Capstone Project: Stealth/Action Game

Morgan McLeod

Hendrix College

A thesis submitted for the degree of

Bachelor of Computer Science

Hendrix College 2022

Acknowledgements

I would like to acknowledge my friends who helped to make progress in the development of this game preceding my work on it during the semester. Jackson Pierce helped to establish the starting code, including the enemies' early system of detection. Haydon Tarvin provided all of the art pertaining to the player: if the player can do it, the art was provided by him. Both were invaluable in establishing the core idea of the game in the planning stages.

Abstract

For my capstone project, I attempted to take work I had done on some simple game mechanics with some friends, and turn it into a full 2D stealth-action platformer game. I wanted the game to be playable and have an acceptable level of polish. These goals were left intentionally vague, as it was difficult to predict how much I could work on the project, and how well I could achieve goals more set in stone. The basic game mechanics had already been created by myself and some friends over the summer, so my job was to get the game to a playable state based on the type of game I was making. Additionally, when creating the level of the game, I wanted to incorporate design principles I find interesting about games of this genre. In the end, I was unsuccessful in creating a full playable game like I wanted. However, what I have created is a prototype that can be transformed into a playable game with more work. Additionally, the pieces that are in my prototype are solidly polished like I had originally hoped.

Contents

List of Figures

Chapter 1

Introduction

Creating a video game from scratch requires work on a plethora of different pieces that come together to make a pleasing product for a user. Both creating and assembling these pieces is only made easier through effective planning and previous experience working in the field. Since I harbor a strong interest in the field, the need for this project arose from my desire to increase my own experience working on video games.

There is a wide ocean of different types of video games that I could have made. However, I needed to find a type of game that would balance complexity and interest with feasibility of implementation. I wanted to choose a type of game with complex enough mechanics that I would be able to exercise my programming skills when making it, but one that would not take a year to build. I settled on a 2D stealth platformer. This works out well since I personally enjoy 2D platformers. Additionally, 2D games are a strong suit of the Unity Engine [4], which I coded my game in.

The idea of the game is that the player controls a character who is tasked with making his way through a level of lethal enemies by utilising their skills in stealth and combat. I have drawn inspiration for this idea from games like *Dishonored* [3] and *Gunpoint* [2], which have similar goals. A concept that those games maintain is that you can approach playing the game in multiple ways that all remain viable. You have a choice both of how stealthy you want to be as well as how lethal you want to be. This is a concept I tried to embrace when designing the environment of my game. I tried to include different paths based on trying to be sneaky and evasive vs direct and aggressive. It is a concept that greatly increases the player's immersion since it grants the player improved agency in the way they achieve the goal of the game. They feel more connected to the character because their decisions as to how they would approach a situation are able to be more accurately expressed by the character.

However, creating a detection system is neither simple nor easy. There are multiple complex pieces that make up the system that need to be tuned correctly to achieve a balance of realism and fun. One might think that striving for perfect realism in designing such a system would be the ideal approach: since it mirrors real life it should be that much more immersive. Though real life simulation has its place in some games, realistic stealth and detection is needlessly punishing to the player in this case. No matter if it is work or play, it feels bad if you are harshly punished for a single mistake, especially if you are inexperienced. Realistically, if someone looked towards the player from far away, they would instantly spot them. However, this makes stealth neither particularly fun nor viable for anyone new to the game since they may not know the movement patterns of the enemies nor would they be perfectly acquainted with all of the games movement and stealth mechanics.

My ideal implementation of detection would mimic the systems that exist in games like *Dishonored*, where the player is detected gradually. The farther the player is from an enemy, the slower said enemy can detect the player [3]. The implementation I achieved for my game was simpler, but I believe still got the job done. If an enemy can see the player, and is within a certain distance, the player is detected, otherwise they are not. This approach tries to balance realism with fun, by not instantly punishing the player for making a tiny mistake while an enemy is far away. However, the enemies are not blind, and will detect the player if they get too close.

I have similarly kept the combat mechanics relatively simple, allowing the player the ability to attack in front of them with a sword. There is not any more to that though that I feel is appropriate for this type of game. In games centered solely around combat, other weapons or types of attacks could be added, adding a massive skill ceiling to the player's combat abilities. For this though, I want something that is not complicated and will thus not dissuade a new player. Additionally balancing combat is much easier when the system for it is simple.

Though I have kept the basic attacking part of the combat system simple, I feel as if the game achieved a solid amount of interesting complexity with the inclusion of the different special abilities that the player has. For example, one of the abilities includes a damaging ranged attack which costs resources to use. These abilities provide the player with more options to use in combat without outright requiring the mastery of said options.

Chapter 2

Background

2.1 Game Genre and Basic Definitions

This game is a 2D side-scrolling stealth-action platforming game. There are quite a few descriptors there, so I will define them here along with other general definitions:

- **2D:** This is an abbreviation for 2 dimensional. This refers to the spacial dimensions that the game inhabits. Everything within the game is contained within a 2 dimensional flat world.

- **Side-scrolling:** This refers to how the player is viewed with the game's camera. In a side-scrolling game, the player is viewed from the side, and they can move left or right to traverse the game world. The camera will follow the player, making the world appear to scroll around the player as they move: giving this genre its name.

- **Stealth-action:** This is a specific sub-genre of the larger genre of action games. In action games, the player must navigate through a level filled with obstacles, whether it be physical obstacles or enemies, to achieve some set goal. For stealth-action games, enemies do not outright detect the player from the start. The enemies have a detection system that allows the player to sneak around them or deal with them silently if the player is skilled and strategic enough. If the player is detected, they play the game normally as if it were just an action game.

- **Platforming:** This refers to how the player moves through the levels of the game. In platforming games, the player is able to run, jump, fly, or otherwise traverse through a level characterized by uneven environments and hanging platforms. The popularity of such hanging platforms gives this genre its name.

- **Level:** This is the physical environment in which the player moves around while trying to accomplish the objective of the game.

- **Game Engine:** These are software frameworks whose design is based around allowing a user to develop of a video game.

- **Gameplay:** The act of the player playing the game.

- **Tile map:** A tile map is a grid of squares where each square contains a piece of some environmental art. A game developer can then create a level for the player out of these squares. The name "tile map" comes from the fact that it is a large map of individual art tiles that can be slotted together in any configuration. Therein lies a subtlety that makes creating a tile map difficult: Each tile should maintain stylistic continuity between the others, and, most importantly, itself so that there is not visual dissonance for the player.

2.2 Game Design Principles and Inspiration

In the initial stages of planning for this game, I wanted to design a game that was feasible in the time allotted while also being an interesting concept based on ideas that I find fun. This led to the idea of the game being heavily inspired by games like *Dishonored* [3] and *Gunpoint*[2]: other stealth-action games that lend many of their gameplay themes and design principles to my game. Both *Dishonored* and *Gunpoint* have the main goal for the player to traverse through an area filled with enemies to achieve some ultimate objective. The fun in these games comes from the moment-to-moment decision making that the player must make to best traverse the area and achieve said goal while avoiding or killing the enemies standing in their way.

These games involve a detection system: a system for handling whether the enemies in an area have detected the players presence. Generally speaking, the more that the enemies detect the player, the more aggressive their efforts to hunt down and kill the player are. This means that it is generally in the best interest for the player to have the lowest amount of detection possible, so that the enemies are more passive and easier to deal with.

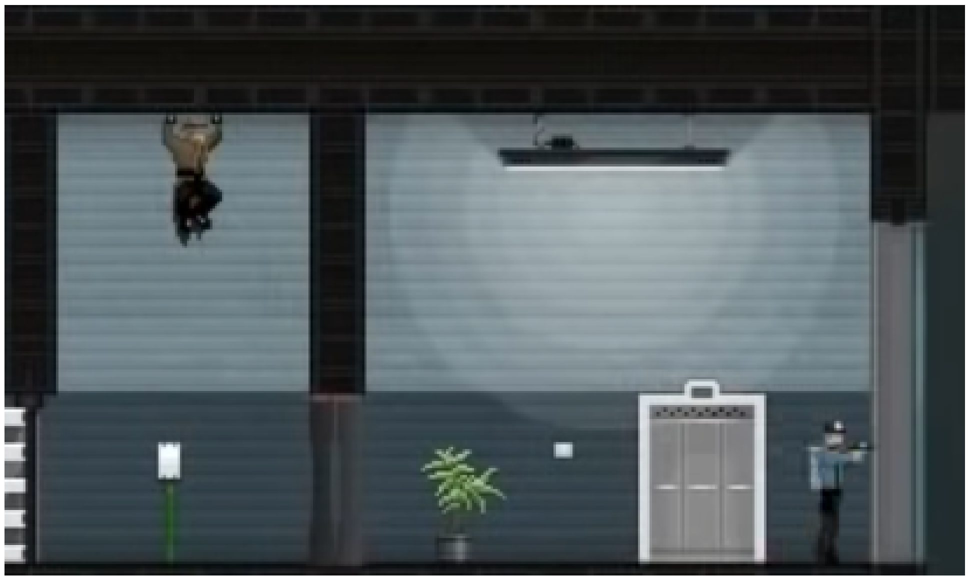
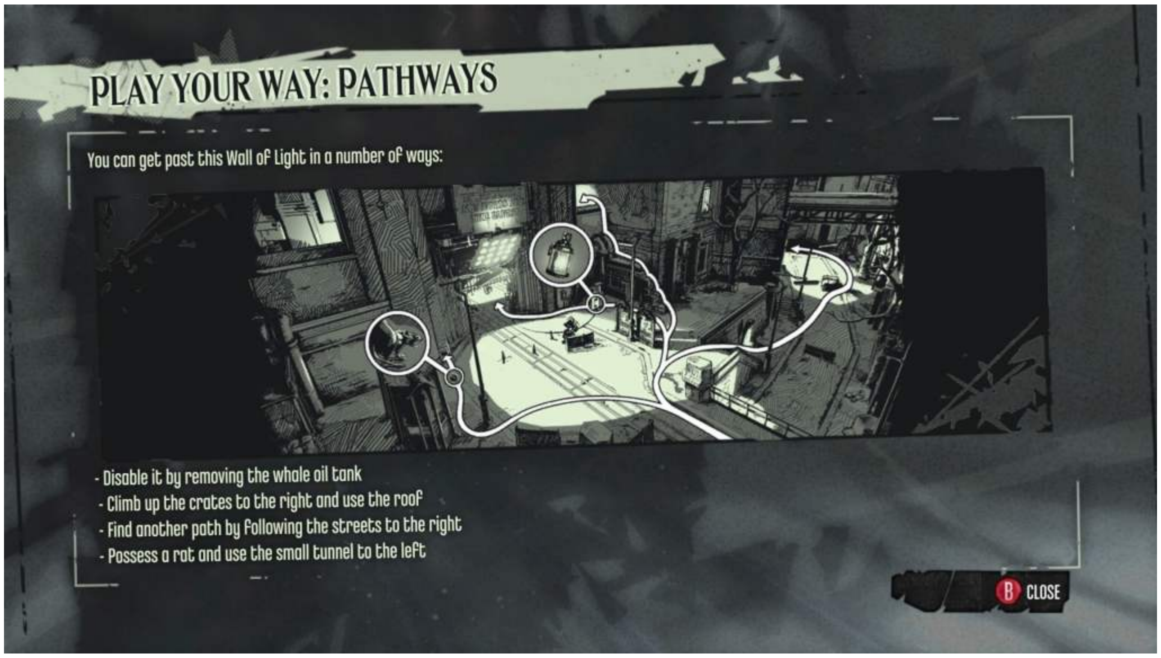


Figure 2.1: Player hiding from a guard in the game *Gunpoint* [2]

Herein lies the foundation of the decisions that make the games fun. When faced with a challenge involving an area full of enemies, the player can make a strategic decision to either embrace or forego stealth. From there, there are countless micro-decisions that the player will make that involve the execution of said strategy. All of these decisions make up the main gameplay loop: the collection of major things that the player does while playing the game.

A design principle from the creators of *Dishonored* is the idea of letting the player "Play your way". I believe this to be an exceedingly powerful concept, and is the driving concept for the design of my game. This idea dictates that all of the different major choices that the player can make when playing this type of game should be equally viable. In other words, a player should be able to go with a stealthy or a direct approach, and both of these options should be able to let the player win the game with about the same level of difficulty as the other. This is such a powerful concept because it lets the player make a choice, which gives them a feeling of power and control over the game, while also ensuring that they are rewarded for this choice. The player gets to win, but they also get to win on their own terms, which is that much more satisfying.



You can get past this Wall of Light in a number of ways:

- Disable it by removing the whale oil tank
- Climb up the crates to the right and use the roof
- Find another path by following the streets to the right
- Possess a rat and use the small tunnel to the left

Figure 2.2: *Dishonored*'s concept of "Play your way"
[3]

This idea of "Play your way" is fantastic in theory, but an interesting thing to tackle in practice. Making sure that multiple approaches to gameplay are relatively equally viable is a technique called balancing. To make sure that players truly can play their own way, the two major approaches, stealth and direct, must be fairly balanced. Since these approaches are so different, it is a slightly complex issue as to how to balance them. The general idea however, is that the direct approach is much simpler in the macro-strategy, since it boils down to running at the enemies in your way and killing them. However, this is balanced by having the micro-strategy of a fight with the enemies be the difficult part. The enemies should not be pushovers when directly confronted, or the direct approach would be the clear choice in any encounter. The stealthy approach can be seen then as a reflection of the direct approach. To remain undetected, the player will have to form a solid plan as to exactly where and when they will move to avoid the enemies. In this way, the macro-strategy is harder than the micro-strategy which just boils down to movement of the character. Fine tuning the number values for the combat and detection systems are what will solidify the balance that is necessary to allow the players to play their own way.

2.3 Unity Engine Specifics

This project has been made entirely using the free Unity game engine [4]. Its robust features, clean user interface, and free price tag make it highly popular among independent developers like myself. Since the software is made specifically for creating video-games, the environment is different to IDEs that I have used in the past like PyCharm or IntelliJ. For that reason, I will define the pertinent terms that made up the work-flow of my project, as well as some examples of how they work in such a project:

- **Scene:** This is where the game developer adds things to create the environment that will make up a level in the final game. Since many games have multiple levels in them, a developer can also create numerous scenes that represent different levels.

- **Game Object:** This is a general term for an object placed within the scene. Everything that is added to a game is a game object. This is an important idea, because it mimics the ideas of object-oriented-programming since everything in the game maintains this underlying identity as a game object.

- **Component:** This is the general term for something that you attach to a game object. These components serve some sort of function to the game object that they are attached to. The most common component is a script, but there are a wide variety of other types that do drastically different things.

- **Script:** In Unity, scripts are components that hold pieces of code. This is highly useful for creating video games because it allows the developer to write code for specific things. For instance, since what I want the player to be able to do is drastically different than what I want the enemies to do, I have assigned scripts to the player and enemies respectively.

- **Sprites** For 2D games, sprites are a term used for the pieces of art used to visually represent different things in the game like players, enemies, or environmental features. For example, the player has a sprite that looks like a hooded man with a sword.

- **Collider** Colliders are another type of component. They are defined by fields around some game object that are able to detect when the fields are touched and by what. These are monumentally useful to game design because we emulate life when we design games. In life, touching things are often the way that we interact with them, and in games this fact is no different. Additionally, being able to tell exactly what touches what is helpful since it allows interactions like a bullet firing and damaging a target. A conditional can be applied to an enemy that specifies that collision will reduce their health only if that collision is from a bullet.

- **Trigger** Triggers are fields that are a form of colliders, but they do not have a physical presence. They are intangible and do not physically collide with anything. However, if something enters their defined field, they can return what is triggering them in the same way colliders return what is colliding with them. These are useful in situations when you want to detect something happening, but not physically stop the thing from happening. For example, if you wanted a musical cue to play when the player entered a room, you could use a trigger so that the musical cue would play at the correct time, but would not impede the player's movement.

- **Prefab** An abbreviation of "prefabricated object". A prefab is an object that has properties already defined about it. It could have any number of components attached to it, with any values in those components defined. A prefab is highly useful if you want to have a lot of some object. For example, bullets and enemies are both prefabs in my game.

The format of attaching components to game objects supports an underlying idea of modularity which I attempted to follow as best I could when making the game. Creating a game in this way is the greatest test of this concept of modular object-oriented-programming I have encountered thus far. To maintain an appropriate level of readability and organization, scripts should only be applied to the objects they affect. Additionally, it was necessary to break up the code for complex objects, like the player, into multiple scripts. Having every line of code within one script would create a file hundreds of lines long that is pointlessly difficult to read. Instead, breaking up the script into specific functions proved to be much more clean. For example, the player has a script that manages statistics like health, a script that controls movement, a script that controls basic attacks, and a script that controls the players abilities. This focus on modularity makes Unity a particularly effective engine to debug in.

Chapter 3

Work During the Summer

The work for this project was tackled over months and through several different stages. I had the idea to create this game long before I decided to work on it for my capstone project. I pitched the idea of creating a video game from scratch to a couple of my like-minded friends for fun and for practice. The three of us went through a rigorous planning stage to make sure we outlined exactly what type of game that we wanted to implement. After the planning stage we worked on the game sporadically throughout the summer: slowly building pieces of a game. After the end of summer, we stopped working on the game as a team, and I pitched finishing the game as a capstone project so I could have an excuse to still work on it during school.

3.1 Planning

While researching the processes that go into creating creative works, like movies, books, or video games, I repeatedly heard that the core of such works should come from one source. That is to say that the main concept of a creative work should be the vision of one person.

I wanted to put this idea to practice myself, so I proposed to friends of mine that we develop a game as a group over the summer. The idea of maintaining one person's conceptual integrity throughout the game was the prime focus of our planning sessions. Almost all of this planning had to be done online because of distance, and we used websites like Trello and MURAL to brainstorm general ideas about the scope and theme of games we wanted to make. Knowing about the dangers of making a creative project without a strong unified vision, we each took a few days to come up with and outline the concept for a game we could make. From there we were each able to vote for one of the projects that was not our own. We felt this was a happy middle ground between making sure that we had a unified concept for our game, and also making sure that there was not one person with absolute authority among us friends. We decided to go with the idea of making the 2D stealth-action platformer I have described already.

After we decided on a decisive concept for the game, it was time to make plans as to how to divide up the labor. On a large-scale macro sense, the decision was easy. Two of us are programmers, and the other is a digital artist. The only difficult part of the division of labor involved the question of what exactly the two programmers should work on. We ultimately decided that I would work on the code for the player, while my friend would work on the code for the enemies. We decided that further division of labor could come later: once we had made significant progress on the project. We also agreed to meet twice a week on Tuesdays and Saturdays. At these meetings, we would discuss our current progress, and discuss what everybody should work on until the next meeting.

3.2 Basic Character Mechanics

My contribution to the game over the summer consisted almost entirely of implementing the mechanics of the player controlled character, and most of the tertiary game systems like the camera and a script that managed certain unseen aspects of the game. With the combination of my scripts and artwork and animations provided by my digital artist friend, we were able to create a basic, but completed, player character prefab. This character had all of the basic mechanics that we wanted, as well as fully implemented animations for each mechanic.

3.2.1 Player Movement

The first thing that I did when coding the mechanics for the player was to create a script that would control the player character's movement based on the input of the player. The basic controls for a standard platformer were present: the player could move left and right and was able to jump by pressing the spacebar key on the keyboard. However, even relatively simple sounding mechanics like those proved interesting to deal with. For example, we did not want our player to be able to jump any time they felt like it. It makes sense that the player would only be able to jump if they are standing on the ground and can jump off of it. To implement this, I placed an invisible collider at the bottom of the player controlled character's model called groundCheck. Unity has a built in layering system that allows you to set any object in a scene to a layer. Additionally, when any collision occurs, you can check if the collision occurred with an object in a specific layer. Thus, I was able to allow the player to jump only if their groundCheck collider was touching the ground.

After creating this implementation, I expanded it to include the possibility of multiple jumps. Many games of all genres feature the idea of a "double jump" wherein the player's character can jump an additional time before they have touched the ground after their initial jump. It has become more of a staple in modern platforming games because it allows the developer to create more complex and interesting challenges to traverse since the player has inherently more freedom of movement. For these reasons, I wanted to keep the door open to including extra jumps to the player's movement. To do this, I simply added an integer to the script called totalJumps representing the maximum number of jumps the player can perform, and a counter within the update method called currentJumps. From here, I changed the logic of testing whether the player could jump to see whether currentJumps ¿ 0. Every time the player jumped, I decremented currentJumps, and when the player touches the ground, it is set back to totalJumps.

The last piece of the player movement script is a function called Flip(), that reverses the direction that the player character's model is pointed. This function gets called when the user inputs a direction to move their character that is opposite to the direction they are currently facing. An easy way to think about the mechanics of this function is to imagine that the player's model is undergoing a geometric reflection over a y-axis. We can use something called the local scale of our player's model to apply such a transformation to the model. Multiplying the x value of our local scale by -1 will give us a reflection of the value we currently have for it.

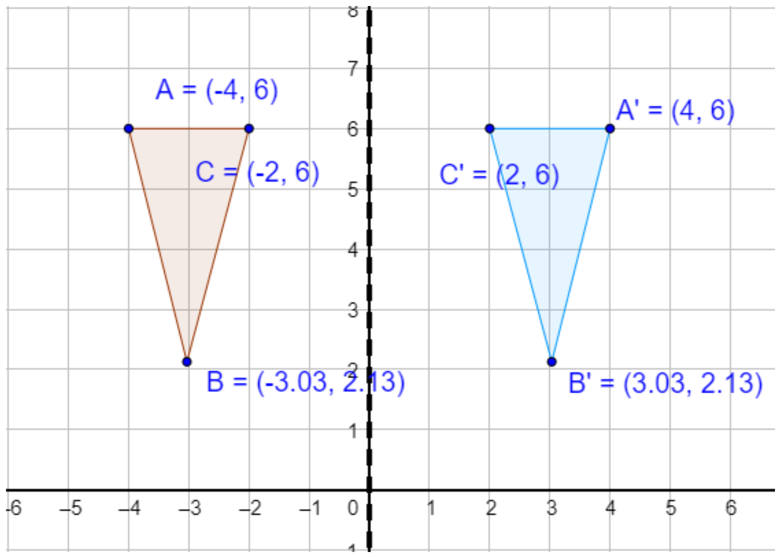


Figure 3.1: Example of flipped object

If we look at the values of the x coordinates in each point in the triangles ABC and A'B'C', we can see that they have undergone the same transformation as would occur with my flip function. This is a highly useful implementation of this idea because it allows me to later write scripts that follow the idea of doing something from the player's front, like shooting a projectile, without having to worry about which direction the player is looking.

To see a YouTube video demonstrating how the player moves, follow this link: https://youtu.be/3n5LgF0NWOg

3.2.2 Player Attack

If the player either fails in maintaining stealth, or forgoes it completely, we have a direct attack that they can use against enemies to damage and kill them. The implementation for this mechanic is relatively basic, but I do not feel as if it needs to be much more complex. There is an invisible trigger, a collider without a physical collision, in front of the player's character that will act as the area in which they swings their sword. When the player clicks the left mouse button, the area in front of the player evaluates if anything is within it. For every object within the area, if it falls under the enemy layer, that object takes a set amount of damage.

3.2.3 Player Animation Web

The last big contribution made by me over the summer was the creation of the animation web for the player. Learning how to create this and implementing it were difficult to wrap my head around, but the results were worth the trouble. With the exception of a few small animations, by the end of summer, the player's character had animations for every action that the player could perform.

The name "animation web" is an appropriate one. The web consists of a series of nodes that represent individual animations which are connected by arrows that represent conditionals.

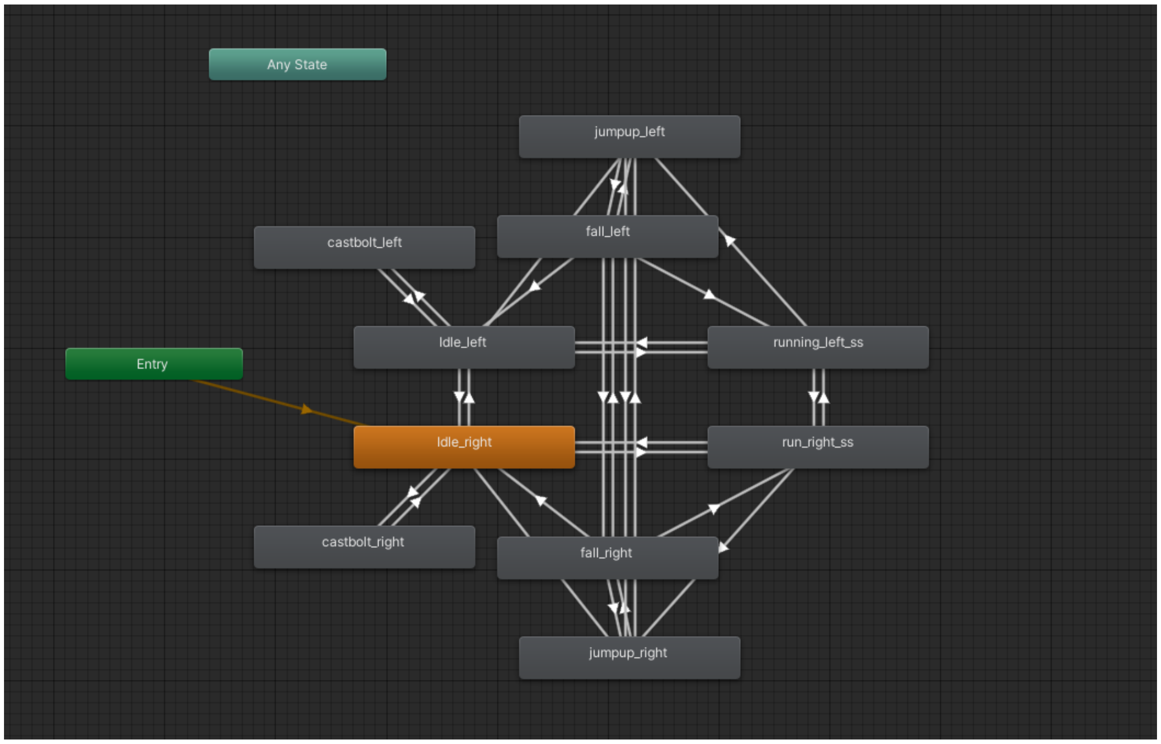


Figure 3.2: Player animation web

Above, we can see the current animation web for the player. Though it may look overwhelming, we can use this web to visualize what animations lead into others. The animation labeled "idle_right" represents the animation that we want to play when the player is standing still and facing right. It is highlighted in yellow because it is the starting animation, and will be the animation that is being played when the player spawns in. However, our game would not be particularly fun if we forced the player to constantly stand still, and so we need to have a way to play a running animation when the player is moving. Looking at the animation web, we can see that we have an arrow pointing from "idle_right" to an animation called "run_right_ss" which is the animation of the player running in the right direction. This arrow represents a conditional statement that evaluates some of the attributes of the player. For the particular arrow mentioned, it detects if the player's horizontal velocity is greater than 0 and if they are on the ground. If those two are both true, the animation "run_right_ss" plays since we know the player is moving right.

The relatively large amount of arrows between the nodes of the animation web should be a solid indicator as to how challenging it was to set up. However, the outcome was well worth the effort. The animated player character is not only visually pleasing to watch in action, but also invaluable in visualizing actions that are being performed for debugging purposes. It is much easier to discern a problem with the player when I am being given visual clues as to what it is doing rather than just having to trace through my code.

A small detail that I think is worth mentioning is that the animations of our player consistently reflect handedness. The character that the player controls holds his sword in his left hand, and if you look at him in any of his animations he will still be holding his sword in his left hand. This was a much harder detail to implement than you might think. The difficulty arose in how our animations interacted with the flip function I described earlier. Since the flip function works like a mirror to reflect the player across a y-axis, it would also reflect the animations that the player is using. If we just had one set of animations, the flip function would swap the handedness of our character every time he turns directions. To fix this, we had to have two sets of animations for the two directions the player can be facing. I also tried to reflect this, no pun intended, in the animation web by making the left animations symmetrical to the right animations.

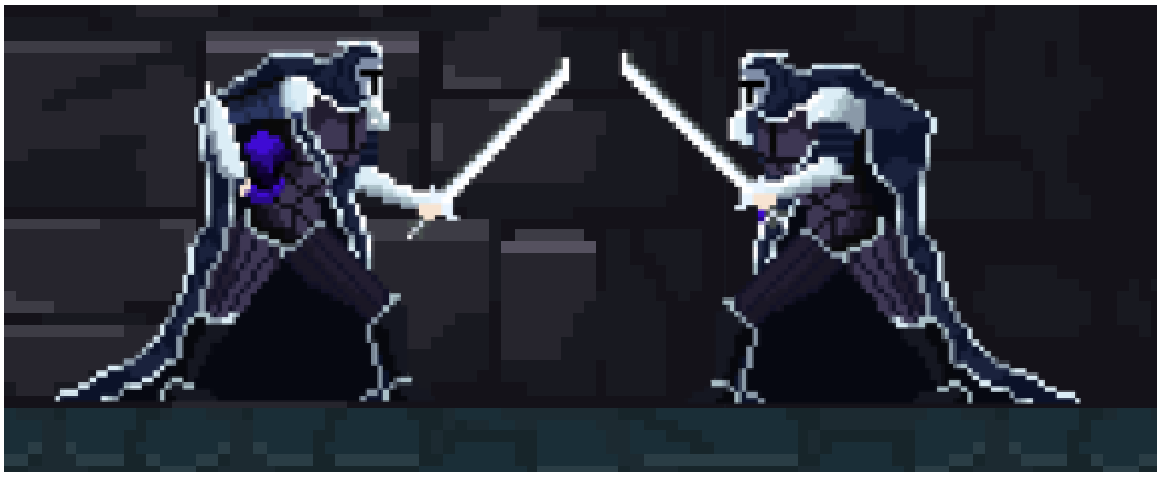


Figure 3.3: Player's character left-handed looking both directions

Chapter 4

Work During the Semester

4.1 Camera

Creating the camera and the scripts that control it took a relatively short amount of time. Unity scenes come with a built-in camera object that the developer can manipulate to control what the player sees. I wanted the camera to follow the player as they traverse through the level, so I made a simple script that moves the camera towards the players position. However, having the camera follow the player perfectly and exactly when they move can make the player feel disoriented and a bit claustrophobic. This is because the player's character would be moving, but it is hard to visualize since the character will maintain a fixed position within the view of the camera. The background would scroll behind them, but the player won't *feel* like they are really moving. To fix this problem, I told the camera to go from it's current location to the players location, but over time using a technique called linear interpolation or Lerp in Unity. The basic idea of this is that the camera will slowly move to follow the player if the player is close to it. If they are far, the camera will move faster. This gives the player the feeling that they are moving through the level, without having them leave the camera behind.

4.2 Player Abilities

Over the summer I developed the basic mechanics that the player controlled character could work with. To add more depth and options to the gameplay, I also wanted to add special abilities that the player could use to aid them in beating levels. Since these special abilities are inherently more powerful than just running up and trying to hit an enemy, their use is restricted in some way. For some, it is by having its use bound to use of a resource, and, for others, using the ability at the wrong time can leave the player more vulnerable than if they did nothing.

If you want a visual demonstration of the player's abilities, see this short YouTube video: https://youtu.be/ewDikZJWbf8. the description of the video will have more information about what exactly is being shown off.

4.2.1 Dodge-Roll

The "dodge-roll" or just "roll" is a staple of action games. The idea of the roll is that the player pushes a button and the character performs a quick movement to a location a set distance from the player. Typically this is accompanied by an animation of the player tucking and rolling on the ground, but my technical artist friend was unable to provide an animation during the school year. In most games, while the player is tucking and rolling they will be immune to damage. This makes the roll a potent defensive technique, as it allows the player to evade incoming damage that would otherwise hit them. The amount of time that the player is invincible is important to balance if a roll is added to an action game because damage immunity is so strong. If the amount of time is too long, the combat is reduced to a farce as the player rolls constantly through enemy attacks, unable to be damaged. However, if the time is too short, the roll can feel too difficult to use for the risk it incurs, causing players to completely avoid using the mechanic as if it did not exist.

Implementing this roll was quite a challenge since there are multiple subtle things happening while the roll is taking place. The basic mechanical outline of the roll is that the player is moved to a location a set distance away over the course of a set amount of time. We restrict the roll to taking a set amount of time to perform because it would be unrealistic and disorienting if the player just teleported to their roll destination. Additionally, a challenge I did not initially think of when creating this mechanic was the idea that, while the character is rolling, no player input should be considered. This reflects the physics of real life: if we dive forward to tuck and roll, we cannot magically turn 180 degrees and go the other way.

To fix the problem of restricting player input, as well as giving damage immunity to the player, I made an isRolling boolean. At the start of the roll, isRolling is set to true. At the end, it is set to false. This way, I was able to add a restriction to the player character's movement script. I just placed all of the code for movement inside of another if statement that just checked "if(!isRolling)". In the same way, an enemy can only do damage if the player is not rolling.

4.2.2 Blink

Though the implementation of this mechanic is relatively simple, it is worth mentioning. This ability is basically like a roll if it were instantaneous. This speed is what gives the mechanic its name, since the player will travel somewhere in the blink of an eye. The miniscule amount of time that this can be performed in requires some sort of limiting factor to balance out its strength. I balanced this ability by tying its use to a resource cost. The player has a set amount of magical power that they can use, and using the blink drains some of it. Once the player is out of magic power, they will be unable to use the blink anymore. This gives the player an interesting cost-benefit analysis that they can make when weighing options to use in a given encounter.

I would not say that the current implementation of the blink is where I would want it to be for a full game. Moving forward, I would change a couple of things.

First, I would delay the teleportation for a fraction of a second after the player hits the button to use the ability. My reason for doing this is explained eloquently by Alan Becker in his video describing the 12 principles of Animation[1]. One of the principles is "anticipation" where a character does something before an action to indicate what that action will be: like raising a hammer high above their head before slamming it down. If the player teleports instantly with no anticipation, it feels disorienting. Adding even a small amount of anticipation would make the ability feel much better to use.

Secondly, I need to implement a check that determines whether the destination for some teleport will end up out of the level. Currently, the player could teleport out of the bounds of the game, or into a wall, effectively ending the game.

4.2.3 Shadowbolt

This mechanic involves the player firing a damaging projectile out of the front of their body. Like the blink, this is a seriously powerful skill since it has the advantage of being ranged. This allows the player to attack enemies from a position the enemies cannot immediately retaliate from. For that reason, I also tied it to the magic power system, so that the player will have to be frugal as to when they want to use their magic power.

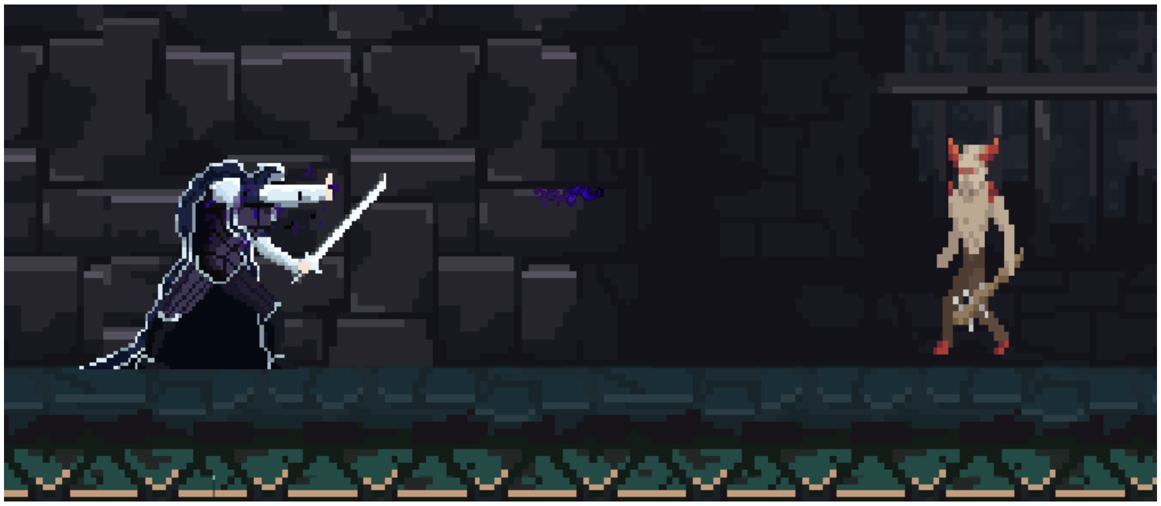


Figure 4.1: Player Firing Shadowbolt at Enemy

Implementing this mechanic took a while, but was not too conceptually challenging. At it's core, the implementation involved freezing the player in their current position and then instantiating a Shadowbolt object that has a starting velocity that can be reversed given if the player is looking left or right. This required me to make a Shadowbolt class, but this did not take too much work. It has attributes for speed and size, and methods for what happens when it hits an enemy.

4.3 Detection

Instead of basing the detection purely off of distance from the player, I learned about and implemented an incredibly helpful tool: raycasts. An easy way to think about how a raycast works is by imagining that it is a laser beam pointed in a direction from some origin point. Based on how you set up your arguments when you create your raycast object, you can detect any number of things that this raycast collides with. This was immensely helpful in allowing me to implement a more realistic detection system because I could set a raycast at the eye level of the enemies and have them detect the player if the raycast hits the player.

Combining the two approaches, raycast and distance based detection, I was able to create the detection system I was looking for. In my ideal detection system, I wanted to mimic the system of many stealth games where there is a gradient to the speed at which the enemies detect the player. What this means is that if an enemy is looking at you, but is very far away, they will become suspicious rather than fully hostile and go to investigate. For me, this approach is ideal since it allows the player to make a mistake in their stealth, but are not harshly punished for it if they made the mistake at an ideal time. On the flip-side, if they make said mistake when there is an enemy right next to them, they are rightfully punished with a full detection.

My implementation represented a relatively basic mimicry of this idea. I wrote a script that both created a raycast object that was centered at the eyeline of an enemy, as well as defined a distance in which the player could be detected. If the player is hit by the raycast and is within this distance, the enemy detects the player and starts moving towards them to attack. If either of those are untrue though, the player is not detected by the enemy. This implementation, while not incorporating the idea of a gradient detection, gets the job done and is a solid approximation of the mechanic I was trying to achieve.

4.4 Level Creation

Without a level for the player to move around in, there is no game. To create the level for the player, I utilized the immensely useful tile palette tool in Unity.

4.4.1 Tile Palette

The tile palette is a tool that enables the easy use of tile maps to create aesthetically pleasing levels. Given the dimensions of the tiles in your imported tile map, you can slice the tile map into individual tiles inside of Unity. In combination with a Grid object providing a rigid structure, I was able to take these individual tiles and place them into the Unity scene to create my level. This proved to be an effective method in tackling the creation of the levels, since the majority of tedious work like lining up tiles was handled by the Grid and tile palette.

4.4.2 Level Design

Due to time constraints, I created one level for the game that served as an area to test out everything that the player could do to interact with the environment as well as enemies. When creating the level, I tried to maintain conceptual integrity with the idea of ”play your way” I mentioned before. To achieve this, I created an area that I feel could illustrate a strategic choice for the player of whether to use stealth or be direct.

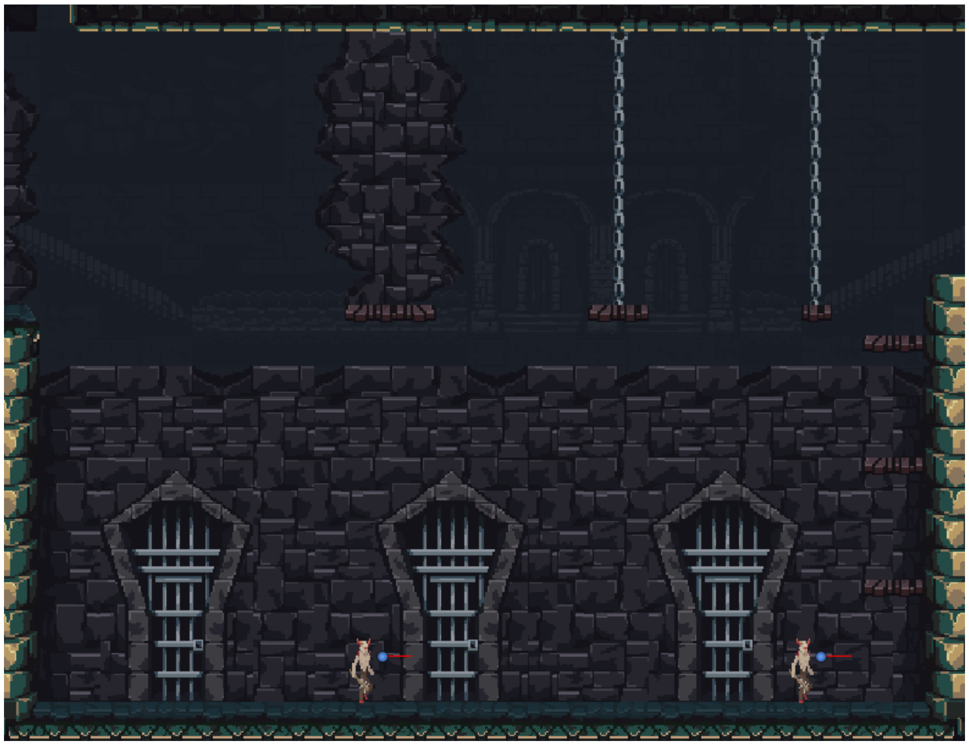


Figure 4.2: Pit with multiple routes

In the above figure, you can see that there are multiple approaches to getting past the pit with enemies in it. The player can drop down into the pit from the left and attack the enemies head-on, or they can try their luck with the platforming to bypass the enemies. This is not a perfect example, given that the choice is not a perfectly balanced one. In the current state, the choice to try and avoid the enemies is the clear best choice. However, I feel as if the example demonstrates the idea I was going for.

Chapter 5

Conclusion

My goals at the beginning of the semester for this project were to create a fully playable game that had a solid level of polish to it. When considering my project in relation to these goals, I would say that I was unable to perfectly realize the goals as I laid them out. However, I would also argue that I achieved the core ideas of what I wanted out of this project. Though I was unable to create a fully playable game, I made a playable demo in which the player can move and interact with their environment and enemies. The main goal of this project, and the reason I wanted to start working on this game over the summer, was to gain experience in creating a game from the bottom up, and I feel I have definitely achieved such experience. Additionally, the demo that exists is relatively polished considering my abilities. I feel that the project has been a success in achieving most of my goals regarding the learning experience, if not a success in making a full game.

As a side note, this is not the last time I will work on this project. I intend to continue work at least into the next year, with a focus on getting a playable, if not polished, game working. That will entail adding a win condition to my demo level, and adding multiple routes to this win condition. While designing these routes, and the level as a whole, I will strive to maintain that idea of ”Play your way” since it is such a powerful concept. From there, it would just be about creating more content in the form of multiple levels, and other features.

Bibliography

[1] Alan Becker. 12 principles of animation (official full series), 2017.

[2] Tom Francis. Gunpoint, 2013.

[3] Arkane Studios. Dishonored, 2012.

[4] Unity Technologies. Unity Game Engine, 2005.