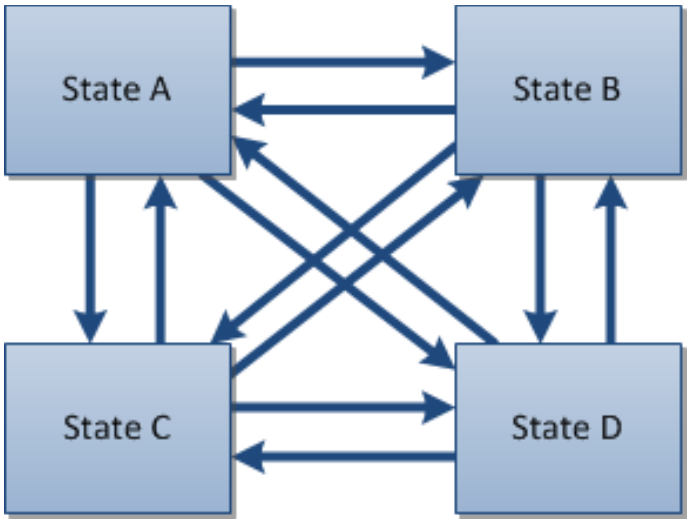


Figure 15. A simple state machine with all possible transitions illustrated (Mark, 2012).

Hierarchical FSM (HFSM)

To be one step ahead, a state machine can be modified into a hierarchical structure. The only difference between a FSM and a HFSM is that in the latter there is a hierarchy of choices, meaning that choices are evaluated at a high-level before getting into the details.

On simple wording, HFSMs are a small modification to FSMs where a distinction of *super states* and *child states* is made, it can be done just by having the child states extend the super state. This makes for a small hierarchy that theoretically helps with organization, as states inside a group don't need to know about specific states in another group, and can just choose to transition to a super state, letting it internally handle which child state to go into. HFSMs come with all the benefits and problems of state machines, with the added boon of slightly better organization. A largely growing HFSM can still become difficult to handle relatively quickly, but they handedly beat the simple FSM.

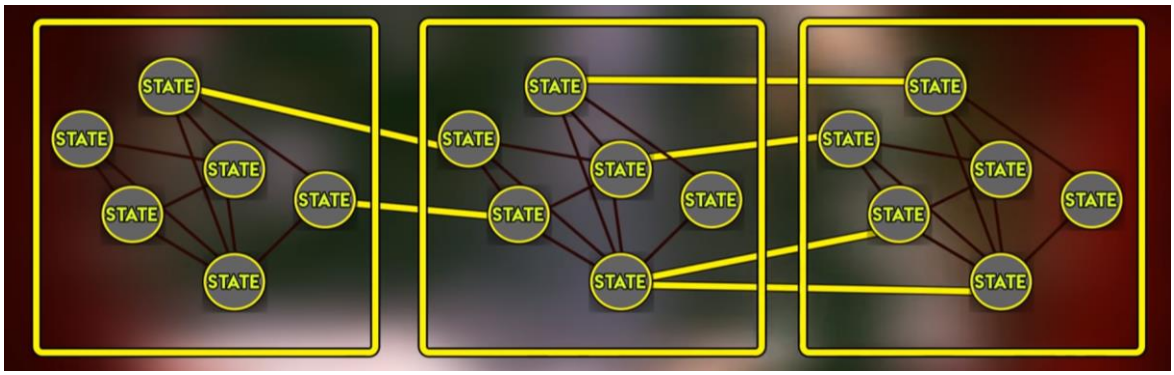


Figure 16. A HFSM, grouping similar states and with transitions between groups (Thompson, 2018).

Behavior Tree (BT)

A behavior tree is like an HFSM, except that the nodes in the hierarchy are modular, uncoupled and self-contained, running on a context basis from any part of any sub-group of the hierarchy. One of the main strengths behavior trees hold over FSMs is that they remove the decision logic from the behaviors and place it into a standalone architecture. This way, all decision logic is in a single location instead of being spread within all nodes, additionally it makes the behavioral nodes uncoupled and self-contained.

Any piece of a behavior tree is modular, and can be called from anywhere else in the tree. This makes it possible to skip to a different area on the tree and do something else if the current evaluation finds that it is no longer relevant. It also makes it easy to construct new behaviors, as old nodes can be chained and reused arbitrarily. One more advantage of BTs is that they favor themselves to be created by visual scripting tools, which makes for faster iteration cycles as well as making them easier to visualize and debug.

One of the signature properties of behavior trees is that they are constructed in a tree-like structure, the flow of decisions is order dependent, instead of relying on explicit transitions. The root node will evaluate its branches until a resolution is reached and returned to root. The next big property of behavior trees is that nodes can return one of three states: Success, Failure or Running. The condition for which status the node returns is defined by the developer.

The nodes of a behavior tree can be one of three types:

- **Composite**: They have one or more children, and will process them in an arbitrary order. They will return the success or failure to their parent, based on the outcome of the child nodes. Sequence and Selector are the most common composite nodes.
- **Decorator**: Also have a child node, but only one. Their function is either to transform the result they receive from their child node's status, to terminate the child, or repeat processing of the child, depending on the type of decorator node. A common example is the Inverter.
- **Leaf**: Lowest level of node and can't have children. They actually perform functions. They can also call other behavior trees and pass in the context.

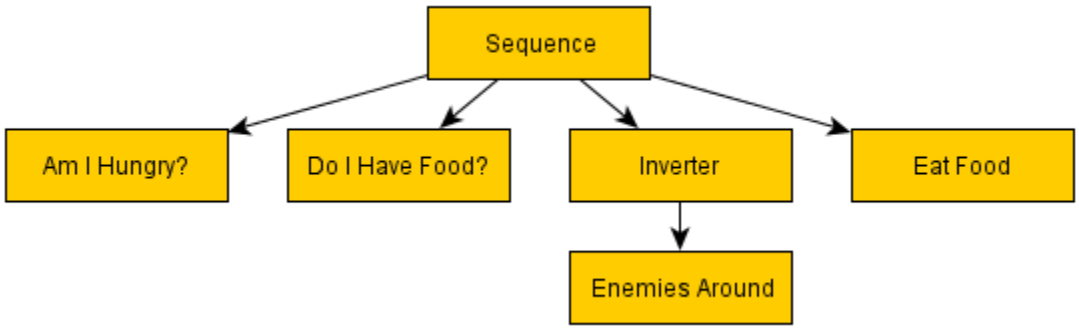


Figure 17. A simple behavior tree featuring Composite, Decorator and leaf nodes (Simpson, 2014).

An analogy is that Composite and Decorators control the flow (like if statements and while loops) while leaf nodes are the game-specific functions that actually execute behavior.

The Halo franchise is notorious for using behavior trees extensively. Halo 2 and 3 had behavior masks to disallow some parts of the tree as available to different types of enemies or when they exhibit certain emotions. Halo also uses a priority list updated on run time to decide on what nodes to execute and even implements an impulse system based on world events: e.g. Leader is killed event initiates a callback to execute flee (Isla, 2015).

Utility AI

The utility system gained popularity when Guerrilla games started using if for their AI in *Killzone 2*. It is optimal for AI with desires, emotions or other decisions based on priorization. They are simple to design because this desire-based approach allows to talk about them in human languages instead of talking about mechanics and transitions. The utility-based code, like behavior trees, is also a reasoner. It selects the state to go to next based on which action has the highest score. The behavioral code is self-contained without any transition logic, all the reasoning code stored in a single place.

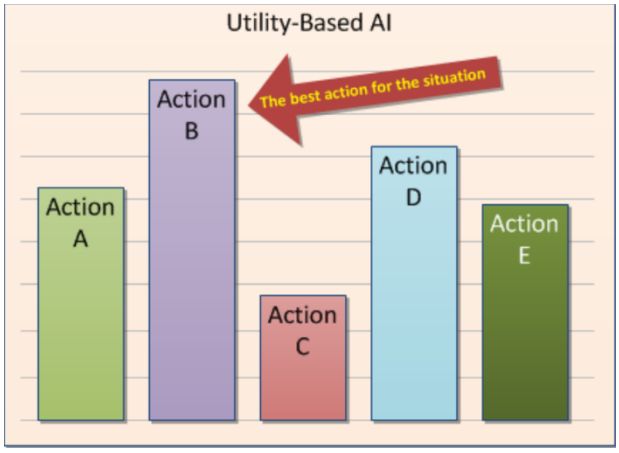


Figure 18. Depicting the behavior of utility-based AIs (Mark, 2012).

While behavior trees and FSMs are completely deterministic, and thus predictable, utility-based AI does not have a pre-determined arrangement of what to do when; in other words, it is non deterministic. Potential actions are considered by weighing a variety of factors, called "Scorers", and selecting the most appropriate thing to do. Since these scorers can be controlled with a simple table of values, where new conditions can be easily added or removed – they only change result, not the operation – It makes Utility AI extremely easy to extend. The drawback to this is that the non-deterministic behavior can lead to unpredictable results, and the designer might have to spend a large amount of time adjusting the table of scorers to get the behavior to act closer to how they want it to.

Action	Scorer	Score
Move to Enemy	Distance to Enemy	0-100
	Gun is not loaded	-100
Fire at Enemy	Proximity to Enemy < 50	75
	Cannot make it to cover	50
	Gun is not loaded	-125
Move to Cover	Is not in cover	50
	Proximity to Cover < 50	50
Load	Gun is not loaded	75
	Is in cover	50
	Gun is loaded	-125

Figure 19. A table of actions with scorers for a character engaging and enemy in combat (Rasmussen, 2016).

Design

To test the differences between these systems, I decided to try and recreate the same behavior in each different system. This would let me know the differences in implementation, the difficulty of it, the efficacy of the program and observe the differences in the behavior of the AI at runtime.

The designed demo follows an orc with main states of being: passive, aggressive or scared. In the passive state, the orc can patrol, wander or stay idle. In the aggressive state, it will try to seek and attack a dummy target. When scared, the agent will attempt to flee away and hide. Additionally, the demo will feature a UI panel where the current state of the agent can be identified for debug purposes. This UI will change depending on the active decision making system being showcased.

Production

As previously stated in the Design section, I created a set of behaviors for an agent to execute and proceeded to attempt implementing each system to rig these behaviors to. In practice, I actually had to recreate the behaviors for each system, since the transitions of the HFSM states made these classes non-reusable.

Creating the state machine interface was simple enough. However, I spent a large amount trying to figure out how to make it a HFSM; there are barely any available sources online for the implementation of these. I eventually figured out that the hierarchical modification comprises simply of having child states, like WanderState, extend super states, like PassiveSuperState. After making the appropriate changes, I had to deal with multiple bugs that came from the fragile structure of the FSM. Finally though, my demo was complete.

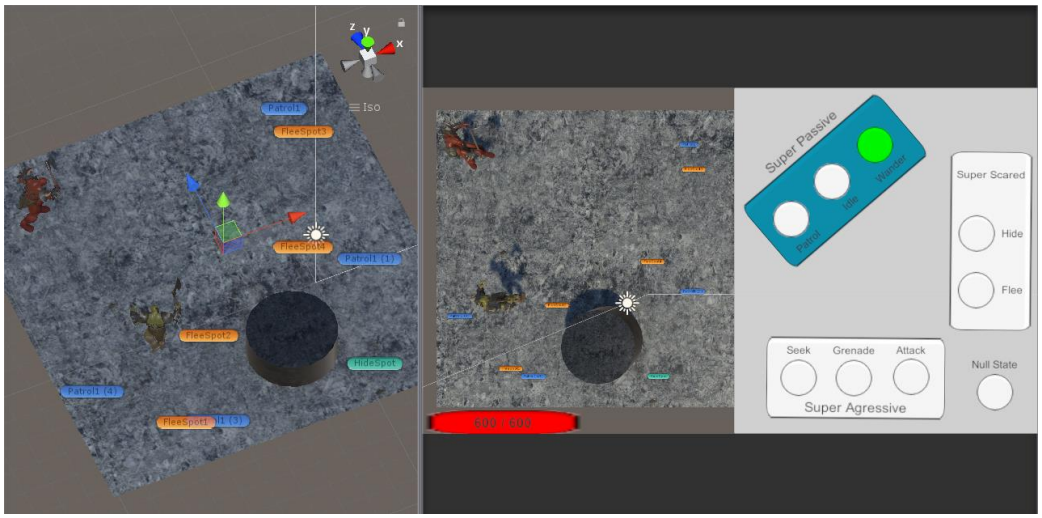


Figure 20. Decision-making systems demo, featuring the HFSM debug.

The behavior tree part of the demo was trickier to achieve. There is an abundance of tutorials on how to work with behavior trees using visual node scripting tools, however I did not count with one and these assumed the framework of the BT to exist already. I managed to build the architecture of a behavior tree, with some composite and decorator nodes, after which, I recreated the required behaviors and rigged everything in a working way. I unfortunately did not have enough time to create a demo a utility AI system.

Implementation

The hierarchical state machine is made up of a HFSM class which holds a list of all states and callbacks to the methods that switch between them. Each state is contained in its own class and implements the Enter, Step, Exit pattern. Transitions code is specified inside each state. States also extend a particular SuperState, which can by transitioned into and will manage the substates it's related to. This quickly became an annoying setup to work with, since I tried to have a large number of states to test my research. I had seven states contained inside three super states and I would not like to touch this code anymore after getting it to work.

For the behavior tree, my framework consists of an abstract Node class with an abstract Run method, which returns a NodeStatus. The NodeStatus is an enum with the standard Success, Failure and Running values. Finally, a node knows of its parent, if it has any, and I added the Enter pattern in the form of a virtual method. Following, I started creating some fundamental composite and decorator nodes. Base classes were made for them that handle basic functionality of holding, adding and removing child nodes.

Composite nodes:

- **Selector**: It will try to execute the children in order. If any child succeeds, it will immediately end execution and return success, otherwise, it a child fails it will move to the next in order and evaluate it. If all children fail, the Selector will return Failure as well. They are akin to if statements and act as an OR (||) check.
- **Sequence**: Children are evaluated in order as well, but it will continue moving through the list of children as long as they succeed. If any children returns failure, the entire sequence fails and returns appropriately. The sequence will only return success if all children return so as well. They can act as an AND (&&) check – The sequence is only continued if it doesn't fail at the start.

Decorator nodes:

- **Inverter**: Inverts the result of the node from success to failure and vice versa.
- **Query Gate**: Will check a Boolean value before proceeding to evaluate its child node.
- **Repeater**: Will reprocess its child note each time the child returns a result. Often used at root.
- **Repeat Until Fail:** Will repeat the child node forever or until it returns failure.

After having my framework completed I started coding the specific behavior in leaf nodes. This was much easier than the HFSM approach as I didn't have to take care of transitions; the code for each behavior was otherwise mostly identical.

Finally, my reasoning logic to change between behaviors made its home into an OrcTree class. This took care of transmitting context data from the agent to the behaviors that needed it. Moreover, OrcTree had the task to construct the tree and run it. Without a visual scripting tool I struggled to adapt into this way of building logic, but I quickly saw its advantages over HFSMs, the logic code is much cleaner and making different trees with unique behavior is trivial.

```
root = new RepeatUntilFail(
    new QueryGate (() => agent.health > 300, new Sequence(
        new MoveToNode(p1, moved),
        new MoveToNode(p2, moved)
        ))
    );
```

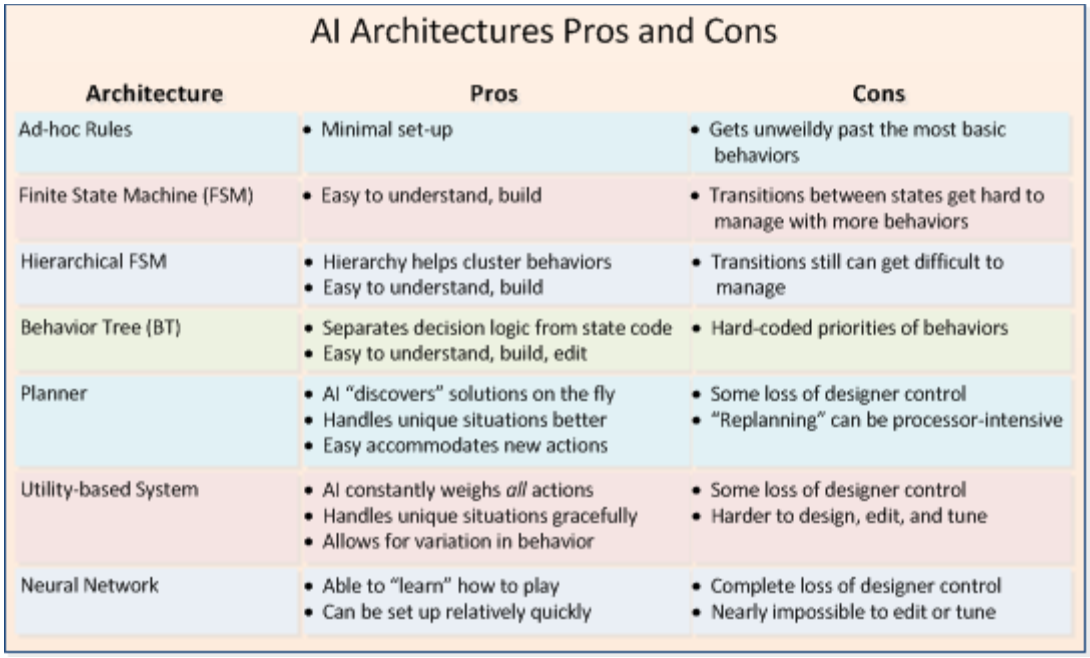
Figure 21. A basic behavior tree that moves an agent left to right, constructed by code.

I have to mention that I ended up wasting a sizeable amount of time atempting to add threading to my behavior trees. I have never worked with threading before, but from my research I repeatedly read that BTs synergise very will with multithreading. Alas, this will have to be an experiment for another time.

Both systems have references to UI buttons in the Unity hirearchy that show the activity of the system. These buttons light up when a node is currently running. For the sake of testing, this code is sprinkled all around the codebase and behaviors, which makes for some ugly spagetthi code. This is, however only temporary and should not pose a problem in a serious implementation of these systems.

Evaluation

While I did not manage to implement a utility-based demo, I am satisfied with my work in this topic. I now understand how to construct and work with behavior trees, as well as the theory behind them. I also learned how to modify a finite state machine into its hierarchical version. Most importantly, I now know the differences, advantages and disadvantages of each system and can hopefully discern which is best to use for a given situation in future projects. In figure 22, I present a chart created by Dave Mark and posted in his website *Intrinsic Algorithm,* that shortly compares all of these and a couple other AI architectures.



AI Architectures Pros and Cons

Architecture	Pros	Cons
Ad-hoc Rules	• Minimal set-up	• Gets unweildy past the most basic behaviors
Finite State Machine (FSM)	• Easy to understand, build	• Transitions between states get hard to manage with more behaviors
Hierarchical FSM	• Hierarchy helps cluster behaviors • Easy to understand, build	• Transitions still can get difficult to manage
Behavior Tree (BT)	• Separates decision logic from state code • Easy to understand, build, edit	• Hard-coded priorities of behaviors
Planner	• AI "discovers" solutions on the fly • Handles unique situations better • Easy accommodates new actions	• Some loss of designer control • "Replanning" can be processor-intensive
Utility-based System	• AI constantly weighs *all* actions • Handles unique situations gracefully • Allows for variation in behavior	• Some loss of designer control • Harder to design, edit, and tune
Neural Network	• Able to "learn" how to play • Can be set up relatively quickly	• Complete loss of designer control • Nearly impossible to edit or tune

Figure 22. A table comparing most AI architectures, their pros and cons (Mark, 2012).

Finally, I would like to keep working on these experiments in my spare time: I still need to delve into the implementation of utility-based systems and might also look into planners, as I did not know of these until near end of this minor. Eventually, I want to create a small simulation with multiple AI-driven characters that interact with each other and operate based on behavior trees or utility systems.

Reflection

While the resulting product did not meet the initial expectations I set for myself, I do feel that I have learned substantially due to the rigorous research I made during the course of the minor. On the other side of the coin, I devoted most of my allocated time to research, which impacted my product in a negative way. My desire to have a proper understanding of the topics before starting to work extended my research time to the point of being intrusive to production.

I made multiple mistakes during the course of this minor that directly or indirectly affected my productivity with it. They made me lose time that I could have otherwise used to improve the quality of it. The biggest time sink I had must have lost me a month; building a proper portfolio and resume along a professional looking online personality to be able to apply for my upcoming internship took much more time than I expected. I have also learned that working on the same project full time, for a long period of time with no one but myself to hold accountable is, unfortunately, a formula that doesn't work so well.

Adding to it all, personal circumstances of the previous year have left me in a state of mental exhaustion. This has taken its toll now, specifically with my work ethic, which lately has been quite poor. It is very much an external circumstance but I cannot ignore the effects it's had in my professional and personal life. That being said, the situation has improved since, and I encounter myself in an upwards personal swing. Time can be a beneficial aspect of life sometimes.

While the past cannot be changed, I would have been a bit less ambitious with what I wanted to achieve. The topics I chose to study are rather immense and could be topics for an entire minor by themselves. I instead chose to try and cram multiple of them in the same project. While I am glad I covered all the topics I did, I hope to continue this research in my spare time and make proper and complete implementations of it to further my own knowledge and expand my existing skillset further.

One area where I definitely have to improve upon is my planning, though the bumpiness of the past months did not help me keep in track with it. Making and sticking to my planning often felt more like a chore than an aid, and having no one but myself to really hold accountable for my success, it led to slacking too much. Self-discipline is definitely an area where I'm seeking to improve for the future.

References

Buckland, M. (2005). *Programming Game AI by Example.* Texas, United States.

Belivacqua, F. (2012). Understanding Steering Behaviors: Pursuit and Evade. Retrieved from https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-pursuit-and-evade--gamedev-2946

Fray, A. (2013). *Context Steering Behavior-Driven Steering at the Macro Scale.* Available from http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter18_Context_Steering_Behavior-Driven_Steering_at_the_Macro_Scale.pdf

Fray, A. (2013). *The Next Vector: Improvements in AI Steering Behaviors* [PowerPoint presentation]. Retrieved from https://gdcvault.com/play/1018230/The-Next-Vector-Improvements-in

Ceipek, J. (n.d.). Game Path Planning. Retrieved from http://jceipek.com/Olin-Coding-Tutorials/pathing.html

Magnusson, J. (n.d.). Navmesh Generation. Retrieved from http://www.gamedevpensieve.com/ai/pathfinding/navmesh/navmesh-generation

Mononen, M. (2009). *Automatic Navmesh Generation via Watershed Partitioning* [PowerPoint presentation]. Retrieved from https://files-cdn.cnblogs.com/files/mattins/MikkoMononen_RecastSlides.pdf

Sambavaram, R. (2011). *Navigation in Insomniac Engine* [Video]. Retrieved from https://www.gdcvault.com/play/1014484/Navigation-in-Insomniac

Miles, D. (2006). *Crowds In A Polygon Soup, Next-Gen Path Planning* [PowerPoint presentation]. Retrieved from https://slideplayer.com/slide/3602002/

Anhalt, J. (2011). *AI Navigation: It's Not a Solved Problem – Yet* [Video]. Retrieved from https://www.gdcvault.com/play/1014514/AI-Navigation-It-s-Not

Pelechano, N. & Fuentes, C. (2016). *Hierarchical Path-Finding for Navigation Meshes (HNA*)*. Universitat Politecnica de Catalunya. Retrieved from https://www.cs.upc.edu/~npelechano/Pelechano_HNAstar_prePrint.pdf

Mark, D. (2012). AI Architectures: A Culinary Guide (GDMag Article). Retrieved from http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/

Thompson, T. (2018). Cyber Demons | The AI Of Doom (2016). Retrieved from https://aiandgames.com/doom2016/

Simpson, C. (2012). Behavior trees for AI: How they work. Retrieved from https://gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Isla, D. (2005). GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI. Retrieved from https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

Rasmussen, J. (2016). Are Behavior Trees a Thing of the Past? Retrieved from https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.ph