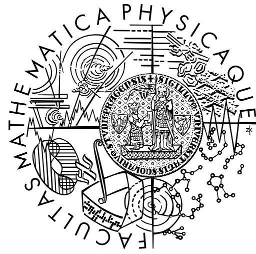

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Ogulhan Bozkir

Just Blade: A 3D melee combat game in a medieval setting

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Filip Kliber
Study programme: Computer Science
Study branch: General Computer Science

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, Mgr. Filip Kliber, who guided me through the arduous process of writing this thesis; and my mother, without her unwavering support, I wouldn't even have the chance to be at this point in the first place.

Title: Just Blade: A 3D melee combat game in a medieval setting

Author: Ogulhan Bozkir

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Filip Kliber, Department of Distributed and Dependable Systems

Abstract: The goal of the thesis is to develop a 3D medieval melee combat game. In the game, the player defends themselves against hordes of enemies coming in waves. The player earns gold with every slain enemy, which can be used to purchase better weapons and armor; or hire mercenaries to help fight against the hordes. The player can choose to buy heavy armor which provides better protection against incoming damage; or remain quick on their feet by wearing lighter armor instead. The game features a four directional combat system (i.e.: the ability to attack and defend in four directions).

Keywords: game, 3D, unity, melee, combat

Contents

1 Introduction **3**
1.1 Similar Games . . . 3
1.1.1 The Last Stand . . . 3
1.1.2 Mount & Blade: Warband . . . 4
1.1.3 Mordhau . . . 5
1.2 Goals . . . 6

2 Analysis **8**
2.1 Agent Movement . . . 8
2.1.1 Rigidbody . . . 8
2.1.2 CharacterController . . . 9
2.1.3 NavMeshAgent . . . 9
2.2 Animation . . . 10
2.2.1 Four Directional Combat . . . 10
2.2.2 Keyframe Animation . . . 11
2.2.3 Physics Driven Combat . . . 11
2.2.4 Combat Coherency . . . 12
2.2.5 Separating the Upper and Lower Bodies . . . 12
2.2.6 Making Movement and Look Directions Different . . . 13
2.3 Items . . . 14
2.3.1 Weapon and Armor Values . . . 14
2.3.2 Weapon Collision . . . 15
2.3.3 Weapon Specific Animations . . . 15
2.3.4 Wearing Armor . . . 15
2.3.5 Movement Speed Penalty . . . 16
2.4 Menus . . . 16
2.4.1 All in One Scene . . . 16
2.4.2 Using Multiple Scenes . . . 17
2.4.3 Managing Player Progress . . . 17
2.5 Combat Scene . . . 17
2.6 Horde Difficulty Progression . . . 18
2.7 Sounds . . . 18

3 Implementation **20**
3.1 Folder Structure . . . 20
3.2 Designing the Agent Entity . . . 22
3.2.1 Agent and Controllers . . . 22
3.2.2 Compound Agent Entity . . . 23
3.3 Agents . . . 24
3.3.1 Movement of PlayerAgent . . . 25
3.3.2 Movement of AiAgent . . . 26
3.3.3 Jumping . . . 27
3.4 Animation . . . 27
3.4.1 Source State Until Fully Transitioned Out . . . 27
3.4.2 AnyState Does not Apply to Transitions . . . 28

3.4.3 Transition Duration Cannot be Changed at Runtime . . . 29
3.4.4 Waist Warping 30
3.4.5 Arm Going Backwards 31
3.5 AgentAC 31
3.5.1 Base Layer 32
3.5.2 AtkAndDef Layer 32
3.5.3 Idle Layer 33
3.5.4 How the Layers Work Together 33
3.5.5 AgentAOC 34
3.5.6 Combat Animations Logic 34
3.5.7 Blend Tree 36
3.6 Combat Coherency Circle 37
3.7 Scenes 38
3.7.1 MainMenuScene 39
3.7.2 InformationMenuScene 39
3.7.3 GearSelectionMenuScene 39
3.7.4 ArenaScene 40
3.8 Weapons 42
3.9 Armor 44
3.10 Horde Prefabs 45
3.10.1 Horde Agent Data 46
3.10.2 ArmorSets 46
3.10.3 WeaponSets 46
3.10.4 CharacteristicSets 47
3.10.5 RewardData 48
3.10.6 InvaderAgentData 48
3.10.7 Adding a New Invader 48
3.10.8 MercenaryAgentData 49
3.10.9 Horde Difficulty Progression 49
3.11 Sounds 49
3.12 Collision Layers 50

4 User Guide **51**
4.1 Installation 51
4.2 Menus 51
4.2.1 Main Menu 51
4.2.2 Information Menu 52
4.2.3 Gear Selection Menu 52
4.3 Gameplay 54
4.3.1 Combat 54
4.3.2 Enemies 55
4.3.3 Gameplay Strategies 57

5 Conclusion **59**
5.1 Future work 59

Bibliography **61**

1. Introduction

The objective of the thesis is to develop a 3D medieval melee combat game. In the game, the player has to defend themselves against hordes of enemies coming in waves. The player earns gold with each fallen enemy. After each time the player survives the onslaught, they are taken to menus where they can use the gold to purchase better weapons and armor; or hire mercenaries to help defend against the next assault. The player can choose to wear light armor to remain quick on their feet; or heavier armor which provides better protection against incoming damage at the cost of movement speed. The implementation of the game features a four directional combat system (i.e.: the ability to attack and defend in four directions).

The resulting video game is called `Just Blade`. The name “Just Blade” is a word play on `Mount & Blade` [1], in the sense that my game doesn't have any mounted combat, therefore it is “Just” (i.e.: “Only”) Blade. The development of my game took inspiration from several games, which are explained in the next sections.

1.1 Similar Games

There are many genres a video game can be associated with, and one of which is called the horde game genre. In this genre, the player tries to play for as long as possible without dying in an uninterrupted session while the game presents them with increasingly difficult waves of challenges [2]. In video games such as The Last Stand, the whole point of the game is the horde game mode itself. However, in some other games, it is simply included as a mini-game mode as in the case of Mount & Blade: Warband and Mordhau. The next sections briefly describe what these games are like.

1.1.1 The Last Stand

Figure 1.1: A screenshot of The Last Stand browser game from my playthrough on Armor Games. There are three survivors (including the player) who fight off zombies in the second night of the game.

The Last Stand is a browser game [3] that was released on Armor Games in 2007 [4]. In this game, the player is positioned behind a barricade and fights off zombies that attack at night. During the day, the player allocates time to find better firearms, repair the barricade, or find survivors to help them fight in the next nights. The goal is to survive for as many nights as possible until the game ends where the survivors get rescued.

Just Blade is a 3D medieval combat game whereas The Last Stand is a 2D shooter game, but the fact that the player fights off hordes of enemies in both games is what makes them similar.

1.1.2 Mount & Blade: Warband

Mount & Blade: Warband (abbreviated "M&B:W") is a video game that was released in 2010. It's a strategy action roleplaying game developed by TaleWorlds Entertainment, and published by Paradox Interactive [5]. M&B:W features an invasion mode on multiplayer which was released on December 20, 2016 [6]. It wasn't around at the release of the game in 2010, so the explanations that I provide below are based on the few minutes of playthroughs that I had.

The game mode can be played by multiple players. Each player chooses their equipment as they normally would on any multiplayer game mode. However, in addition, the player has two computer controlled companions to choose from. These companions are the same ones as you would find in the single player campaign. Each companion has certain proficiencies, meaning while one companion could be good at fighting with two handed weapons, others may prefer one handed weapons. The player can also choose the class of their companions, which are based on the factions that they sided with while configuring the game mode. These classes are usually infantry, ranged or cavalry.

Figure 1.2: An in-game screenshot that I took of M&B:W's invasion game mode.

The invasion mode consists of enemies attacking in waves. The players are given some time to prepare before the wave approaches, in which the player chooses equipment for themselves and their companions. Once the time is up, the wave of enemies begin their attack. Each enemy killed provides the player gold. Sometimes, the enemies can drop loot chests, from which contain special

items that cannot be obtained by normal means (e.g.: boots that make you move faster, knives that steal life from enemies, etc.). The players can choose to keep these items for themselves, or assign them to one of their companions.

The gameplay is seamless, in the sense that there are no pause menus between the waves. Each wave must be thwarted within a time frame such as 5 minutes, or else the next wave will begin without finishing the current one. Some waves are special, where they contain characters from the singleplayer campaign (i.e.: lords, counts, boyars, etc.). The arrival of such enemies are announced by the in-game text during the countdown. Every now and then, a "supply caravan icon" appears in the game. This indicates that any fallen players and companions will have a chance to respawn soon. However, if every player and companion dies, the enemies win and the game is over.

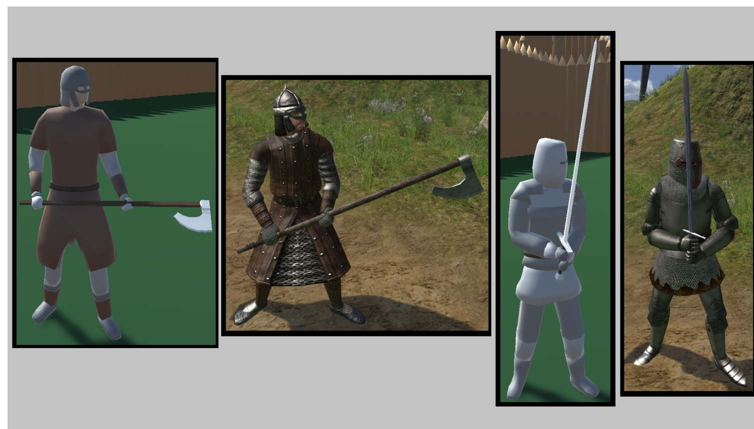

Figure 1.3: A comparison of the visual similarities of models in Just Blade (left) and M&B:W (right) from the in-game screenshots that I took.

Just Blade was heavily inspired by M&B:W's four directional combat system, as well as its visual aesthetics. While M&B:W's invasion mode is similar to Just Blade horde gameplay, there are also some differences. One of those differences is the fact that M&B:W's gameplay is seamless as explained above, whereas Just Blade takes the player to menus after fights where they can make purchases.

1.1.3 Mordhau

`Mordhau` is a video game that came out in 2019. It's a multiplayer medieval hack and slash fighting game developed by `Triternion` [7]. It contains a game mode called horde.

Mordhau's horde mode is very similar to M&B:W's invasion mode. In Mordhau, the game mode is played with other players. The enemies come in waves after a certain preparatory time period has passed. Unlike M&B:W, the players do not have AI controlled companions. In addition, there is an objective goal in Mordhau's horde game mode, which is to protect the "Noble". The Noble is a computer controlled warrior, which wanders near the starting point. If the enemies kill him, the game ends, regardless of how many players are still alive. Luckily, the Noble is a tough warrior himself, and he can fend off enemies as well. When a player dies, the other players can revive the player by holding a button for some time.

Figure 1.4: An in-game screenshot that I took of Mordhau's horde game mode.

For each enemy killed, the players earn gold; and with each wave beaten, the players earn skill points. The gold is used to purchase weapons and armor; while the skill points are used to specialize the player's character in the following skill trees: melee, tank, support, ranger. They contain skills which give passive bonuses throughout the game. At the end of each skill tree, there are active skills which give more useful effects, but suffer a cooldown period. For example, the melee tree has an active skill that makes the player invulnerable to damage for a few seconds.

Much like M&B:W, Mordhau's horde mode is also seamless, in the sense that there are no pause menus once the game has begun. Similar to M&B:W, the purchases are done through an in-game menu. Even though Just Blade is not seamless, it was inspired by Mordhau's horde game mode, as well as its weapon damage and armor system.

1.2 Goals

The implementation of the game is centered around a few key goals. The game may contain features other than described below, simply because those features are relatively minor and need not be mentioned. The goals are as follows:

1. All agents must have the ability to do the following:
   (a) Attack while moving.
   (b) Move in a direction which is different from their look direction.
   (c) Look up/down while attacking in order to target different body parts of their opponent.
2. Agents must be able to attack and defend in four directions (i.e.: up, down, right, left). This should be done in a coherent manner. For example, the agents should not be able to block an attack if their back is turned against their opponent, even if the directions match.
3. The weapons must have varying features such as damage, length, and speed. In addition, the animations must be suited according to the way a particular class of weapon is held (e.g.: swords, spears).

4. There must be varying levels of armor, with each level providing more protection against damage at the cost of movement speed. There must be multiple armor pieces to cover different body parts (e.g.: head, torso, legs).

5. The scene in which the agents fight is expected to have the following properties:

   (a) A mostly flat surface to move around, with some small slopes.

   (b) Some impassable obstacles (e.g.: boulders, trees, fences) which the agents must move around.

   (c) A background landscape such as a mountain with trees or the battlements of a fortress.

   (d) The starting locations of the combatants on each side must be situated at opposite ends of the scene.

6. The hordes of enemies must be harder to tackle as the game progresses, yielding better rewards when beaten. There must be a difficulty slider to further adjust the difficulty of the game.

7. There must be several menus that are seen before each fight, which are as follows:

   (a) An information menu where the player is given information about the next fight.

   (b) An item shop menu to buy weapons and armor.

   (c) A mercenary menu to hire mercenaries to aid the player in combat.

2. Analysis

This chapter goes over some of the challenges that are expected in order to implement the features that were described in section 1.2. It covers various ways to address the issues that are described, and the one which is believed to be the most suitable solution is selected.

Note that the text focuses on the `Unity Engine` [8] as a tool for analysing the implementation of the game's various aspects. While it is possible to use other known engines to implement the game with some adjustments to accommodate the needs of their tools, I decided to focus on Unity simply because I have some prior experience in using it.

2.1 Agent Movement

The humanoid characters that move around and fight are called agents. An agent that is controlled by the player is called a player agent (or `PlayerAgent`); and one that is controlled by the computer is called an AI Agent (or `AiAgent`).

We're going to have to design an `Agent` entity which allows the player and the computer to control an agent in the game. There is one important aspect that might prove rather challenging, and that is the movement of the agents. In general, Unity offers three solutions to consider when implementing the movement of any agent:

- Rigidbody
- CharacterController
- NavMeshAgent

These solutions are examined in the next sections.

2.1.1 Rigidbody

In order to let a `GameObject` [9] be affected by Unity's physics engine, it is necessary to attach a `Rigidbody` [10] and a `Collider` [11] `Component` [12] to it. A collider is the physical representation of the game object in the scene. The preferred shape of the collider is usually a capsule. This is so that the object doesn't get struck onto the edges and corners of scene geometry while moving (e.g.: when climbing a staircase), which is something that can happen when using a rectangular collider for movement. Once the components are set, the various methods provided by the `Rigidbody` can be used to move the game object in the scene.

In general, it is important for the game engines to update the physics simulation at constant time intervals. This is in order to preserve the correctness of the simulation. Therefore, it makes sense to use Unity's `FixedUpdate` method to apply the movements on the `Rigidbody`. However, in the case of the player agent, reading the inputs in `FixedUpdate` would cause input lag [13]. Therefore, it's a good idea to read the inputs in `Update`, and apply the effects in `FixedUpdate`.

I believe that moving the player agent in the scene using this approach is reasonable. The only issue I can think of is related to camera jitter (i.e.: blurriness when the camera is moved). Usually, it makes sense to move the camera in `LateUpdate`, which is invoked after the `Update` method is used to receive inputs. However, since the `Rigidbody` is moved in `FixedUpdate`, the player agent might appear blurry to the user whenever the camera is moved. It's possible to fix this by moving the camera in `FixedUpdate` as well. The problem is, if the other agents in the scene are moved in a different method (e.g.: `Update`), then they will appear blurry when the camera is moved. If these issues occur, it might be a good idea to use Unity's `CharacterController` component instead (which is explained in a later section).

Moving the AI agents using `Rigidbody` is trickier, because they need to be able to avoid obstacles in the scene. It's possible to write a pathfinding algorithm [14], but I believe that would require too much work to get it right, especially since Unity already offers a solution: `NavMeshAgent`. It is explained in a later section.

2.1.2 CharacterController

Unity's `CharacterController` component is another way to move an agent in the scene. As Unity's documentation page says, it allows "movement constrained by collisions without having to deal with a rigidbody." [15]. It is still affected by collisions, but the upside is that we can implement the movement in `Update` instead of `FixedUpdate`. This could fix any camera jittering issues mentioned in section 2.1.1, should they actually occur.

Unlike movement using `Rigidbody`, it is not necessary to attach a collider to a `CharacterController GameObject`, as the component already comes with its own collider.

If the `Rigidbody` approach for movement causes problems, it's reasonable to use the `CharacterController` to implement the player agent's movement. However, it is still tricky to use this for AI agents, as we're still lacking a pathfinding solution for them in this case.

2.1.3 NavMeshAgent

Unity's `NavMeshAgent` (`NMA`) component [16] is a movement solution which uses Unity's navigation system [17]. It allows game objects to move around in the game scene with obstacle avoidance. This is achieved by baking the `NavMesh` [18] of a scene using the tools in `Unity Editor`. The navigation system then uses this `NavMesh` to determine which parts of the scene are moveable or not.

The `NMA`s do not get affected by physics colliders. Instead, we're meant to use a `NavMeshObstacle` component [19] if we want to create obstacles for the agents to avoid. The `NMA` can be told where to go in the `Update` method with its `SetDestination` method.

In the case of AI agents' movement, the other methods require us to come up with a custom pathfinding solution; whereas the `NMA` already solved this problem. Therefore, I believe implementing the movement of the AI agents using this tool is quite reasonable.

The `NMA` component can also be used to implement the movement of the player agent, but I believe there would be some issues. The first problem is implied by the `SetDestination` method, which accepts a position vector as an argument. This type of movement is usually best suited for `Real-time Strategy` games [20], where the user decides where to go by choosing a position in the game world. Secondly, it is ultimately up to Unity's pathfinding algorithm to move the player, which may not be as precise as the user wishes. It is possible to disable the pathfinding for the player agent, but that would make it pointless to use a `NMA` in the first place. For these reasons, I believe the movement of the player agent should be implemented with either a `Rigidbody` or the `CharacterController`.

2.2 Animation

In order to animate the agents, there are a few options:

- Unity's Mecanim.
- Using C# script to animate the agents with Unity's animation engine.
- A third party asset such as Animancer.

The animation system offered by Unity is called `Mecanim` [21]. It comes with a graphical user interface to create a state machine for the animations to be played as needed. This seems like a reasonable approach to set up the combat animation logic of the game. Mecanim also comes with a `Blend Tree` tool, which allows one to blend multiple animations [22]. It seems like an easy way to set up the movement animations of our game, in a way that is also visually pleasing to the eye.

The second option is to write a custom state machine using C# scripting and Unity's animation engine. Doing so would most likely offer the best control to animate the agents compared to the other alternatives. Though, the time it takes to implement and debug such a system from scratch may not be worth it, as Unity's Mecanim seems like it already offers a good solution for animations.

Another option is to use a third party asset that is made by other users of the Unity engine. An example of this is `Animancer` [23], which seems to have received a good rating in the `Asset Store` [24], but requires the user to pay a fee to use the full features. These might act as a good alternative if Unity's Mecanim proves to be quite unusable.

Since Unity's Mecanim seems like a good solution to set up the animations, I plan to use it in my game, as I also had some experience using it in the past.

2.2.1 Four Directional Combat

There are four directions in which every agent can attack or defend: up, down, left, right. For example, if an attacker performs a swing from the left, the defender must look at the attacker and block from the left side in order to successfully deflect the attack. I believe that being able to choose the correct directions in the heat of the battle adds depth and challenge to the combat of the game.

Using a mouse and keyboard as the input mechanism is probably ideal for this type of combat system. Because the left and right mouse buttons can be

used to send attack and defend signals respectively; while the combat directions can be chosen by moving the mouse in the appropriate direction. The player's movement signals can be sent by pressing the "WASD" buttons on the keyboard, which is a common practice in many video games. It's possible to use other input methods as well. For example, if it had to be implemented using a controller, the user could use the right analog stick to control the camera and choose a combat direction, and press the necessary buttons to perform the attack or defense. The movement would be implemented by moving the left analog stick to accelerate in the appropriate direction. However, for the purposes of this game, I believe the main input method should be mouse and keyboard.

I know of two approaches to implement such a combat system, which are explained in the next sections.

2.2.2 Keyframe Animation

The first approach is by using keyframe animations [25]. The idea is to save the pose of the skeleton's bones in certain key poses (called keyframes), and then let the underlying software (in this case, Unity) interpolate between the keyframes to generate a frame. These animations are then played when necessary. For example, if we want the agent to attack from the right, we play the "attack_right" animation when the user moves the mouse to the right and presses the attack button.

This is probably the most common way to animate characters in most video games. From the looks of it, other medieval combat games such as M&B:W and Mordhau also use keyframe animations to animate their characters.

In my case, I plan to use the models and animations which I made in 2016 using `Blender` [26]. The animations are also keyframe based, and they are visually similar to M&B:W, as I was inspired by that game at the time. If I didn't have these assets available at hand, I would have looked at Unity's Asset Store to use some third party assets, as it's too time consuming to create such assets from scratch in general. In that case, the third party assets would probably not have looked as visually consistent, as it's hard to find game-specific assets made by other people, but still would have got the job done nonetheless.

2.2.3 Physics Driven Combat

The second approach is by driving the animations purely by a physics system. In this case, no keyframe animations are used, and the skeleton of the agent is controlled by applying physical forces. As far as I know, the only game that does is `Exanima`, which is a game developed by `Bare Mettle Entertainment` [27]. In Exanima, the characters can attack in four directions. Since the game is driven by physics, it's possible to do even more than that. They develop the game using their in-house game engine, which most likely helps them optimize the performance caused by physics related calculations.

For the purposes of our game, it would be too much work to write a physics based combat system from scratch, as Unity is more of a general purpose tool, and its built-in physics engine may not be efficient enough to handle such calculations. Moreover, the keyframe animation approach is enough to cover the needs for our

game, as an ultra realistic combat simulation is not the main point of it. For these reasons, I believe it's reasonable to use the keyframe animation approach instead.

2.2.4 Combat Coherency

As mentioned in Goal 2, the four directional combat should be done in a coherent manner. This point is elaborated further with another example. Consider a scenario with an attacker and a defender, and they're both facing each other. If the attacker is attacking from his right side, the defender has to defend from his left side in order to block the attack. This is the expected combat dynamic.

Now, consider the case where the attacker is situated on the right side of the defender. In this case, if the attacker attacks from his right side, then the defender should not defend from his left side as explained above. This is because the attacker is on the right side of the defender, which means that the defender should defend from his right side instead. In fact, if the attacker decides to attack from his right side in this situation, the defender should not be able to block at all, as the attacker is too much on the right side of the defender. The correct course of action is for the defender to look at the attacker and center his view on him.

Achieving this degree of coherency can be done by comparing the positions and look directions of the combatants. Once the attacker's relative position to the defender is determined, it can be decided whether or not the defender should be able to block an attack.

2.2.5 Separating the Upper and Lower Bodies

In order to implement the features described in Goal 1, it is necessary to separate the control of the upper and lower body of the skeleton. I had some experience doing this in 2016 on a prototype I made for a similar combat system. There were some issues that I expect to see again as I am making this game, which are explained below.

In Unity, a child game object will follow its parent game object. That is, the child will maintain its position and rotation relative to its parent. If the parent object rotates, the child object will also rotate (in world space) to maintain its relative rotation.

The agents use a skeleton rig which I made in Blender. This skeleton rig is made out of "bones", which are just transform objects in a hierarchical order. Therefore, for example, if the shoulder bone is rotated, the arm bone and its children will also be rotated.

When I first rigged the skeleton in 2016, the spine bone was a child of the pelvis bone in the skeleton hierarchy. This meant that, in Unity, the pelvis bone had full control of the spine bone, which made it impossible to separate the control of the upper and lower bodies.

I needed the spine bone to inherit the position of the pelvis bone, while disregarding the rotation of the pelvis bone, despite being the child of the pelvis bone in the skeleton hierarchy. Therefore, I had made it so that the spine bone was no longer a child of the pelvis bone. This allowed me to separate the

control of upper and lower bodies, and introduced a new problem. When the agent was moving and attacking at the same time, his waist would get stretched. This is because the spine bone is no longer attached to the pelvis.

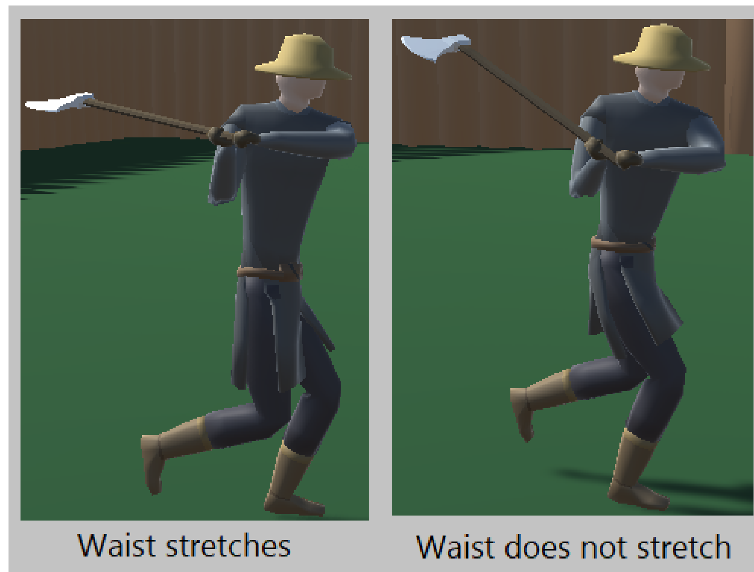

Figure 2.1: On the left, the waist stretches downwards due to the pelvis not being connected to the spine via code. On the right, the pelvis is connected to the spine bone, and the problem is solved. This picture was taken after the game was developed.

In order to fix this new problem, I wrote a C# code which manually connects the spine bone to the pelvis bone. This is done as a post processing effect after Unity plays the animations, so the code is invoked from a `LateUpdate` method. The idea is the following: When an agent is instantiated on the scene, save the initial spine and pelvis bone positions and rotations. Then, after animations are played every frame, connect the spine to pelvis using the initial information.

Since the upper and lower bodies are separated, it is possible to attack while moving. In addition, the agents are able to look up/down while attacking to target different body parts of their opponents. This solution worked really well, so I plan to use it again as I am making this game.

2.2.6 Making Movement and Look Directions Different

In some older games, it is usually the case that the characters can only move in the direction they're looking at. This is especially seen in computer controlled characters. When they decide to move in a different direction, they must do a full body turn, which doesn't look visually appealing to the eye.

In my game, I plan to make it so that the agents can move in a direction that is different from their look direction. Back in 2016, I had made my walking and running animations specifically for this purpose. There are 8 variants of each walking and running animations, which cover all of the cardinal and intermediate directions. The plan is to plug these animations into a blend tree, and control the blending via the user defined parameters in Mecanim.

The player's movement direction is acquired from keyboard input which covers the 8 directions mentioned above. The mouse controls the camera, which then controls the look direction of the player agent. Hence, the player agent always looks in the direction of the camera. However, if the player decides to run to the side while looking ahead, the correct animation in the blend tree is played based on the keyboard input. This separates the look and movement directions for the player agent.

The AI agent's velocity is driven by Unity's `NavMeshAgent`. The AI agent's look direction is based on the position of the enemy agent. By transforming the velocity vector to the AI agent's local space, and feeding this information to the blend tree in Mecanim, it is possible to make it so that the look and movement directions of the AI agents appear separate. This allows them to change movement directions without having to do full body turns.

2.3 Items

The items that the agents equip consist of a single weapon, and armor pieces that protect specific parts of the body. The agents are to be instantiated in the game with their selected items, and they won't be able to change them until the next battle.

2.3.1 Weapon and Armor Values

One of the key aspects of the game is how much raw damage the weapons deal, how armor reduces the incoming damage, and what the underlying mechanic is for determining the final damage output when an agent is struck with a weapon.

The first approach is to come up with a system like M&B:W, where each weapon can deal different damage types (e.g.: slash, pierce, blunt). These are then paired with a number which determines the potency of the damage type. In this case, the armor would also be associated with a defense value which provides damage reduction when the agent is struck. In most games, it's usually the case that slashing damage is effective against lightly armored enemies but falls off as the armor value increases. In the case of heavily armored enemies, the piercing or blunt damage types work better. The upside is that the formula which determines the final damage output won't be easily predictable, and it may add a feeling of depth to the game. The downside is that getting the formula to feel right requires a lot of fine tuning.

The second approach is one that is similar to Mordhau's system. In this case, the armor pieces are given non-numeric defense levels such as light, medium and heavy. Then, the weapons are given specific damage values for each armor level. For example, a sword could deal 100 damage to a light armor piece, while only 30 against heavy armor. The upside of this approach is that the fine tuning of the damage values is easy, as you just write out how much damage is dealt against each armor level. The downside is that the final damage output is relatively easy to predict, and may detract from the game's sense of depth.

2.3.2 Weapon Collision

Agents use weapons to attack other agents. The weapon needs to be able to detect the defender agent's hitboxes, which can be done by using Unity's physics system to detect collisions. Regarding the implementation, two approaches come to mind.

The first approach is what I call the "cone method". The idea is, whenever an agent attacks, an invisible Unity collider is instantiated in front of the attacker agent. The defender agent is deemed to be struck if the collider makes contact with it. A cone shape for the collider would probably work better for this, as it can be pointed out towards the enemy. There might be some issues. Firstly, it might be difficult to reflect the visual length of the weapon using an invisible collider. Secondly, since the weapon itself wouldn't actually be making contact with anything, the user may not get a feeling of impact when agents are struck. For these reasons, perhaps this approach is better suited for a game with simpler combat mechanics.

The second approach is to attach a collider to the weapon itself. This allows the agents to manipulate their attacks to target specific body parts of their opponent. The collider's dimensions can easily be adjusted according to the visuals of the weapon. There might be some issues if the agent attacks at a very high speed. In that case, the weapon might end up being swung too quickly for the physics engine to detect any collisions. Therefore, the attack speed values of the weapons should be carefully considered.

2.3.3 Weapon Specific Animations

The game features different classes of weapons, such as `Two Handed` weapons and `Polearms`. It doesn't make sense to wield and use a polearm weapon like a spear as if it was a big two handed sword. Since I am planning to use keyframe animations in Unity's `Animator Controller` [28] (i.e.: Mecanim) to implement the combat of the game, I need a way to use different animations for each weapon class. Luckily, Unity already offers a solution for this, which is called the `Animator Override Controller` [29]. It allows swapping the animations in an Animator Controller without changing the original structure of the state machine. It seems like a good way to solve this problem, so I plan to use it when developing the game.

2.3.4 Wearing Armor

In order to add depth to the game's combat, there are several armor pieces to protect each body part such as head, torso, and legs. The armor models are mesh objects which are rigged with the same skeleton as the human model. Unity refers to such objects as `Skinned Mesh Renderers` (`SMR`) [30]. In order to make the agents wear an armor piece, the bones of the armor `SMR` has to be set equal to the bones of the human skeleton. This is the planned approach to make the agents wear armor in the game.

2.3.5 Movement Speed Penalty

As stated in Goal 4, the damage protection provided by wearing armor must come with a penalty to the movement speed. Two approaches come to mind in order to implement this feature.

The first approach is similar to what M&B:W does, which is to assign a numerical weight value to each armor piece. Then, come up with a formula which reduces the movement speed as the total weight from all armor slots increases.

The second approach is similar to what Mordhau does, which assigns movement speed penalty values for each armor level (e.g.: light, medium and heavy). In this case, the formula would probably have to take into account the differences of armor slots. For example, a heavy torso armor should provide a higher movement speed penalty than a heavy head armor.

2.4 Menus

The player needs to interact with some menus before each fight. These menus are the following:

- Information Menu
- Item Shop Menu
- Mercenary Menu

The `Information Menu` mainly contains text which explains what the next battle is going to look like. For example, if the next battle contains a boss wave (i.e.: a more challenging enemy), then the player should be informed in this menu.

The `Item Shop Menu` should contain controls to inspect various weapons and armor. It is in this menu that the player should purchase such items in exchange for gold. Once purchased, the player spawns with the selected weapons and armor in the next battle.

The `Mercenary Menu` is where the player can hire mercenaries to fight the enemy hordes. The menu should contain controls to see the various mercenaries that can be hired. For example, some mercenaries could have lighter armor, while the others could be heavily armored. The player should be able to see their hire costs and pick a mercenary accordingly.

In order to implement the menus, two approaches come to mind, which are explained in the next sections.

2.4.1 All in One Scene

The first approach is to use a single scene for the entire game. In this case, all of the menus would be implemented on the combat scene. The menus would be made invisible during combat, and set to visible after the combat ends. The upside of this approach is that we wouldn't have to worry about managing the player's progress across multiple scenes, as there is only one scene. The downside is that it requires us to show and hide the necessary user interface (UI) widgets according to each situation, which would be very messy to do so in a single scene. This approach is perhaps better suited for a much simpler game.

2.4.2 Using Multiple Scenes

The second approach is to use multiple scenes for each situation. In this case, all of the menus would have their own scenes, and the combat area itself would be a separate scene as well. Larger games most likely use this approach, as it becomes easier to manage the UI widgets in each scene where they're needed. I believe this game is complex enough so I plan to use this approach in my game.

There is one downside of this approach which should be discussed. In Unity, when the game transitions from one scene to another, the contents of the previous scene are completely destroyed. In our case, it would mean that the player's progress would be lost during scene transition. For example, the information that the player bought some items in the Item Shop scene would be lost once the scene is destroyed. There are ways to tackle this problem, which are explained in the next section.

2.4.3 Managing Player Progress

In Unity, it is possible to mark game objects so that they're not destroyed during scene transition [31]. This would allow us to maintain player progression data (e.g.: items bought, mercenaries hired) across multiple scenes. It is necessary to do this in every scene so that the data is preserved constantly. This might cause us to accumulate a lot of "baggage" data for each scene, just because they are needed eventually. This may not be desirable, which is why I can think of two approaches to tackle this issue.

The first approach is to create a manager game object which systematically gathers and stores data generated from each scene. It would act as a central authority where each scene reports their portion of the player progression data to the manager object. This is most likely the preferred approach in larger games, as they probably have to track several data across multiple scenes (or similar structures). It's important that this system is carefully planned and debugged, as we wouldn't want some data to get lost during scene transition.

The second approach is to store the player data in the static memory. Since this section of the memory is available throughout the entire lifetime of an application, it is guaranteed that the data is preserved across multiple scenes. The static memory is smaller compared to heap memory, which may cause issues if the data becomes too large. In that case, the `Singleton` design pattern [32] can be used to have static references to data objects in heap memory. Another problem is that this approach is not well suited for a large game. Because the management of data can become harder to control as the amount of data increases. However, in the case of my game, there is only one player, and the only data that must be tracked across scenes is the player's progression. Therefore, it wouldn't be unreasonable to consider this approach for this game.

2.5 Combat Scene

The combat scene is where the actual fighting takes place among the agents. Unity's terrain tools [33] should be sufficient to generate a mostly flat surface with some small slopes around which the agents can walk. In addition, it can be

used to raise mountainous scenery in the background, as well as brush tools to paint the terrain with textures. The textures should probably be single colored in order to keep it consistent with the rest of the game's artstyle.

The obstacles should consist of boulders, trees and fences. They must be impassable, and the agents must walk around them in order to continue. I plan to use the scene props which I made in Blender back in 2016. Unity's terrain tool seems to offer a way to place 3D tree objects in a performance efficient manner, which should come in handy.

The scene must be constrained with invisible barriers which bars the agents from passing through. The player can be locked in this manner by surrounding the intended gameplay zone with large box colliders which block the player. In the case of the AI agents, their `NavMesh` needs to be baked in such a way that they can only traverse the intended fighting area, which should not be a problem using Unity's navigation tools.

The spawn locations of each team should be based on some transform game objects whose positions can be customized in Unity Editor. The C# script which manages the horde game mode can use the position of these objects to spawn the agents side by side on each team.

2.6 Horde Difficulty Progression

The difficulty of the game should increase as the game progresses. To achieve this, several approaches come to mind. Note that we are not forced to pick only one approach from below. It is possible to combine several approaches accordingly.

The first approach is to make the AI agents smarter in combat. In this case, they would be able to block the attacks more often, and they could prefer attack directions which are not defended by their enemies. They could also be programmed to work together and eliminate their opponents as a group. This approach heavily relies on programming and debugging the AI of the agents well.

The second approach is to improve the items and combat attributes of the AI agents on higher difficulties. For example, the agents could start wearing heavier armor, and use more lethal weapons. Moreover, their attributes such as maximum health, damage resistance, attack speed, damage etc. can be increased to make them more formidable.

The third option is to add a difficulty slider in a menu which determines the overall difficulty of the game. On lower difficulty settings, the player's team could take reduced damage and earn more gold; and vice versa as the difficulty increases.

2.7 Sounds

The sounds are to be implemented using the audio tools provided by Unity [34]. Some examples of the kinds of sound effects that should be in the game are:

- Metal/wood clanking
- Armor/flesh being struck

- Grunting
- Footsteps

There are two approaches to acquire these sound assets, as I do not have them readily available. The first approach is to record them myself, which requires a quiet environment and a decent quality microphone. While it might be easy to record the clanking sound of metal objects, it is harder to record footsteps sounds on grass due to background noise. Moreover, it might be necessary to use audio editing software to reduce unwanted sounds.

The second approach is to look for sound effects on Unity's Asset Store, preferably ones that are free for general use. It's unlikely to find a single source which provides all of the sound effects that are needed, so I might have to listen to sound effects from multiple asset bundles. As a result, I may end up with sound effects with a bit less than ideal harmony. Still, I believe that this approach would get the job done for the purposes of this game, which is why I plan to go for it.

As a final note, I do not intend to add any music to the game because I think playing music on loop would be too distracting for the user.

3. Implementation

In this chapter, I describe some of the technical aspects in the development of the game. When it comes to C# scripting, most of the programming details were already described within the code. One could create an auto generated documentation with a tool such as `Doxygen` [35] to see these details. For this reason, this section goes over the parts which I was not able to mention from within the code. It also covers some of the non code related details, as well as give a broad overview as to how different parts of the project are connected.

To make the game, Unity version 2021.3.6f1 was used. The models were made using Blender version 2.76b. Note that while Unity supports multiple platforms, the game was developed and deployed for `Windows` with mouse and keyboard as the input method. The approaches and techniques described in this text would work on other platforms as well, though there would have to be some adjustments to accommodate the needs of each platform.

3.1 Folder Structure

There are two topmost folders in the project. These are called “Assets” and “Packages”. I never used the `Packages` folder, as it’s mostly used by the Unity engine itself. This section focuses on the `Assets` folder.

The `Assets` folder contains every other folder, which are as follows:

- Animator Controllers
- Avatar Masks
- Imported Assets
- Materials
- Models
- Prefabs
- Scenes
- Scripts
- StreamingAssets
- TextMesh Pro

The “Animator Controllers” folder contains the `Animator Controllers` (`AC`). These are used to drive the animations of anything with an `Animator` component attached. Currently, the only objects which use `AC`s are the agents who can fight and move around the game scenes. It also contains `Animator Override Controller` (`AOC`), which can be used to swap animations in a given `AC`. Finally, it contains a text file that describes some of the details as well as numerical values about how the `AgentAC` is set up.

The “Avatar Masks” folder contains the `Avatar Masks` [36] which are used by the `AC`s. These are used to enable/disable the use of some bones in the skeleton when animating the agents.

The “Imported Assets” folder contains the imported assets which were downloaded from Unity’s Asset Store. Currently, the only third party assets that were used are related to sound effects. Note that for any given third party asset, the whole package was not downloaded. For the most part, only the assets that were used in the project can be found in these folders.

The “Materials” folder contains materials which are used to give color to the objects in the game. Most of the models were already imported with their own materials, so this folder doesn’t contain much.

The “Models” folder contains the weapon, armor, scene props, and the human agent model. This folder also contains the animation clips which are used by the agents. An important thing to note is that every imported model has its `Scale Factor` value set to 0.29. This is because I may have made the models a bit too big in Blender back in 2016, and this was a way of making them look a reasonable size. If you add new armor to the game, you’ll have to use the human’s skeleton for it to be animated, which means that you’ll have to set the scale to 0.29 as well. This is not strictly necessary for weapon models.

The “Prefabs” folder contains the prefabs which are used throughout the game. These can be weapons, armor, agents, friendly and enemy agent data, sound effects, and so on. This folder also contains several special folders with the name “Resources”. The Resources folder is used to instantiate game objects from within code, rather than dragging and dropping them onto the scene by hand in the editor. The name of this folder must be “Resources” because it is required by Unity. The prefabs which are instantiated by code are done so in the `PrefabManager.cs` script.

The “Scenes” folder contains the scenes of the game. The scenes which are actually used by the game are:

- MainMenuScene
- InformationMenuScene
- GearSelectionMenuScene
- ArenaScene

There is one more scene, which is called “DefaultScene”. It was the very first scene that was added by the `Unity Editor` when the game project was created for the first time. It can be duplicated to create a new clean scene if needed.

The “Scenes” folder also contains child folders with the same name as some of the aforementioned scenes. These folders contain the lighting and `NavMesh` data. The lighting data is used to bake lighting into a scene. The `NavMesh` data contains the `NavMesh` which was generated by Unity for the AI agents to navigate themselves around the scene.

The “Scripts” folder contains the C# scripts which were used to make the game. It also contains a file name `"DocumentationMainPage.md"`, which was used by Doxygen to create the main page of its auto generated documentation.

In this file, you can find a “Credits” section, which gives credit to the authors of the sound effects that were used in the game.

The “StreamingAssets” folder is a folder created by Unity [37], and I never used it myself. The “TextMesh Pro” folder contains the assets which are used by the `TextMesh Pro` tool, which is a user interface text solution in Unity [38].

3.2 Designing the Agent Entity

The `Agent` entity that we have to design has some common components which should be relatively straightforward to implement. An idea as to what these components are can be seen as follows:

- **EquipmentManager:** Responsible for spawning and managing the weapons and armor of an agent.
- **HitboxManager:** Manages the hitboxes of an agent so that weapons can detect them and inflict damage when an agent's hitbox is struck.
- **AnimationManager:** Determines when and which animations should be played for a given agent.

In section 2.1, we concluded that it'd make sense to use `NavMeshAgent` to implement the movement of the AI agents; and `Rigidbody` or `CharacterController` for the player agent. This affects the design of the `Agent` entity. This is because while the `Agent` entity has some common components, the implementation of the movement is what truly differs the player agent and the AI agent. I propose two approaches to achieve this goal:

- Agent and controllers
- A compound Agent entity

3.2.1 Agent and Controllers

In this approach, there are three key entities:

- Agent
- PlayerController
- AiController

In addition to the common components, the `Agent` contains an interface to move the agent. Then, the “controller” entities use this interface to control the movement of the agent.

For brevity, suppose that the player agent's movement is controlled by a `Rigidbody` instead of a `CharacterController`. The movement interface in the `Agent` entity contains both the `NavMeshAgent` (`NMA`) and `Rigidbody` components. It understands the details of how to move an `NMA` and a `Rigidbody`. Then, it defines its own methods which unifies the understanding of these components. For example, an `NMA` is moved with the `SetDestination`

method which accepts a position vector as an argument; whereas the `Rigidbody` can be moved using the `AddForce` method which accepts a direction vector as argument.

The controller entities then use the interface defined by the `Agent` entity to move a player or an AI agent. For example, the `AiController` can decide where to go in its `Update` method using some kind of finite state machine, and use the interface to move the agent afterwards.

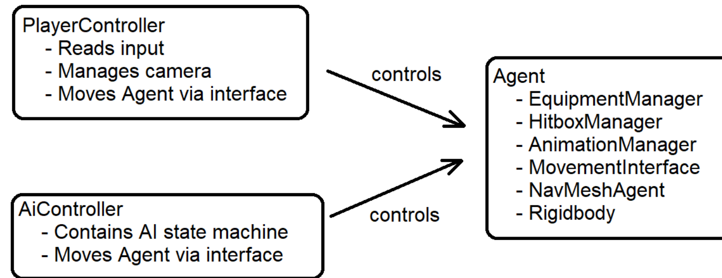

Figure 3.1: An Agent entity that is controlled by a PlayerController or an AiController entity.

The upside of this approach is that the `Agent` entity only contains the components which are necessary to control it. Meaning, it doesn't know anything about being a player or an AI agent, as it defines its own movement interface which must be used by the controller entities. The downside is that it needs to understand the details of how to move an `NMA` and `Rigidbody` component at the same time, which may cause some bloating in its code.

An alternative approach is explained in the next section.

3.2.2 Compound Agent Entity

There are three key entities in this approach:

- Agent
- PlayerAgent
- AiAgent

The `Agent` entity is an entity which contains the common components that every agent must have. The `AiAgent` and `PlayerAgent` entities are derived entities which inherit from the base `Agent` entity. These entities add the different parts required by each type of agent to become fully functional.

Again, for brevity, suppose that the player agent's movement is controlled by a `Rigidbody`. In this case, the `AiAgent` entity would contain a `NavMeshAgent` component, and the `PlayerAgent` would contain a `Rigidbody` component. Each entity would contain code that's necessary to operate whatever movement method they're using. For example, the `PlayerAgent` would read inputs in Update to control its `Rigidbody` in `FixedUpdate`; and the `AiAgent` would contain a finite state machine which determines the combat and movement decisions, and apply the movement in Update.

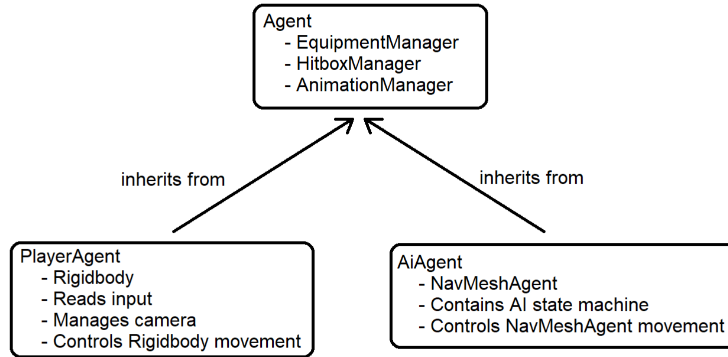

Figure 3.2: The inheritance diagram of an `Agent` entity and the derived `PlayerAgent` and `AiAgent` entities.

The upside is that the derived entities only know what they're responsible for. For example, the `AiAgent` doesn't need to know anything about a `Rigidbody`. The downside is that the base `Agent` entity simply acts as a compound entity and nothing more. Meaning, the code that is otherwise mostly similar may have to be duplicated in each entity. For example, even though the movement of the player and the AI agents have differences, the parts that might be similar could possibly be duplicated when writing the code in the derived entities.

3.3 Agents

Agents are the human characters that are controlled by the player and the computer. They are found in the `"/Assets/Prefabs"` directory. There are 3 types of agents:

- PlayerAgent
- AiAgent
- MannequinAgent

The `PlayerAgent`, `AiAgent` and `MannequinAgent` are controlled by the `PlayerAgent.cs`, `AiAgent.cs` and `MannequinAgent.cs` scripts respectively. All of the scripts inherit common data from the abstract base class `Agent.cs`. Every agent comes with the following components, which are C# scripts:

- **EquipmentManager.cs:** Manages the weapon and armor pieces of the agent.
- **AnimationManager.cs:** Drives the agent's animations by changing the states of the animator's parameters.
- **LimbManager.cs:** `Limbs` are the hitbox colliders which are detected by weapons to make combat related decisions. There are three `Limbs`: head, torso, legs. The initialization and the dimensions of these are managed by the this script component.

- **CharacteristicsManager.cs:** Acts as a way to group characteristics such as health, extra damage multiplier, etc.
- **AgentAudioManager.cs:** Manages the sound effects generated by an agent, such as footstep and grunting sounds.
- **NavMeshAgent:** This is a component provided by Unity. It is attached to every agent, no matter how they're controlled. This is mainly so that the AI agents recognize the presence of the player agent. Because the player agent uses `CharacterController` to move around, which is not detected by the `NavMeshAgent`.

The `Compound Agent Entity` approach mentioned in section 3.2.2 was used to implement the agents. This is because `PlayerAgent` is moved via Unity's `CharacterController`, and the `AiAgent` is moved by Unity's `NavMeshAgent`. Thus, it seemed reasonable to let each class handle the implementation details of each movement solution.

On top of the above, the `PlayerAgent` is also attached a `CameraManager` component. The camera movement must be done after the animation post processing effects in `LateUpdate`, which is why the `PlayerAgent` controls it. Still, the camera controls are contained compactly in the `CameraManager.cs` script.

The `AiAgent` is controlled by the code written in the `Update` callback. The fact that the upper and lower body of the agents are separated allows them to move while attacking, as well as the ability to move in a direction which is different from where they're looking at. The code in the `Update` callback tries to mimic a simple state machine that governs movement and combat separately, and the basics of which are as follows:

- **Movement:** If the `AiAgent` doesn't have an enemy, then it stops moving. If it has an enemy, then it asks if the enemy is close enough. If the enemy is too far, the agent moves towards the enemy. If the enemy is close enough, the agent stands still. Finally, if the enemy is too close, the agent moves backwards to maintain a certain distance, which depends on the weapon's length and the agent's size in the world.
- **Combat**: If the `AiAgent` doesn't have an enemy, or if the enemy is too far, then the agent does not attack. If the enemy is close enough, the agent attacks in random directions. If the agent has friends nearby, it prefers vertical (i.e.: up and down) combat directions to avoid having the weapon get stuck to the other agents. If the `AiAgent` gets struck, there's a random chance that it tries to defend itself for a short period of time, after which the attacking continues.

The `MannequinAgent` is used in the `GearSelectionMenuScene` to show the player how the selected weapon and armor pieces look. As its name suggests, it is merely a mannequin, and it has no other purpose.

3.3.1 Movement of PlayerAgent

The very first movement related implementation that I ever did was the movement of the `PlayerAgent`. To do this, I used Unity's physics system (i.e.: `Rigidbody`)

at first. It worked really well, until I realized some screen jitter when I moved the camera.

The camera follows the agent controlled by the player. The camera can still be rotated to look around by moving the mouse. So, whenever I rotated the camera while looking at other agents that were moving, I would notice some screen jitter. I thought I just had to synchronize the movement of the camera and the movement of the agents, so I moved the code responsible for camera control into `Update`, `LateUpdate` and `FixedUpdate`. None of the methods fixed the screen jitter problem.

At this point, I decided to re-implement the player agent's movement using Unity's `CharacterController`, since it doesn't make use of a `Rigidbody` [39]. The movement itself was good, similar to my `Rigidbody` implementation. However, the camera jitter problem didn't go away, because that problem was actually related to input smoothing.

The camera's movement is done by mouse input. At first, I was using raw input data, which caused the screen jitter. Later on, I started to smooth my input data using the `SmoothDamp` method [40], after which the camera jitter problem was gone.

In the end, when it comes to player movement, I didn't think there was much difference between using `Rigidbody` or `CharacterController`. Though the camera jitter problem was caused by lack of input smoothing, trying to solve this issue was the main reason why I switched from using `Rigidbody` to `CharacterController`, as I didn't know the cause at the time.

3.3.2 Movement of AiAgent

The movement of the `AiAgents` are done using Unity's `NavMeshAgent` (`NMA`) component. Unity's `NavMesh` system allows you to bake a `NavMesh` in your scene, which the `NMA`s can use to navigate around. I believe it's a fairly decent tool that is easy to use, but I'll mention a few problems I encountered.

Firstly, even though the `PlayerAgent` itself uses the `CharacterController` component to move around, I still had to attach a `NMA` component to it. This is because I needed the `AiAgents` to recognize the presence of the `PlayerAgent`, which can only be done by attaching this component to it.

The second problem has to do with the movement and animations. The `NMA` component moves the `AiAgent` around at a certain velocity. I use this velocity to animate the agent. In other words, depending on the magnitude of the velocity vector, the agent walks or runs. However, whenever the `NMA` reaches a given destination, it tends to stop too abruptly, which causes the movement animation to instantly go from walking/running to standing still. The `NMA` component has options so that it slows down smoothly, but none of those options were enough for me. In the end, I had to write some code which slows down the animation to make it less visually jarring. The idea is, even if the agent stops, you use the velocity data from the previous frame to smooth out the animation. I can't say the problem is completely solved, but I believe it looks better than before.

3.3.3 Jumping

Jumping was a feature which I wanted to implement for all agents. I wanted the agents to be able to kick (also not implemented), and jumping would have been one way to avoid getting kicked. I had to abandon both because of development time restrictions, but also due to a quirk regarding the `NavMeshAgent` (`NMA`) component.

The positions of the `AiAgents` are driven by the `NMA`. So, even if I made them jump, it wouldn't work, because the `NMA` has strict control of the agent's position, and it attaches the agent to the underlying `NavMesh`. One workaround is to disable the control of the `NMA` when jumping [41]. This didn't seem like an elegant solution to me, so I just decided not to implement jumping at all for `AiAgents`.

The jumping feature is still present on the `PlayerAgent`, despite having attached an `NMA` component. It works because the `NMA` is permanently disabled on the player agent, and it is only there so that the other `NMA`s recognize the position of the player. I didn't remove this feature, as I didn't want to risk breaking anything, and there's no harm in keeping it.

3.4 Animation

Unity's Mecanim was used to implement the combat and movement animations of the agents. In order to separate the upper and lower bodies, the approach explained in section 2.2.5 was done. The expected problems were addressed using the solutions described in the same section. However, some additional and unexpected problems occurred during development, which are detailed in the next sections.

3.4.1 Source State Until Fully Transitioned Out

In Unity's Mecanim, there are states and transitions. The states contain the animations to be played, while transitions are used to transition from a source state to a target state. When leaving a source state and entering a target state, the transitions are there to make it so that this process is visually smooth. A transition occurs if a set of conditions are satisfied, which are defined in Mecanim by the developer.

As the title says, Mecanim considers the control flow to be in the source state until it has been fully transitioned out from the source state to the target state. This caused me two problems, and I explain one of them now.

For example, when an agent is performing an overhead swing, the "atk_u p_release" animation is played. When the animation is complete, the control switches from "atk_up_release" to "idle", using a transition.

While the "atk_up_release" animation is played, the boolean `IsAttackingFromUp` is true. When the animation is over, the control transitions to the idle state. However, during the transition, the `IsAttackingFromUp` boolean is still true, despite expecting it to be false. The reason for this is explained in the second paragraph above.

This is a problem because I do not want to be considered attacking while I'm transitioning to the idle state. The only solution that I was able to come up with is to specifically check every single one of these transitions, in every single scenario. Meaning, similar to attacking, if I stop defending and start transitioning to the idle state, I don't want to be considered "defending" during the transition itself.

I became aware of this problem much later into the project, which is why the code regarding the animations are kind of bloated. It's because I have to make sure every transition is checked specifically in order to avoid unintended behavior (and there are a lot of transitions in this project).

3.4.2 AnyState Does not Apply to Transitions

`AnyState` is a special state defined in Mecanim. It exists for the situation where you want to go to a specific state regardless of which state you are currently in. As Unity's documentation page says, "this is a shorthand way of adding the same outward transition to all states in your machine." [42].

There are two obvious cases where `AnyState` would be useful:

- The agent gets hurt.
- The agent dies.

In both of these cases, we'd like to play an animation. However, here is the problem. `AnyState` is meant for states, and transitions are not states. They're transitions. Therefore, for example, if `Agent A` is in a transition while being struck by `Agent B`, then the AnyState conditions are simply ignored, and `Agent A` does not play the "getting_hurt" animation as intended. This is because transitions are uninterruptible by default.

I could not come up with a satisfying enough solution. At first, I tried to put the `getting_hurt` animations in a different layer in Mecanim, and play with the weights of the layers whenever an agent gets hurt. The idea is to smoothly lower the weight of the "combat animations layer" to 0%; while increasing the weight of the newly added "getting hurt layer" to 100%. The end result was not visually acceptable to me, but there was another problem. Since the control flow in the combat animation layer was still doing whatever it was doing, the fact its weight was temporarily reduced didn't matter, as it just picked up where it was left as soon as the layer's weight was increased.

I solved this problem by going to every single transition in Mecanim, and setting their `Interruption Source: None` property to `Interruption Source: Current State`. This made it so that all the transitions are now interruptible, which allowed the agents to transition from `AnyState` to the `getting_hurt` animations. However, the fact that the transitions are interruptible introduced yet a new problem.

Recall from the previous section that Mecanim considers the control to be in the source state until it has been fully transitioned out. Now, since the transitions are interruptible, the transition can actually be changed until one of them ends up being lucky enough to "escape". This depends on the order of the transitions listed on a given state, which is because the `Ordered Interruption` property is

checked [43]. Unchecking it would have made things less predictable, which is why I left it checked.

I'll explain with an example. Suppose you have an agent in the idle state, and you want to start attacking. The transitions are listed from top to bottom as follows:

- atk_up
- atk_right
- atk_down
- atk_left

Note that when interrupting a transition from a source state, Mecanim prioritizes the transitions in a descending order. That is, `atk_up` has the highest priority; while `atk_left` has the lowest priority. This means that an `atk_left` transition can be interrupted by anything above it; whereas an `atk_up` transition cannot be interrupted by anything below it.

Now, let's imagine that you wanted to attack from the right, so you move your mouse to the right and do a left click on the mouse. The agent will begin the transition from idle to `atk_right`. However, before the transition was completed, let's assume that you moved your mouse forward and did another left click. This will interrupt the existing transition, and the agent will begin to start attacking from up instead.

Normally, the intended behavior is to fully commit to an attack direction once it was chosen. However, since transitions can now be interrupted, you can start from the bottom-most transition and keep interrupting it until one of them lasts long enough to reach its targeted state.

Since every transition was made interruptible, the behavior described above may be experienced in other circumstances, and cause a "janky" feeling while playing. Therefore, this is not really a solution, but merely a workaround with its own downside. However, the alternative is to have the agents ignore being struck, and continue what they were doing without flinching. That sounds worse to me, so I decided to keep this workaround.

3.4.3 Transition Duration Cannot be Changed at Runtime

Later in development, once I set up the animations, I decided to make it so that some weapons are faster than others. For example, I wanted to make swords attack faster than axes. Since my animation state machine in Mecanim makes heavy use of transitions, the way to achieve this goal is to change the transition duration of attack related transitions at runtime, based on the properties of the weapon.

This is simply impossible, as the title suggests. Unity does not allow changing the transition durations in Mecanim at runtime. Apparently, the reason is "simply because it was not requested by our client.", according to this Unity forum thread [44]].

One idea is to include the `UnityEditor` namespace, which allows the transition durations to be altered via code, as can be seen discussed in this forum thread [45]. However, the problem is that the `UnityEditor` namespace is not included in a final build [46], which is why this approach would not work.

Since the speed of animation states (not transitions) can be modified at runtime with the use of animator parameters, another idea is to use handcrafted animations in place of transitions. However, it's not feasible to create enough animations for every possible transition while still maintaining the visual coherency of the movements. This is because transitions can be interrupted at any moment, so one would require infinitely many animations, which is impossible.

For these reasons, I had to settle with using hardcoded transition duration values for every transition. This means that every weapon will have the same attack speed, no matter what.

3.4.4 Waist Warping

One of the consequences of separating the control of the upper and lower body of the agents is what I call "waist warping". In extreme situations such as the one which can be seen in figure 3.3, the agent's waist is warped.

Figure 3.3: In the picture to the left, the agent is moving right while attacking from the left. In the picture to the right, the agent is moving left while attacking from the right.

One solution I can think of is the following. Use the blend tree as usual, but create some states which are meant to be transitioned to whenever there is an extreme situation which would cause the agent's waist to be warped. When the extreme situation is no longer the case, the control can switch back to using the blend tree.

I had anticipated this problem in 2016, which is why I had created four animations:

- run_e_backwards
- run_w_backwards

- walk_e_backwards
- walk_w_backwards

For example, if the agent decides to attack from the right while running to the left, this could be recognized in the code, and let the animator know by changing a parameter. In this case, the control would switch to `run_w_backwards` until the extreme conditions are gone, in which case the control would return back to using the blend tree as usual.

I did not have a chance to try this approach due to time restrictions, but I think it would work. However, note that I only have these four animations readily available, so even this solution may not be sufficient in some cases, which is why it might be necessary to create more animations for other extreme cases.

3.4.5 Arm Going Backwards

The animations were done using keyframe animations. Due to the way the keyframe animations are interpolated, there could be some issues. For example, when interpolating the rotation of a bone between keyframes, it is often seen that the software will prefer the shortest path available. This can sometimes cause issues. For instance, if the rotations of a bone are not set properly between two keyframes, then you could have a character whose arms twist in an unnatural way, resulting in unrealistic visuals. One possible solution to such a problem is to add yet a third intermediate keyframe that explicitly points out which path the software should prefer when interpolating between the starting and ending keyframes.

However, there are cases where you cannot simply add more keyframes. An example of this would be the transitions in Mecanim. When a transition occurs from a source state to a target state, I imagine that Mecanim uses a similar interpolation method as the one I described above.

I encountered an issue like this in the following case: As can be seen from the picture, when transitioning from `idle_pole` to `atk_polearm_up`, the left arm of the agent would go from his back instead of the front. To my knowledge, I cannot simply add keyframes on a transition. Firstly, the animation was made in Blender, so Unity just launched Blender whenever I tried to edit the animation from within Unity. Secondly, since the agent could decide to make this transition in several different timings, I don't even believe it makes sense to try something like this.

I solved the problem with a very simple solution. I went into the `idle_pole` animation, and changed the rotation of the left arm a little bit. As figure 3.4 shows, the change was enough to make it so that the interpolation causes the left arm to go from the agent's chest, and not behind his back.

3.5 AgentAC

The movement and combat animations of every agent are set up in Unity's animator state machine called Mecanim. Every agent has an `Animator` component named attached. The Animator component is driven by an

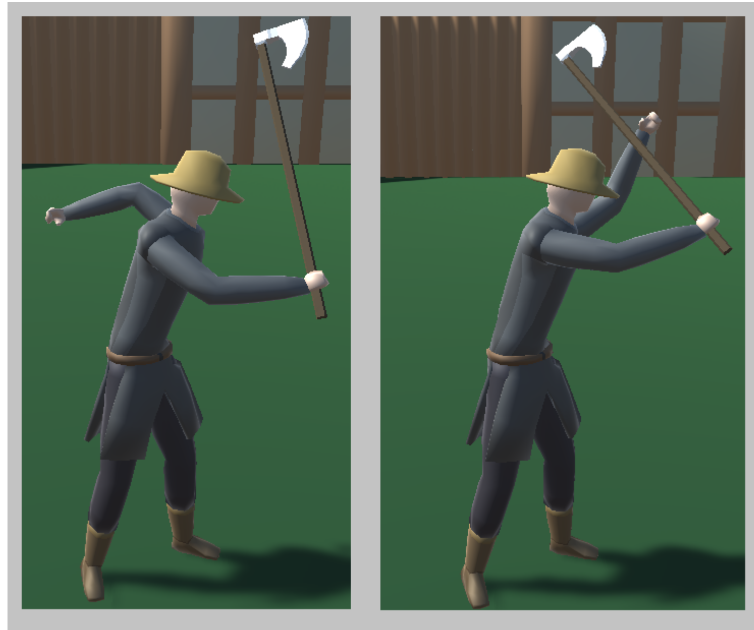

Figure 3.4: In the left picture, the left arm of the agent goes from the back. In the right picture, both arms go from the front as expected, thanks to the solution.

`Animator Controller` (`AC`). The name of the `AC` which the agents use is called `AgentAC`. This section briefly explains what `AgentAC` looks like.

There are three animation related layers used in `AgentAC`:

- Base Layer
- AtkAndDef
- Idle

3.5.1 Base Layer

The name of this layer is “Base Layer”. I will not be calling it “Base Layer layer”. Instead, I’ll just write `Base Layer`.

The `Base Layer` contains the blend tree which is responsible for the walking and running animations. It also contains the death animation, as well as the jump related animations. Note that the jumping feature was abandoned due reasons explained in section 3.3.3, but the player agent can still jump.

The blend tree in the base layer is controlled by two animator parameters: `moveX` and `moveY`. The parameters are controlled by the `PlayerAgent.cs` for player’s movement animations. The `AiAgent.cs` uses the velocity vector obtained from its `NavMeshAgent` component to modify the two parameters.

The base layer does not use an `Avatar Mask`. This means every bone of the agent’s skeleton will be affected by any animation being played on this layer.

3.5.2 AtkAndDef Layer

The `AtkAndDef` layer contains the combat related animations. It uses the `Avatar Mask` named `AttackAndBlockMask`. This mask marks the spine bone and all of

its children to be used by this layer. The unchosen bones in the avatar mask are unaffected.

The `AtkAndDef` layer controls the upper body of an agent; while the lower body is controlled by the `Base Layer`. This is done implicitly. To elaborate, the `Base Layer` is listed higher above the `AtkAndDef` layer, which gives it a higher priority over the control of the bones [47]. It is also a layer layer without an avatar mask, which means that it affects all bones. The `AtkAndDef` layer is below the `Base Layer`, but it uses an avatar mask which gives it control on the upper body bones. Combining these, we get the conclusion that the `Base Layer` controls the lower body whereas the `AtkAndDef` layer controls the upper body.

The combat logic in this layer is explained in the section 3.5.6.

3.5.3 Idle Layer

The final layer is called `Idle`. It uses the avatar mask named `IdleMask`. This mask controls the left and right shoulder bones, and their children. This layer is used to control the idle animation of the arms while the agent is not attacking.

3.5.4 How the Layers Work Together

When an agent is not attacking, the `Base Layer` and the Idle layer work together. At first, the movement animation in the `Base Layer` controls every bone in the agent's skeleton. After that, the idle animation in the `Idle` layer kicks in. The `Idle` layer only controls the left and right shoulder bones (and their children). The final result is an agent which walks/runs while idling with the correct idle animation. This is important. Without the `Idle` layer, the walking and running animations by default makes the skeleton run and walk normally, waving his arms around. Because that's what the movement animation clips look like. When the `Idle` layer kicks in, the agent doesn't wave his arms around. Instead, he holds his weapons in his hands.

The above is what happens when an agent is not attacking. This is achieved by controlling the weights of the layers in `AnimationManager.cs`. The `Base Layer` doesn't use an avatar mask, so it doesn't have a weight. However, the Idle layer has an avatar mask, which is why its layer weight is set to 100% while the agent is not attacking.

When the agent is attacking, the `Base Layer` and the `AtkAndDef` layer work together. As explained above, the `Base Layer` controls every bone in the agent's skeleton. Then, the animations in the `AtkAndDef` layer kick in. The bones which are affected are the spine bone and its children. This makes it so that the agent can attack while moving in any direction. This is achieved by setting the weight of the `AtkAndDef` layer to 100%.

The weights of the `Idle` layer and the `AtkAndDef` layer are changed whenever an agent starts/stops attacking. For example, let's assume that the agent is running left without attacking. The `Base Layer` controls the movement, and the `Idle` layer makes sure that the arms are holding the weapon correctly while facing the direction of the movement. When the agent attacks, the weight of the `Idle` layer is smoothly brought down to 0%; while the weight of the `AtkAndDef` layer is brought up to 100%. This ensures that the upper body plays the combat

animation while the lower body continues to run to the left as usual.

3.5.5 AgentAOC

The `AgentAC` and `AgentAOC` are different things. `AgentAOC` is an `Animator Override Controller` (`AOC`). It allows swapping some animations in a given `Animator Controller` (`AC`).

The combat animations were excruciatingly set up once in the `AtkAndDef` layer. The animations that were used are `Two Handed` weapon animations. They resemble a person holding a large weapon with two hands, and performing combat moves. However, the game also includes `Polearm` weapons. Polearms are melee weapons such as spears and long axes. The animations for these are different, and it's necessary to use the correct animations for the appropriate weapon types.

Unity's `AOC` comes to the rescue. It allows picking an `AC` (in this case, `AgentAC`), and cherry picking which animations need to be replaced using an `AOC`. This saves us from having to duplicate the logic of the animation states, just because the animations are different.

The information regarding an agent's weapon type can be found in the `EquipmentManager.cs` script. When a weapon is equipped, it informs the `AnimationManager.cs` script, and the appropriate animations are used with the help of `AgentAOC`.

3.5.6 Combat Animations Logic

Just Blade uses a four directional combat system, where the agents can attack and defend in four directions: up, right, down, left. The "down" attack is simply a thrust, while the other attacks have a swinging motion. The agents can perform these actions while moving or standing still. These are the core aspects which were heavily inspired by M&B:W's combat.

The player can attack using the left button of the mouse (i.e. "left click"); and defend using the right button of the mouse (i.e. "right click"). The direction is chosen via mouse movement. For example, if the player moves the mouse to the right and does a left click, then the agent in the game will attack from the right. Doing a right click instead will cause the agent to defend from the right.

The logic of the combat animations are placed in the `AtkAndDef` layer in `AgentAC`. The names of the animations look like "atk_2h_up_hold" or "def_2h_up_hold". This is because, by default I used the `Two Handed` weapon animations. The "up" direction is one of four, as described above.

In this section, I outline the basic logic of how the attack and defend animations are set up in `AgentAC`. The combat is done with the use of states and transitions in Mecanim. The weapon's name and the combat's direction will be omitted from the explanations. For example, I will write "atk_hold" instead of saying "atk_2h_up_hold".

Attacking occurs over several stages, which are explained below:

- **Windup:** In this stage, the agent is preparing to attack from a chosen direction. This is represented by a transition from a source state to an `atk_hold` state in `AgentAC`.

- **Hold:** Once the windup stage is over, the agent is allowed to wait indefinitely before releasing the attack. This is done by making the `atk_hold` animation wait indefinitely (i.e., by not letting it loop).

- **Release:** This is the stage in which the attack is actually performed. The agents are able to hurt other agents in this state. This is represented by an `atk_release` state in `AgentAC`.

- **Recovery:** Occurs right after the release state. When the release state is over, the agent smoothly returns to the idle state thanks to the recovery state.

The attack recovery stage itself can come in two forms:

- **Release recovery:** This is when the released attack is not interrupted, i.e. either the attack was missed or the opponent was unable to deflect it and got hurt. This is represented by a transition from the `atk_release` state to the `idle` state.

- **Rebound recovery:** This is when the released attack comes in contact with an object in the scene or when the opponent successfully deflects the attack. This is represented by a short transition from the `atk_release` state to the `atk_bounce` state, followed by having to fully play the `atk_bounce` state itself. After the `atk_bounce` state is played, the control transitions to the `idle` state.

Similarly, defending also takes a few stages:

- **Windup:** In this stage, the agent is preparing to defend in a chosen direction, and is unable to deflect attacks, even if the direction is correct. This is represented by a transition from a source state to the `def_hold` state in `AgentAC`.

- **Hold:** Once the windup stage is over, the agent is allowed to wait indefinitely to deflect any attacks from the appropriate direction. This is done by allowing the `def_hold` animation to hold indefinitely (i.e., by not letting it loop).

- **Blocking:** This occurs when an attack was successfully deflected. In this state, the agent is still allowed to defend against attacks from the appropriate direction. This corresponds to the `def_blocked` states in `AgentAC`.

- **Recovery:** This is the state where the agent stops defending and returns to the idle state.

The defending recovery stage itself can also come in two forms:

- **Hold recovery:** This is when the agent decides to stop blocking and return to the idle state after holding the defensive position for some time. It is represented by a transition from `def_hold` to the `idle` state.

- **Block recovery:** This stage occurs when the agent decides to stop blocking after having deflected an attack, and return to the idle state. This is represented by playing the `def_blocked` animation, followed by a transition to the `idle` state.

Both attacking and defending are usually performed from an idle state, i.e. the state in which the agent is not doing anything at all. However, there are exceptions. For example, an agent can transition to an `Defend Windup` immediately after an `Attack Recovery`, without having to transition to the `Idle` state at all. This was done so that defending is a bit easier compared to performing an attacking.

All of these can be performed while moving or standing still. Moving the agent does not prevent it from attacking or defending, since the upper and lower bodies are separated. In some games (such as Dark Souls [48]), the agents can either move or attack. Doing these together is not an option (with very minor exceptions).

The transitions between the above combat states are controlled by the following animator parameters: `combatDir`, `isAtk`, `isDef`, `isAtkBounced`, `isDefBounced`. The `isAtk` and `isDef` parameters become true when the agent wants to attack and block respectively. If an attack gets rebounded, `isAtkBounced` is set to true. If an agent successfully blocks an attack, `isDefBounced` is set to true.

The `combatDir` is an integer parameter used to specify the four combat directions, which are as follows:

- Up is 0 (zero)
- Right is 1
- Down is 2
- Left is 3

These combat directions and their corresponding integer values are consistent throughout the entire project.

The transition durations between the combat states were meticulously chosen and written down in the `AgentAC` for each transition. These numerical values will not be detailed here. For more information, refer to the text file named "AgentAC Mechanics Notes.txt" under `"/Assets/AnimatorControllers"`.

3.5.7 Blend Tree

The movement animations are done using a blend tree, which can be found in the `Base Layer`. This tool blends the walking and running animations of the agent seamlessly.

I learnt how to use this feature back in 2016, which is the reason why the movement animations were made specifically to be used in a blend tree.

There are walking and running animations for each cardinal and intermediate direction. The blend tree is controlled with two parameters: `moveX` and `moveY`. The idle animation is at the center, with (0, 0) coordinates. The walking animations are placed away from the center, using a distance of 0.5; and the running animations use a distance of 1.0 to be away from the center.

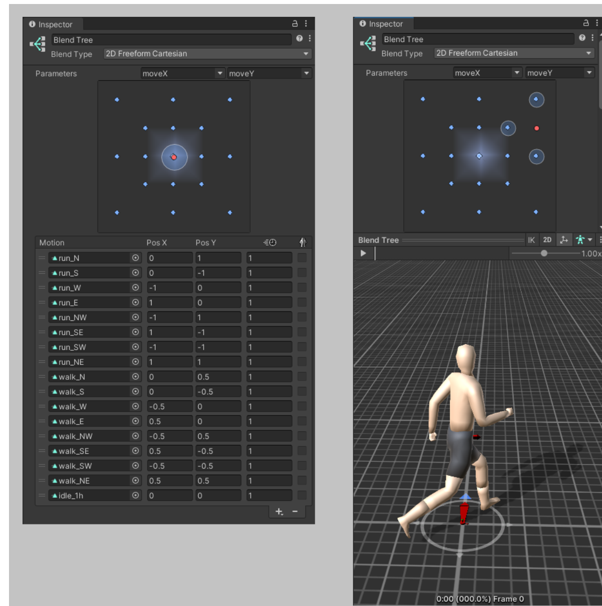

Figure 3.5: The box on the left represents the blend tree which I set up for the agent's movement. The box on the right is an example of the blend tree "in action", where `moveX` is set to 1, and `moveY` is set to 0.5.

3.6 Combat Coherency Circle

As mentioned in section 2.2.4, one of the goals was to implement the four directional combat coherently. This is done by taking the attacker's position relative to the defender into account.

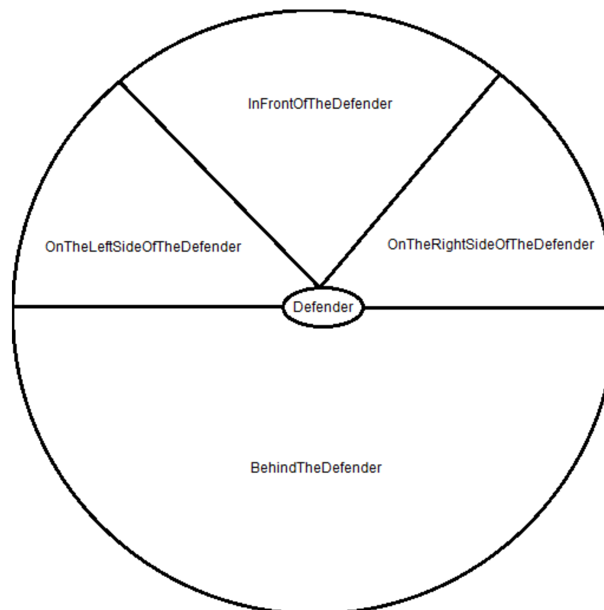

Figure 3.6: A visual representation of the combat circle. The borders are not exact, as it's merely meant to give an idea about the underlying mechanics.

There are parameters which determine the angles and borders of the combat circle. There isn't an actual circle in the game. They're just values which are

taken into account when combat decisions are made.

For example, suppose that the attacker is in front of the defender. If the attacker is attacking from his left, then the defender must block from the right. However, if the attacker is on the right side of the defender, the situation is different. In that case, the right side of the defender is exposed. Meaning, if the attacker attacks from his left, then the defender is unable to defend his own right side, as it is exposed. If the attacker is behind the defender, the defender is unable to block any attacks, even if the combat directions match. The implementation details are in the `CombatMechanics.cs` script.

3.7 Scenes

The game loop is achieved with the use of game scenes in Unity. There are four main scenes:

- MainMenuScene
- InformationMenuScene
- GearSelectionMenuScene
- ArenaScene

As mentioned in section 2.4.3, the approach of using static memory is used to track the player's progress in the game. I didn't think that exceeding the static memory would be an issue, which is why the data are organized under static classes with static fields rather than the `Singleton` approach. The important static classes which play a role in tracking the player's progression are as follows:

- **PlayerCharacteristicProgressionTracker.cs:** For each wave beaten, the player gets bonuses such as increased health, damage, resistance, etc. This class is responsible for tracking that information.
- **PlayerInventoryManager.cs:** Keeps track of the player's gold, as well as the indices of the player's selected weapons and armor. The indices are used by the `HordeGameLogic.cs` to spawn the player with the chosen equipment.
- **PlayerPartyManager.cs:** Tracks the number of mercenaries according to their categories, as well as the maximum party size of the player as more enemy waves are beaten.
- **PlayerStatisticsTracker.cs:** Stores information such as total enemies beaten by player, total gold earned, etc. These numbers are presented in the `InformationMenuScene` when the game is over.
- **StaticVariables.cs:** Stores information regarding the camera rotation speed, graphics preset settings, etc.
- **HordeGameLogic.cs:** While this class is mainly responsible for driving the logic of the horde game mode, it also contains some static fields such as the number of waves beaten; whether the next wave contains a boss battle or not, etc. These static fields can be used by other classes across other scenes.

Since the game is relatively small in size, it was possible to implement the tracking of player's progression in static memory. It would have been more elegant and scalable to implement an overarching manager class to track such information as mentioned in section 2.4.3, but I was unable to do so due to time restrictions.

3.7.1 MainMenuScene

The `MainMenuScene` is the first scene the user sees when the game is launched. From here, the user can navigate to view the key bindings, adjust the settings, or start a new game.

The logic of the controls of the UI widgets are found in the `MainMenuUI.cs` script. It is also responsible for starting a new game, which initializes the necessary variables to start a clean game state. Some simple settings regarding graphics and camera rotation sensitivity are also found here, under the `Settings` submenu. The graphics preset settings can only be changed here, and cannot be modified once the game is started. This is because I didn't want visual distortions and popping to appear in the game due to changing graphical presets. Instead of detailed graphics settings, I offered a simple graphical preset option, as it would have been too time consuming to create UI widgets that change every little setting.

3.7.2 InformationMenuScene

The InformationMenuScene is loaded under the following conditions:

- When a new horde game is launched for the first time
- After a certain number of waves have been repelled
- When the game is over

This scene mainly contains text which conveys the player what to expect in the future. After the first fight, it also provides information about the player's automatic "level up" system. The script which is responsible for this is called `InformationMenuUI.cs`. When the game is over, a list of statistics about the player is shown, which is tracked by the `PlayerStatisticsTracker.cs.`

3.7.3 GearSelectionMenuScene

This is yet another UI related scene. It is a simple scene, much like the other UI scenes, but it has more effect over the game than the others. The logic of this scene is managed by the `GearSelectionUI.cs` script.

This is where the player can buy items such as weapons and armor, and the UI widgets necessary for this task are found on the top right corner of the screen. The item shop acts both as a shop and as an inventory for the player. For example, if a weapon hasn't been bought yet, the widgets will act as a shop for that item. Once it is bought, the item will be in the player's inventory, and the widgets can be used to navigate through the items that were bought.

There is no “shop” class. The items are loaded by the `PrefabManager.cs`, and stored in C# Lists. When an item is bought, its `isPurchasedByPlayer` private field under the `EquippableItem.cs` script is set to true.

The player’s “inventory” is managed by the `PlayerInventoryMananger.cs` script. It doesn’t save which items were bought in memory. Instead, it contains indices which point at the current index of a particular list in the `PrefabManager`. For example, the weapons are loaded into the `Weapons` list under the `PrefabManager`. The `PlayerInventoryManager` has a `PlayerChosenWeaponIndex` field. This field is used to go through the weapons. When the player starts the game, the weapon chosen with this indexer will be equipped by the player. If the weapon was not bought, the UI logic will disallow the player from starting the game. What I described here about weapons are the same for armor purchases as well.

The reason for this is due to time restrictions. The user interface was made at a much earlier time in the game’s development, when the overall concept of the game was more vague. An object oriented approach in the code would necessitate remaking the shop and inventory user interfaces from scratch, as it would be rather difficult to forcefully use the current static memory based implementation of the code on the new interface.

The scene contains a so-called `MannequinAgent`. It’s an agent which inherits from the `Agent.cs` base class. As its name suggests, it is simply there to act as a mannequin for the player. The currently selected weapons and armor will be worn by this agent to demonstrate what they look like.

Due to time restrictions yet again, I was unable to implement a separate UI scene to hire mercenaries. Therefore, the player can hire and upgrade their mercenaries in the `GearSelectionMenuScene` as well. The information regarding the hired troops are managed by the `PlayerPartyManager.cs` script. It contains the number of mercenaries hired. There are simply four types of mercenaries which are categorized by their armor level. These armor levels are basic, light, medium and heavy. The basic armor level means that the mercenary is unarmored (i.e.: no damage reduction from armor whatsoever); whereas the other categories mean full suits of armor in their respective armor levels. Since there was very little room left for more UI widgets, the player cannot see the appearances of the mercenaries. Thus, they will have to guess what a mercenary might look like judging by their names. Though, after the game begins, the player will be able to see exactly what the mercenaries look like.

When the `Fight` button is pressed, the game will transition to the `ArenaScene` where the combat gameplay actually happens. The player will be spawned with whatever items were selected in the “shop” section of the user interface.

3.7.4 ArenaScene

The `ArenaScene` is where the actual gameplay occurs. The player fights against the incoming waves of enemies, earning gold with each enemy felled. When the wave is finished, the game transitions to the `InformationMenuScene`, and the loop continues.

The spawning of all agents are managed by the `HordeGameLogic.cs` script. In the scene, you can find a game object named `HordeGameLogic` to which the

aforementioned script is attached. This is where the game designers can customize the properties of the invader enemies, as well as the horde waves. This is explained in section 3.10.

The arena is a simple rectangular area where the agents fight. The spawn locations can be chosen using the spawn point child objects under the `HordeGameLogic` game object.

The game can be paused mid combat by pressing the M button. This shows a pause menu where the player can customize some settings. The script responsible for this is called `InGameUI.cs`. The same script also shows the player's health and gold on the screen.

The scene contains hidden box colliders to prevent the agents from escaping the intended gameplay zone. The `NavMesh` that was baked defines the areas on which the `AiAgents` can traverse. The player can move freely, so the invisible box colliders are especially used to prevent the player from leaving the gameplay area.

Figure 3.7: An in-editor screenshot of the arena which the game uses as the combat scene.

The floor of the arena is made out of many little cubes in green color. There is no terrain that is being used anywhere in the game, which means there aren't any small slopes as mentioned in section 2.5. At some point, I tried making a mountainous map using Unity's terrain tool, but it took way too much time, so I decided to use an old arena scene which I had made during development.

One of the goals was to put obstacles in the game scene. I ended up not adding any obstacles at all, as it didn't feel right for this arena. If obstacles need to be added, it would not be an issue. For the case of the `AiAgents`, the `NavMesh` of the arena would have to be re-baked so that the `AiAgents` can recognize and walk around the obstacles. In the case of the `PlayerAgent`, it is necessary to attach a collider (preferably with an appropriate shape) so that the `CharacterController` can detect collisions with them and prevent the player from passing through. Finally, the static lighting of the overall scene would have to be re-baked so that the shadows of the obstacles are cast properly [49].

The scene itself is organized under the `Scene` game object. The `SceneDecors` child object contains towers, battlements and gatehouses around the scene. These are scene props which are simply for show. They cannot be climbed over by any agent. The second child object is called `NavMeshGeometry`. It also contains a child object named `Real Floors`, which contains the walkable

Figure 3.8: An in-editor screenshot of the mountainous combat scene which was discontinued due to time constraints. It lacks the textures and environmental decors such as trees and boulders.

cubes as mentioned earlier. It is possible to use these game objects to re-bake the `NavMesh`, if necessary.

3.8 Weapons

If you have a weapon model that you wish to add to the game, create an empty game object in the Unity Editor on any open scene. Then, drag your weapon model to be a child object of the empty game object. Add a `Weapon.cs` script as a component to the empty game object. Enter its shown name, purchase cost, and whether or not it's a starter item which is available at the start of a new game (i.e., does not need to be purchased with gold). Customize the sound options. For example, it'd make sense for an axe with a wooden shaft to make a wooden sound when it is used to block an attack. To fill the `Weapon Visual` field, select your weapon model on the scene, and drag & drop onto the `Weapon Visual` field.

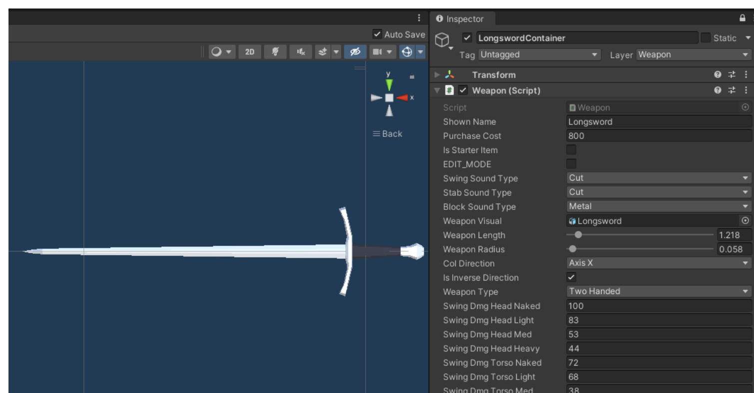

Figure 3.9: A screenshot of the Longsword weapon showing some of its properties in the Unity Editor.

Next, add a box collider component to the empty game object. Use the `Weapon Length` and `Weapon Radius` fields in the `Weapon.cs` script to adjust those properties of the weapon. Use the `Col Direction` and `IsInverseDirection` fields to further adjust the trigger collider of your weapon. In order to see the

changes in real time, check the EDIT_MODE checkbox defined in the component. This will allow you to see the dimensions of the trigger collider of the weapon in real time on the scene. Note that, when EDIT_MODE is enabled, you may receive warnings in Unity's Console window. Feel free to ignore these warnings, as they are related to the Unity Editor itself. When you are satisfied with these fields, make sure you uncheck the EDIT_MODE field, so that Unity doesn't spam you with any more warnings.

If your weapon is a two handed sword and want those animations to be played by the agents, select the Two Handed option in the Weapon Type field. If the weapon is more like a spear or a poleaxe, then it'd make sense to use Polearm as the Weapon Type.

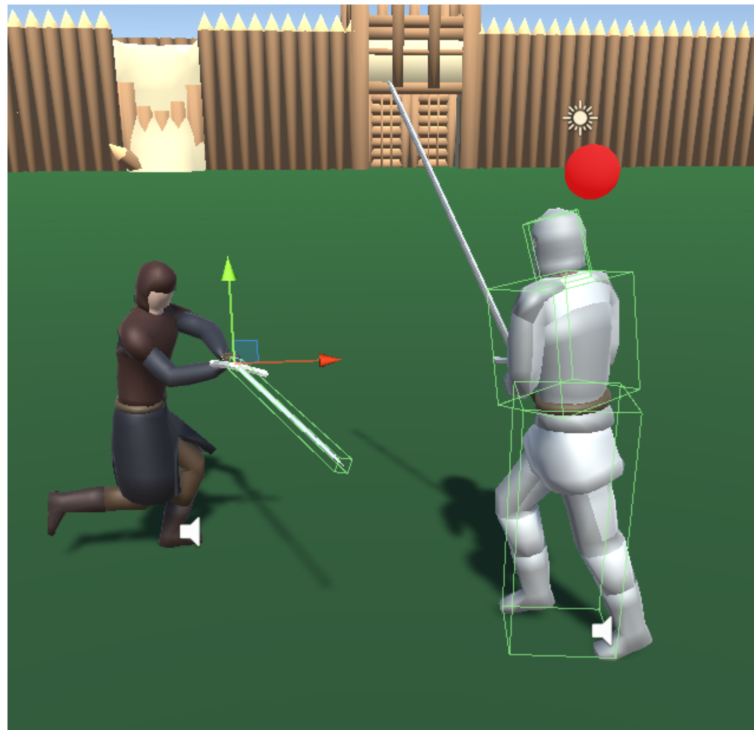

Figure 3.10: The PlayerAgent (on the left) is attacking an AiAgent. The weapon's trigger collider will detect one of the AiAgent's limbs and inflict damage, because the AiAgent is not defending.

Enter the damage values of your weapon. You'll be entering the swing and thrust damage values separately. The damage values against each body part and armor type is entered one by one. For example, the SwingDmgHeadNaked field represents the damage value of a swing which targets an unarmored head.

You may wish to rename the empty game object to something more recognizable in the project folders. Once you are finished, drag and drop your renamed game object under the "/Assets/Prefabs/Resources/Weapons" folder. Your weapon will now be available to the player, thanks to the PrefabManager.cs script. To make it available for the other agents, refer to section 3.10.

For any given agent, it is the responsibility of the EquipmentManager.cs script to instantiate a chosen weapon into the hands of the agent. More specifically, the weapon game object becomes the child object of the item bone named item_R.

This is the item bone in the right hand of the agent's skeleton. The weapon is able to deal damage thanks to the trigger collider that is attached to it, which is managed by the `Weapon.cs` script. When the weapon hits something in the game, the decisions are made in Unty's `OnTriggerStay` callback, which uses helper methods that are in the `CombatMechanics.cs` script.

The design of the damage values are similar to that of Mordhau, as mentioned in section 2.3.1. It was partly due to time restrictions, as it would have taken too much time to come up with a custom damage formula and test that it works as intended. Moreover, using set damage values for the weapons allows one to carefully choose how much damage should be dealt in each situation. Note that the damage values themselves in Just Blade were heavily inspired by the actual damage numbers in Mordhau, though they can easily be customized by using the process described above.

3.9 Armor

To add armor which can be worn by the agents in the game, you need to make sure that your armor model is rigged with the same skeleton as the human model. The human model is called "Human 2022 - v1.08", and it's under the `"/Assets/Models"` directory. The model and its skeleton was made and rigged using the Blender software. While I'm not a professional artist, I'd recommend opening the human model file, and creating an armor model which fits the agent's body dimensions while making sure that your armor mesh uses the same skeleton as the human. For example, if you're modelling a body armor, you can duplicate the body mesh of the human model, and sculpt it to make it look like a body armor, and so on. Once you're done, save your work, and import it in Unity under the `"/Assets/Models/Armor"` directory.

Once your armor model is imported, click on it once to see its properties in the Inspector menu. Go under the `Model` tab, and set the `Scale Factor` to 0.29. As noted earlier in section 3.1, the human model uses this scale factor when it was imported into Unity. Save your work and proceed.

It's not necessary (and perhaps not recommended) to create an empty game object on the scene to add your armor. Simply drag your armor model into any scene in the `Unity Editor`, and add an `Armor.cs` script to it. Enter its shown name, purchase cost, and whether or not it's a starter item which can be used without purchasing by the player.

To fill the `Skinned Mesh Renderer` (`SMR`) field, inspect the child objects of your armor model on the scene. You should see a transform game object which represents the root bone of the skeleton. In the human model, this root bone is called `Armature`, so your armor model probably uses the same name (unless you changed it). You do not need to deal with this `Armature` child object. What you need is the other child game object, which contains an `SMR` component (if you did things right). Drag and drop the game object (with the `SMR` component) into the `SMR` field of the `Armor.cs` script. This will make it so that your armor is wearable by the agents in the game.

It makes sense to choose an `Armor Type` which suits your armor mesh. For example, if you make a pair of boots, I wouldn't recommend using `Head` as the armor type. I never tried this, so you should also just choose `Leg` as expected. The

`CoversTheEntireBodyPart` field can be checked if you want the corresponding body part to be invisible when the armor piece is worn.

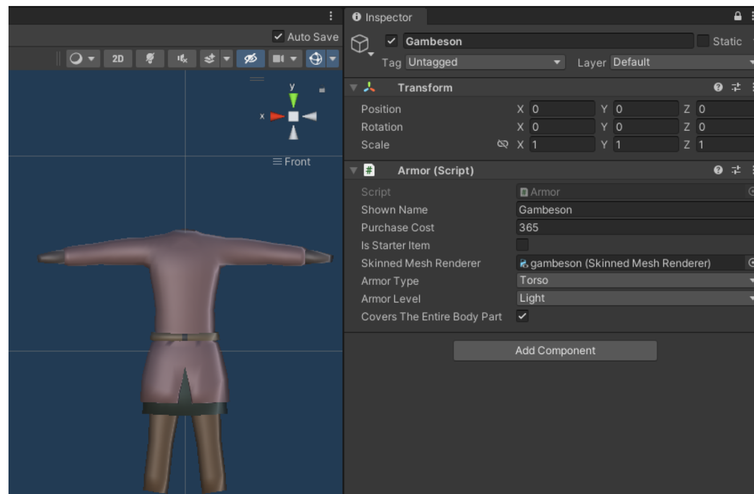

Figure 3.11: A screenshot of the Gambeson torso armor showing its properties in the Unity Editor.

Once you're done, drag and drop your armor model under the `"/Assets/Prefabs/Resources/Armors"` directory to save it as a prefab. Your armor piece will now be available for the player, which is loaded by the `PrefabManager.cs` script. To make it available for the other agents, refer to section 3.10.

The `EquipmentManager.cs` script is responsible for the technical details of making an armor worn by an agent. It does this by setting the bones of the armor's `SMR` equal to the bones of the agent's bones. The same script also contains the method which determines the movement speed penalty due to wearing armor. The design of the armor system is similar to that of Mordhau, which was explained in section 2.3.1. This decision was due to time restrictions mentioned in the Weapons section earlier, and also the fact that I thought it complemented the weapons' damage system well.

3.10 Horde Prefabs

The horde game mode is made out of prefabs which are contained in this folder, and its directory is `"/Assets/Prefabs/HordePrefabs"`. This section explains how to add new waves and enemies to the horde game mode, by going over the following subfolders:

- ArmorSets
- WeaponSets
- CharacteristicSets
- RewardData
- InvaderAgentData
- MercenaryAgentData

3.10.1 Horde Agent Data

Before the contents of each folder is explained, it might be helpful to go over the basic class hierarchy of the horde agents.

The agents that can be seen in the horde game are called “horde agents”. These are just regular agents which use one of the derived classes of the `Agent.cs` script. However, in the context of the horde game mode, these agents get their properties from the various horde agent data.

Every horde agent uses the following data (found in `HordeAgentData.cs` script), whether it’s a friend or foe to the player:

- Armor set
- Weapon set
- Characteristic set

Horde agents which are hostile towards the player are called “invader agents”. They use the `InvaderAgentData.cs` script, which inherits the above data from `HordeAgentData.cs`. These agents, in addition to the above, contain reward data when slain (in `HordeRewardData.cs`).

The friendly horde agents who fight by the player’s side are called “mercenary agents”. They use the `MercenaryAgentData.cs` script, which inherits from `HordeAgentData.cs`, but also contain several new fields.

The next sections describe how to put together a horde agent using various pieces.

3.10.2 ArmorSets

To create an armor set, create an empty game object on any scene, and attach the `HordeArmorSet.cs` component to it. For each armor list, add the appropriate armor piece, as many as needed.

For example, let’s assume that you want a horde agent to have 5 helmets to choose from. To do this, create 5 entries in the `Head Armor Prefabs` list in the attached component. Then, navigate to the `"/Assets/Prefabs/Resources/Armors/HeadArmors"` directory, and drag and drop a helmet from this directory into the appropriate entry in the list. Do this 5 times for 5 different helmets. If you want to add torso, hand and leg armors, follow the same procedure, and add them to their respective lists. Once your work is done, drag and drop the game object from the scene to the `"/Assets/Prefabs/HordePrefabs/ArmorSets"` directory. After this, any horde agent which uses this armor set will randomly choose an armor for each of the armor slots, as described in the lists that you provided. If you don’t want the armor to be chosen randomly, then simply put one armor piece in each slot.

3.10.3 WeaponSets

To create a weapon set, create an empty game object on any scene, and attach the `HordeWeaponSet.cs` component to it. It contains a list which can be filled with weapon prefabs found in the `"/Assets/Prefabs/Resources/Weapons"` folder.

If you add 3 weapons in the weapon list, then any agent who uses this weapon set will choose one of the 3 weapons listed when spawned. The choice is done randomly. If you don't want the choice to be random, just make sure that you only have one weapon in the list.

To save your work, simply drag and drop the game object from the scene into the `"/Assets/Prefabs/HordePrefabs/WeaponSets"`.

3.10.4 CharacteristicSets

Each horde agent has the following characteristics:

- Health
- Model size multiplier
- Extra movement speed multiplier
- Extra damage infliction multiplier
- Damage taken multiplier
- Poise

To add a characteristic set, create an empty game object on any scene, and add a `CharacteristicSet.cs` component to it. Configure the necessary fields, and save your work by dragging and dropping the game object from the scene to the `"/Assets/Prefabs/HordePrefabs/CharacteristicSets"` directory. Below is a brief explanation about what each characteristic does.

Maximum Health determines the starting health of any horde agent. When their current health value reaches zero, the agent dies.

Model Size Multiplier acts as a multiplier to the agent's model size. For example, if this value is 1.5, the agent will appear 50% bigger in scale, in every dimension. The agent size affects various things, such as the size of equipped weapon, artificial intelligence's combat decisions, and so on. Note that the movement speed does not depend on the agent's model size.

Extra Movement Speed Multiplier acts as a secondary multiplier to an agent's movement speed. Normally, every agent's movement speed is affected by how heavy their overall armor is. However, this field acts as a way to multiply the final movement speed value. It can be used to create special enemies with high movement speed.

Extra Damage Infliction Multiplier is a secondary multiplier to an agent's damage. Normally, an attacker agent deals damage to a defender agent based on the equipped weapon's damage values; and the defender agent's armor level. However, this field acts as a way to further multiply the damage output.

Damage Taken Multiplier is a secondary multiplier which affects how much damage an agent takes. For example, suppose that this value is set to 0.5. Then, once the usual damage calculations are done, the final damage output will be further multiplied by 50%, which effectively halves the damage taken. Note that if this value is higher than 1, then the agent will take increased damage.

Maximum Poise determines the starting poise of an agent. Normally, when an agent is struck, the agent flinches by playing a "getting hurt" animation. This

happens if they have zero poise. However, suppose that an agent has a poise value of 3. When this agent is struck, he will not flinch. Each time he is struck, he loses poise by 1. When poise reaches zero, the agent will flinch, and the poise value will be reset back to the `Maximum Poise`. This characteristic can be useful when creating boss enemies.

3.10.5 RewardData

When an invader agent is killed, the player gains gold based on the reward data. To add reward data, create an empty game object on any scene, and add a `HordeRewardData.cs` component to it. The gold reward data is chosen randomly between the `Min Gold Reward` and the `Max Gold Reward`. To save your work, drag and drop the game object from the scene to the `"/Assets/Prefabs/Horde Prefabs/RewardData/"` folder.

3.10.6 InvaderAgentData

To create an invader agent data object by combining the pieces from the previous sections, use the following steps. Create an empty game object on any scene, and add an `InvaderAgentData.cs` component to it. Then, simply drag and drop the data pieces crafted from the previous sections into the necessary fields. Additionally, if the `IsAggressive` field is checked, then the invader agent who uses this data will attack relentlessly, without ever thinking about defending. To save your work, drag and drop the game object from the scene to the `"/Assets /Prefabs/HordePrefabs/InvaderAgentData"` directory.

3.10.7 Adding a New Invader

The enemies in the horde game mode come in “wave sets”, which contain “waves”. A wave set is completed by repelling all of the waves defined in it. Each wave is repelled by killing all of the invader agents defined by the wave. When a wave set is complete, the game switches to the gear selection scene, so that the player can buy items or hire mercenaries.

The previous sections explained how to create a new enemy data. This section explains how to actually add the enemy based on this data.

Open the `ArenaScene` in the editor, and find the game object named `HordeGameLogic` in the scene. Click on it to view its properties in the inspector. It contains a list named “Wave Sets”.

Click the + icon to add a new wave set. In this wave set, click the + icon to add a new wave. You can add multiple waves in a single wave set. Check the `IsBossWaveSet field` if this is a boss wave set, so that the player is informed about it in the `InformationMenuScene`.

In a given wave, click the + icon to add a new invader data list. A single wave can contain multiple types of invaders.

In a given invader data list, click the + icon to add a new invader data. Drag and drop the invader data found in the `"/Assets/Prefabs/HordePrefabs/Inv aderAgentData"` directory into the `Invader Agent Data Prefab` field. You can use the `Invader Count` field to determine how many invaders of this type should invade.

Using the description above, it is possible to add any number of wave sets, waves, and invaders.

3.10.8 MercenaryAgentData

Mercenary agents help the player defeat the invaders. The user interface defined in the `GearSelectionMenuScene` forces us to have only 4 possible mercenary agents. In other words, there are only 4 button widgets on the user interface, which means that the player can only hire `Basic`, `Light`, `Medium` and `Heavy` mercenaries. Their data can be found under the `"/Assets/Prefabs/HordePrefabs/Resources/MercenaryAgentData"` directory.

The mercenary data are loaded via code in the `PrefabManager.cs` static class. The loaded data is managed by the `PlayerPartyManager.cs` static class. During gameplay, the `HordeGameLogic.cs` script uses the party manager class to spawn the player's mercenaries.

The way I set up the user interface so that there can only be 4 types of mercenary agents to hire is due to my design decisions. Due to time restrictions, I had to come up with a simpler design to hire mercenaries, and this is the result. To add more variety to the player's mercenaries, the user interface must be changed. After that, the C# script files mentioned above, as well as the `GearSelectionUI.cs` script will also have to be edited to accommodate the new changes.

3.10.9 Horde Difficulty Progression

The difficulty progression of the game is done by giving better equipment and characteristics to the enemies in later waves. As the waves progress, the enemy `AiAgents` do not become smarter. In other words, they don't form tactical formations to better tackle the player's team; nor do they fight smarter by blocking in the correct direction. Instead, they get heavier armor, more lethal weapons, and better characteristics. Finally, there is a difficulty slider which can be found in the `MainMenuScene` that further adjusts the difficulty of the game.

3.11 Sounds

To add a sound effect which can be heard in the game, create an empty game object on any scene, and add the `PlayAndDestroy.cs` component to it. When done so, this will automatically add an `Audio Source` component [50] as well.

Navigate to the `PlayAndDestroy.cs` script and enter its `Sound Name`, which will be important later on. Drag and drop the `Audio Source` component attached to the game object in order to fill the `Audio Source Component` field. Find the location of your own audio clip, and drag and drop it onto the `Audio Clip` field. Customize the volume and the pitch of the sound effect to your needs. Once you're done, save your work by dragging and dropping the game object on the scene to the `"/Assets/Prefabs/Resources/SoundEffects"` folder.

To actually play the sound effect in the game, it is necessary to do some C# scripting. Open up the `SoundEffectManager.cs` file, and examine the source code. If you're just adding a variation to an existing sound effect (e.g., a new

grunting sound), then just enter the name of your sound effect (i.e., the important one mentioned above) into the appropriate array, and it will be played at random when needed. If your sound effect is entirely unique, then create a new array by examining the source code, and insert the name of your sound effect to it. Note that you will have to invoke the method which plays your sound effect in the new array.

All of the sound effects in the game are free to use assets (at least, they were free to use during the development of this game) acquired from Unity's `Asset Store`. The credits are given to their respective authors in the `"Documentation MainPage.md"` file, which can be found under the `"/Assets/Scripts"` directory.

3.12 Collision Layers

The collision layers in Unity are used to fine tune which objects should detect or ignore contact with one another. The collision matrix [51] can be viewed in the editor. This section briefly explains the layers that are used, and their purpose:

- **Default** layer is a built-in layer added to every project by Unity. In this game, it's mainly used by the scene geometry objects (such as floors, invisible walls, etc.). It is also assigned by default to any other game object, but they may not necessarily be using it.

- **Agent** layer is mainly used by the agent prefabs that reside in the `"/Assets/Prefabs"` directory. It is simply there to avoid using the `Default` layer for the agents, so it doesn't really have a special purpose.

- **Weapon** layer is used by the weapon prefabs found in the `"/Assets/Prefabs/Resources/Weapons"` folder. This layer detects contact with the Limb and `Default` layers.

- **Limb** layer is used by the three hitboxes (head, body, legs) that are on the agents. The combat decisions are made when a Weapon object makes contact with a game object which uses the `Limb` layer.

- **NoCollision** layer is used by the weapon prefabs. When an agent is releasing an attack, the weapons are put into the `Weapon` layer at runtime, so that it can detect collisions. At other times, weapons are put into the `NoCollision` layer, so that they can ignore all collisions.

4. User Guide

This part of the text explains how the game itself is actually played. The reader is assumed to be a regular player, and therefore the text does not contain any technical details regarding the project.

4.1 Installation

To install the game, it is necessary to use a program which can extract an archive file. Most machines nowadays come with operating systems which already include such programs. However, in case you do not have a program which can work on an archive file, I recommend installing `7zip` first [52]. The installation instructions written here assume that the user has `7zip` installed.

After making sure you have means of working with an archived file, follow the rest of the instructions below:

- Download the game's archive file file.
- Extract the archive into a folder.
- Run Just Blade executable in the newly created folder.

4.2 Menus

This section explains the several menus that are encountered throughout the game.

4.2.1 Main Menu

The main menu is the first screen that is seen when the game is launched. It contains a few buttons whose functions are explained below.

The `Start Game` button starts the game. Once the game is started, there is no way to return back to the main menu without quitting a playthrough altogether. Keep in mind that the game doesn't have a "save game" feature.

The `Key Bindings` button opens up the submenu which shows the key bindings. This menu cannot be opened when the game is started, so be sure to read the key bindings beforehand.

The `Settings` button opens up the settings submenu where the user can customize some settings. Their functions are listed below:

- **Camera Sensitivity:** Changes the in-game camera's rotation speed. If the value is too low or high, the camera will rotate very slowly or quickly, respectively.
- **Sound:** Controls the sound level (i.e. "volume") of the game's sound effects. Setting the slider to zero mutes the game entirely.

- **Field of View:** Adjusts the field of view [53] of the in-game camera. Higher values increase peripheral vision, but may cause so-called “fisheye effect” [54].
- **Difficulty:** Adjusts the difficulty of the game. See section 4.3.3 section for an explanation as to how the difficulty setting works.
- **Quality Setting:** Changes the graphics preset of the game. Lower settings may increase performance.

Note that the first three of the aforementioned settings can be changed after the game starts (by pressing M key and opening the in-game pause menu); whereas the last two cannot.

The `Exit Game` button closes the game application.

4.2.2 Information Menu

The Information Menu is seen in the following cases:

- When a new game is started for the first time
- After repelling a set of waves
- When the game ends

This menu shows text as to what should be expected in the upcoming waves. After the first combat encounter, it also shows values about the player’s “automatic level up” progress, which depend on how many waves the player has beaten. These values are increased health, extra damage, extra damage resistance, extra movement speed and party size.

When the game ends, this menu shows statistics about the player, such as how many enemies were killed, total gold earned and spent, and so on.

The user can click the `Next` button to navigate to the `Gear Selection Menu`. Note that it is possible to return to the `Information Menu` before the combat begins.

4.2.3 Gear Selection Menu

The `Gear Selection Menu` allows the player to buy weapons and armor, as well as hire and upgrade mercenaries.

On the top right corner, the player can choose from a variety of weapons and armor. If the player does not own a selected item, its price is shown on a button. The player can double click on the purchase button to permanently buy the item for the entire playthrough.

On the top left corner, there is information about the selected weapon. It contains the name and the length of the weapon, as well as some damage values. Every weapon can be swung and thrusted in combat. The average damage values for swings and thrusts are written for the selected weapon. The shown value is an “average”, because the actual damage depends on the body part by which the weapon was struck; and the armor level of the said body part.

On the left side, the armor level of each armor slot is shown. There are four armor slots:

- Head
- Torso
- Hands
- Legs

Each armor slot can be filled with an armor piece that protects the targeted body part. In other words, head armor protects the head; torso armor protects the torso; and leg armor protects the legs. Note that hand armor does not protect the hands, but provides a general damage resistance to all body parts.

There are four armor levels:

- None
- Light
- Medium
- Heavy

The `None` armor level provides no damage resistance whatsoever, but the character incurs no movement speed penalty. On the flip side, the `Heavy` armor level provides the highest damage resistance with the highest penalty to the movement speed. The movement speed penalty varies based on the armor slot. For example, a heavy torso armor will yield a higher movement speed penalty compared to a heavy head armor. Note that the movement speed multiplier which is based on worn armor is different from the extra movement speed multiplier that can be seen in the `Information Menu`. Both of these are separate values which act multiplicatively.

On the bottom left corner, the information regarding the mercenaries hired by the player is shown. There are four types of mercenaries:

- Basic
- Light
- Medium
- Heavy

The `Basic` mercenaries wear no armor and use simple weapons. The `Heavy` mercenaries have the highest armor level in all armor slots, and use much more lethal weapons. The `Heavy` mercenaries are also inherently stronger, as their characteristics have increased health, damage and damage resistance. The equipment and characteristics of the other mercenary types lie somewhere between `Basic` and `Heavy`, as their names suggest.

The player can hire and upgrade mercenaries, with no way to disband them. Note that the number of mercenaries that can be hired is capped by the player's party size, which is written at the end of the mercenary info section.

On the bottom right corner, the player's gold is shown. It is necessary to have enough gold to perform the monetary transactions which were described above.

The `Back` button can be pressed to return to the `Information Menu`. The `Fight!` button starts the combat with no way to return to the menus until the waves of enemies are beaten.

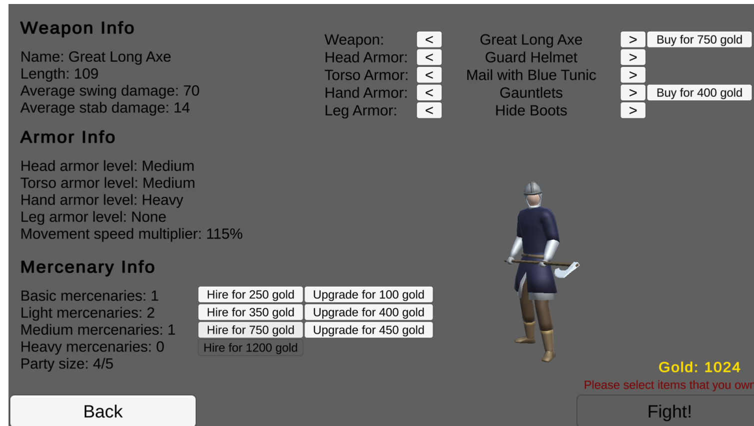

Figure 4.1: An in-game screenshot of the Gear Selection Menu.

4.3 Gameplay

The goal of the game is to beat the enemy hordes coming in waves. You earn gold for every fallen enemy, which can be used to buy weapons and armor; hire and upgrade mercenaries to help you fight in battle. The next sections explain the parts about the game that are related to combat.

4.3.1 Combat

The combat consists of two primary moves: attacking and blocking. Attacking can be performed by clicking the left mouse button (i.e.: “a left click”), and blocking is done by clicking the right mouse button (i.e.: “a right click). There are four directions in which any character can attack or defend: up, right, down, left. The direction is chosen by mouse movement. For example, if the mouse is moved to the right, the “right” direction is chosen. It is then possible to perform an attack by doing a left click, or block by doing a right click. When a character attacks from “down”, it is simply a thrusting maneuver; whereas the other directions are swinging motions. The damage of the weapon depends on whether it was swung or thrusted, as mentioned in the `Gear Selection Menu`.

Figure 4.2: An in-game screenshot of the the player defending against an overhead attack by the enemy.

When a character is attacked, they can defend themselves by blocking in the correct direction. For example, if an enemy is attacking you from above, you have to face the enemy and start defending in the same direction. Note that it is not possible to defend against attacks when your back is turned, even if the direction and the timing is correct.

Blocking isn't the only way to avoid getting damaged by enemies. It is possible to strafe left or right to dodge the enemies' attacks, depending on the direction. Moving backwards is also an option. Note that the player character incurs a 50% movement speed penalty while moving backwards, in order to prevent them from kiting the enemies. This penalty is only applied if the movement speed penalty from armor is greater or equal than 100%. The non-player characters (`NPC`) do not suffer a movement speed penalty while moving backwards in any case.

The mercenaries hired in the `Gear Selection Menu` help the player fight the hordes of enemies. By pressing the Q button, it is possible to order the mercenaries to hold position at the player's current location, or let them attack enemies at will.

The player's health bar is shown at the bottom right corner of the screen during combat. If it reaches zero, the player character dies and the game will end. The gold earned by the player is also shown at the bottom right corner, which increases as each enemy is fallen.

4.3.2 Enemies

This section gives information about the various types of enemies which can be found in the game.

There are two types of enemies in the game:

- Common enemies
- Boss enemies

There are four types/tiers of common enemies:

- Basic
- Light
- Medium
- Heavy

The enemies attack in the following pattern:

- Two waves of common enemies.
- A single wave of boss enemies.

For example, when the game starts, the first two waves consist of common enemies. Once they're repelled, the player is given a chance to take a break. During which, it is possible to buy weapons and armor; and hire/upgrade more mercenaries. When ready, the player can start the next battle, which consists of

a single wave of boss enemies. When this wave is also beaten, the game repeats the pattern by sending two waves of common enemies, and so on.

The names of the types of common enemies might give a good idea as to what they're like. The basic enemies use basic weapons and armor, while heavy enemies have lethal weaponry and wear heavy armor. The other tiers of enemies lie somewhere between the two cases.

Figure 4.3: An in-game screenshot depicting a common enemy.

In the very first wave, the enemies are basic. As the waves progress, the basic troops start to use better weapons and armor over time. After a few more waves, enemies of higher tiers start showing up. Therefore, it's a good idea to hire better mercenaries to counter them. It is important to note that higher tiers of enemies and mercenaries have better characteristics, such as increased health, damage, resistance, movement speed, and so on.

There are seven boss waves in total:

- Tutorial boss
- Light armor unit
- The berserker and his bodyguards
- Medium armor unit
- Heavy armor unit
- The turtle knight and his bodyguards
- The berserker and the turtle knight, and their bodyguards

The light, medium and heavy armor units consist of several high level enemies which use armor types as suggested by their names. They are better than the common soldiers of their type, and wear the same suit of armor to show off their unity. The tutorial boss wave contains one member from each of the aforementioned units, as a way to introduce the player to the game.

The berserker is a unique boss who wears no armor (except for a helmet), and wields a giant axe. He is much taller than most enemies, moves around very quickly, and can inflict serious damage with his attacks. He hardly ever flinches

Figure 4.4: An in-game screenshot of the boss enemy "Turtle Knight" who stands taller compared to others.

when struck, as he has 3 units of `Poise` (explained below). He is surrounded by his followers who look similar to him, but aren't as strong.

The turtle knight is another unique boss who wears full plate armor. Unlike the berserker, he moves very slowly due to his armor, but is very much resistant to damage. He has 5 units of `Poise`, thanks to his armor, so he won't flinch too many times when struck. His bodyguards who look like him also have 2 units of `Poise`.

`Poise` is a mechanic that is exclusive to the enemies described above. It allows the characters to withstand damage without flinching. Normally, enemies flinch when they're struck, causing them to stop what they're doing and squirm briefly in pain. However, for example, if an enemy has 2 units of `Poise`, they don't flinch for two hits. After being struck for the third time, the enemy flinches, and his `Poise` resets back to 2 units.

4.3.3 Gameplay Strategies

This section gives a few strategies as to how the game can be played. The strategy depends on the game's difficulty, which is why they're explained as such.

The game's difficulty is based on a slider in the `Settings` section of the `Main Menu`. The slider ranges between 25% and 150%, where the former is the easiest difficulty setting and the latter is the hardest. When the difficulty is low, the player's team takes reduced damage from enemies, and the fallen enemies yield more gold. On higher difficulty settings, the amount of gold earned becomes less, but the player's team does not take increased damage.

For the purposes of this section, the difficulty setting will be categorized in the following way:

- Values less than 100% are considered easy
- 100% is considered normal
- Values above 100% are considered hard

On easier difficulty settings, the player earns more gold, which can be used to hire a lot of heavily armored mercenaries. They will pretty much carry the player to victory with barely any input from the player.

As the difficulty setting approaches normal, the player has to put some effort to win the game. It is crucial that the player doesn't spend all the gold on buying items for themselves. Instead, it's a better idea to hire and upgrade mercenaries. Commanding the mercenaries (using the Q key) so that they stay together becomes a useful strategy, though it's important not to always keep them in a straight line. This is because some mercenaries might have shorter weapons, which means they may not be able to reach their enemies if they are ordered to stay in a line. While the mercenaries are holding off most of the enemies, the player should try to find opportunities to get behind the enemies, and get a few clean hits in.

Figure 4.5: An in-game screenshot of the player's mercenaries lining up to anticipate the enemy onslaught.

On the harder difficulties, the reduced amount of gold earned poses a significant challenge to the player, which is why it's not recommended on a first playthrough. The player will pretty much have to play alone in the earlier waves. This is in order to save gold to hire mercenaries for the boss waves. The importance of commanding the mercenaries effectively becomes more pronounced, as there is less gold spare to hire them. Regarding combat, blocking isn't always the best choice. Strafing left and right to dodge attacks, as well as keeping distance from the enemy in general becomes more useful. For this reason, it might be necessary to use lighter armor for the entire game. This is because while heavier armor gives better protection, the movement speed penalty caused by it can become deadly in a mob.

5. Conclusion

In this chapter, I briefly go over the goals in section 1.2 to review how well they were implemented in the actual game:

- **Goal 1:** This was accomplished. Though, after the development of the game was finished, I found out that Unity seems to offer an alternate way to address the issue described in section 2.2.5 in the form of Transform Constraints [55].
- **Goal 2:** This was accomplished as well. Using keyframe animations as described in section 2.2.2; as well as making the combat coherent as described in section 2.2.4 and section 3.6 is definitely a viable approach to implement this goal.
- **Goal 3:** This was accomplished with the exception of assigning different attack speeds to each weapon. The reasons for this were described in section 3.4.3.
- **Goal 4:** This was accomplished by using the second approach described in section 2.3.1.
- **Goal 5:** The sub-goal `5d` was accomplished, but the other sub-goals proved challenging due to time constraints, which were explained in section 3.7.4. The game works fine with the arena scene it ended up having, as this goal was mainly about the visual aspects of the game scene, and not having obstacles in the scene doesn't affect gameplay in a significant way. Despite that, I think I could have done a better job predicting how much time it takes to create a scene from scratch.
- **Goal 6:** This goal was accomplished as described in section 3.10.9.
- **Goal 7:** This was accomplished for the most part, with the exception of sub-goal `7c`. The reasons were explained in section 3.7.3.

5.1 Future work

I believe that the core of the game is implemented successfully, but no game is perfect. Therefore, in this section, I offer some ideas about what might be added or changed as future work:

- **Animation:** As described in section 3.4.1 to section 3.4.3, there were some issues as a consequence of using Unity's Mecanim to implement the combat animations of the game. While the other methods to implement the animations as described at the start of section 2.2 could be used to mitigate these problems, I believe writing a custom state machine in C# to drive the combat animations might be a better approach. Put simply, the idea would be a "hybrid" approach, which is to use Mecanim for movement animations, and use the `State` design pattern [56] to write a custom C# script to implement the combat animations using Unity's

`CrossFade` method [57] in its animation engine. I believe this approach would grant the best control over the combat animations, but requires some time to properly implement and debug it.

- **Tracking Player Progression:** As mentioned in section 3.7, the static memory approach was used to track the player's progression across scenes. As a future work, it would be a better idea to implement an overarching manager system which tracks such information across scenes, which was briefly described in section 2.4.3.
- **Game Difficulty:** The game's difficulty was implemented as described in section 3.10.9. However, I believe this can be further improved by making the `AiAgents` smarter, which was the first approach described in section 2.6.
- **Combat Scene:** Several new combat scenes could be made and put in a new user interface where the user can select them before starting a new game. This would help keep the game "fresh", in the sense that the player would not get tired of playing in the same scene over and over.
- **User Interface:** As mentioned in section 3.7.3, the menu to hire mercenaries was combined with the menu to buy weapons and armor. Therefore, creating several user interfaces with each of them having their own purpose is another idea of some future work. Note that this might require refactoring the underlying code in an object oriented manner rather than the static memory approach, which was also briefly touched upon in section 3.7.3.
- **Items:** As with any game, it might be a good idea to add more variety to the weapons and armor. It's not necessary to add entirely new models. Copying the existing weapons and armor meshes and changing their dimensions and colors respectively could also work.
- **Enemies:** More waves and enemy types could be added to the game. In particular, if the implementation of the combat animations can be improved (perhaps with the approach described above), it would be possible to change the attack speeds of the agents based on the weapon and characteristics that they have. This would allow adding new enemy types who can attack very slowly or quickly.

Bibliography

[1] Mount & Blade - TaleWorlds Entertainment .
https://www.taleworlds.com/en/Games/MountAndBlade .
[Online; accessed 15-02-2024] .

[2] Survival Mode - Wikipedia .
https://en.wikipedia.org/wiki/Survival_mode .
[Online; accessed 13-02-2024]

[3] Browser Game - Wikipedia .
https://en.wikipedia.org/wiki/Browser_game .
[Online; accessed 13-02-2024]

[4] The Last Stand - Play on Armor Games .
https://armorgames.com/play/269/the-last-stand .
[Online; accessed 13-02-2024]

[5] Mount & Blade: Warband - Wikipedia .
https://en.wikipedia.org/wiki/Mount_%26_Blade:_Warband .
[Online; accessed 08-01-2024]

[6] Mount & Blade: Warband - Invasion Mode - Steam News .
https://store.steampowered.com/news/app/48700/view/2905340928358501663 .
[Online; accessed 08-01-2024]

[7] Mordhau (video game) - Wikipedia .
https://en.wikipedia.org/wiki/Mordhau_(video_game) .
[Online; accessed 08-01-2024]

[8] Unity Engine - unity.com .
https://unity.com/ .
[Online; accessed 26-02-2024]

[9] Unity - Scripting API: GameObject .
https://docs.unity3d.com/2021.3/Documentation/ScriptReference/GameObject.html .
[Online; accessed 15-01-2024]

[10] Unity - Scripting API: Rigidbody .
https://docs.unity3d.com/2021.3/Documentation/ScriptReference/Rigidbody.html .
[Online; accessed 12-01-2024]

[11] Unity - Scripting API: Collider .

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/Collider.html .

[Online; accessed 06-02-2024]

[12] Unity - Scripting API: Component .

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/Component.html .

[Online; 15-02-2024]

[13] Input lag - Wikipedia .

https://en.wikipedia.org/wiki/Input_lag .

[Online; accessed 06-02-2024]

[14] Pathfinding - Wikipedia .

https://en.wikipedia.org/wiki/Pathfinding .

[Online; accessed 06-02-2024]

[15] Unity - Scripting API: CharacterController .

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/CharacterController.html .

[Online; accessed 12-01-2024]

[16] Unity - Scripting API: NavMeshAgent

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/AI.NavMeshAgent.html .

[Online; accessed 12-01-2024]

[17] Unity - Manual: Navigation and Pathfinding .

https://docs.unity3d.com/2021.3/Documentation/Manual/Navigation.html .

[Online; accessed 06-02-2024]

[18] Unity - Scripting API: NavMesh .

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/AI.NavMesh.html .

[Online; accessed 15-02-2024]

[19] Unity - Manual: Nav Mesh Obstacle .

https://docs.unity3d.com/2021.3/Documentation/Manual/class-NavMeshObstacle.html .

[Online; 06-02-2024]

[20] Real-time strategy - Wikipedia .
https://en.wikipedia.org/wiki/Real-time_strategy .
[Online; 06-02-2024]

[21] Unity - Manual: Animation system overview .
https://docs.unity3d.com/2021.3/Documentation/Manual/AnimationOverview.html .
[Online; accessed 07-02-2024]

[22] Unity - Manual: Blend trees .
https://docs.unity3d.com/2021.3/Documentation/Manual/class-BlendTree.html .
[Online; accessed 07-02-2024]

[23] Animancer - Unity Asset Store .
https://assetstore.unity.com/packages/tools/animation/animancer-pro-116514 .
[Online; 10-01-2024]

[24] Unity Asset Store .
https://assetstore.unity.com/ .
[Online; accessed 07-02-2024]

[25] What is a keyframe? - gamedesigning.org .
https://www.gamedesigning.org/animation/keyframe/ .
[Online; accessed 10-01-2024]

[26] Blender project - blender.org .
https://www.blender.org/ .
[Online; accessed 15-02-2024]

[27] Games - Bare Mettle Entertainment .
https://www.baremettle.com/games/ .
[Online; accessed 07-02-2024]

[28] Unity - Manual: Animator Controller .
https://docs.unity3d.com/2021.3/Documentation/Manual/class-AnimatorController.html .
[Online; accessed 08-02-2024]

[29] Unity - Manual: Animator Override Controllers .
https://docs.unity3d.com/2021.3/Documentation/Manual/AnimatorOverrideController.html .
[Online; accessed 08-02-2024]

[30] Unity - Manual: Skinned Mesh Renderer .

https://docs.unity3d.com/2021.3/Documentation/Manual/class-SkinnedMeshRenderer.html .

[Online; accessed 08-02-2024]

[31] Unity - Scriping API: DontDestroyOnLoad .

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/Object.DontDestroyOnLoad.html .

[Online; 09-02-2024]

[32] Game Programming Patterns - Singleton .

https://gameprogrammingpatterns.com/singleton.html .

[Online; 09-02-2024]

[33] Unity - Manual: Terrain tools .

https://docs.unity3d.com/2021.3/Documentation/Manual/terrain-Tools.html .

[Online; accessed 09-02-2024]

[34] Unity - Manual: Audio Source .

https://docs.unity3d.com/2021.3/Documentation/Manual/class-AudioSource.html .

[Online; accessed 09-02-2024]

[35] Doxygen .

https://www.doxygen.nl/ .

[Online; accessed 13-01-2024]

[36] Unity - Manual: Mask .

https://docs.unity3d.com/2021.3/Documentation/Manual/AnimationMaskOnImportedClips.html .

[Online; accessed 12-02-2024]

[37] Unity - Manual: Streaming Assets .

https://docs.unity3d.com/2021.3/Documentation/Manual/StreamingAssets.html .

[Online; accessed 14-01-2024]

[38] Unity - Manual: TextMeshPro .

https://docs.unity3d.com/2021.3/Documentation/Manual/com.unity.textmeshpro.html .

[Online; accessed 14-01-2024]

[39] Unity - Manual: CharacterController .

https://docs.unity3d.com/2021.3/Documentation/Manual/class-CharacterController.html .

[Online; 12-01-2024]

[40] Unity - Scripting API: Mathf.SmoothDamp .

https://docs.unity3d.com/2021.3/Documentation/ScriptReference/Mathf.SmoothDamp.html .

[Online; accessed 12-01-2024]

[41] Unity Forum - Player jumping on NavMeshAgent .

https://forum.unity.com/threads/player-jump-not-allowed-on-navmeshagent.718364/ .

[Online; accessed 12-01-2024]

[42] Unity - Manual: Animation States .

https://docs.unity3d.com/2021.3/Documentation/Manual/class-State.html .

[Online; accessed 20-02-2024]

[43] State Machine Transition interruptions - blog.unity.com .

https://blog.unity.com/engine-platform/state-machine-transition-interruptions .

[Online; accessed 14-01-2024]

[44] Unity Forum - Change transition settings at runtime .

https://forum.unity.com/threads/change-transition-settings-at-runtime-with-script.544062/ .

[Online; accessed 10-01-2024]

[45] Unity Forum - Change the transition duration between two animation by scripting .

https://forum.unity.com/threads/change-the-transition-duration-between-two-animation-by-scripting.397939/ .

[Online; accessed 10-01-2024]

[46] UnityEditor assembly in build - stackoverflow.com .

https://stackoverflow.com/questions/67298578/whats-the-point-of-using-unityeditor-assembly-if-you-cant-build-project .

[Online; accessed 10-01-2024]

[47] Unity - Manual: Animations FAQ .

https://docs.unity3d.com/2021.3/Documentation/Manual/MecanimFAQ.html .

[Online; 14-01-2024]

[48] Dark Souls - Bandai Namco Entertainment .
https://en.bandainamcoent.eu/dark-souls .
[Online; accessed 12-02-2024]

[49] Unity - Manual: Generating lighting data .
https://docs.unity3d.com/2021.3/Documentation/Manual/UsingPrecomputedLighting.html .
[Online; accessed 12-02-2024]

[50] Unity - Scripting API: AudioSource .
https://docs.unity3d.com/2021.3/Documentation/ScriptReference/AudioSource.html .
[Online; accessed 18-01-2024]

[51] Unity - Manual: Layer-based collision detection .
https://docs.unity3d.com/2021.3/Documentation/Manual/LayerBasedCollision.html .
[Online; accessed 18-01-2024]

[52] 7zip.org .
https://www.7-zip.org/
[Online; accessed 22-01-2024]

[53] Field of view - Wikipedia .
https://en.wikipedia.org/wiki/Field_of_view .
[Online; accessed 22-01-2024]

[54] Fisheye lens - Wikipedia .
https://en.wikipedia.org/wiki/Fisheye_lens .
[Online; accessed]

[55] Unity - Manual: Constraints .
https://docs.unity3d.com/2021.3/Documentation/Manual/Constraints.html .
[Online; accessed 07-02-2024]

[56] Game Programming Patterns - State .
https://gameprogrammingpatterns.com/state.html .
[Online; accessed 14-02-2024]

[57] Unity - Scripting API: Animator.CrossFade .
https://docs.unity3d.com/2021.3/Documentation/ScriptReference/Animator.CrossFade.html .
[Online; accessed 10-01-2024]