Implementing Non-Player Characters in *World Wizards*

A Major Qualifying Project Report

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science by:

Nathaniel Carter

Project Advisors:

Professor Jeffrey Kesselman

Table of Contents

Table of Contents

Table of Figures

Abstract

World Wizards is a virtual reality world building tool developed in the Unity game engine. Users can create their own worlds by placing tiles and objects using the HTC Vive controllers. Users can import their own assets into the project to create specialized environments for their specific interests. This project expands the existing World Wizards project to support the creation non-player characters. A curve editor is created that allows users to edit the personalities and behaviors of the characters, modifying their decision-making process without having to write any code.

Introduction

History of Artificial Intelligence in Video Games

Ever since the earliest days of video games, there has existed a need for the simulation of in-game opponents and characters. While some games have used human players exclusively to control each character, many more rely on artificial intelligence to produce complex behavior and provide interaction for the player. Most importantly, the use of simulated opponents in video games allows them to be played alone, without the help or involvement of any other human. As almost every game project implements their own artificial intelligence system, it is valuable to look back on the history of artificial intelligence in video games and observe how the landscape has evolved over the years.

Some of the first and most revolutionary video games, *Tennis for Two* (1958) and *Pong* (1972), are examples of 'human-only' games. Players competed directly against one another similar to how a game of tennis is played in the real world. These games relied on multiple players being present and could not be played alone. This precedent, however, quickly shifted to favor a single-player experience as the game industry developed. Games such as *Space Invaders* (1978) and *Pac-Man* (1979) pitched player against machine, producing an entirely different dynamic of competition. According to game designer Raph Koster, there are a few primary elements that contributed to this shift. The difficulty of sitting multiple people in front of a small screen, the invention of cooperative play, and the introverted personality types of the players and developers lead to the popularity of player-versus-computer gameplay [2]. As the interest and audience of video games rapidly grew, so did the need from stronger opponents. In order to

provide a compelling and novel experiences for these players, the decision-making processes of these games became increasingly complex.

The term artificial intelligence, commonly referred to as AI, is used to describe behavior generated by a program that is either analytical, intelligent, or humanlike. AI, within the context of video games, can mean anything from scripted character movement, to complex machine learning that attempts to emulate human personalities. Because of the limited perception skills of humans, simple AI implementations often seem much more complex to the player than they actually are. This is useful as it allows for game developers to keep solutions simple. If the player thinks that an AI is intelligent, then it might as well be.

Chaos Theory and Emergence in AI

A critical concept to understand when developing game AI is emergent behavior. The idea of emergence comes from the study of chaos theory, which attempts to predict chaotic and seemingly random behavior that can arises in various systems. While many systems appear to be random, behavior in deterministic systems can be modelled and understood mathematically [5]. Emergence refers to the surprising behavior that the systems can produce that is not represented by each of its components. Emergence is particularly useful for AI design because the combination of a few basic elements can create a product that is much more intricate than the apparent sum of its parts. A powerful example of this is the enemy AI in the popular game *Pac-Man* (1980). Each ghost has their own extremely rule mechanic that governs its behavior. One ghost follows a few tiles behind Pac-Man, while another tries to move to the tile a few squares ahead [3]. When all of these unique ghosts, “Blinky”, “Pinky”, “Inky”, and “Clyde”, are combined together, each with their distinct personality, they produce what seems to be an intelligent and coordinated assault. Specifically, their cooperative behavior produces a “deep and

challenging game that players still strive to master, 30 years after its release" [3]. The simple sets of rules that dictate the movement of each ghost come together to create a robust and challenging AI system, despite its simplicity.

Figure 1: Ghosts chasing the player in Pac-Man (1980)

Another example of emergence can be found in the flocking simulation developed by *Craig Reynolds* in 1986. When placed in groups, his geometric creatures, called boids, performed incredibly realistic flocking behavior. The combination of three basic functions, separation, alignment, and cohesion, produced movement that was extremely similar to the actual flocking patterns produced by birds and fish. While each boid has access to the data of the entire flock, they are only concerned with the information of their nearby neighbors in their movement calculations [6]. This simple approach creates a compelling result that is far more efficient to develop than a planned or scripted system.

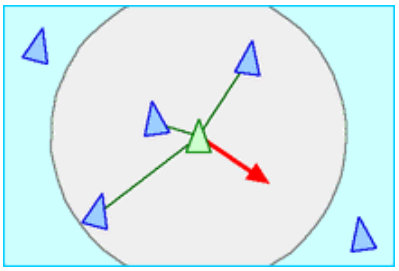


Figure 2: Separation: Moving away from nearby boids [6]

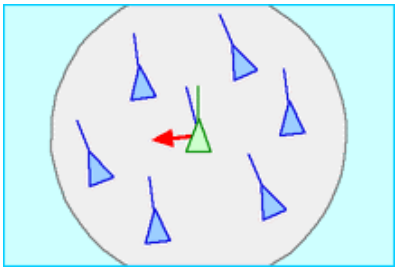


Figure 3: Alignment: Moving in the same direction as nearby boids [6]

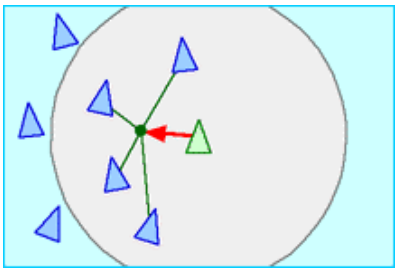


Figure 4: Cohesion: Move to the average position of nearby boids [6]

Ultimately, the Boids project reveals the high potential for emergence in AI systems. Adding or removing components from a system can create bizarre, unexpected, and sometimes extraordinary consequences. It can be valuable to embrace and explore the chaos, rather than try to hardcode specific results. Understanding this concept of emergent behavior when designing AI is imperative because it allows the designer to fully explore the problem at hand and can potentially lead to simple yet innovative solutions.

Non Player Characters

As the years went by, and technology rapidly advanced, there became an ever increasing demand for more complex AI. With the market transitioning from the 2D to the 3D world, the focus on realism and human-like characters became more prominent. This change, however, revealed a weakness in game character intelligence. With humans being experts at identifying and evaluating other humans, 3D characters that looked like themselves were easy to be picked apart. It became even easier for players to spot flaws and notice unnatural performances in games. This change pushed future AI implementations to become more complex, matching the rising player expectations.

AI controlled characters in video games are commonly referred to as non-player characters, or NPCs. While this term can be vague, covering many different types of characters in game, almost all NPCs can be reduced down to two core elements: an AI component that controls behavior and a graphical component that demonstrates its current state to the player.

Despite video games having existed for almost half a century, there is still no surefire method that is successful in all applications. The design and implementation of these characters is often dependent on the goals and requirements of the current game. This can be clearly seen in the comparison between AI in the modern games *Alien Isolation* (2014) and *FIFA 17* (2016). In *Alien Isolation*, the monster has a complex menace system which controls when to scare the player and when to back off [9]. The monster moves throughout the player in a non-scripted pattern that keeps the player filled with suspense. In contrast, NPCs in *FIFA 17* abide by a very specific set of rules regarding spacing and football game strategies [10]. These characters attempt to emulate how actual football players would approach the game. When developing non-player

characters for a game, the specific needs of and rules of the game dictate the implementation of the character

World Wizards

World Wizards is a virtual reality world building tool made in the Unity game engine. The user places tiles with the HTC Vive controller simply by pointing their arm. These tiles snap to a grid for simple and intuitive level construction. Technically inclined users can import their own tilesets which contain many different tiles and objects that they have created. This modularity allows form users with many different interests to take advantage of the World Wizards system. Additionally, focusing on user generated content allows the project to grow far beyond its initial implementations.

World Wizards uses virtual reality to establish a sense of immersion for the user, allowing them to construct worlds from a first person perspective. Traditional level editors as seen in Unity and Unreal engine place the designer in a more birds eye view, making for an unnatural experience. Placing the designer directly into the scene gives them a newfound appreciation for the scale of the environment, allowing them to understand exactly how player will experience the world. Additionally, Placing tiles in virtual reality is a fun experience that keeps the user engaged with their work.

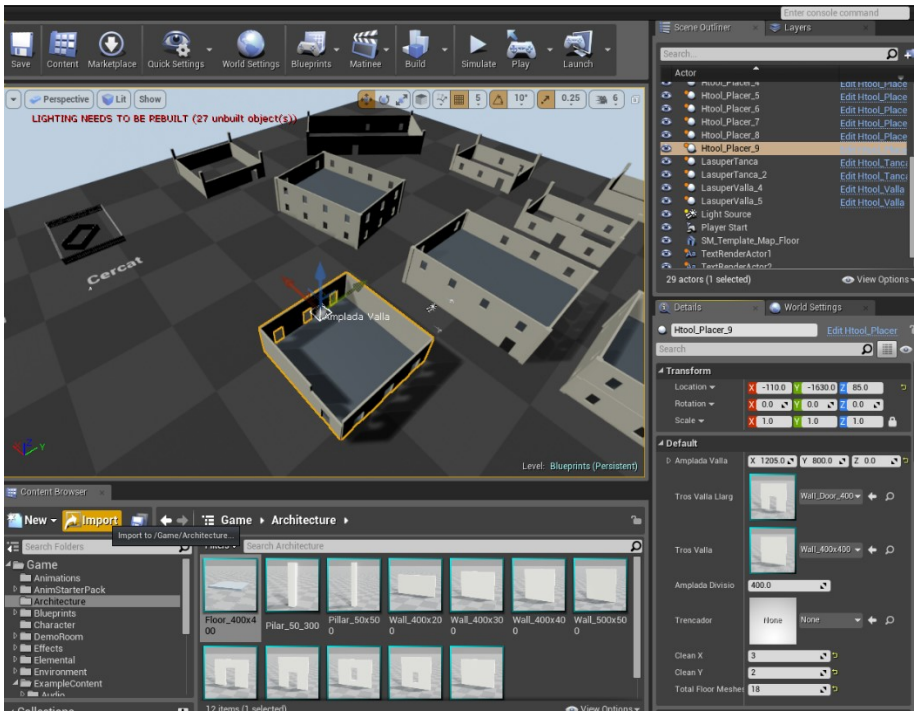


Figure 4: Top down view of level editor in Unreal Engine

Software and Packages

Unity

Unity is a popular game engine that supports the creation of both 2D and 3D experiences. It features the powerful scripting language C#, sleek user interfaces, and compatibility with all major platforms [11]. Among other things, Unity is known for its community. There are many forums dedicated to tutorial content and debugging support. This is extremely helpful for developers who are just starting to use the engine. Unity provides a build in marketplace that allows users to share both free and paid assets. This encouraged developers to share work with one another, improving the capabilities of the community as a whole.

A* Pathfinding Project

The A* Pathfinding Project is an existing Unity package that provide a wide variety of heavily optimized pathfinding systems. I chose to use this existing library to help with pathfinding as it offered robust, user tested algorithms and advanced geometry scanning. The functionality used from the A* Pathfinding Project will be gone over in more detail throughout the paper.

Implementation

Designing a NPC Model

It is important to note that the goal of this project is not simply to implement characters into the World Wizards project. More specifically, the goal is to create a model that the user an work with to implement many kinds of characters. This model needs to support many different types of character that a user might want to create, as well as the various personalities that they might have. It is critical that this functionality be achieved without the users having to write any new code.

One of the restrictions that I placed on NPC model was limiting it to bipedal characters. This decision was made with the consideration of scope, simplifying the characters and decreasing the amount of functionality to be developed. Additionally, bipedal characters can all be animated with the same Animation Controller which will be detailed later. Through this section I will explain the challenges involved in creating this model and how they were overcome.

NPC Components

Decision-Making

When designing an AI for a NPC there are three main problems to solve, these being decision-making, planning, and locomotion. The decision-making component of an AI has one major purpose: to consider all relevant game information and decide on the best action to take. While the planning stage deals mostly with pathfinding through the physical game world, decision-making is the stage in which the goals are set. It is important to understand that the goal is not create an AI that always makes the right choice, but to create one that acts based on its own interests. To clarify, the most compelling and realistic characters are the ones that sometimes make suboptimal choices because of their timid or aggressive personalities. From a design standpoint, it is better to make opponents compelling and realistic rather than elite and unbeatable. This can be explained through the concept of Flow as described by psychologist *Mihaly Csikszentmihalyi* in 1970 [7]. His research found that when the challenge of a game and the skill of the player are balanced, a state of focus called Flow is achieved. In this heightened state, the player loses self-awareness and becomes hyper-focused on the tasks at hand.

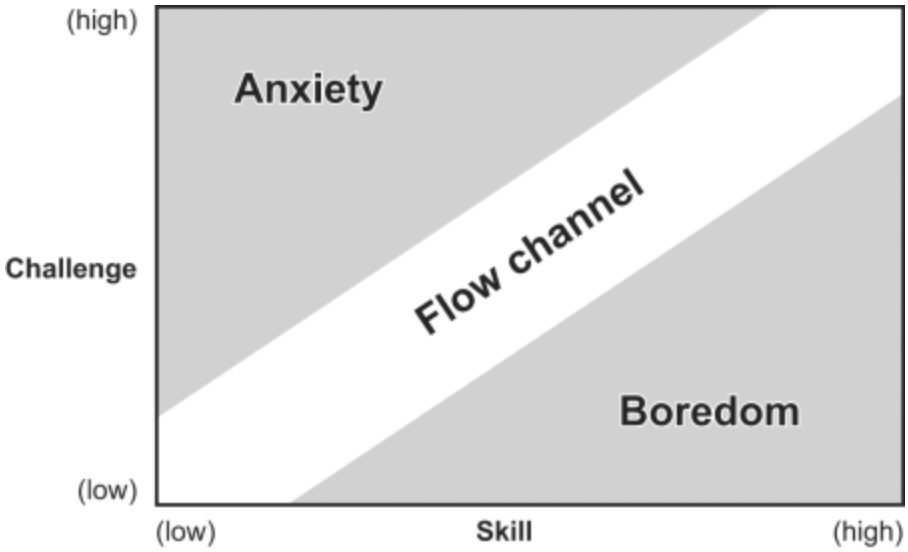


Figure 5: User engagement during play

The ultimate purpose of a character in a game is to offer a specific level of challenge to the player. Too much challenge and the player will become frustrated and quit. Too little and the player will become bored and uninterested. Taking flow into account is important when designing the decision-making process because the characters choices need to keep the player in this state of Flow as effectively as possible.

Planning

Planning refers to how the character takes in information about their environment and creates a course of action, or path, from that data. For example, if the goal of a character is to grab a sword from the basement of a house, they must be able to observe their environment, creating a path down the staircase and avoiding obstacles along the way. While this problem seems trivial to a human due to our subconscious planning ability, it can be a complex problem to solve, especially in dynamic environments.

Locomotion

Once a plan has been formed, the character needs a mechanism for moving throughout the world. While running or walking are the first things that come to mind, locomotion can include any action that helps the character move from one location to another. Crawling, jumping, swimming, are all forms of movement that can be important to consider depending on the environment of the game. Locomotion is conceptually the most simple problem to overcome but can present challenges when it comes to animation.

3D model and Animation

It is critical that characters have a graphical representation so that the player can properly perceive their current state. In the context of 3D games, this usually manifests as a 3D model that is animated based on its current state. Accurately displaying AI behavior to the player is imperative because it allows the player to understand and predict its future behavior. Given the limitations of human perception, the only method through which the player can understand the AI is observation. Because of this, the entertainment value of AI systems is directly limited by their observability. It does not matter how complex or dynamic a character is if the player cannot perceive what is going on.

Locomotion

In my initial locomotion implementation Unity's Character Controller component was used. After difficulties moving over terrain, this system was replaced by Unity's Rigidbody Component. This component allows the charcter to be affected by physics forces and move in a realistic way. Because of the physics simulation that it calculates, events like explosions or knockbacks can influence the character similar to how they would be affecting in the real world.

The Rigidbody is controlled using the A* Pathfinding Project's AIPath script. This script physically accelerates and turns the Rigidbody to arrive at a given destination. The speed, rotation, and acceleration of each specific character are set by the user.

Planning

A* Pathfinding

The A* search algorithm is one of the most popular and commonly used pathfinding methods. The graph used in the A* pathfinding project uses this algorithm to calculate paths around obstacles. The algorithm scores each tile of a graph by combining g(n), the distance from the start, and h(n), the value of the heuristic function.

$$f(n) = g(n) + h(n)$$

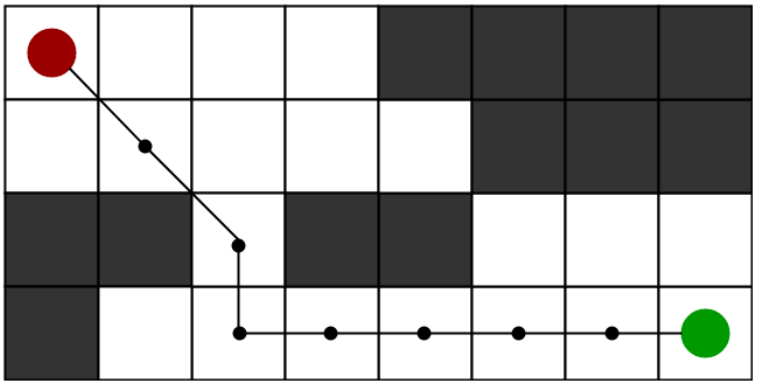


Figure 6: A grid search*

The heuristic function is usually specific to the problem at hand but Euclidian and Manhattan distances are common examples. Manhattan distance is a very fast heuristic because it simply runs comparisons on x and y values rather than use the Pythagorean Theorem. This method is faster because it avoids the need to calculate squares which are computationally slower

to perform. The A* algorithm is optimally efficient, given a heuristic function *h* [8]. This means that an A* algorithm will complete a search in the same amount of moves as any other search algorithm that uses the same heuristic function. While other search algorithms can compute paths more quickly, they do so at the cost of not being optimal. This means that the path they return may not be shortest path available. A* offers a good balance of speed and optimality, while also being conceptually simple.

A common problem with path produced on a grid is the zig-zag pattern that occurs when moving in a diagonal direction. This can be seen in **Figure 7**. A character following this path would move in an inefficient and completely unnatural manner. This issue can be resolved by applying a smoothing algorithm to the A* path. The A* Pathfinding Project offers multiple smoothing algorithms in the package.

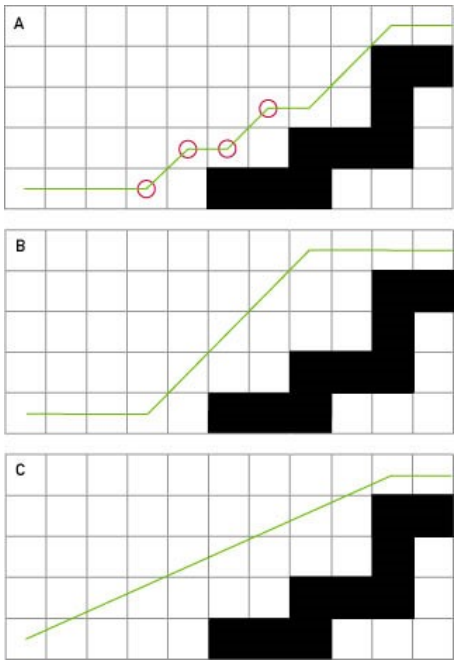


Figure 7: Diagonal movement problem

The Simple Smooth Modifier does a good job of creating a realistic path from the jagged diagonal path generated by the A* algorithm. It accomplishes this by subdividing path points and using curved lines called splines when necessary. Once these paths are created they can be used by the character or a movement plan.

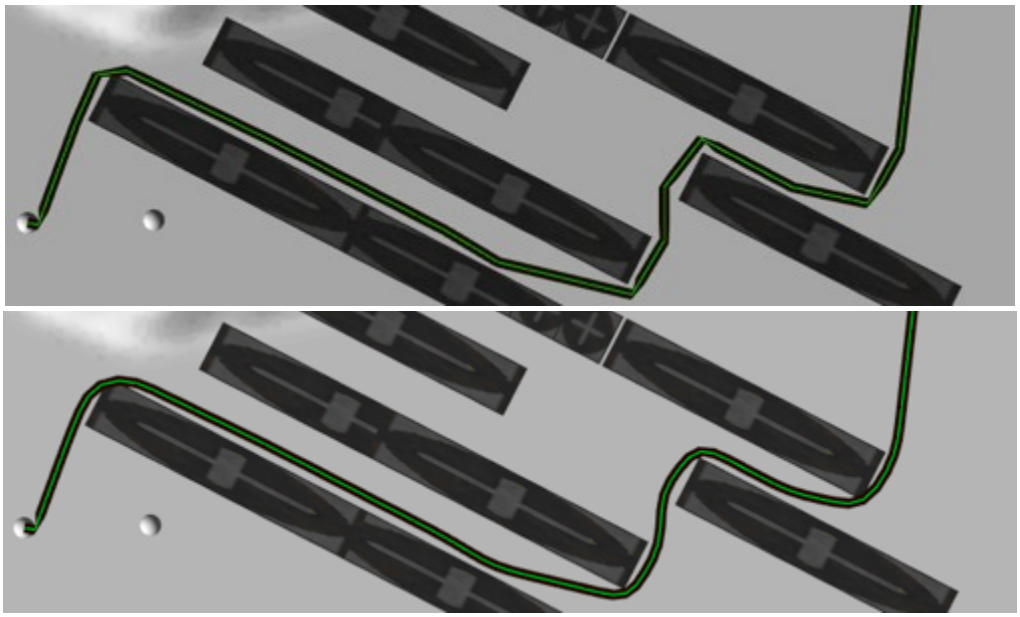


Figure 8: A path with and without the Simple Smooth Modifier*

Layered Grid Graph

While the A* pathfinding project offers multiple different graph types, the Layered Grid Graph, or LGG, was the most reasonable choice for use in World Wizards. Of the advantages, the most important was that it allows for vertically overlapping graphs, meaning it supports pathfinding on multiple floors stacked atop one another. This allows for verticality in environments as seen in houses or dungeons. The multi-storey building shown in **Figure 9** could only be supported with the functionality built into the LGG.

Figure 9: Multi-storey building

Another quality of the LGG is its scanning speed. While other graph systems have to be methodically placed or precompiled, The LGG can be quickly generated during play. This allows for tiles to be added and deleted during runtime as the system can quickly create a new graph. While traditional pathfinding implementations can get away with these precompiled solutions, the dynamic nature of the World Wizards tile system makes this impossible. The runtime scanning of the LGG makes it the perfect fit for World Wizards.

The LGG was integrated into World Wizards by including a layer type in each tile. By marking each tile as ‘terrain’ or ‘obstacle’ the scanning system create a graph of the path-able terrain. Tiles that are market on a different layer are ignored in the scanning. **Figure 10** shows a graph generated during runtime. The stairs and floor are path-able terrain, while the walls represent obstacles.

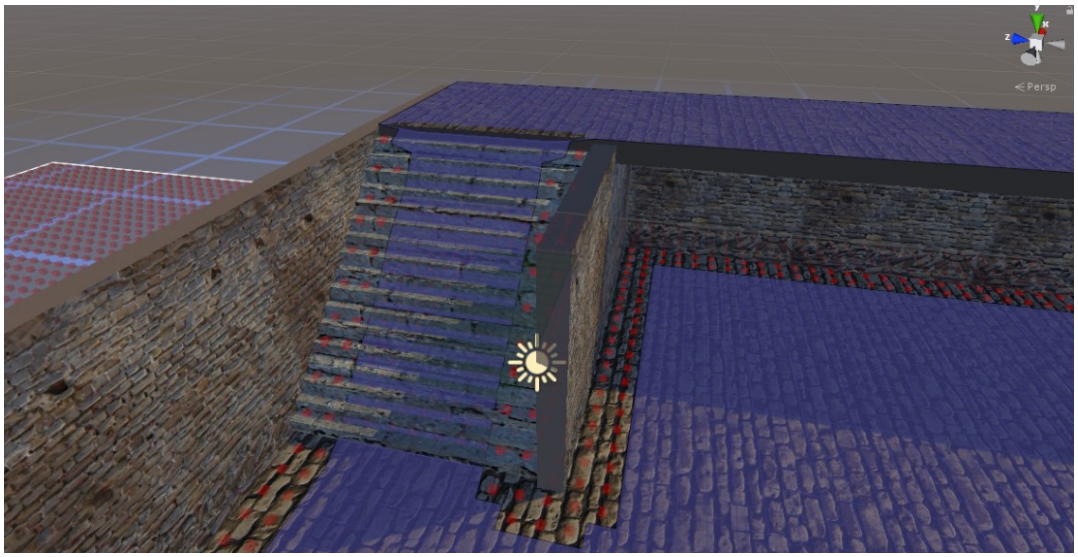


Figure 10: Layered Grid Graph creation with a staircase

Figure 11 displays grid scanning on complex geometry. This scanning system can support intricate environments such as doorways or rocky terrain. The terrain was placed at runtime using the existing Word Wizards tile building functionality. The generated grid leaves enough space around obstacles so that the characters will not path through them.

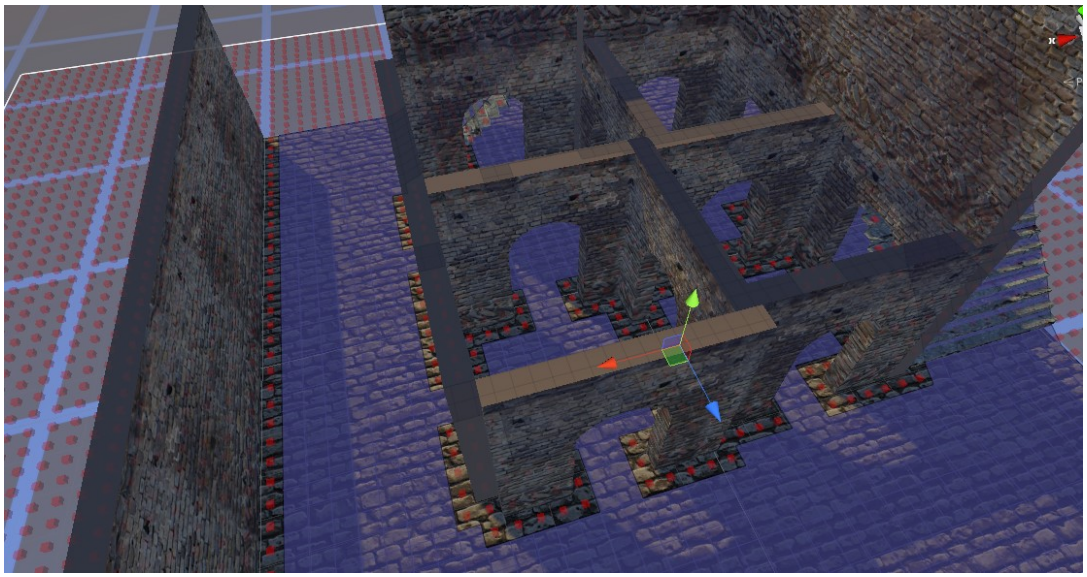


Figure 11: Graph creation with complex geometry

Dynamic Scanning

Once the functionality to properly scan the scene was established, the final step was to rescan upon changes to the terrain or movement of the player. The graph covers a large radius around the player that can be changed depending on hardware performance. The larger the graph, the more memory that will be needed. Additionally, larger graph sizes will take longer to recalculate. While the gird is only of a finite size, it can map a world of infinite size by rescanning whenever a player moves a certain distance from the center of the grid. The catch being that pathfinding is only possible in a small area around the player. This limitation is rarely relevant because NPCs do not need calculate paths when the player is not nearby. If the player is not interacting with or cannot see an NPC, they do not need to be updated. Disabling NPCs that are far from the player, decreases the total amount of computations performed by the player's machine. The LGG has a built in grid moving script, only recalculating the new area and moving the rest. This optimization allows moving the grid to be computed faster than an entire rescan.

Decision-making

The goal for my AI was to highly modular and editable by the user. There are many different approaches to solving this decision problem, each with their own strengths and weaknesses. In the next section I will go over the AI implementations that I considered and exactly why I made my final decision.

Behavior trees

Behavior trees, or BTs, are a hierarchical model of AI that formats conditions and actions within a tree structure. Starting at the root node, the tree is evaluated, branching at each

condition. One of the clear advantages of BTs are their conceptual simplicity and clear readability. Because of their tree structure, BTs are graphically very easy to represent. Sophisticated user interfaces, such as the behavior tree editor in Unreal Engine, can make BTs highly accessible to the user.

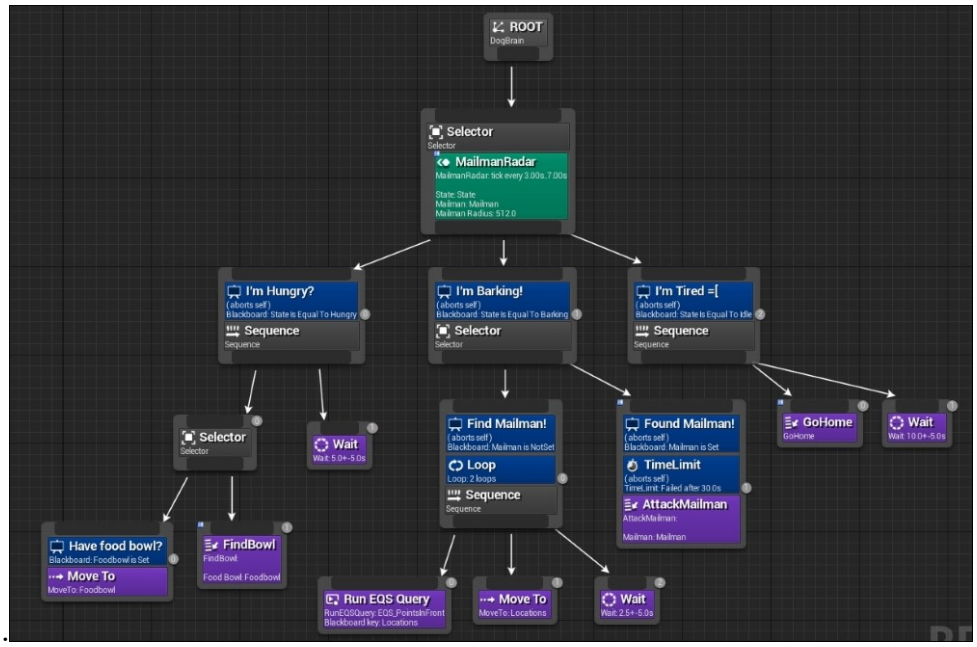


Figure 12: Behavior tree editor in Unreal Engine 4

BTs, however, have some notable disadvantages. Firstly, complex trees of significant depth, can become costly to evaluate. While this can be averted by implementing sub trees, regions of the tree that can be evaluated without needing to return to the root node, it is not always the most elegant solution. Even the most extensive of behavior trees all have the same fundamental problem: Abstracting the complex process of decision-making into 'true-or-false' Boolean logic is not always successful. Specifically, fuzzy decision-making concepts like 'prioritization' cannot be expressed [1].

Finite State Machines

Finite-state machines, or FSMs, are an intuitive, lightweight approach to creating an AI. The FSM model consists of multiple states connected by specific transitions that move from one state to another. As only one state may be active at a time, these transitions and their direction are just as important as the states themselves. While within a state, an agent is only concerned about performing its action and checking if the requirements to transition have been met. If no transitions are available, the agent will remain in its current state. Because of this single minded nature, FSMs tend to be highly predictable. If the designer is not careful, connected states can end up bouncing back and forth with each other, giving the agent unnatural behavior and breaking the immersion of the player.

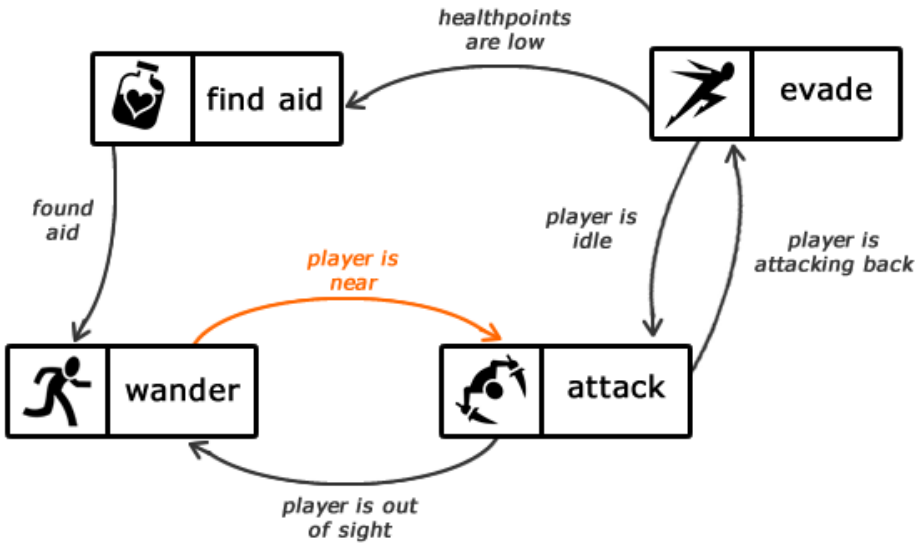


Figure 13: Basic state machine

FSMs are popular in game AI as they are excellent at abstracting intricate behavior into a very basic form. A good example of this abstraction is with human behavior. While human intelligence is currently impossible to implement into an agent, FSMs can be used to represent high level aspects of human behavior. States and transitions like eating when hungry, sleeping when tired, and working when out of money, are all well represented in FSMs. For NPCs with only a few actions, FSM do a much better job of producing the desired behavior.

One of the fundamental issues with the FSM architecture is its lack of scalability. Highly complex problems turn into a convoluted spider web of states and transitions. In a similar vein, FSMs also have problems with modularity. Adding or removing new states can fundamentally change the model as a whole, making live modification effectively impossible. Ultimately, FSMs are good for conceptualizing problems and small scale implementations.

Utility Based AI

While BTs and FSMs operate within a discrete framework, only existing in one state at a time, Utility AI offers a different structure. Utility score are calculated for all possible actions and the agent probabilistically chooses from the higher scored actions. This model is much more difficult to get right because it essentially based on fuzzy logic. In contrast to the traditional 'true-or-false' Boolean logic used by BTs and FSMs, fuzzy logic allows for a better representation of how decisions are made by humans. Additionally, emergent behavior frequently spawns from the adjustment of the scoring functions. Small tweaks can propagate throughout the system leading to unforeseen results. Utility AI, however, does not come without its limitations. The success or failure of the algorithm relies heavily on the quality of the functions [1]. Heavy tuning of these functions is required to achieve specific, planned results.

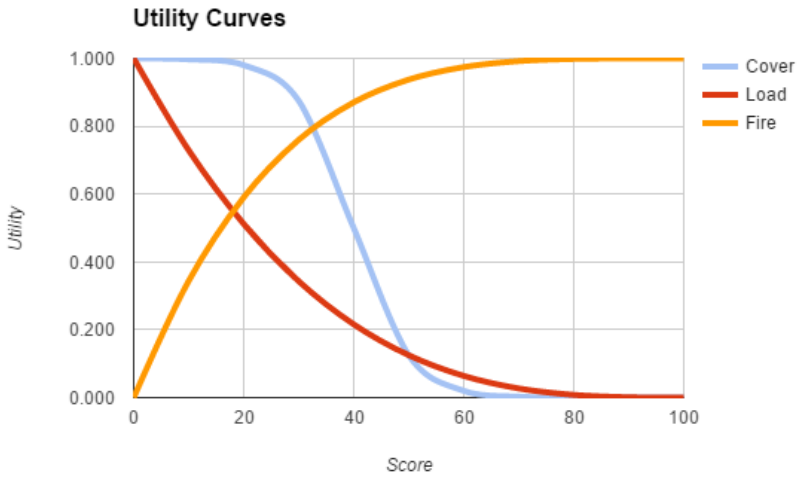


Figure 14: Utility curves representing the value of each action based on the score

The fuzziness of the scoring functions, and the infinite amount of shapes that they can take, affords Utility AI with an incredibly wide range of behavior. The advantage of this being, there are many possible models that can be constructed within the same framework, making customization and adaptation always possible. As previously mentions, however, this can become a nightmare for the developer to test and optimize, effectively limited the AI to the intelligence of the developer [1]. This problem can be solved with the instrumentation of machine learning. By treating the scoring functions as weights, Utility AI can also be easily trained in a process similar to that of back propagation in a neural network. The weighting or shape of each curve would be continuously adjusted autonomously based on the performance of the character. Establishing metrics to evaluate optimal AI behavior is a highly complex problem in its own right making this an extremely difficult task. Such implementations would be well out of the scope of this project but could very well be the future of Utility AI.

Curve Based AI

User

Because this system is meant to be used to create many different characters, I will be using the term “user” to describe the potential designer who is developing a new character. These users would be familiar with Unity and game AI.

Actions and Contexts

The entire Utility AI system is based on what I call actions and contexts. Actions refer to the predefined maneuvers that the character can perform. Currently this includes is attacking, fleeing, or regrouping, but can be expanded to include more specific abilities. Contexts describe the world information that the AI is considering when making its decisions. The current contexts are the health of the character, the amount of nearby enemies, and the amount of nearby allies. While the current actions and contexts are focused on combat, additional actions and contexts can be made to further increase the complexity of the AI. As these require specific world information and modify the low-level functionality, these would need to be added to the code by a developer.

Curve editor

The most important goal of the AI design was to create a system that offered maximum creative control to the user, while at the same time being simple in nature. To accomplish this, I devised a form of utility AI where the user is given a range of actions and contexts which they can map together as they please. The user selects two variables and is then prompted to create a

curve between them. When the AI is evaluating its surroundings, it scores the value of each action based on the created curves. As this system is meant to be modular and support the development of many different types of characters the user can choose to map relationships between as many or as few of these actions and contexts as they please. Additionally, the Unity curve editor allows for highly complex curves that give the user more than enough control over the mapping.

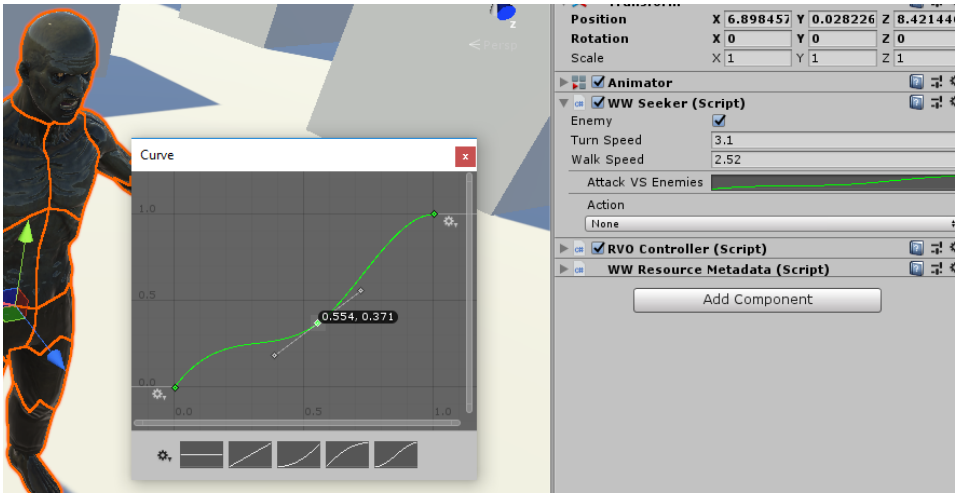


Figure 15: Curve creation between Attack VS Enemies

In this system, increasing the amount of available actions and contexts increases the number of combinations exponentially. While this is great for raising the potential of the AI, it can start to be a danger to performance. While there may be many potential curves that can be created, the user can make as many or as few as they like. It would be unreasonable to expect the user to create curves for every combination, as not all combinations would make sense for every character. Taking this into account the system was optimized so that the AI only evaluates information specific to the selected graphs. This architecture allows the system to scale well as the number of actions and contexts increases, maintaining runtime performance. **Figure 16** is a visualization of how the AI functions. The curves are evaluated and then values are averaged to

decide which action was scored the highest. The blank cells represent relationships that were not selected by the designer.

	Attack	Flee	Regroup
Health	0.3	-	0.2
Allies	0.1	0.9	-
Enemies	-	0.9	0.4
Average	**0.2**	**0.9**	**0.3**

Figure 16: Utility scoring visualization

The design of the AI allows for it to be as simple or complex as the user desires. Too much freedom can, however, be a bad thing. A negative aspect of the Utility model is that, because there are so few restrictions, it is quite easy for the user to shift the model out of balance. Increasing the weighting of one action can overpower other curve data and upset the system as a whole. Ultimately, curve editor is a system that has the potential to create many different possibilities, but the responsibility on the user to find the most compelling combinations.

Animation

In order to make the NPC model universal, a specific list of animations had to be established that would be shared by every implemented character. While finding this core group of animations is difficult, the fact that all the models are bipedal makes this achievable. Unity's

Animator Controller, shown in **Figure 17**, is used to establish a relationship between all animation states.

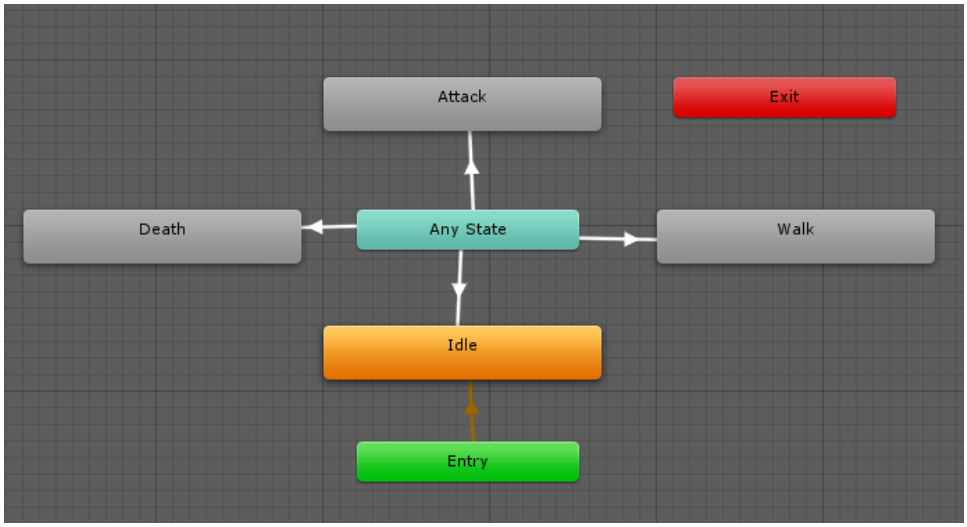


Figure 17: Animator Controller

While this controller is used for every character, each character also has its own Animation Override Controller. This component allows each characters individual animations to override the Animator Controller. For the purposes of testing the model, attack, death, idle, and walk animations were the chosen animations common among all characters.

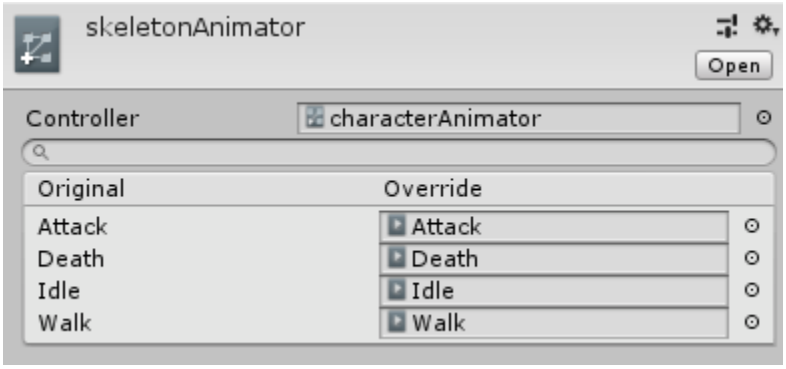


Figure 18: Animation Override Controller

Implemented Characters

In order to test my NPC model, I implemented three characters that would have different personalities and behaviors. These characters functioned as a proof of concept, indicating that many more characters could be made using the same process. A Skeleton, Zombie, and Archer were chosen because they fit the theme of the existing World Wizards content and they all represent slightly different behavior. The Zombie is unintelligent and aggressive. It does not care about how low its life is. Archer is a ranged unit that attacks from afar and groups with its nearby allies. The skeleton is bold and likes to fight alone, but flees when it is damaged. These personalities are all produced by the curve editor with no additional coding.

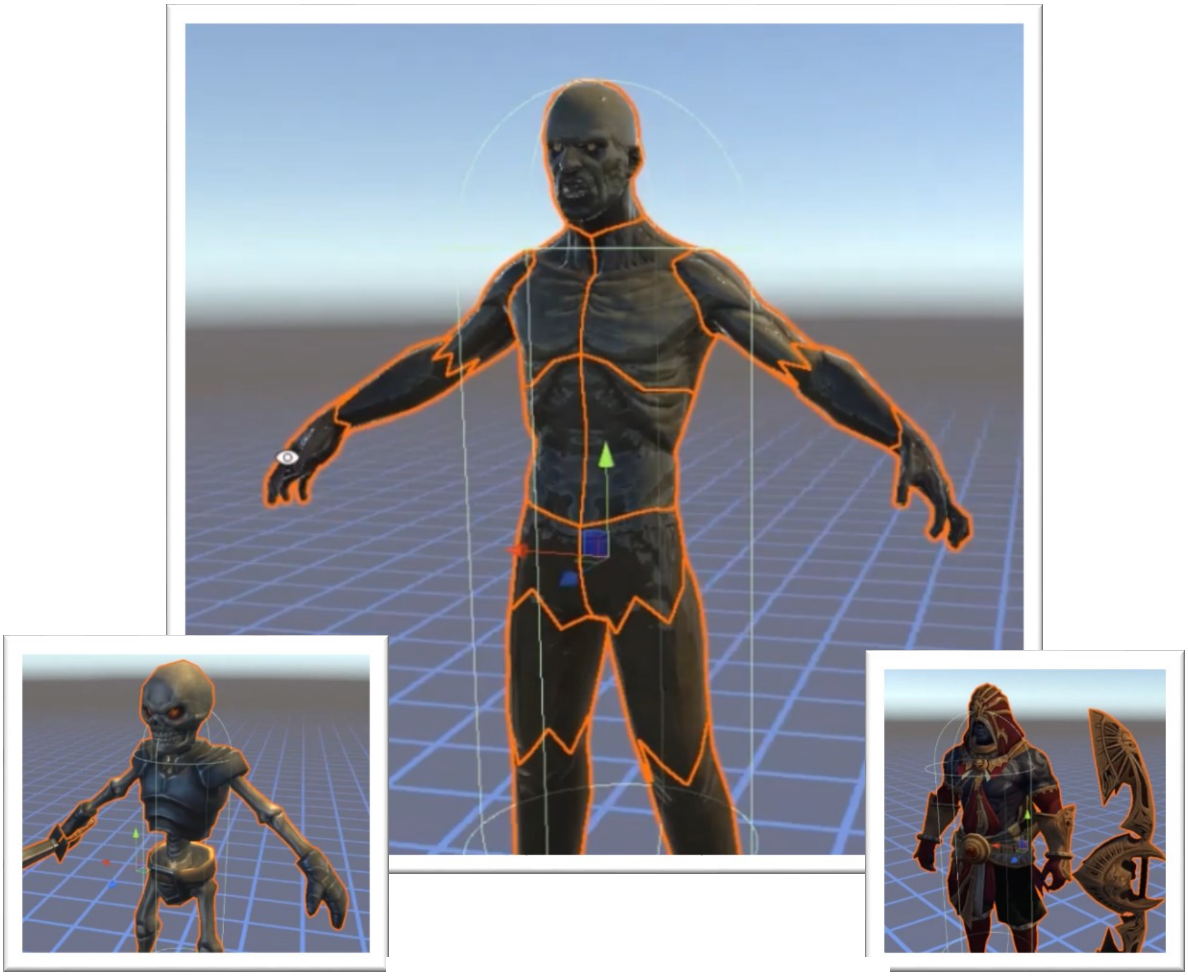


Figure 19: Zombie, Skeleton, and Archer characters

Figure 20 shows the user mapping a relationship between an action and a context. In this example the willingness to attack is being mapped against the current health of the character. As the health decreases, so does the willingness to attack. As the character becomes critically injured, the willingness to attack drops away completely.

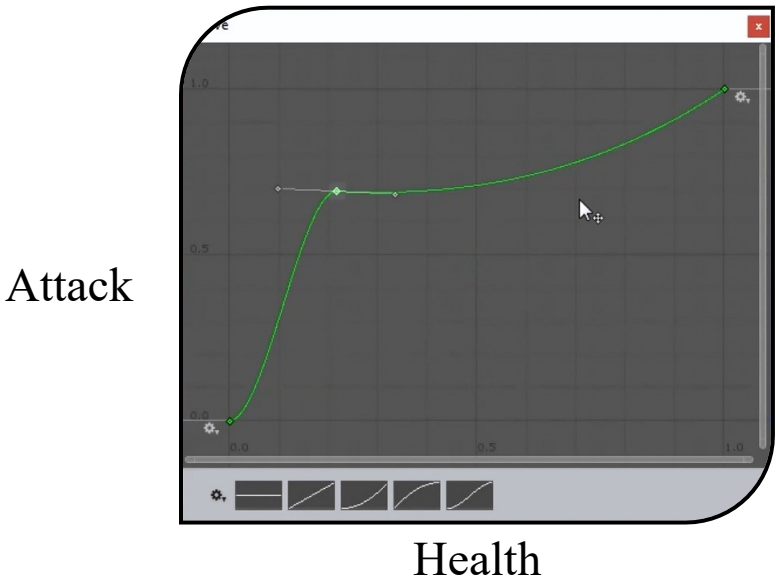
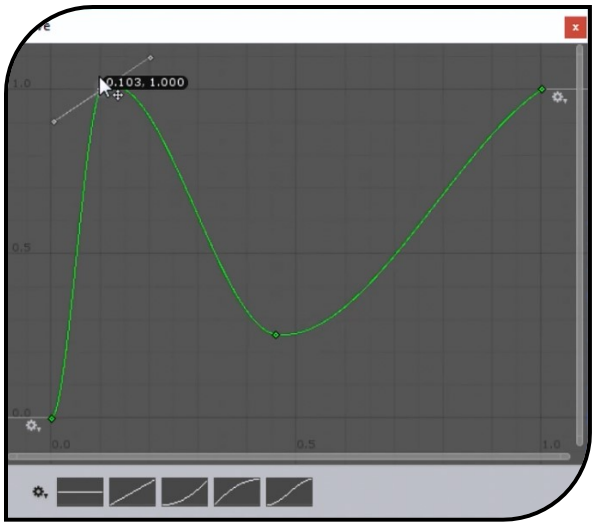


Figure 20: Willingness to attack V.S. character health

Another, more complex, example of this system can be seen in figure 21. The user is mapping the characters willingness to regroup against the amount of nearby allies. If there are many allies nearby, the character will want to stay grouped up. However, if the character is somewhat split up from the pack, they will stay split up. When only a few allies left, or the group is very split up, the willingness to regroup rises again. Finally, when there are no allies left, the willingness to regroup goes away completely. This example highlights just how much behavioral data is being abstracted in the shape of the curve. With fine adjustment, the user is able to tune these scoring functions and ultimately create complex character personalities.

Regroup



Allies

Figure 21: Willingness to regroup V.S. nearby allies

Results

With the grid generation created and multiple characters implemented, all the mechanisms could be combined and tested in the game world. From the very beginning of testing, different patterns could be clearly identified from each character. Modifying the curves in real time showed immediate changes in decision. What was especially fascinating was the interactions between characters that created patterns and reactions. This emergence was expected, the more actors that were added to the performance, the more involved and complex the scene became. Something that I had not previously considered quickly became obvious; The character's decision-making systems functioned very differently when they were alone compared to when they were in a group. The consequences of this being that the AI must be tested and tuned in the many settings that they would be involved in. Making a character behave in a specific, premeditated way was seeming to be an increasingly challenging task for a designer. This could be perceived as a weakness in the system, but, perhaps this emergence should be

embraced by the designer in the character development. **Figure 22** and **Figure 23** show some examples of characters interacting in a scene.

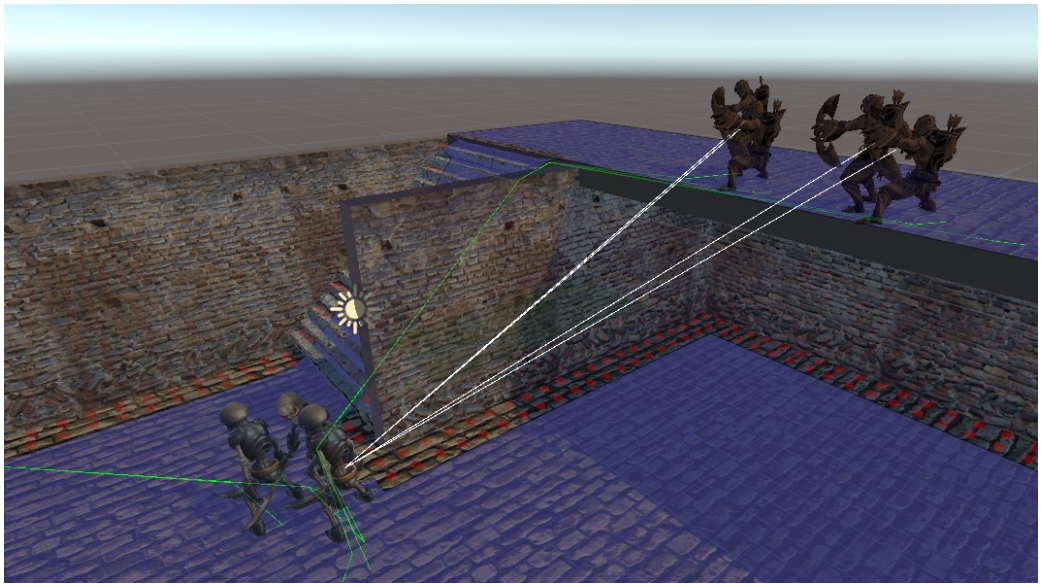


Figure 22: Archers firing at Skeletons

While it is difficult to represent the AI behavior in these pictures I will try to explain some of the things that are going on in **Figure 23**. In the top left skeletons are fleeing as their health is low or there are too many nearby enemies. In the center many zombies swarm and attack a single skeleton. On the right the archers fire on enemies but keep their distance.

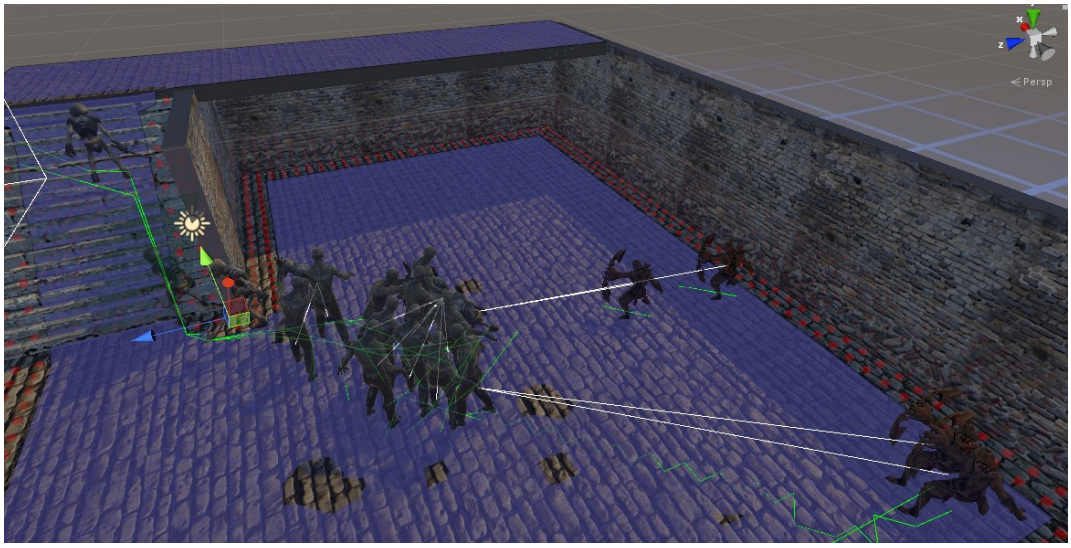


Figure 23: Archers, Skeletons, and Zombies battling

Conclusion

NPC Model

The goal of this project was to simplify the lengthy and convoluted process of creating game characters, ultimately supporting user generated characters in World Wizards. The system that I created is not as elegant as I had hoped but will allow for future users to implement their own characters into the World Wizards engine. While the system only supports bipedal characters, it could be expanded in the future to support many more types of characters through modification. Animation and locomotion would be the major aspects that would need to be changed to support quadrupedal or airborne characters.

Curve editor

The curve editor was the most ambitious aspect of this project. I was presented with a very niche problem that needed a somewhat unconventional solution. I was able to transform the idea that I had in my head into a functioning system. I am especially proud of this because I was working off my own design ideas rather than referencing existing solutions. Working in uncharted territory is a risky but rewarding venture. Through my implementation of the curve editor I was able to try and break new ground in the field of artificial intelligence.

In testing, I found that the curve editor was not quite as simple of a process as I had expected it to be. A large amount of tweaking was required to create behavior with complexity. Despite being aware of this weakness from the beginning of development, I was unable to overcome this problem. Ideally, the training process would be done by machine learning rather than human testing. While my creative approach to the problem may have been too optimistic, it was worthwhile due to the potential innovation that taking a risk affords.

Work Cited

1. Rasmussen, Jakob. “Are Behavior Trees a Thing of the Past?” *Gamasutra Article*, 27 Apr. 2016, http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php.

2. Koster, Raph. “Are Single-Player Games Doomed?” *Raph's Website*, 10 Feb. 2006, http://www.raphkoster.com/2006/02/10/are-single-player-games-doomed/.

3. Birch, Chad. “Understanding Pac-Man Ghost Behavior.” *GameInternals*, 2 Dec. 2010, http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior

4. Iwatani, Toru. “Toru Iwatani, 1986 PacMan Designer.” *Programmers At Work*, 29 Nov. 2010, https://programmersatwork.wordpress.com/toru-iwatani-1986-pacman-designer/.

5. Boeing, Geoff. (2016). Visual Analysis of Nonlinear Dynamical Systems: Chaos, Fractals, Self-Similarity and the Limits of Prediction. Systems. 4. 37. 10.3390/systems4040037.

6. Reynolds, Craig. “Boids - Background and Update .” *Boids (Flocks, Herds, and Schools: a Distributed Behavioral Model)*, 1995, http://www.red3d.com/cwr/boids/.

7. Baron, Sean. “Cognitive Flow: The Psychology of Great Game Design.” *Gamasutra*, 22 Mar. 2012, https://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php

8. Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*". Journal of the ACM. 32 (3): 505–536. doi:10.1145/3828.3830

9. Thompson, Tommy. “The Perfect Organism: The AI of Alien: Isolation.” *Gamasutra*, 31 Oct. 2017, http://www.gamasutra.com/blogs/TommyThompson/20171031/308027/The_Perfect_Organism_The_AI_of_Alien_Isolation.php

10. Electronic Arts. “Active Intelligence System - FIFA 17 Developer Interview.” *Electronic Arts Inc.*, Electronic Arts, 8 June 2018, https://www.ea.com/games/fifa/fifa-19/news/fifa-17-active-intelligence-system

11. Unity. “Products.” *Unity*, 24 Apr. 2019, unity3d.com/unity.

Appendix

Character Pipeline

Creating new characters is a lengthy but conceptually simple process. While there are many tasks to complete, all of them are trivial for a technically inclined user. The user begins with a fully rigged and animated 3D character. After the 3D model is imported into Unity, there are four components that need to be added to its prefab.

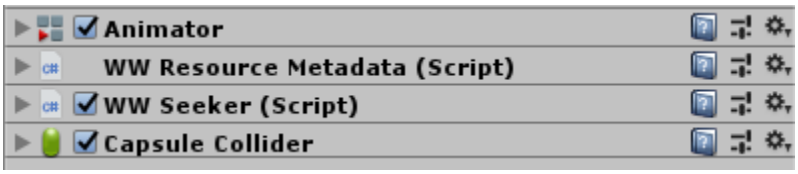


Figure 24: Four components

The first of components to add is an Animator. This controls the animations system for the character. An Animation Override Controller needs to be made that links characters animations to the global Animator Controller used by all characters.

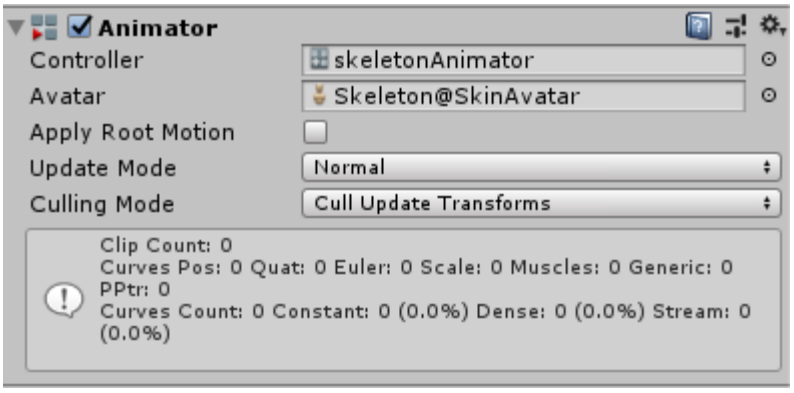


Figure 25: Animator Component

The second component is a WW Metadata script that allows the character to be loaded into the World Wizards engine. Characters are currently loaded in as props so that they can be placed anywhere and do not snap to the grid.

Figure 26:WW Resource Metadata

The third component is a WW Seeker script that handles the decision-making, planning, and locomotion. This script has an interface that allows the user to customize the character with sliders and the curve editor.

Figure 27: WW Seeker script

The final component is a capsule collider that enables collisions. The radius and height of this component needs to be adjusted to match the dimensions of the character.

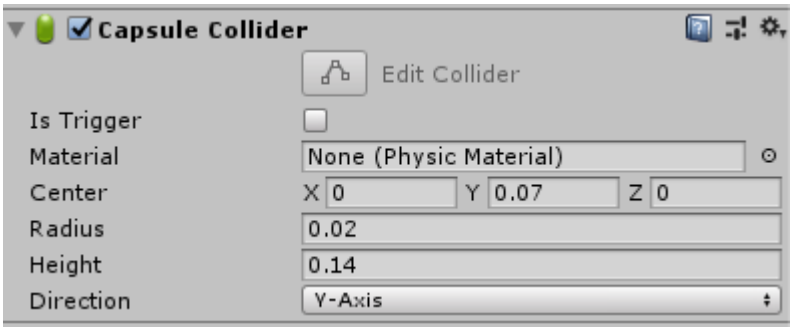


Figure 28: Capsule Collider

Once all the components have been added to the prefab and all of the variables have been set, the prefab must be serialized into an asset bundle. All other character components are added to the prefab at runtime so they do not need to be edited by the user.