Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Game Design and Development

Capstone Final Design & Development Approval Form

May 07, 2025

Student Name: **Samuel Burgoyne**

Research Title: **Choreography of Engaging Combat in Games**

Keywords: **Combat Design, Player Control, Enemy AI**

Sten McKinzie
Lead Capstone Advisor

Chris Egert
Research Advisor

David Schwartz, Ph.D.
Director, School of Interactive Games and Media

Choreography of Engaging Combat in Games

by Samuel Burgoyne

Paper submitted in partial fulfillment of the requirements for the degree of
Master of Science in Game Design and Development

Rochester Institute of Technology

School of Interactive Games & Media

B. Thomas Golisano College of Computing and Information Sciences

May 07, 2025

Choreography of Engaging Combat in Games

Abstract

Combat plays a critical role in many games and is used as a mechanic for challenge and progression, while supporting emotional and narrative engagement. This is essential for my MS GDD Capstone project, *Grim's Coffin* since diverse combat is a core design pillar whose goal is to support gameplay and story. This paper explores ways to choreograph combat in a way to make each encounter engaging, meaningful, and well crafted. Designing combat that is consistently fun and impactful can be a challenging task. By drawing from both technical and design perspectives, my research looks into how pacing, choreography, enemy behavior, responsive player controls, emotional stakes and the environment can contribute to making compelling combat design. Taking all of this information, it will allow me to build a framework for design combat that feels intentional, dynamic, and emotionally investing. Information gained from my research will directly inform the development of *Grim's Coffin's* control scheme and enemy behavior to create a cohesive and satisfying player experience.

Problem Statement

Combat is a central focus to lots of games, and diverse combat is a major design pillar for the game I'm developing, *Grim's Coffin*. One can develop some core technical systems, but making those systems fun and engaging has been a persistent challenge during the design phase. If combat is boring, players can simply walk away from the challenge to progress. Sometimes players are gated from moving on and are forced to fight, but at point it may not engage the player to fight, but only to progress. This can cause frustration and disconnection from the game's experience.

Combat in games can be complex. With player input, it can be hard to script out specific moments, but there are ways to choreograph the fight to make combat feel well-structured, intentional, and satisfying between the player and AI. This can be developed through orchestrating moves like combos or attack patterns, pacing encounters, and providing feedback. There are other attributes like making the player controller feel responsive, allow for a sense of mastery, and reflect the game's theme (Knowll, 2019).

This leads to my core question throughout my research, what design principles make combat engaging, and how does media across movies and games inspire the current understanding of these principles?

Significance

Combat is a core design pillar for the game my group and I are developing, *Grim's Coffin*. Combat is used to engage the player in the core gameplay loop. It helps push the player forward through the game as it provides challenge and progression. It also provides narrative context because *Grim* is trying to save spirits from the unknown forces of the world.

As a core component to engage the player, if the combat feels boring and bland, people will simply stop playing the game or not engage in combat. We want to control the player's emotions to make sure they stay engaged. With sequences that seem exciting, grow anticipation, and grow intensity, players will be more invested in fighting enemies (Everin, 2024). There is a story to be told with each fight, and if players are engaged they are willing to participate in the core gameplay loop of the game. It can be hard to provide these emotions for the player, but with proper pacing and choreography, these emotions can start to be built up.

Another useful part of combat is to add stakes to the game. Adding stakes can be done in a variety of ways, whether it is through combat, time, not reaching specific criteria, and so on. With combat as a core pillar in *Grim's Coffin*, we knew that intensity and interactions were going to be derived from it. The player shouldn't just be fighting enemies for no reason, they should be fighting with intention to progress through the story (Scissortail, 2018). Along with that, the game needs to evolve and have a variety of interesting choices. If combat is always repetitive, it can grow tiresome. Having the environment change, be useful or be dangerous, can be a successful way to change up combat (Scissortail, 2018). This means combat needs to be carefully considered heavily with the level design. Understanding combat is necessary to build out the level, and if the level does not have combat in mind, each component can fall apart.

With stakes, there needs to be a fine balance between being challenging, but not overwhelming. If there is too much emotion, the game can be too intense at all times. There should be a build up over time, eventually leading to a large conflict. This typically falls in with a level engagement curve, but just adding more and more enemies does not always reach high peaks. Not only does combat need to be intertwined with the level, it needs to be considered with the narrative design to help support higher level engagements (Scissortail, 2018). In *Grim's Coffin*, this is a constant question with balancing the story with mechanics. By making sure they are intertwined, it will help the game be more cohesive and allow for better solutions for both, like fighting off enemies to protect and save spirits or inspiring an intense boss battle.

Background

Historical Context

Combat has been a core of video games for a long time. It started off very simple, where games only had one button for inputs, along with basic and predictable enemy patterns in games like *Space Invaders*, *Asteroids,* and *Pac Man*. Eventually technology grew and some of the first fighting games were created. Some original games like *Heavyweight Champ* and *Karate Champ* introduced one on one fights, with more complex controls that allowed for a range of kicks and punches (Stuart, 2019). Later in 1991, one of the most important fighting games, *Street Fighter II* released, showing off a variety of unique moves stemming from distinct button combos to create a dance of attacks and counters (Stuart, 2019). These games started to show off how in-depth combat could become. From here, games like this inspired how combat could evolve to be engaging for the player, while promoting a sense of mastery that the players could learn and improve upon. Eventually enemy systems caught up with player mechanics where in games like *Mega Man X*, players had to attack and dodge attacks while memorizing attack patterns and rhythms. Player controllers have started to become a lot more refined and polished, with games like *Celeste* showing off how tight movement can be to create an engaging experience for the player (Sellers, 2018). With the combination of choreographed enemies and polished player controls, combat design evolved from just a challenge to artful choreography. Combat has been turned into an art form, Ubisoft's creative director for the game *For Honor* said he "believed that combat is an art form…hand-to-hand combat as a lethal, dance-like form of art" (Sellers, 2018). At this point, combat needs to have emphasis on its design. In most games, combat is no longer simple. They've been turned into performances with telegraphed cues, rewarding flows of motion, and peaks of engagement.

General Combat Pacing

Combat in games hasn't just evolved all on its own. Combat in games take reference from action films where fights are meticulously choreographed to build tension while making sure not to overwhelm the viewer. Jackie Chan is a pioneer when it comes to fight scenes in movies. Pacing is important, especially when fighting large groups of enemies. Even though large groups can be overwhelming, it isn't always the group attacking at once. Jackie Chan notes how when he fights multiple people, there is only one person at a time that attacks him (Teo, 2017). A part of this gives the illusion of intensity and allows for something to look intense, while providing proper pacing for one to fight off a swarm of people one person at a time. This illusion helps make a fight feel more dangerous than it actually is, by controlling who engages and when.

This technique helps inspire games to achieve a proper state of flow and to help pace the player. One useful way of doing so is to set an engagement limit. This forces a limit on how many enemies can attack the player simultaneously while making others stay at the edge of a fight, known as a spectator circle, waiting for their moment to attack (Teo, 2017). It allows designers to control the intensity of the attack and make the player feel that sense of danger, while making sure they aren't overwhelmed. This can be seen in a variety of modern games like *God of War* or *Batman: Arkham* where the player's attention is focused on one or two enemies at a time in a crowd fight (Teo, 2017). This turns the fight into a dance and forces a rhythm on the player, where there is a set series of attacks and a response. With proper design, there are clear beats for the player to engage with. By leveraging fight choreography in movies and implementing them in games, designers can create their own flow and direct a scene even though it's in a non-scripted environment.

Design Techniques

Honing in on a Good Player Controller

Players and enemy behavior goes hand in hand. Making sure the player controller is responsive and intuitive is foundational for having engaging combat. Movement should feel fluid and immediately responsive. When the player puts out a set of actions, the character should be able to easily move as the player envisions it. Knowll denotes how, "The first crucial part of getting Combat correctly has nothing to do with fighting. It begins with getting the movement right." (Knowll, 2019). To achieve this state, inputs need to be tuned, have responsive animations, and solid camera feedback to keep controls feeling tight (Everin, 2024). Designers spend a lot of time iterating and honing in on mechanics like input buffering or input/event canceling. An example of this is allowing the player to cancel an attack to dodge or change attack inputs to quickly react and make the controls feel smooth (Everin, 2024). Another technique is giving each ability a clear purpose to avoid redundancy. Every attack or skill should have a distinct role so that players know they have a variety of meaningful choices. In many games, attacks are balanced by risk and reward, where quick attacks are safe, but don't deal as much, where heavy attacks deal higher damage, but require longer wind-ups or recovery time (Lambottin, 2012). This gives the player options and makes it feel like they can build their own story of interactions. This also encourages mastery with the player to combine techniques in a satisfying way that allows them to progressively get better at handling tough situations. Along with this, attacks need to feel precise and be predictable. A game that does this well is *Hollow Knight*, where attacking timing, knockback forces, and recovery time feels fair and skill based (Kraj, 2020). When player input feels responsive and helps relate the player's thoughts to

onscreen results, the player controls will feel good and provide players an opportunity to focus on strategy and plan execution, instead of having to fight the control interface.

Honing in on Proper Enemy Behavior

Well designed enemies should have clear, yet varied challenges for the player. Enemies should also have behavior that is recognizable so that players can learn and strategize. A key design principle is to have enemies telegraph their attacks that tells the player when they are about to do something. For example, an enemy might lean back before a heavy attack or have some sort of visual cue. This allows for the player to react to defend themself and maintain a sense of fairness (Vossen, 2015). Vossen puts emphasis on enemy animations, making sure they are clear and distinctive, especially if they have a variety of attacks. Especially if attacks are stronger, the tells need to be more obvious. These tells may give away an attack, but it reinforces the player's sense of danger and keeps it feeling intense, while not feeling random and unavoidable.

Another aspect is providing variety within enemies. Enemies should have different archetypes that challenge and reinforce different player skills. For example, one enemy may carry a shield and force the player to either find the right time to attack or stagger an enemy to break their defense or one enemy might have a quick succession of attacks, forcing the player to dodge away quickly to avoid damage (Lambottin, 2012). Each enemy should have some sort of defined or precise challenge. This helps define how the player can attack or defend against them. While this can help build intensity for the player, it allows for a sense of mastery of the mechanics. With unique encounters, it also encourages players to use their full skillset. Instead of the player being able to attack the same way every single time, it forces the player to find a new way around the enemy. This helps tie together the player controls with the excitement of combat, making sure that the two systems feel cohesive.

Technical Implementation and Best Practices

Player

To get a solid feeling player controller, there needs to be emphasis on careful timing and clear animations. Some useful techniques include having an input buffer which stores button presses while animations or actions are still playing out. There is animation cancelling which allows the player to move from one attack to another in quick succession. These help make the controller feel tight and responsive. For example, the player could press a series of buttons to execute a combo, and have this information saved to follow out the moves after each animation is finished. But, if the player needs to dodge, they can quickly cancel the series of inputs for the combo so they can move out of the way. Animation canceling can also be a design choice not to

implement. If animation canceling is not used, it reinforces that players need to be more intentional with their attacks and make sure they don't overcommit, which is reinforced by attacks not being cancelable. Either way is a design choice, but should be considered for either case to make sure the controller is intentional.

The physics of the player is also important to consider. Player movement should be intentional and also clear compared to the environment. If the player is walking on the ground, the controller should be tuned to avoid unnecessary inertia or sliding. But, if they are in the air or running on ice, it should be the opposite where momentum is conserved and reflects the environment they are interacting with. This can be done with basic acceleration or deceleration formulas that note the environment and the speed that the player should have. Optimization is also very important, making sure that there is little to no input lag to make sure the visual feedback fits specific button presses. Optimization can be a larger issue, but the controller systems should be accessible and not demanding. The goal of this is to make the character feel like an extension of the player.

Enemies

Enemy AI can be implemented in a variety of ways and use a variety of algorithms. Some typical methods include state machines, or general if statements for behavior. More advanced techniques include behavior trees or utility-based decision making for more interesting or complex decision making. Pathfinding is also very important for enemies. With characters, we can rely on the player or human input to get them from point A to B. Humans can easily navigate and note important parts of an environment, but this needs to be translated to AI. Different ways to do this is to leverage the A* algorithm or built in navmesh and steering behaviors for navigation (Russell & Norvig, 2010). Location targets can be scripted or even leverage target points for enemies to move, whether they are locations on the map or the player (Gallant, 2019). These locations and points can leverage different data and weights to choose where they may want to move. With this, these technical implementations should be used to reinforce the design intentions referenced in previous sections. Their decisions should have distinct behaviors, tells, and triggers that make the fight meaningful and engaging, there can be a variety of successful backends, but on the front, it should still respect these behaviors.

Methodology

Hypothesis

By following proper design principles for combat set in movies and games, combat can be used to drive interaction and engage the player with the game. The goal is to take the information I have received from my background research and use it to drive the

decisions for designing combat in my capstone project, *Grim's Coffin*. Over time based off of my team and I's analysis, along with feedback received from playtests, will guide me to see which implementation is most effective within my project.

Technical Design

Player

The player was one of the first things I started to develop when working on this project. I wanted to make sure the player controller was accessible and easy to use/read. I also knew that the player controller should be data driven. This would allow for quick iteration and modification of any different speeds, timings, and so on. I also knew the player controller was going to be the center of the game, considering this is what the user interacts with for most of the game. I knew from here that there should be a state list that made understanding what the player was currently doing was easily accessible to anything that needed it.

I first started with the controller, but after some developing the controller off of my own knowledge, it didn't quite feel right. I was modifying a lot of velocities for the player and the movement didn't feel as responsive or as intuitive that I needed it to be. From here, I knew I needed to rewrite a lot of my controls and this is where a lot of my research started. After going through a lot of documentations and papers, I leveraged learnings from Knowll (2019) and Everin (2024) to better understand the current state of what I needed and eventually came up with this framework seen below.

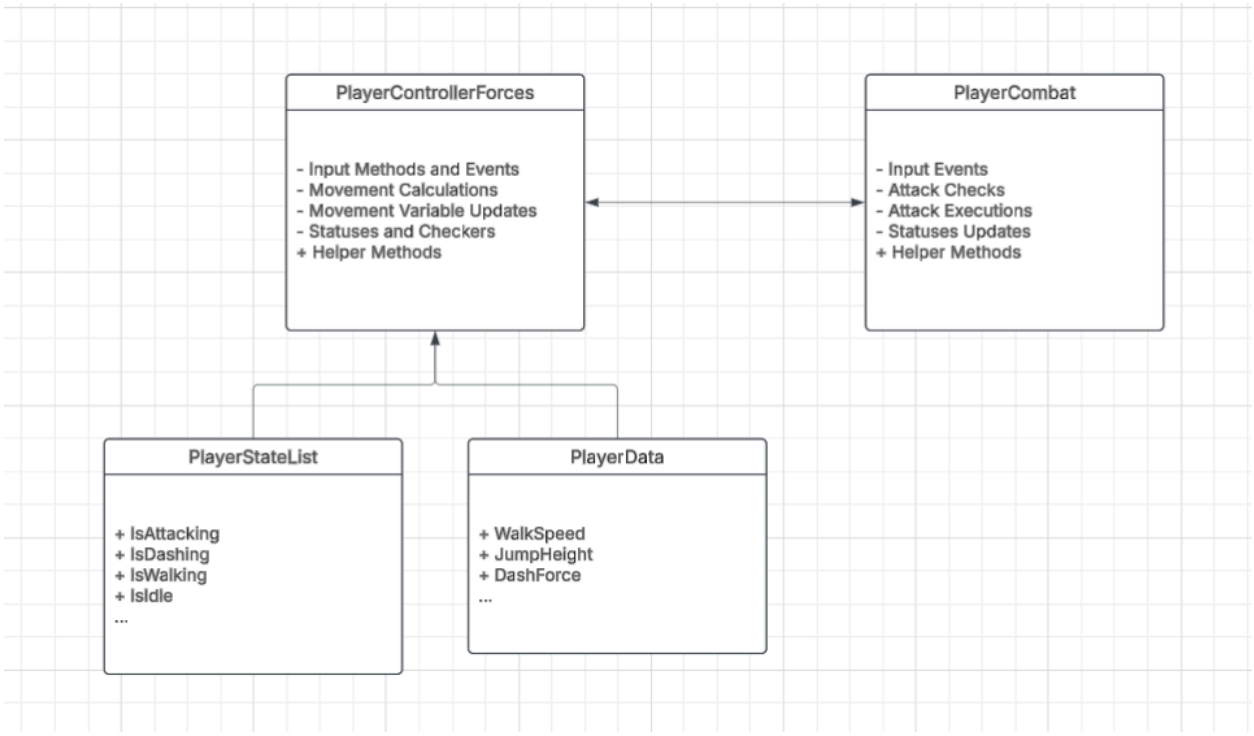

Figure 1: Class diagram showing the simplified structure of the player controls

At the top I had the script `PlayerControllerForces` that handled anything with moving around the player. Another large change I made from my first iteration is making sure it is force focused. This allowed for more control and allowed for modifications on the type of feel the player might have. There were plenty of timers and buffers that went into the implementation to help with the general controllers feel. From here the inputs would update the player's state which is tracked in the `PlayerStateList` script and any data determining how the player moved came from `PlayerData`.

The final list of controls that my group decided on were: walking, jumping, double jumping, wall jumping, and dashing.

For combat, this took a lot more consideration. Taking our core pillar and some of the research I had done, we decided on having a basic attack with a combo, along with having an up attack, and two down attacks which depends on if the player is on the ground or not.

The `PlayerCombat` script is a little more unique. The forces for each attack are still handled in the `PlayerControllerForces` script, but the `PlayerCombat` script had its own state machine which handled all of the states and inputs of the attack. Since the combat could be overlapped with the movement, I decided to separate the two since the handling is different between each system. There is only one attack button as well, so once the input is detected, I leveraged the left analog stick or direction the player is holding to determine which attack should be executed.

Enemies

After completing a basic structure for the player, I moved on to enemies. A lot of thoughts went through my head on how I could implement enemies. I knew I wanted to leverage polymorphism principles and create an abstract enemy class to stem all of the enemy variations from. From here I also wanted to have a parallel with the player, which includes the state list and how damage and collisions were being handled.

After the basics, I had to explore my options for the enemy behavior. I've had some general experience with AI before, but this was my first time working on something large scale. During my Undergraduate years at RIT, I had spent some time working and testing with different algorithms. After doing some more research to refresh myself, I found a paper written by Kießling (2022), who wrote an interesting paper about current trends in the market. They noted how Finite State Machines as well as Behavior Trees are still the best known methods and most popular in today's industry. They are fairly simple and flexible and allow for quick iteration, especially where enemies have plenty of overlapping behaviors.

I have built behavior trees in Unreal Engine and thought this would work well. The only downside is that I am working in Unity, and Unity does not have a pre-built structure for building behavior trees. I contemplated building my own visual structure for some time, but decided I was more focused on the actual behavior instead of the basic algorithm. I found a package built by Opsive (2025) and leveraged their basic algorithm structure utilizing the pre-built composite and decorator nodes, while being able to build out my own custom actions and conditions. Using all of this I was able to build out my basic structure shown in the diagram below.

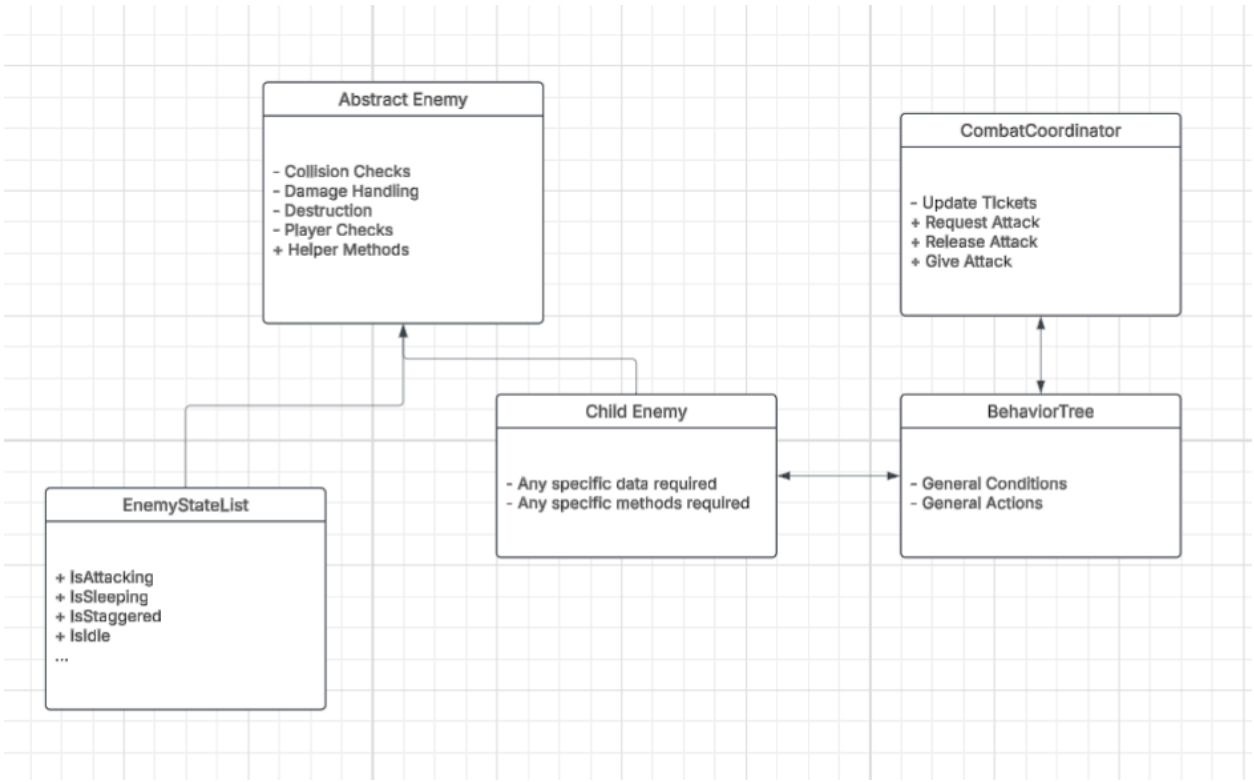

Figure 2: Class diagram showing the simplified structure of the enemies

I started off with the abstract `Enemy` class. This handled all of the basics like checking for collisions, damage, and destruction with itself. There were also a variety of methods that checked information from the player. These include methods that checked the direction of the player to the enemy or the exact distance away the player was. These methods are pretty general to all enemies and have important information that is utilized in the behavior tree. The `EnemyStateList` is pretty straightforward, it holds all of the states that the enemies were currently in, like if it is attacking or is staggered. This information is leveraged in the behavior tree as well and is important for determining what the enemy should do. Each enemy has its own child class that has any specific information that it needs to know, like if it is a boss, it has its own arena with information on the bounds of the room and so on.

Moving on, there is the behavior tree which is the main driver for all of the enemy behavior. It accesses information from the `EnemyStateList` and `Enemy` to help make its own decisions. From here, I can build out a variety of actions and conditions like moving, seeking a target, attacking, and so on. These actions and conditions are applicable to any enemy type as well. This part is important because most enemies have overlapping actions, like seeking the player. This allows for quick iteration between enemies instead of having to build something custom for each one.

From here there is also a `CombatCoordinator`. This coordinator collaborates with the different types of actions in the enemy behavior trees. It manages a list of tickets and as enemies are engaged in and out of combat. The combat coordinator handles how tickets are gained and given out. When an enemy is engaged with combat, it requests an attack and is added to a list of waiting enemies. From here there are two options, the enemy is eventually given an attack or some other condition happens where the enemy no longer needs an attack and releases its request for a ticket.

Implementation

After setting the structure of both the player and enemies, I set out developing and testing all of the different mechanics of my game. This took a long time to build and feedback was received throughout the majority of the time working on *Grim's Coffin*. This section will go into detail on all of the different types of data driving my designs and the end result of each piece of data.

Player

For the player, all of the information is stored in a scriptable object. The data is easily accessible and works well with other requirements for *Grim's Coffin*, like making sure the data is savable. Some of this data is pretty straightforward, like when a player is dashing, adding a specific dash force. Some of the data is not as straightforward and was found from sources like Knowll (2019) and Everin (2024) to add an extra level of polish to the player controller.

Gravity

Gravity is important and is pretty broad. This is controlled when the player is falling, whether they walk off a platform, get knocked up or when they jump.

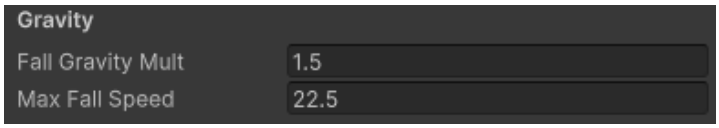

Figure 3: Gravity Data

Fall Gravity Mult is the default gravity multiplier for the player. There is a general gravity multiplier that gets changed throughout multiple other states. After states are

completed, like a player is done jumping, this is what the multiplier gets set back to. This allows for consistency and allows for the gravity to change to put more emphasis on different feelings. With the Max Fall Speed it makes sure the player doesn't continuously gain more and more speed as they fall. In other words this is the terminal velocity of the player. This helps keep consistency and makes sure the player is still easily able to control the player after falling a far distance. This allows players to consistently reach platforms when falling and gives them an extra buffer to make sure they aren't getting overwhelmed while falling.

Walk

Walk is the main player movement, it is how the player moves back and forth. There is no run, so the player only has one option for the speed of their character.

Figure 4: Walk Data

The movement data is also pretty straightforward, with the top walking speed and how fast the acceleration is. After testing, I kept the acceleration pretty high to help make sure the controller was responsive, and left the air acceleration timers at 1. This is pretty important as we wanted to include forms of aerial combat. Since the player is fighting in the air, keeping the acceleration high helped keep the player responsive, no matter where the character was. Conserving momentum was an option, but after testing, it was found that the character could build up too much momentum and my group and I didn't want to have the game be momentum based, especially since we wanted to have ability gated progression and that could be used to bypass certain areas.

All Jumps

For all of the jumps, there was specific data that helped control the player, especially with their speed.

Figure 5: All Jumps Data

Starting off, there is the Jump Cancel Gravity Multiplier. If the player holds the jump button, it will gain them more height. But if they only tap or let go of the jump they will cancel it. To make the controller feel more responsive, if they cancel the jump early, the gravity will increase to help revert the action the player just did. This gives more opportunities for the user to bail out of the action or quickly move around. There are some extra options for Jump Hang. If the players hold onto jump, they get less gravity. We didn't rely on this as much for the controller of the game because we didn't want the player character to feel too floaty. We wanted the character to still be responsive to the ground and only float in the air when they were attacking.

Jump

The jump is mainly for the single jump off of the ground, although the Jump Time to Apex is pretty general to the other types of jumping.

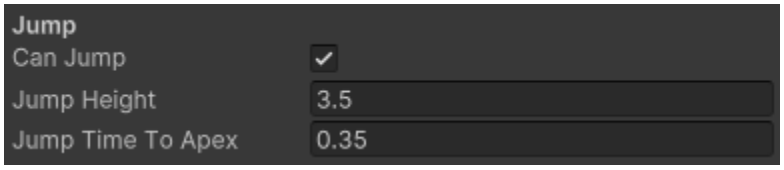

Figure 6: Jump Data

For a lot of the movement and attacks past this section, there is a check mark whether they can execute it or not. This isn't as important to the feel of the controller so they won't be brought up from here on out. The jump height is straightforward, it controls how high the player jumps. This aspect was important to get down the exact height since the height influenced aspects for platforming and combat. With this height, it allows the player to jump over most enemies, except for one of the bosses. This gives the player another option to dodge incoming attacks and is essential to keep the height fairly high. The Jump Time to Apex is very important as well. When pairing this with jump canceling, it gives the player more control over their movement. If they want to try and jump over the enemy, they can commit to the .35 seconds to gain the extra height. If they change their mind they can easily cancel and execute another action that they see fit.

Wall Jump

Wall jumps is a modification of the jumping function, but only for when the players are attached to a wall. Wall jumps can be challenging for the players to get right so we added plenty of extra polish and data to make sure this feature is accessible for most users.

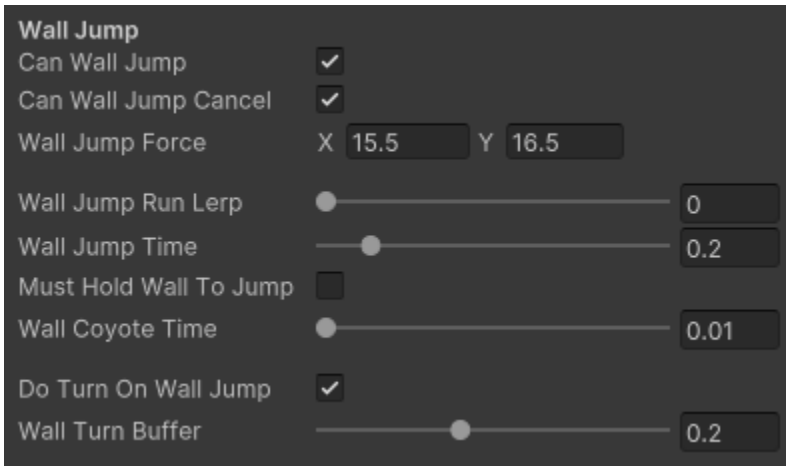

Figure 7: Wall Jump Data

For the wall jump, there are some similar features to the base jump. Wall Jump Cancel is similar to being able to hold the jump or not for extra distance. There is the jump force, but it is a Vector2 since it includes the X direction to jump away from the wall. The Lerp feature ended up not being leveraged, but was a multiplier for the walk after the player had jumped. We leveraged the Wall Jump Time instead and made the 0.2 seconds a time where the player could not move back towards the wall. This reinforced the fact that the player was jumping away from the wall and could not consistently attach themselves to the wall. Another important feature is Holding the Wall to Jump. This feature was important for accessibility. Although it made more sense to have to hold the wall to jump so it was separated from the double jump feature, it was a lot harder for the players to engage with. This feature was unintuitive and not accessible for the players and ended up having to be checked off. The next feature was the coyote time, but it ended up being not as important as the general coyote timer and allowed for more control of the player when moving off the wall. The last two data points were focused on turning the player around automatically, along with a buffer when they could turn back around. This feature was more visual and showed to the user that the player was trying to jump away from the wall. The goal of these data points was to help the player move to the next section after jumping, but still allowed for the players to move back if they decided they didn't want to commit to the wall jump.

Double Jump

There was a section for double jumps as well. This allowed for the player to jump an extra time if they were in the air.

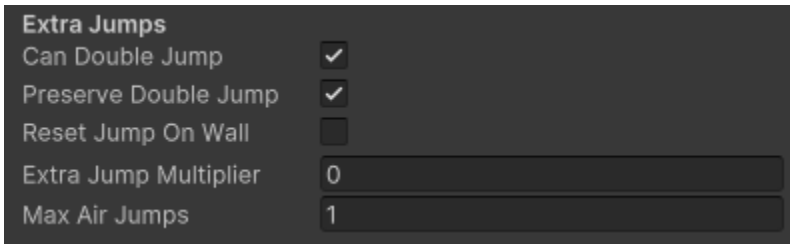

Figure 8: Double Jump Data

The double jump is fairly simple and relies on a lot of data from the main jump. The data driving this feature was reflecting if the player could actually double jump. The Preserve Double Jump boolean is an accessibility feature that allows the player to keep the double jump after they wall jump. This allows the player to plan out their inputs and feel like they don't lose an important toolset due to them wall jumping. There is the Reset Jump On Wall which we ended up turning off since we didn't want to allow players to infinitely jump up just using one wall. There is an option to have a multiplier for the height of the extra jumps, but we decided to keep this consistent and have any extra jumps stay consistent to the first one. The Max Air Jumps allows for triple or quadruple jumps, but that allowed for too much air control and we wanted to balance that feature out and make sure they only could double jump.

Assists

These assist features are an extra level of polish to make sure players get an extra buffer and don't have to rely on their input not working or timing inputs as they exactly need to.

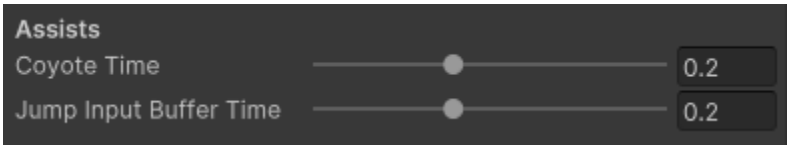

Figure 9: General Assists Data

These two data points are small, but an immense help in translating the player's thoughts onto the screen. With games, a lot of the checks can be pretty rigid and timing can be pretty specific. With Coyote time, if the player falls slightly off the edge, it gives them a slight buffer to use their first jump while in the falling state. This makes sure the player does not have to rely on specific timing and makes sure it removes any visual confusion for the player in case they thought they were still on the platform when they weren't. The jump buffer does something similar, but allows the player to jump

even when they haven't quite touched the ground yet. This is another visual issue that the player has to work around, where they preemptively press the jump button thinking they're on the ground when they are still technically in the fair, which makes the player feel like their jump just never worked. This extra time and feedback helps reconnect the player with the character and translate their thoughts into the controller. Being extra forgiving was useful in making sure the player connected with the controller and wasn't fighting it for getting the proper inputs.

Dash

Dash is a very important skill, especially with combat. A dash allows the player to move laterally in a quick succession. It also turns the player invisible and dodge attacks, which is very important for dodging and reacting to incoming enemy attacks.

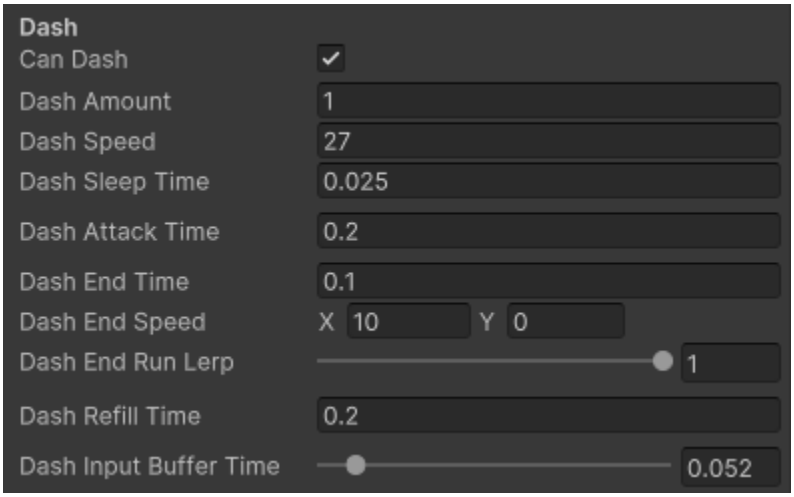

Figure 10: Dash Data

For the Dash, the first point allows for how many dashes they can save maximum and the second data point is the speed. The sleep time is utilized as a build up time, it is short, but helps put emphasis on the power of the dash. It is small, but helps with the general feel of the dash and make sure the player is aware of what is about to happen. The attack time is the duration for how long the player is in the dashing motion and the end time is for when the player is recovering from the dash. The End Speed and Lerp are the forces for after the dash is finished. The lerp is a multiplier for the player speed, but after testing we noted that we wanted to give full control back to the player and didn't want to slow them down during their recovery. The last two parameters have to deal with the recovery time to make sure the player can't endlessly spam the dash and have to be conscious about it when trying to dash away from enemies because we don't want it to be a resource to constantly dodge incoming attacks.

General Attacks

These features are simply checkboxes, but allow for interesting movement in the air and keep the player afloat.

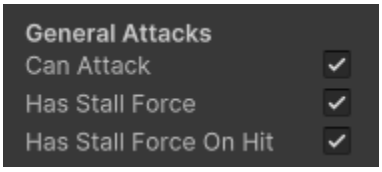

Figure 11: General Attack Data

These two checkboxes are fairly similar, but are very important for impact when attacking in the air. If the player is in the air, if they attack they will float in the air for a period of time. We took inspiration from the *Devil May Cry Series* (Capcom 2019), and this allowed for more dynamic combat, no matter where the player is positioned. This helped align our jumping movement with combat and give more complexity for combat, while making attacking more accessible in the air and giving players more time to react since they are suspended. The two options make it when the player hits the attack input, no matter if they hit an enemy or not, they will stall in the air.

Combo Attacks

The combo attack includes the default attack, along with any progressing attacks. By pressing the attack input once, it executes the main attack, but if the user presses it multiple times in one succession, it'll execute a combo.

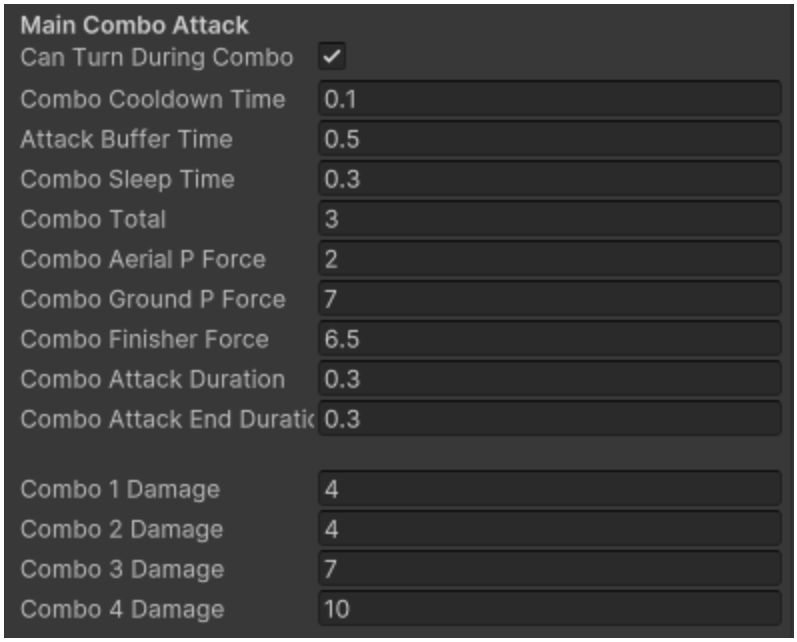

Figure 12: Combo Attack Data

There is a lot of data in this section, but it handles multiple attack iterations. The first few data points handle the timing between attacks and resetting the combo. The Attack Buffer time is especially important, because it gives the player enough time to progress through the combo even if they hit the input buttons slowly. This helps with the general accessibility of the controller and makes sure the user does not have to spam the attack button too hard. The next important section is the Combo Ground and Aerial Player Force. This section adds force to the player and gives them momentum for the attack. Although it may take away some control from the player it puts emphasis on the power of the attack and helps tie the controller with the visuals together. It gives a sense of impact for the attacks, and is combined with the Combo Attack Duration that sleeps the player for the specific amount of time detailed. The increasing damage of the combo gives incentive to progress through the combo to deal a larger amount of damage. This helps offset concerns with the sleeping of the movement, but it is risky to try and attack for that long. It helps increase the complexity of combat and forces the player to weigh their options if they should commit to the full combo or not.

Up Attack

The up attack is essential for two main reasons. It allows for attacking in a different direction which helps if there is an enemy above them. It also allows the user to stagger enemies and move them up into the air.

Figure 13: Up Attack Data

The up attack is interesting since it can stagger enemies. To help make sense of staggering enemies, sending enemies in an upward direction using the Ground Upward Enemy Force helps tie together the directional input and the animation. This attack didn't need to show any force towards the player, except for the sleep duration holding the player still since the movement mostly deals in the vertical direction. The Up Attack delay is the amount of recovery time before using another upwards to make sure the player can't spam this attack and helps balance it out from the others.

Ground Down Attack

The ground attack has fairly similar implications to the up attack. The attack allows for staggering enemies and adds extra knockback force on enemies.

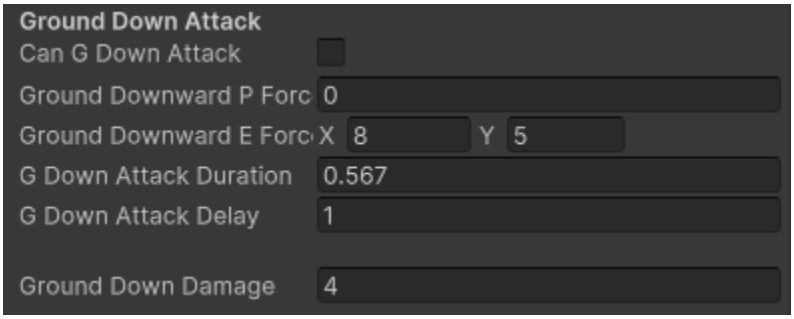

Figure 14: Ground Down Attack Data

The data from the ground data attack is very similar to the upwards attack. Since the attack is directional, it doesn't affect the player very much except for the cooldown before using this attack again and the sleep timer during the attack duration. The Ground Downward Enemy Force sends enemies a lot more horizontally and just a little upwards for its stagger effect. This is the opposite of the Upwards attack and helps provide more variation and ways for the player to deal with enemies.

Aerial Down Attack

This attack is for when the player is in the air and slams the player down to the ground while attacking. This attack is fairly powerful and has more weight to it compared to other attacks.

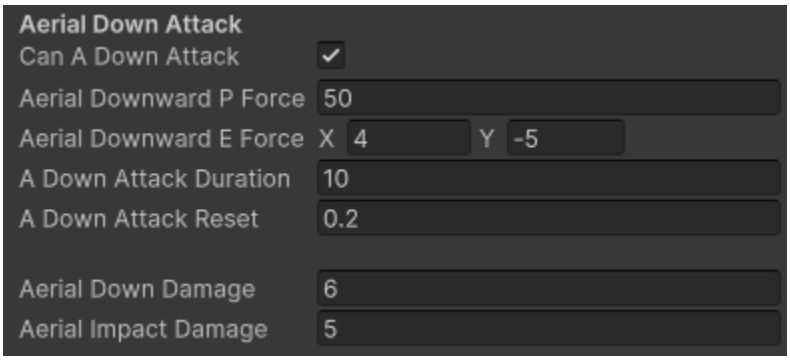

Figure 15: Aerial Down Attack Data

For the final attack, it is a more powerful force that sends the player down quickly to slam the ground. The downwards force sends the player to the ground and quickly changes the location of the player. This helps the player change height quickly and keep their movement a lot more dynamic, while also attacking enemies. There is some downward force, but it sends enemies down and slightly out. This helps make the enemies follow a similar direction while sending them slightly away from the player. There is also a small cooldown just like the rest of the attacks to give some more weight to this attack.

Extra Data

This covered the majority of the data that went into designing the player controller, but it wasn't everything. Two other major sources of data include animation canceling and hitstop. Animation canceling is another tool that allows the player controller to feel responsive to the user's decisions. If they decide to bail out of an attack due to an incoming attack, we allow the user to do so. In some games, especially soul likes, players have to commit to an attack or input. We want the players to have the choice as soon as they want and animation canceling allows for the player to be more reactive to any oncoming attacks and quickly adapt to any new oncoming action. With hitstop, it allows the player more breathing room, while making the attack feel more impactful. Hitstop literally stops the game, both the player and enemy stop from moving. This puts more weight onto the attack and helps visualize the damage that the player dealt or received. It also allows time for the player to process the current situation and plan their next move.

Controller as a Whole

With all of the data, my group and I have a solid controller for the player. This part is super important to get down, considering this is a game not a movie. The player has to feel like they are a part of the action, that this isn't some scripted scene they get to watch. With a controller, they get to be a part of the action and feel the intensity. Having a refined controller allows them to easily dodge and attack how they see it within their head. By keeping the controls responsive, players can imagine a movement or input of actions and their thoughts immediately happen on the screen. At this point, the players aren't fighting themselves and the game and they can focus on fighting the enemies as their movement slowly becomes second nature. The controller has found a good balance of being complex, but also straightforward. There are multiple approaches for the player to attack and defend enemies, while not confusing them in the same motion.

Enemies Behavior Nodes

For the enemies, the final implementation leverages the behavior trees. Over the course of the second semester I spent time building out a custom library of conditions and actions that could be utilized across any enemy. Since they could be used by any enemy, I was able to simply drag and drop these actions and modify them as needed for quick iteration. This allowed me to quickly go through different behavior variations and hone in on which ones that felt the best. In this section I will go through some of the actions and conditions driving my behaviors.

Conditions

Conditions for the behavior trees are information that would check for certain statuses and pass either true or false. These were very useful for determining which behaviors should execute and which ones should not. These conditions analyze both the status of the player and enemy and leverage the helper methods in the base abstract enemy class. Here is a base list of conditions that have been developed:

- IsDead
    - This was useful in checking for death and making sure any asynchronous tasks were ended and stopped from running in the behavior tree to help stop any weird bugs.
- IsStaggered
    - This is similar to IsDead, where it would check to see if the enemy was in the staggered state and stop it from executing any other behaviors.
- CanSeePlayer
    - This condition checks to see if the player is still in range and verifies if they should continue executing their behavior or stop since they can't see or reach the player.
- IsClose

    - This condition would check to see if the player was close to the enemy. If they were that close, it could force an enemy to attack to defend. themself.
- IsAlone
    - If the enemy is all alone, they could decide to do a certain action since they had no support from other enemies. This also means they don't have to wait for support from enemies to engage with the player.

Seeking

There are a couple iterations of the seeking action. The goal is to find the player and reach a specific target. To find the player, I leveraged the A* Pathfinding Project (Granberg, 2025) instead of having to implement my own A* algorithm. This helped accelerate the process and all me to focus more on the behavior side of trying to track the player

- Seek
    - This seeking behavior is fairly basic. The enemy would draw a path from the enemy to the player and follow the path. For most enemies, the seeking pattern was only lateral and enemies could not jump up ledges. This helps keep the enemies consistent and gate them from certain sections. This seek was kept simple and would be done when reaching a specific target range of the player.
- FlyingSeek
    - This FlyingSeek is very similar to the base Seek behavior, but it leverages both the X and Y axis. The behavior is kept to be simple and consistent to allow players to see the pattern of the movement easily and keep it predictable.
- SeekTargetFacingPlayer
    - This seek is a bit different from the other two. This seek leverages the Combat Coordinator script. As it waits for a ticket to attack the enemy it targets a specific location that is offset from the player. The enemy will walk backwards if the player tries to get too close. Making sure the enemy faces the same direction helps keep the pattern recognizable to show that it is waiting for its chance to attack.

Combat Coordination

The Combat Coordinator is a ticket and queue system that the enemies can leverage. Over time the coordinator generates tickets that the enemies can request to use. Once an enemy requests an attack, they will be added to a list who are ready to receive an attack. When the Combat Coordinator generates a ticket, it'll give the ticket to the enemy in the front of the queue and then generate a new ticket. Even if the coordinator has multiple tickets at one time, there is a small timer that delays giving tickets out to

make sure enemies don’t move at the same time. There are also methods that will release enemies from the list for when they don’t need the ticket any more since they are out of range or are dead.

Attacking

- BasicAttack
    - This attack is straightforward. Whenever the action is set off, it'll have the enemy execute the attack animation. There are no forces involved with this attack and is the most basic feature to keep it adaptable to every enemy.
- ForwardAttack
    - This attack action builds upon the previous attack. The attack plays the animation for the attack, while also adding a force to the enemy to move it in a certain direction.
- NonStaggerableAttack
    - For the other two attacks, if the player uses an Up attack or Down attack, they are able to stagger the enemy out of their own attack. With this attack, the enemy is unable to be staggered and will follow through with their attack. This adds a level of complexity and forces the player to use a strategy that doesn’t revolve around staggering an enemy.

Enemy Behavior

Basic Skeleton

The basic skeleton is the main enemy in the game *Grim’s Coffin*. This enemy went through a lot of variations to get where it is at the current state. In some of the original iterations, the enemy would either just walk back and forth without direction dealing damage on collision. Another iteration was when the skeleton would just seek the enemy and attack when it was close enough. There was a core issue with this behavior. Although this enemy was supposed to be basic, the player controller was too complex to engage with the player. This is where the patterns of the skeleton needed to be upgraded.

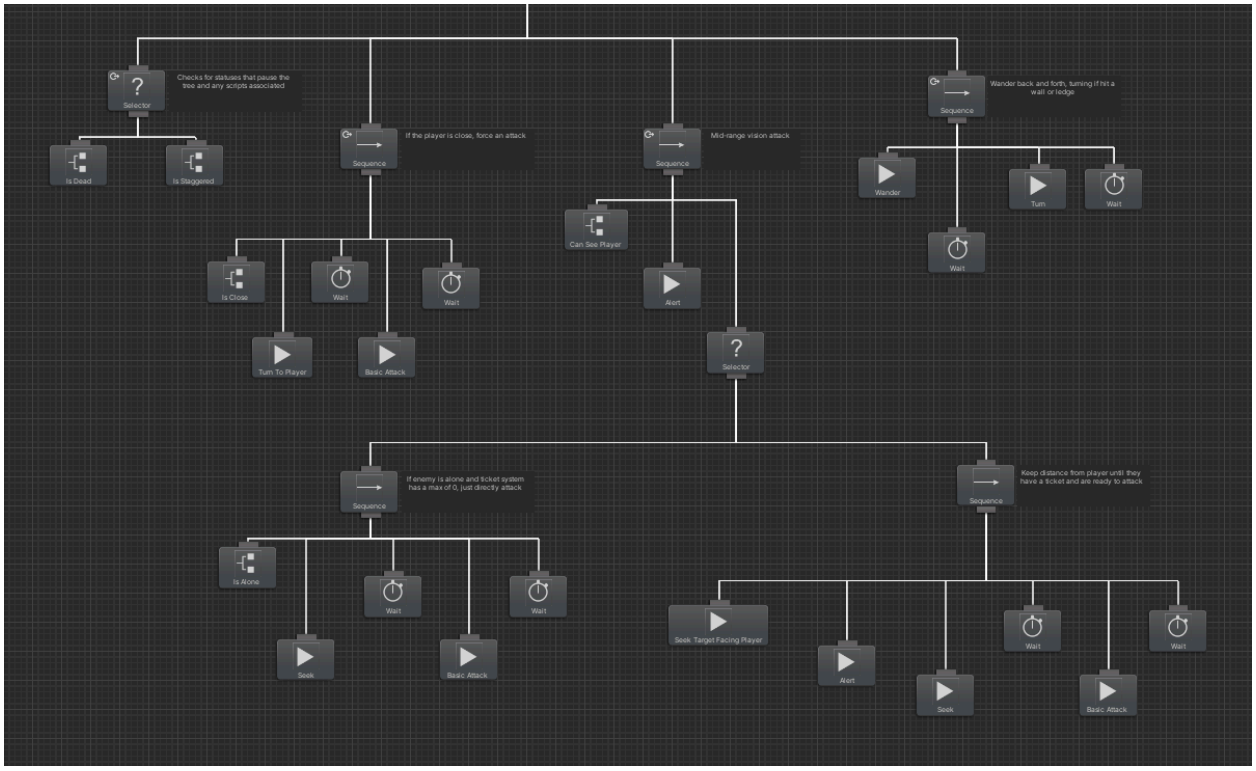

Figure 16: Final Basic Skeleton Behavior

The behavior needed to match the complexity of the player, as the choreography between the two was too simple. The player wasn't challenged in any way and didn't have to think hard to understand the pattern of the enemy. Even though this is the starting and basic enemy, it should still engage with the player. With this, it helped inspire the current state of the behavior tree

For the behavior, going from left to right (how the behavior tree executes), it starts off with some basic conditions that check to see if the enemy is dead or staggered which stops any of the following actions from executing. Next there is a conditional check to see if the player is close to the enemy. If the player is close, it forces an attack. In the later attacks, it checks to see if the player is in range. From there it checks to see if the enemy is alone and if it is, it seeks the player normally. If there are multiple enemies that are engaged in combat, it seeks the player from a distance and only when it has a ticket, it attacks the player. With the combat coordinator, it helps reflect the nature shown in movies as detailed earlier in the background section of this paper. The enemies anticipate action and make sure they keep a safe distance. They make sure to wait for a ticket so that the player doesn't feel overwhelmed all at once. This task was tough to accomplish since this was in a 2D space. By using some random variation in the target distance the enemies won't overlap if they happened to be in similar locations to make sure the player is aware of how many enemies they are engaging with. To make sure the enemies stay reactive, if the player gets too close it will force an

attack, even if it is still waiting for a ticket. This helps give a feel of reaction from the enemy and makes it feel like the enemy isn't stupid and making it too easy for the player. This reaction helps with the engagement and makes it feel like the player is heard and engaging with the moves that the player is making. From here, if the enemy has nothing else to do, it will wander around.

Overall, this behavior is straightforward, while being both planning ahead attacks and being reactive. Just by adding these few tweaks, the enemies feel a lot more alive and start to match the energy driven by the complex player controller.

Heavy Axe Skeleton

The heavy axe skeleton is similar to the basic skeleton, but has a much different approach to engaging with the player. This skeleton has a lot more health and deals a lot more damage. Just with this knowledge, it starts to derive the behavior and show off how it is more bold, but slower. It will need extra support for dodging player attacks and will be a lot more momentum based.

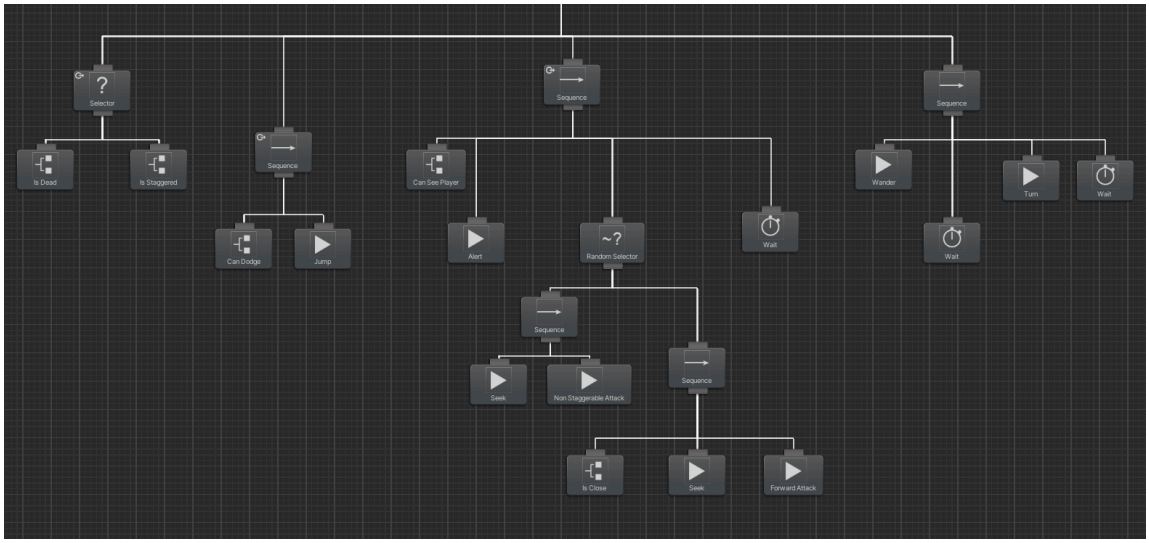

Figure 17: Final Heavy Axe Skeleton Behavior

With all of this in mind, I was able to end up with this enemy behavior tree. It starts off in a similar way as the basic skeleton, where it checks to see if it should be dead or staggered to make sure none of the following tasks execute. Following that, this skeleton has its own condition, CanDodge. This condition checks to see how many times it has been hit over a certain amount of time. If this condition passes, it'll run a random chance generator and jump backwards. This is another reactive action to the player. It builds on some of the choreography of the fight and shows that the enemy is engaging with the player. It knows it is hurt and should try to dodge backwards to try and avoid any further attacks. This helps build the spacing for this enemy that is used in the next section. If the player is in range, the enemy has a goal to attack the player,

but has two options. If the player is within a certain minimum range away from the player, it will lunge forward to try to catch the player off guard. This makes sure the player is paying attention and knows how to move out of the way quickly. The other option is doing a large swing attack that is not staggerable. This attack forces the player to make sure they don't overcommit for an attack and get caught in the range, reinforcing that the player should have patience when attacking. If none of these conditions are met, the enemy will wander around, just like the basic skeleton.

This enemy's behavior may not look as complex as the basic skeleton, but this skeleton is much more assured in its attacking style. It has plenty of opportunities to react to the player and engage in a way that may be slightly harder to read, but still predictable enough so that the player can react accordingly.

Results

Player

Over the course of the semester, the player controller ended up being a quite polished product. We received a lot of feedback at the end of the first semester noting how refined the player controller felt. This allowed us to progress towards working and finalizing different systems like fighting enemies. Once these rules were set, we could be a lot more intentional with the data driving the enemies.

It is interesting to note the progress over each Playtest build we had over the semester. After reviewing all of the Playtest builds compared to some of the final builds, each controller felt immensely different. Polish was constantly being added across the year, but even basic movement speeds and weights were very different. This changed the mood and pacing of the game quite a lot.

In early iterations, the movement felt fairly slow and floaty, along with some mechanics not working as intended. In the second iteration, movement was turned down a lot. The player was very slow and didn't move very far with some of the extra movements. It felt sluggish for the player and the movement felt like it was holding back a lot of the players. The third iteration was a lot more close to the final result, with movement being turned back up. The controls felt a lot more responsive and the pacing matched the style we were looking for.

For a long time the dash was an issue across all playtests. After the first semester and third iteration, we thought we hit a decent spot with the movement. For a long time though, the dash didn't quite feel right. For a long time it was because it was way too fast and hard to track the character. We had to slow the player down a lot and extend the time the player was in the dash. While the distance was the same, it was a lot easier for users to visualize and understand what was going on. Another major issue with the dash was animation canceling. Players would queue up the dodge input, but

have to wait for an attack to finish to execute it. A lot of the time, the player was trying to react to an oncoming attack and would input to dodge away in time, but they had to wait for the controller to finish its execution. This created a large disconnect for the player so we eventually had to add in the animation canceling. The animation canceling helped solve this issue and reconnect the player with the character.

Enemies

The enemies have reached a decent spot at the end of the semester, but there is still plenty of room for growth and engagement. Enemies are fairly complex, but easy to read and learn. This pattern allows for satisfaction when engaging with these enemies. There are few enemies in *Grim's Coffin* so far that leverage my research, but there are more enemies and iterations that could be added to the game. There isn't a large quantity of enemies in the game, so the combat doesn't allow for the player to test and practice all of their mechanics. This happened due to time constraints and didn't allow for a full comprehensive implementation of enemies. Although time may have been a large drawback, the current enemies test and engage with the player well and do what they were set out to do.

Something major I did learn about was fixing the damage of the player and health of the enemies. For a long time players would be quick to just dodge enemies to progress. After talking with the group and committee, the data we collected on people running away from enemies wasn't necessarily because the controls felt bad or the behavior wasn't interesting, but the time it took to engage and finish a fight. Although we wanted to have more complex and interesting enemies, they didn't need to have a ton of health and take a long time to kill. We wanted the player to eventually feel pretty strong and reactive and the enemies having too much health took away from this. It took too long to kill them and players didn't engage with the aspect of the game. Just by tweaking these small values we were able to drastically improve player engagement with combat. Sometimes something so simple can make a large difference. It can take a lot of time to come to this conclusion, and it was very easy to assume that I needed to tweak behavior even more. It comes down to a fine balance and making sure one understands the root issue of any problems.

Conclusions

Over the course of the year, I think I have ended at a good result for combat within *Grim's Coffin*. I had basic knowledge of player controllers at the beginning of the year, and intermediate knowledge with enemy behavior, but since this project I have deepened my knowledge on both aspects. This process went decently well, but in some moments it was slow and took a lot of time writing and rewriting my implementation. Along with that, it was an iterative process. A lot of the features were slowly added, slowly adding polish to both the player character and enemies. Although it was good to add and test features as they were implemented, weights had to be

constantly changed due to new features. It took a long time to feel like a complete package, so for a long time data went back and forth, especially during the first few months of development. There was a learning curve with some of my implementation, so if I were to go about this again, I would be a lot more confident and time efficient. But as a learning process for anyone who is iterating on it slowly, is to be flexible with the data and controller. And having a complete controller set is important, especially with those fine details because they make the controller feel so much more solid and refined which drives engagement with the player.

Another issue I ran into was that for a while, the data I used was tested only by me. It took a while to get my team involved to come to different agreements. If one were to go through this process, they should make sure to get constant feedback. With the feedback, having two or more control options to compare with is important so that playtesters have a frame of reference. Not only should the team be testing it, but making sure other new players get to as well. A pair of fresh eyes is always important, especially for accessibility. The controller should be fairly easy to pick up, but with the complexity making it hard to master. If the basics are at a good spot and playtesters pick it up quickly, that is when you start to know the controller is intuitive and reacts well to the player's thoughts.

Another aspect that would have made this process smoother is adding better tools for testing and debugging. Although the features I wanted got to where I needed, it was hard to visualize a lot of things. Sometimes I couldn't tell which enemy had a ticket and was actively attacking or even visualizing which behavior they took, so sometimes there was confusion on the implementation side. I was able to figure out a lot of problems utilizing other debugging skills, but the process was still a lot slower than it needed to be. When one spends less time debugging, they can put more time and emphasis on the design of features, making sure different weights and data gets to where it needs to be.

One of the most useful things I learned is to give yourself options. It is ok to test and add features in and out. As long as these features are data driven and accessible, it will allow for quick iteration and make sure you don't have to commit to any idea. Having this flexibility made it easy to respond to feedback received. And, if any of the changes ended up needing to be reverted, nothing was deprecated and could easily be added back into the game.

Overall, everything I have researched and learned over the past year has been super helpful. Making a controller accessible and intuitive is an important process. Player controllers are the one of the most forward facing features and are typically one of the most used features by a user. If the player doesn't have to fight the controller, they can spend more time working and figuring things out in the game. Enemy behavior was a more challenging problem for me, but ended up in a solid spot. One of the most challenging things for enemies is pathfinding and without being able to move, the

behavior of enemies is pointless. The extra tools and packages helped me focus on the behavior and they have gotten to a good point where enemies are engaging and exciting to fight against. The research I've done has been successful in improving combat with *Grim's Coffin* and there is plenty of knowledge I will be taking into future projects.

Future Work

At this point, the player controller is at a very decent spot. There is a variety of movement and attacks that the player gets to engage with that gives a layer of complexity that allows the user to approach combat however they see fit. Yet, if my group and I were to expand upon *Grim's Coffin*, we would definitely add a larger variety of movement sets and abilities. With a metroidvania, ability gated progression is an important part of our design. My group and I would have to collaborate to make sure any future abilities or movement sets fit the design I have set forward. They would have to make sure controls are tight and responsive and not hinder any other current movement.

For enemies, there is plenty of future work that could be done. In the current state, I do have access to other enemy variations with complete animation states. There are some extra animations that I would need like a staggered state for most enemies, but with an almost complete package there is a lot of work that I could do on the technical side. I could test and implement a variety of new behaviors and target more mechanics of the player to test and challenge them with.

Due to time constraints with my capstone project, *Grim's Coffin*, I wasn't able to get to the features I just mentioned. But, with the research I have completed, I feel comfortable to tackle these challenges in the future and execute them well.

References

Aron Granberg (2021-2025) A* Pathfinding Project. Unity AssetStore.

Capcom. (2001–2019). Devil May Cry Series. [PC/Console game]. Capcom.

Everin, I. (2024, September 28). *Combat Design, Mechanics and Systems for Satisfying Game Combat*. Game Design Skills. https://gamedesignskills.com/game-design/combat-design/

Gallant, M. (2019, April). *Combat Design & AI in Uncharted 4*. Gangles. https://gangles.ca/2019/04/15/combat-design-uncharted-4/

Kießling, T. (2022). *Implementation strategies for battle AI and its usage to increase replayability in action RPG games that focus on PVE combat* (thesis).

Knowll, H. (2019). *Game Design: Combat as Dance*. https://www.knowll.com/e/142/game-design-combat-as-dance

Kraj, N. (2020, April 25). *Keys to Combat Design: Anatomy of an Attack*. GD Keys. https://gdkeys.com/keys-to-combat-design-1-anatomy-of-an-attack/

Lambottin, S. (2012, August 15). *The Fundamental Pillars of a Combat System*. Game Developer. https://www.gamedeveloper.com/design/the-fundamental-pillars-of-a-combat-system

Opsive (2019-2025) Behavior Designer - Behavior Trees for Everyone. Unity AssetStore.

Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A modern approach*. Prentice-Hall.

Scissortail, E. (2018, August 10). *Making Combat Interesting*. The Gauntlet. https://www.gauntlet-rpg.com/blog/making-combat-interesting

Sellers, M. (2018). *Advanced Game Design: A Systems Approach*. Addison-Wesley.

Staurt, K. (2019, June 1). *Kapow! The History of Fighting Games*. The Guardian. https://www.theguardian.com/games/2019/jun/01/kapow-the-history-of-fighting-games

Teo, R. (2017, April). *Combat Design : Basic Choreography*. RayTeo Active. https://rayteoactive.weebly.com/combat-design/combat-design-crowd-choreography-02

Teo, R. (2017, April). *Combat Design : Learning from HK Movie Choreography*. RayTeo Active. https://rayteoactive.weebly.com/combat-design/combat-design-crowd-combat-choreography-01

Vossen, B. (2015, May 4). *Enemy Design and Enemy AI for Melee Combat Systems*. Game Developer. https://www.gamedeveloper.com/design/enemy-design-and-enemy-ai-for-melee-combat-systems