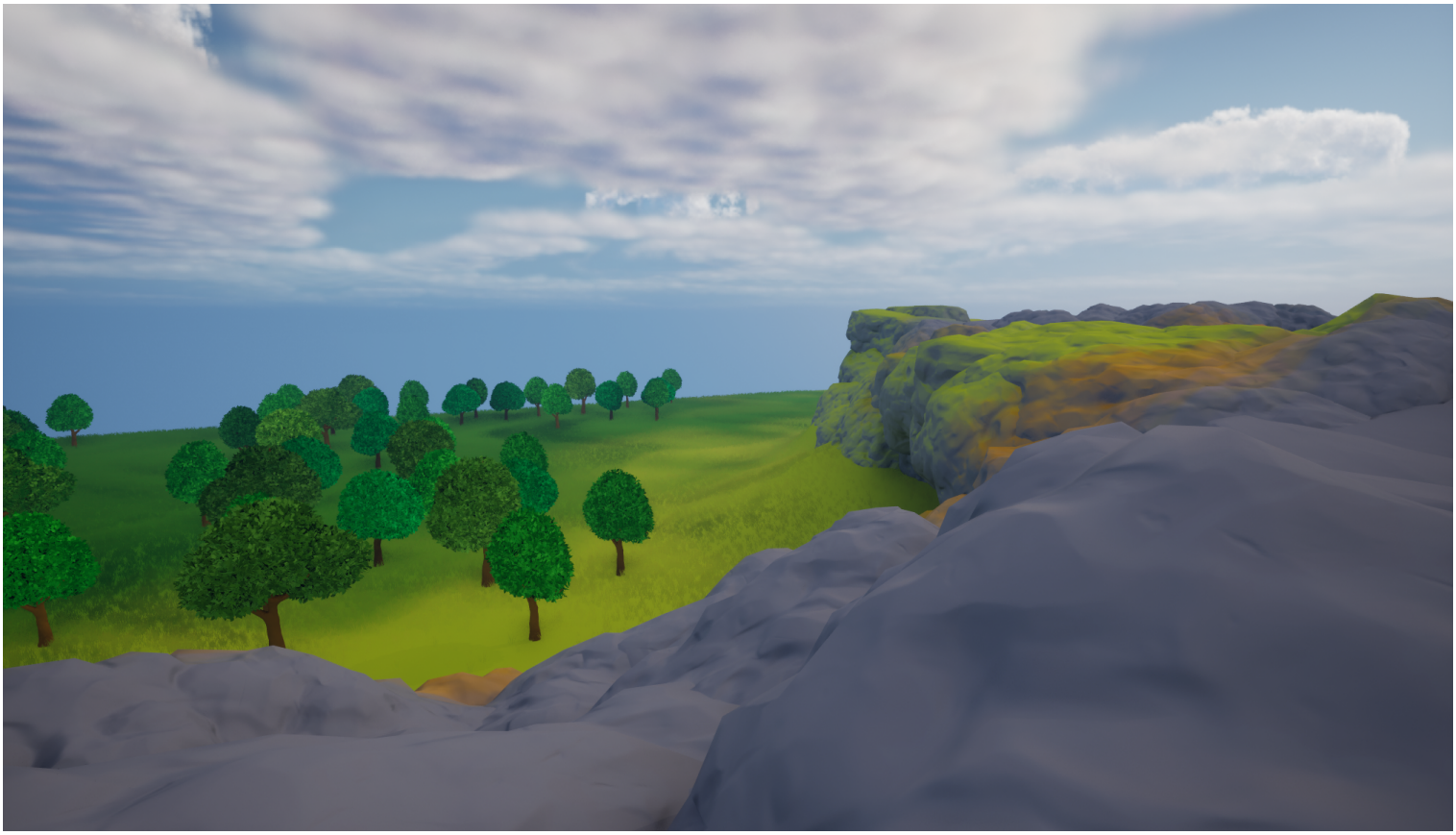
Kantonsschule Im Lee Winterthur

Maturitätsarbeit HS 2023/24

COMBAT DESIGN IN A 3D OPEN WORLD VIDEOGAME



Raffael Scagnetti (4f)

Supervised by: Thomas Graf

Winterthur, 8.1.2024

Contents

Part I.

Prologue

1. Introduction

In the last decades, the videogame industry has grown enormously. Today, a high number of games get released by large studios every year trying to satisfy the huge demand for games. There is also a large number of small teams or even solo creators developing and releasing their own game. This leads to a market where players have a wide choice. That means that a game must meet the expectations of a player exactly and provide a promising game experience to be successful. Creating such a game is time-consuming and difficult even for large studios and generally requires massive effort. There are however differences in the level of difficulty and complexity depending on the type of the game to be created: creating a massive multiplayer online game (MMO) for example requires significantly more time and effort than creating a simple offline puzzle game like Tetris. One of the main reasons for this is that a MMO requires a much higher amount of mechanics and functionalities than a puzzle game. An open world game is another example of a game type requiring a massive work to create, since the premise of such a game is that the player can make their own decisions on how to move through the game's world. Because I am personally fascinated by open world games, I decided to work on such a game as my Maturaarbeit. Of course, creating a complete game doesn't fit in the limited time of a Maturaarbeit. Thus this work focuses on one key aspect that significantly influences the overall gameplay experience of an open world videogame: the combat design.

Combat design focuses on how the player experiences encounters with opponents inside the game world. Although combat is not necessarily a part of every open world videogame, most games include this mechanic. Often, the goal of combat is to reinforce the feeling of a real world and at the same time test the player's skills by challenging them. Required skills often include timing and strategy. Combat can also be used to give the player a certain feeling, like Santa Monica Studio's "God Of War" games, which want the player to feel powerful and therefore encourage offensive actions. Other videogames like FromSoftware's "Dark Souls" games want the player to think more strategically and thus discourage overly offensive behavior and rewards a considered style of play.

1.1. The Goal of this Work

The overall goal of this work was to create the combat elements of a 3D open world videogame. The combat system should be easy to understand and get used to and make the player feel powerful while also allowing skill expression. The game should also include a small world with different terrains to test the mechanics. In detail the goal of this work is to create a combat system that:

- is easy to understand (intuitive for averagely experienced gamers)
- gives the player the required support to maintain control and confidence in their abilities
- consists of mechanics and behaviors that feel natural
- makes the player feel powerful
- allows players to improve their gameplay when playing more skillfully
- facilitates varied gameplay by allowing the player to execute a number of actions
- includes various opponents which interact with the player and can attack them
- makes combat interesting by having distinct opponent characters with a variety of different attacks

1.2. Motivation

I have been interested in computers and technical aspects since I was four years old. As a child, I liked producing simple Scratch programs. Later I started programming small snippets with Node.js. Then I switched to programming a Raspberry Pi until I became interested in videogame development. Here, both my passions for complicated board games and computers meet. Complicated board games always held a great fascination for me. I would often attempt to improve the games I liked by adding interesting new rules to the point where no one in my family would even have a chance of understanding them. In a videogame it is however possible to create an incredible amount of rules without the need of the player to understand them all in detail as the computer handles the rules and the player only has to understand how they affect the game. This fascination led me to do my Maturaarbeit on the topic of creating a videogame.

1.3. The Contents of this Work

This work will include discussions of design problems and of how they were solved in the practical part of this work as available on GitHub (R. Scagnetti 2024). The project contains over 9000 lines code written over the course of this work. This number does not even include the significant work done inside the used framework's visual editor. Due to the limited size of this text, it was decided to not include the code itself in the text but make it accessible on GitHub.

The exemplary code shown in figure 1.1 is part of the functions used to change the character's appearance dependent on its distance to the camera. As explained later on page 17 this turns the character invisible when the camera is close to it and fades it in at a greater distance. This however is only a part of the required setup. Corresponding work is required in the editor specifically implementing the actual change of the character's visibility based on the value generated here. Explaining this functionality in detail would involve not only a comprehensive discussion of the intention of the code but also a basic explanation of the functions and classes provided by the framework. To limit my thesis to a feasible length, the work will focus on discussing the problem which the code solves and showing the conceptual idea and the basic structure of the solution.

```
//this function gets called every frame
void AGeneralCharacter::Tick(float DeltaSeconds)
{
    Super::Tick(DeltaSeconds);
    FadeMeshWithCameraDistance();
}

void AGeneralCharacter::FadeMeshWithCameraDistance()
{
    if(!IsValid(CameraPlayerController))
    {
        CameraPlayerController = GetWorld()->GetFirstPlayerController();
        if(!IsValid(CameraPlayerController)) return;
    }
    FVector CameraLocation;
    FRotator CameraRotation;
    CameraPlayerController->GetPlayerViewPoint(CameraLocation, CameraRotation);
    const float OldOpacity = GetMeshesOpacity();
    const float Distance = FVector::Distance(CameraLocation, GetActorLocation());
    if(Distance > MaximumFadeDistance)
    {
        if(OldOpacity != 0)
        {
            SetMeshesOpacity(0.f);
        }
    }
    else if(Distance < MinimumFadeDistance)
    {
        if(OldOpacity != 1.f) SetMeshesOpacity(1.f);
    }
    else
    {
        const float DesiredOpacity = 1.f -
            pow((Distance - MinimumFadeDistance) / (MaximumFadeDistance -
    MinimumFadeDistance),InputFadeStrength);
        if(abs(DesiredOpacity - OldOpacity) >  0.001) SetMeshesOpacity(
    DesiredOpacity);
    }
}

float AGeneralCharacter::GetMeshesOpacity() const
{
    return GetMesh()->GetCustomPrimitiveData().Data.IsEmpty() ? 0.f : GetMesh()->
    GetCustomPrimitiveData().Data[0];
}

void AGeneralCharacter::SetMeshesOpacity(float DesiredOpacity)
{
    GetMesh()->SetCustomPrimitiveDataFloat(0, DesiredOpacity);
    for(USkeletalMeshComponent* MeshComponent : RelevantMeshes)
    {
        MeshComponent->SetCustomPrimitiveDataFloat(0, DesiredOpacity);
    }
}
```

Figure 1.1.: Extract of the code used in-game to fade the character's visibility based on their distance to the camera

Part II.

Creating the Game

2. Game Theory

Before creating a game, it is important to first understand some basic things about the creation of a game. This will make the creation process more efficient and target oriented.

2.1. Gameplay design

The main focus of this work is combat design, which is one part of a videogame's full `gameplay` design. Talking about the quality of a videogame, one of the most commonly cited reasons for why a game is perceived as good or bad, is how its gameplay feels. Depending on who you ask, the exact definition of gameplay may vary. But most people would agree that, in essence, gameplay is how the player interacts with the game. This includes the situations the player is put in as well as the available options to interact within the game (Tayler 2023). Furthermore, it is also strongly influenced by how a player chooses to use the given tools to solve a problem at hand, as there can be a multitude of valid answers to a situation depending on the game. Even so, it can be seen in many existing games that players generally tend to use the most efficient seeming solution to a problem posed by the game when given the choice (even if that option may not be the most entertaining thus resulting in a worse gameplay experience).

Gameplay design is an extremely important part of game design as it strongly influences the players' reception of the game. And it is thus extremely important to understand how the gameplay can be changed and adapted by the designer of the game. One of the main factors influencing gameplay are the game's `mechanics`. In essence, mechanics are a game's functionalities. A game's mechanics show in many different aspects. These range from how the game reacts to player inputs to the subtle techniques a game might use to get you to play the game in a certain way. All these mechanics are ways to directly influence the gameplay of a videogame. Often, designers have a specific gameplay experience in mind when creating a game and develop its mechanics in a way so the game will produce exactly that experience. Sometimes, games trying to belong to a certain genre might implement mechanics to make them feel more intuitive when compared to other titles of the same genre.

Since a certain mechanic is only one of hundreds of different solutions to a given design problem and a videogame consists of many different mechanics, this is a big part of what makes a game unique. But it is also an important criteria for categorizing games into genres since a group of videogames which have similar mechanics are said to belong to the same genre. For example, the racing game genre consists of games that feature vehicles, one of which is player controlled and distribute rewards for reaching the finish line faster than the other drivers. Whereas role-playing games (RPGs) are expected to include a player character that can fight and/or use other skills to advance in the game, improving the character's abilities.

Thinking about these genre conventions is also important when designing a game: People normally expect games featuring set of mechanics typical for a certain genre to share an overall similar gameplay. But, if the game was not designed with that genre in mind, it can lead to player expectations that are widely different from what the designer actually intended and promised, which in turn often leads to the game being poorly received by the community.

2.2. Game Engines

A complete videogame typically uses a large set of rules, defining how the player interacts with the game, what he can and can not do as well as how different objects inside the game interact. While some of those rules are specific to the game or the genre it belongs to, most are generally required. There is a limited number of possibilities how the player can communicate his intentions to the device running the game. Furthermore, most games that implement object interactions seek to simulate the real world behavior of physical objects. Reproducing real world behavior mostly includes the simulation of gravity, friction, the notion of impulse and energy preservation. Furthermore, all computer games provide some output to the player, which is typically done by showing the game on a screen. Although creating these basic functionalities by yourself is possible, it would require a large amount of time and work. This effort is not in the interest of a game developer, since he wants to focus and spend most time on the functionalities making his game unique.

Game engines aim to solve exactly that problem: they are program frameworks that provide the central elements used by most videogames such as rendering, input devices, object collision, gravity and many more. Using game engines dramatically speeds up the process of creating a new game. Furthermore game engines often come with a suitable visual development interface and a level editor that simplifies

putting together the levels (or in case of this work the world) for your game. Today, engines are used by almost all game developers. While large studios like Guerilla Games use a proprietary engine like the `Decima Engine`, many of today's games are developed using a publicly available engine, such as Unreal Engine, Unity or Godot. Even though fundamentally providing similar functionalities, every engine has its own specialties, supports different programming languages and has a different pricing model.

Unreal Engine, for example, is created in C++ and therefore mainly uses this language but also provides basic support for python. One of the outstanding features of this engine is that it offers sophisticated tools for photo realistic and high quality rendering. The engine's costs are dependent on the money you are making with your game. Differently, Unity is primarily used with C# and is often praised for its high number of available free extensions and assets as well as being easy to pick up on. Starting in 2024, the licence model changes to a complicated monetization model based on the number of installations of a game. Very differently from the two other examples is Godot. Being released a considerable time after Unreal Engine and Unity (2014) and being completely free and open source, it has slightly less tools but is therefore also easier to get started with (*Godot Engine 4.1 documentation in English* 2023). It technically supports modifications in both C++ and C# but is best used with its own language GDScript (Godot Engine 4.1 documentation in English, 8.10.2023). This being said, all engines can be valid choices to create a new videogame. Evaluation of a game engine should therefore include the preferences of the development team as well as the specific advantages relevant to the planned game.

This work in particular will be based on Epic Games' Unreal Engine. Some of its advantages are its full C++ support as well as a big community, where you can easily find answers to most of your questions. Furthermore, Epic Games' engine includes an incredibly large number of advanced functionalities and systems by default. Plus, the regular updates along with innovative features make it a quite powerful engine. However, this choice also has disadvantages, the most important being the lack of official documentation about non-trivial functionalities.

2.3. Unreal Engine: Blueprints or C++

The Unreal Engine provides many features aimed at making the development process of games more easy. One of the ways Epic Games simplifies the development process is by allowing creators using the Unreal Engine to access a powerful visual editor. This editor allows for visual code creation as well as using custom C++

classes. C++ Classes created by a game developer can specify public and protected variables as being a **UPROPERTY()**. On compilation additional code is generated internally for such variables which allows the engine to access these variables. Using this macro as such on pointers to **UClass**es (all unreal engine objects are children of **UClass**) allows the engine to manage the content and garbage collect it. The **UPROPERTY** macro can also be used for many other reasons by adding further arguments to the macro. Developers for example use **UPROPERTY(EditAnywhere)** to make any variable show up in the visual editor. In the engine the object can then be placed inside a level. An Unreal Engine level is the space where the game takes place and can contain the entirety of an in-game-world or simply a small part of it. Values of the object marked with **EditAnywhere** can then be seen and edited both when making changes to the level as well as when testing the game inside of the editor. Other specifiers allow setting unit suffixes for values, limiting the value to a range with a maximum and a minimum and many more.

The Unreal Engine allows for even more visual customization of classes through `blueprints`. A `blueprint` can be a child of any Unreal Engine based class (all **UClass**es) and is an in-editor representation of it. In many ways, they act exactly like a native C++ class. They have associated variables that can be public or protected and have a default value. They can have internal logic and even functions defined simply by connecting different blocks. Each of those blocks itself represents a blueprint or C++ implemented function or operation. This allows creating most functions possible in C++ also in the engine's visual language. Blueprints can also not only be based on C++ classes but also other blueprint classes. Using these visual Unreal Engine constructs is generally slower as compared to pure C++ but they seem undoubtedly easier to understand more convenient to use. This raises the question which of them you should use and if both, which one for what purpose.

As often, there are different answers to this question. In this work, C++ will be used for creating all basic game mechanics and interactions. This also includes all of the more complex and computationally expensive calculations. It allows the usage of some additional functions and tricks that are not possible in blueprints. This also gives the option to override more essential engine classes to modify things like the editor's interface. And it also results in a slightly faster execution of the more demanding aspects of the game.

Blueprints on the other hand will be used to easily and quickly expand on the game mechanics and interactions defined in C++. This mostly includes graphical design but also means setting the exact values of objects' and characters' relevant proper-

ties. In a multi-person development team, this allows for designers to focus on the design and ignore the technical implementation and logic written in C++. That is useful as blueprints do not allow for game breaking code while C++ definitely does so and designers are often not trained to use C++ properly. Using blueprints also allows for much faster iteration when tuning values, as C++ code always has to be compiled which can potentially require a lot of time while blueprints do not. Sticking to such a defined principle when creating the code generally has the benefit that you have to think about why a certain way of implementing something is preferable to another which often results in better code.

3. Character Templates

When creating an Unreal Engine 5 C++ project with the Third Person template and using the unreal Engine starter content, the project opens a simple template world as seen in image 3.1. The template comes with a default Unreal Engine character. When simulating the game, this character will automatically be used as the player character. This means the character is placed in the empty world at the location of the controller icon as seen in the image. The character can then be controlled using the default key bindings on a keyboard, controller or using a touch screen. This default character lets the player jump, walk around and rotate the camera at a normally fixed distance around it. Since this work will use the models from the *Quadruped Fantasy Creatures* 2018 pack as both player and opponent characters, the first thing to do is replace the template character.

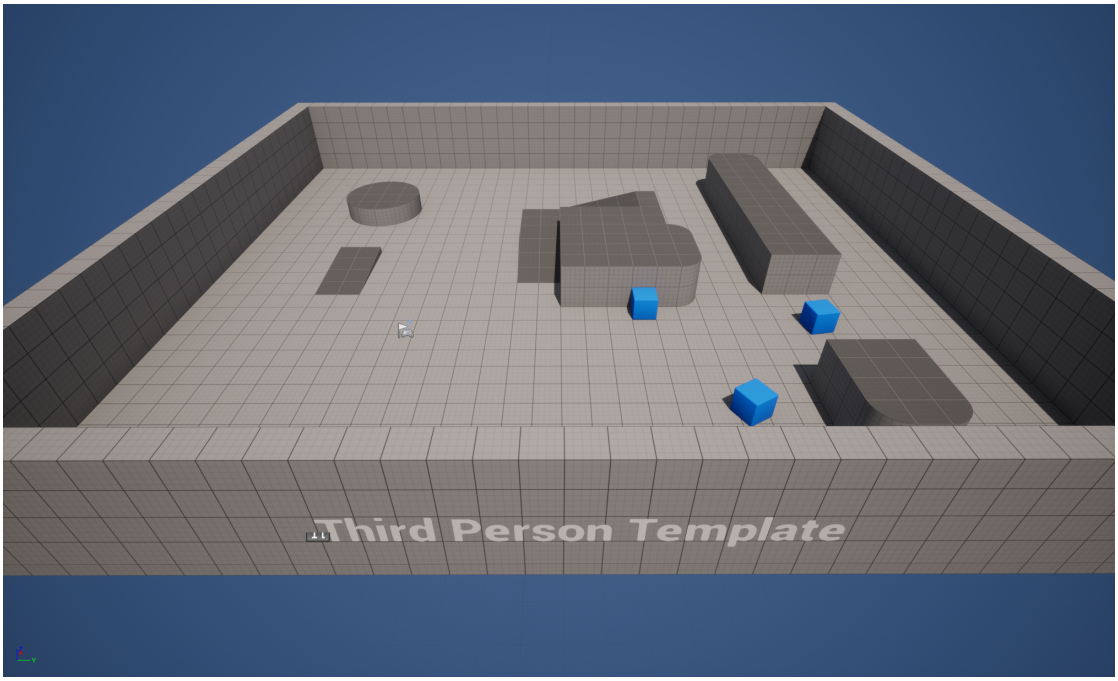


Figure 3.1.: Unreal Engine's Third Person template opens to this default world. In the center left, a small controller icon can be seen, signifying the player's start location.

Before creating the player or opponent character, it makes sense to create a base class for all characters. There are multiple advantages to having one common parent structure for non-player characters (NPCs) and players. The main one is that this allows you to define systems and rules for all characters at once instead of copy-pasting the code multiple times.

For this game, it even makes sense to create an additional class in between the general character and the players and opponents. The reason for this is that although this game is focused on combat, there should still be room left for creating some NPC's relevant mainly relevant for story telling purposes not capable of combat at all in the future. Having one parent structure in common for all characters and a separate for characters capable of combat is useful to assign the combat related functions to combat participants only.

3.1. A General Character

Unreal Engine provides many functionalities for characters such as movement, collision handling and animations in their `ACharacter` class. Most of these functions are specified as virtual and can therefore easily be overridden to fit the more specific needs of a project.

3.1.1. Camera behavior & Using Character's Eyes

For a combat focused third person game, the default camera behavior of Unreal Engine's cameras is often not ideal. The engine uses a simple system with a camera attached to the player's model to determine what the player sees. If any object is between the model and the camera, the camera will automatically move closer to the player so the object is not in view anymore. This is useful to prevent the camera from clipping into walls or the floor but it does not work very well with moving objects close to the player. Opponents in combat are exactly that, which forces the camera to change distances often and quickly.

A simple solution is to disable camera collision on all character models, which would lead to the camera not detecting other characters as objects in its way. This however leads to a new problem. What should the game display if the camera comes to rest withing of a character model? The default behavior is that all pixels of the model that are looked at from inside the midel become completely invisible while the pixels that are looked at from outside still show. This leads to awkward behavior as seen in image 3.2.

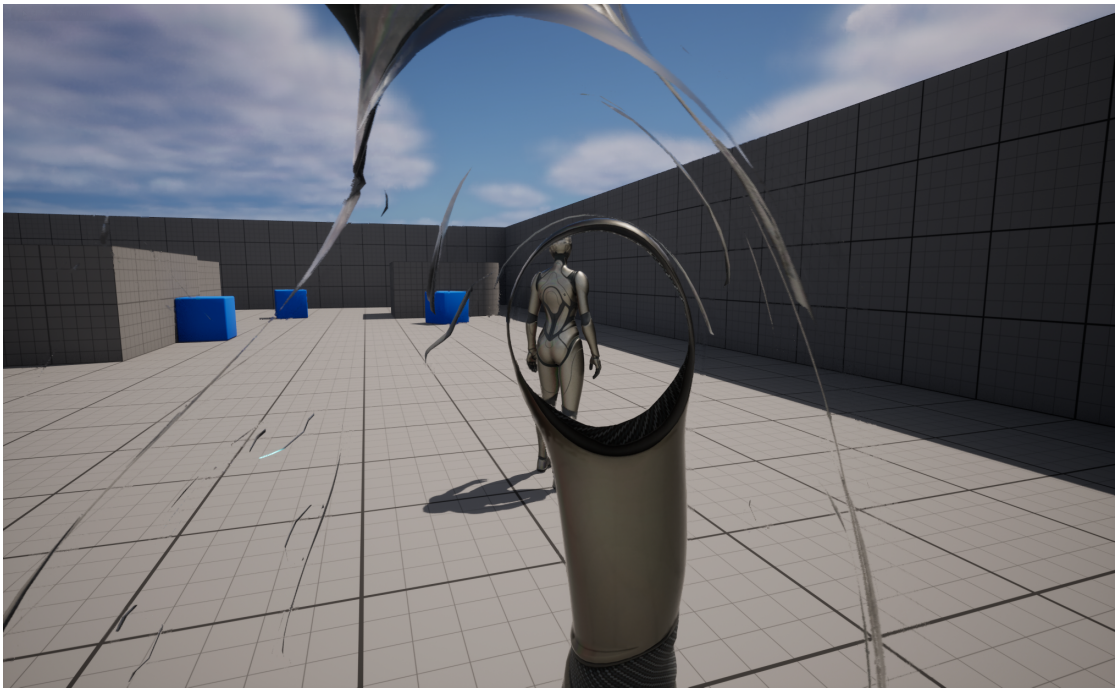


Figure 3.2.: This is how camera clipping looks with the default third person character

This strange side effect can again easily be prevented by fading the model's visibility based on the camera's distance to the model, so that the model is completely invisible when the camera is close and slowly fades in with increasing distance from the camera. An efficient way of doing this is calculating the camera's distance from the character every time an image is rendered. Depending on the distance defined for the character inside the editor, a value between one and zero is generated. Zero means that the character is fully shown and one means the character is fully invisible. This value can then be used accessed in the object responsible for the character's appearance to change the character's visibility accordingly.

When making a game that uses sight detection, overriding the provided **`GetActorEyesViewPoint`** function in the general character is another very useful point to consider. This function is used by the engine to get the location of the character's eyes as well as the direction of their view. By default the location is tied to the general position of the character and direction absolute to the world is used. This is far from where the eyes of a character are expected to be. Overriding the function to get the location and rotation based on a socket placed on the character's mesh in-editor gives you the expected properties and follow the character's eyes.

3.1.2. Limiting Inputs

There are some other systems that should be considered to be implemented in the general character class depending on what non-combat-NPC's should be able to do. In my game, all characters should be affected by combat events such as status effects and disabling of certain abilities (such as walking) for a limited amount of time. The only difference between a non-combat-NPC and an opponent in this game is the ability of opponents to deal and take damage.

The first of those mentioned systems is the option to prevent a character from executing certain actions. For example, characters might be given a status effect that forces them to walk instead of run or stops them from jumping. Other use cases include preventing a player from executing ground attacks while in air or interacting with the world while they are staggered or dying. A stagger is a reaction emphasizing the impact of a received attack as explained in chapter 3.2.2. For the movement-related functions, Unreal Engine provides a set of movement properties to achieve exactly that. But expanding or changing this system directly is not possible but often useful. A solution is therefore to create a system managing all the actions that can be limited so that everything is managed the same way.

The implementation used in this project consists of a structure containing mainly one integer value. This value stores whether each input type is currently allowed to execute or not as one of its bits. Each of the input types as defined in the code seen in figure 3.3 is therefore associated with a the position of its value in the integer. This value uses Unreal Engine's option for the editor to interpret the values as `Bitfalgs` as explained in Ben Humphreys article about **`UPROPERTY`**s (Humpherys 2016). That interpretation of the value allows the engine to efficiently handle and display the options in its editor user interface. Also this way of storing the values allows easily expanding the system by adding more input types. The following code snippet shows the definition of the input types. The types Undefined, Death, Force and Reset are marked as not visible in the editor. These are intended strictly for use in specific situations which are defined inside the C++ code. Processing limitations of input is either done by applying a limitation for a limited time or by applying it indefinitely until it is removed explicitly. A limit contains three pieces of information:

- the type of input the limiter itself was triggered by
- the inputs that will be allowed while the limit is applied
- the duration for which the limit will persist (zero if it should persist indefinitely)

```
UENUM(meta=(Bitflags, UseEnumValuesAsMaskValuesInEditor="true"))
enum class EInputType : int32
{
  Undefined       = 0 UMETA(Hidden),
  Walk            = 1 << 0,
  Sprint          = 1 << 1,
  Camera          = 1 << 2,
  Jump            = 1 << 3,
  Attack          = 1 << 4,
  SwitchOut       = 1 << 5,
  Stagger         = 1 << 6,
  HeavyStagger    = 1 << 7,
  Death           = 1 << 8 UMETA(Hidden),
  Force           = 1 << 9 UMETA(Hidden),
  Reset           = 1 << 10 UMETA(Hidden)
};
ENUM_CLASS_FLAGS(EInputType)
```

Figure 3.3.: Extract from the code of the game defining the different input types

This allows attacks to simply store the limit information and applying them on execution. It can also be used to prevent the character from attacking or moving after being hit by a strong attack to emphasize the impact.

3.1.3. Status Effects

When applied to a character, status effects can change the behavior, internal values and sometimes the look. They can be represented by a simple **`UActorComponent`**. These components are Unreal Engine objects that can be attached to any object. The engine uses the **`AActor`** class to represent the most general independent object possible. Similar to all other classes marked with the prefix "**`A`**", the **`ACharacter`** class also inherits from this general object class. Being associated with exactly one object at the time and having a simple way of detection built into the engine, components are a suitable representation for status effects. The base version for status effects is fairly simple. It essentially contains a variable referencing the target of the effect, a thumbnail to represent the status effect and a value representing the effect duration. A status effect is applied to a character by specifying the type of the effect in the **`ReceiveStatusEffect`** function. The status effect component then gets spawned and attached to the character. If the same type of effect is already active on the character, the effect will be restarted rather than having two instances of the same effect simultaneously. If the effect was newly applied, the character then calls a blueprint overridealbe function. Removing the effect, another blueprint overrideable function gets called. The fully blueprint customizable nature of these two functions allows all the specifics of the status effect to

be defined inside the exact status effect blueprint. This makes it easy and fast to implement new status effects and tune their exact effects. It also corresponds with the principle of writing code in both blueprints and C++ according to their function.

In this game, the general character also includes a death function, which could theoretically be moved to the fighter character. However, the only property needed in this function that is not already present in the general character is the information about the animation to be played on death. All other required functions are already present. This means adding the function here is possible. Plus, it might be useful also for non-combat-NPCs to have this functionality when they are finally used in game.

3.2. A Fighter Character

The fighter character expands the functionalities of the general character by adding functionalities specifically focused around combat, most important of all the ability to take and cause damage.

3.2.1. Character Stats

To implement these abilities, multiple values which are called `stats` are required. The combat relevant ones are stored along with the character's walk speed and its run speed in the `FCharacterStats` structure. The most generic values used for combat (health (HP), attack (atk) and defense (def)) are largely self-explanatory: The damage ΔHP_2 dealt by character 1 to character 2 can be described as:

$$\Delta\text{HP}_2 = -\frac{\text{atk}_1 * \frac{\text{atkDmg}_1}{100\%}}{\text{def}_2} \tag{3.1}$$

The atkDmg_1 represents the damage multiplier number as defined by the percentage number in the executed attack of character 1. There are also two other combat relevant stats in this game, that are less common and whose meaning often varies between games: `toughness` and `interruption` resistance. The interruption resistance is given as the probability in a thousand attempts that a stagger triggered by an attack is successful. The toughness value represents a similar but independent value to the health points: when a character is hit by an attack, the toughness (tgh) decreases by the exact amount defined by the attack's toughness break value as shown in equation 3.2.

$$\Delta\text{tgh}_2 = -\text{atkTghBreak}_1 \tag{3.2}$$

If the character's toughness value reaches zero or lower, a toughness break is triggered. The character then looses 10% of their maximal health points, plays the according animation and gets applied the **`HeavyStagger`**'s default input limits for the complete duration of the animation. Additionally, the character receives a temporary stats decrease (debuff) for the duration of the animation reducing its defense to half of its original value. The implementation varies slightly from player and opponent characters, as loosing control of the character for any amount of time is already punishing enough for a player who does not expect it (which will always be the case as they otherwise would not get hit).

Moving at the Right Speed

Walk and run speeds are separately defined in the character stats. Having different movement speeds for different situations makes characters feel more dynamic. Applied to the player character, this makes movement more interesting and less tedious as they can move faster when it is required or helpful. Applied to an opponent character dynamic movement speeds makes combat more interesting and pursuit scenarios more realistic. Problematically, Unreal Engine characters only know one speed for walking which is called **`MaxWalkSpeed`**. When a player (or the computer for non-player characters) gives a movement input, the character is accelerated by a constant amount until they reach that maximal speed. After accelerating, a default Unreal Engine character has therefore exactly one speed. To change the **`MaxWalkSpeed`**, the fighter character has two functions which set it to the according value as defined in the character stats. This is enough in most cases to make dynamic speeds possible. Theoretically, interpolating between the two speeds when changing between them would result in a slightly better result. But as a maximal acceleration of the character is defined separately anyways, the change will not be instant anyways.

Attack Trees

The character stats also contain information about all the attacks the character can execute. As it is implemented in the fighter character, the attack information should be available in a way useful for both player and non-player characters and easily editable and readable inside the editor. The solution used is inspired by LeafBranchGames' tutorial video *How to make combo attacks - Beginner tutorial Unreal Engine 5* 2022. In this tutorial, LeafBranchGames uses the Unreal Engine's `state machine` found inside each character's `animation blueprint`. Animation blueprints are built to allow context sensitive changes in the character's animation. Such context sensitive changes might be playing the idle animation when not moving and the run animation when moving fast. One of the tools used to make such

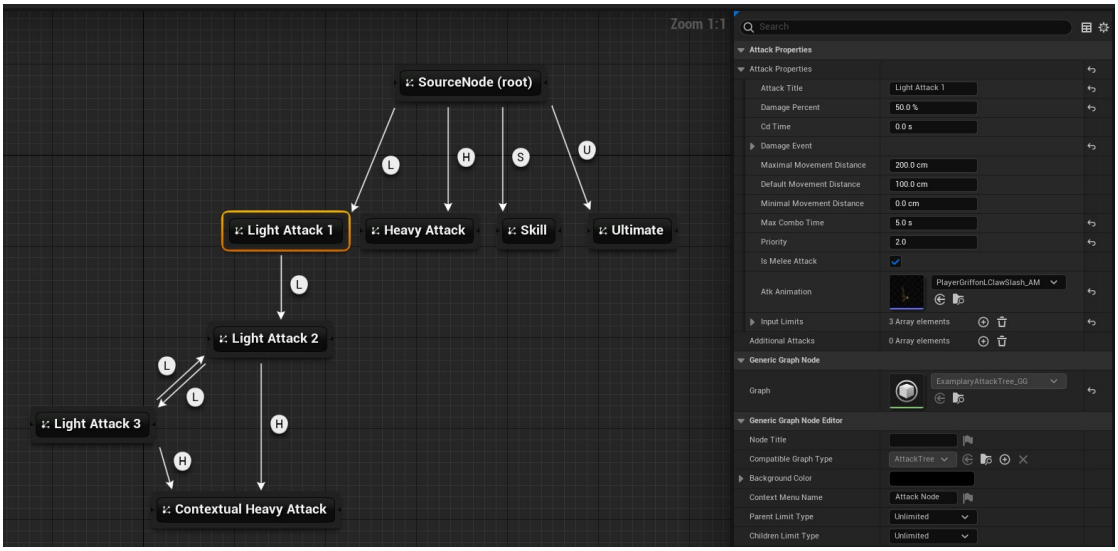


Figure 3.4.: An exemplary attack tree for a player character; The letters L, H, S and U represent the inputs expected from the player to execute the related attack. On the right, all the properties of the selected node are shown.

state based animations easily readable is the state machine. LeafBranchGames used the visual tree-like interface inside the `animation blueprint` to represent the attack animations. This implementation is easily readable and maintainable and even allows complex combo chains. The problem with using this system in the game produced for this work is that the nodes in the state machine can only be used to store animations and no other information such as the attack's damage multiplier.

Using the third party genericGraph C++ plugin (jinyuliao 2023), a fully custom graph can be created that allows setting any type and number of properties. This also makes the attack management simpler since the state machine was used only for the interface and the transitions from one node to another are not optimized for such use. The genericGraph plugin allows the creation of fully custom tree-based interfaces in C++ by allowing programmers to override both the nodes and the connections between them as seen in image 3.4. Each node represents an attack including all its information. Relevant mainly for player character, the links between them can be given indices between zero and three to determine the input needed for the system to use the link. These indices relate to different attacks triggered by the player. When the player presses a key to execute an attack, the index representing the pressed key is passed to the `FAttacks` structure. this structure exists

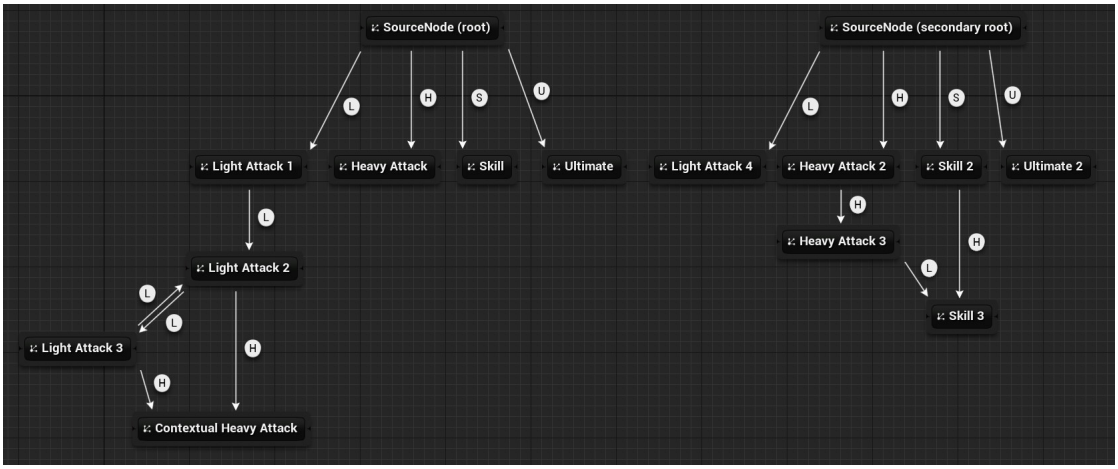


Figure 3.5.: An exemplary attack tree for a player character using two root nodes to signify different states

to coordinate the usage of all attacks from the same character. Given an index, it will determine which attack this input should result in. For this, the context of the attack is determined first: If the **`MaxComboTime`** of the last executed attack has not been exceeded, the branches of the last attack's node are considered first. To check which one if any of the branches should be executed, the system checks for each connection starting from the current node if its index matches the one given. If a valid connection is found, the corresponding attack node is executed. If the **`MaxComboTime`** has been exceeded or there was no branch marked with the given attack index on the first context node, the same process as described above is repeated with the root node of the tree as the context node. In such a tree, the root node is the lowest node of the tree and has no connections going to it but only connections going from it to other nodes.

In some action games, there are abilities that do not primarily apply damage to an opponent or apply status effect to the executing character or their target but change all of the attacks available to the character. For example, an opponent might have an aggression mode which will change all of their animations to look more aggressive and the character's attacks to do more damage. One option allowing such changes is to allow multiple root nodes to exist in one attack tree. To still be able to tell them apart, the root nodes have a separate class from the other nodes. This allows them to set their unique identifier and set one of them as the default root node. This will result in an attack tree looking like the one shown in image 3.5. When an ability such as the above mentioned is triggered, the current identifier is changed which results in a switch of the root node. This

enables fully switching between two sets of attacks. Such a full switch will also enable the player to do all the attacks in their arsenal as these attacks naturally have different cooldowns form the ones connected to the other root node. An attack's cooldown is the time required for the attack to be executed consecutively. It is also possible to have some attacks connected to both root nodes. This means that the cooldown will be kept between the different root nodes which can sometimes also be desired.

In some cases, the ability might not change all attacks, but only a selected number. For example, an ability might improve all normal attacks but have no effect on heavy attacks, skills and ultimates. This could theoretically be done by switching the root node. But having multiple root nodes inside an attack tree has the potential to make the visual representation cluttered. Therefore each attack also has the option for context sensitive attacks to be added. Such a context sensitive attack is associated with a status effect which will cause the context sensitive attack to execute when the node's attack gets executed. An attack tree using such context sensitive attacks as seen on image 3.6 is also significantly smaller as when trying to achieve the same effect with two attack nodes. Another difference between the two is that context sensitive attacks share their cooldown with the other attacks on the same node. This allows context sensitive attacks to remain on cooldown even if the context changes. That behavior further differentiates the usage of context sensitive attacks from the usage of the root swapping option.

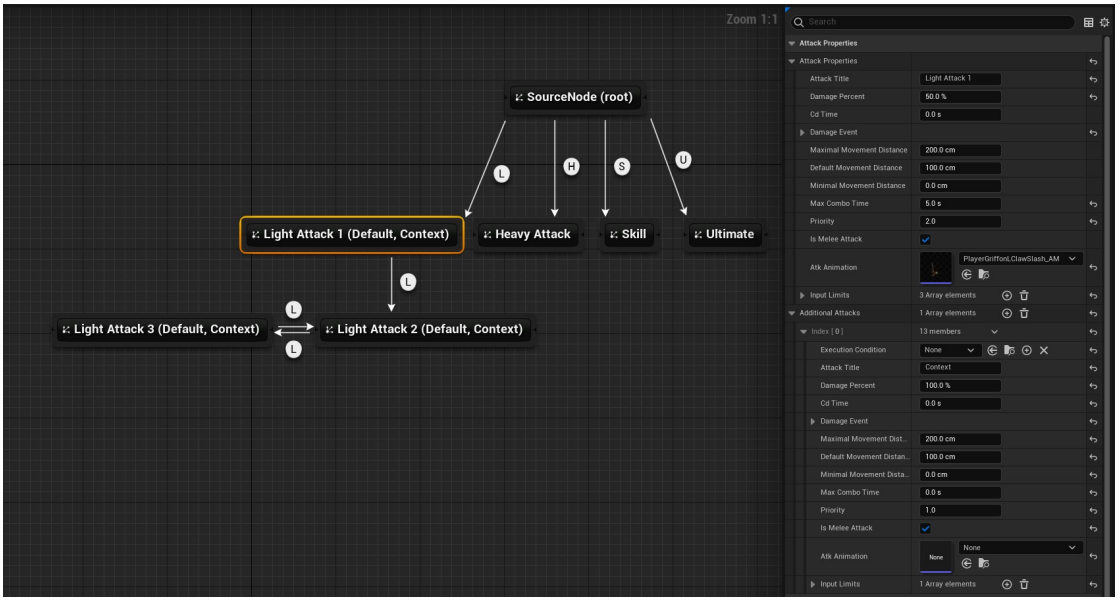


Figure 3.6.: An exemplary attack tree for a player character using context sensitive attacks on all 3 light attacks; On the lower right, the selected node's context sensitive attack properties are shown.

Attack Properties

Every attack in this tree is defined by a variety of values, primarily the damage multiplier and a reference to the animation of the attack. But there are a variety of other defining properties. Most attacks have the executing character move some distance towards their target. As such, the **`DefaultMovementDistance`** represents the distance the character will move when executing the attack in an empty room. To simplify combat, there is also a **`MaximalMovementDistance`** $\geq$ **`DefaultMovementDistance`**. This value defines the distance that the attack can maximally move the character towards their target. Using this value, the character can move a longer distance which allows a higher likelihood of hitting the target. There is also a value representing the cooldown time (**`CdTime`**) of the attack. The cooldown time describes the time before an attack can be used again. An attack which is still on cooldown can therefore not be executed. As the attack generally cannot be used again while it is being executed, there is also a function to get the total cooldown time. This combines the duration of the attack with the cooldown time defined to provide a full overview of how long the attack cannot be used again. Negative values of the cooldown time variable can be used to give the attack a shorter cooldown or even none at all. Also, the **`MaxComboTime`** value used in the attack trees (3.2.1) is defined here, allowing each attack to set a time window for the next attack of the combo to work. This ensures that the time

frame where the next attack of the combo can be executed overlaps about with the duration of the impression the attack created.

There are also a **`Priority`** and a **`bIsMeleeAttack`** value defined for each attack, even though they are only used in non-player code. They are still included in the general attack properties as creating a different type of attack properties for non-player characters would make the code more complex. Simply explained, the **`Priority`** value of an attack compared to the other attacks signifies the attack's likelihood of being executed while the **`bIsMeleeAttack`** value tells the non-player character whether the attack can only cause damage if the character directly touches the player or not.

Attacks also have a number of input limits associated with them. When an attack is executed, some abilities such as walking are blocked for the duration of the attack. This prevents strange and unrealistic behavior and enhances the feeling of impact from the attack. However, applying limits for too long can feel limiting for a player. An attack can therefore add multiple input limits to a queue. This way, an attack can be divided into multiple sections with different input limits. Generally, there are three phases inside an attack animation:

- wind-up (the time the character requires to be ready for the attack without actually causing damage)
- execution (the time the character executes the attack)
- exhaustion (the time the character takes after the attack to return to the original pose)

Using the mentioned system that allows different consecutive limits makes it possible to implement more specific limits. One such example would be preventing the character from attacking in the first two phases and allowing it in the exhaustion phase while still preventing the character from walking during the entirety of the animation.

Technically, this is implemented by saving the input limits in a sorted list which can be viewed and edited from within the editor. This allows defining any number of consecutive limits that are needed for the specific attack. When the attack is executed, the first limit on the list gets applied while the following limits get added to a queue. If the first limit's time ends, the next limit of the queue gets applied. If one of the limits gets overridden from an outside source before their time ends, the queue is aborted. This implementation makes the system flexible and can be easily displayed in the editor.

The remaining properties related to the effect of the attack are stored inside a `FAttackDamageEvent`. This damage event is useful to separate the properties defining how the attack is executed from the ones defining the reaction of the target. The structure inherits the Unreal Engine structure `FDamageEvent`. This structure is an argument of the native **`TakeDamage`** function, whose function is discussed in chapter 3.2.2 of this work. The damage dealing entity generates the relevant data for the attack and passes it to the target along with the damage to be dealt. In the games implementation, the damage receiving character then checks which type of damage event it received and interprets the data accordingly. This can be done by overriding the engine provided standard functions **`IsOfType`** and **`GetTypeID`**. These functions make clever usage of static indices manually defined in each child structure of `FDamageEvent`. When using pointers or references, this usage of virtual functions enables easily differentiating between the different damage types. Using this structure inside the attack properties reduces redundant conversions between types, as the data can be set directly inside the attack definition and then directly passed to the **`TageDamage`** function. Only some of the properties defined in the attack damage event can be edited though. The editable properties are the attack's visual effect scale factor, the chance for the attack to trigger a stagger and the toughness break value of the attack. The usage of these values is explained at the beginning of chapter 3.2.1. The location and direction of the hit are automatically set for each hit and therefore not editable inside the Unreal Engine editor. There is also a delegate function contained in the attack damage event. This delegate is called when the hit connects to give the damage causer the option to react differently when attacks did hit but did not trigger a stagger compared to when they did. This information can then be used to emphasize the impact of an attack that did trigger a stagger as discussed in chapter 3.2.2.

Setting the Stats' Values

When the fighter character gets spawned, the stats are created completely void of useful data as they are not editable inside the Unreal Engine editor. Instead, designers can modify the `FCharacterBaseStats`. These are significantly less complex than the complete `FCharacterStats` as they allow only setting a base value for each stat. All values stored in the full character stats can be divided into a base value, a percentage bonus and a flat bonus. The base value thereby represents a fixed value while the flat bonus and the percentage bonus are modifiers changing the base value by an absolute value or a relative percentage. This results in one combined value that easily accounts for bonuses and changes of the total value through status effects. Allowing to change the simple bonus or the percentage bonus inside the editor would not really make sense then. Furthermore, the properties health and toughness have a current value in addition to their maximal

value which also makes no sense to edit manually.

So after creating the stats object, it has to be initialized with its **FromBase** function. This function takes a `FCharacterBaseStats` object, a set of direct modifiers (such as level) and a reference to the character that will be using these stats. This sets the `FCharacterStats` to the values indicated in the base stats, scaled by the modifiers.

The character stats also include functions for temporarily increasing and decreasing the resulting values of properties using the mentioned bonus values. This is done by passing a `FCharacterStatsBuffs` structure to the corresponding function. This structure contains number a flat and a percentage bonus for each of the properties described in the character's stats. The function can then use this information to change the bonuses for each property.

Displaying the Values

To show the most relevant information about both the player character and the NPCs to the playing person, Unreal Engine's `widgets` are used. Unreal Engine widgets are simple user interface (UI) classes that can be designed in editor using the provided UI elements. More complex elements can be defined in C++ and then automatically show up in the widget editor. In the fighter character's code, only the base class of the stats displaying widgets is referenced to allow the fighter character access to the relevant properties while still allowing the player and opponent characters to use largely different user interfaces. The fighter character class thus is responsible mainly for displaying the character's current health and toughness. For this purpose, the relevant functions for registering health and toughness changes inside the widget are bound to the corresponding delegates defined in the character stats. These delegates then enable the character stats to call the bound functions when the toughness or health values change.

3.2.2. Causing Damage

The functions for health and toughness reduction are also included in the `FCharacterStats`. For causing damage, Unreal Engine includes a built-in **TakeDamage** function. This function can be overridden by the programmer and exists on every Unreal Engine object. This prevents the need of casting between types every time when causing damage and therefore makes damage handling much simpler and even slightly faster for the computer to execute. When a fighter character should take damage, the **TakeDamage** function is called, which first considers three conditions:

- whether the damage causer is friendly to the damage receiving character
- whether the damage receiving character is marked as invincible
- whether the damage information given is sufficient to work as input for the system

If one or more of these conditions are not fulfilled, the function was called will not apply any damage. Otherwise the damage is calculated using the receive damage function of the character stats structure using equation 3.1. Additionally, the system triggers the relevant delegates to reflect the change of the character's health. The function then checks if the attack satisfies the even higher information standard of an attack damage event. An attack damage event will additionally report the damage to the damaged character, making computer controlled characters actually notice they were attacked.

The implementation of the damage dealing functionality theoretically allows both melee and ranged attacks. Currently, only the melee attacks are implemented due to the time limitations of the development process. Melee attacks can be triggered from any animation montage using an `AnimNotifyState`. Montages are a type of character animations that allow interrupting the default animations by smoothly applying the new animation through a blending process. `AnimNotifies` are functionalities that can be set inside every type of animation to execute at a certain frame in the animation. Similarly, `AnimNotifyStates` are tasks that execute over a duration inside the animation as defined by the starting and finishing frame set in the animation file. The class responsible for enabling melee damage in that way is the **`UAnimNotifyState_MeleeAttack`** class. Inside the animation file, it can be configured to enable certain parts of the character's model (defined by the character's so called bones) to cause damage during the time the notify state is active.

On a technical level, the notify state registers the array of predefined bones with the character to enable them. The fighter character itself checks the list of bones registered to cause damage every frame. It then checks if the bone would collide with any object when continuing some distance in the direction it is currently moving. This is done by checking for collision with the `destructible` channel. This means that only objects marked as colliding with destructible objects can be detected. This prevents the system from detecting invisible objects such as the capsule that characters use for navigation. The damage dealing character then generates the attack damage event that will later be passed to the **`TakeDamage`** function. Generating the damage result, the system uses the hit result generated by the collision check to determine the location and direction of the hit. Calculating this information is the main reason why that kind of collision check is used

even though it uses a lot of calculation time and is not very accurate. The other information such as the stagger chance, the toughness break and the hit effect scale are all defined by the executed attack and simply copied from its settings. The outgoing damage is then calculated by multiplying the character's attack value with the attack's damage multiplier as shown in equation 3.1. To prevent the same target from taking damage from the same slash of the attack multiple times when not intended, targets are registered to an array of actors hit by that one slash of the attack and then prevented from being damaged by the same attack again.

Emphasizing Attack Impact

To emphasize the impact of attacks, fighter characters also have a custom function called **`BlendTimeDilation`**. This function allows the damage causer and the damage receiver to freeze or slow time for a split second in order to give hits more weight as mentioned in "Dreamscaper: Killer Combat on an Indie Budget" (Cofino 2022). To achieve this, the character simply applies a multiplier to the time information that is passed to the character every frame. With this trick the engine-provided function makes it seem to the character that time passes a lot slower than it does or even completely stops. The blend time dilation function starts a smooth interpolation process between the target speed and the current speed which ensures that the time dilation feels like a natural effect of the hit and not like an annoying stutter. To make the transition even smoother, only the characters participating in the hit are applied time dilation as suggested by Confino.

Another technique used to emphasize the impact of an attack are hit effects. A hit effect gets spawned when a character gets attacked and can be divided into visual effects (vfx) and sound effects (sfx). The visual effect is spawned at the location of the hit's impact. Its shape can be set specific to the character attacked but the effect gets scaled based on the character's internal setting as well as the hitting attack's settings. This **`HitFXScaleFactor`** property allows the designer to make an attack stand out more by changing the visual response to it. The sound effect played is dependent on the part of the character that was hit. This is implemented by looking up the bone closest to the impact of the attack on a list defined for each character and playing the sound associated. If the bone is not found on this list, it means that it has no manually defined sound. The system then searches the list for the bone connecting the determined bone to the rest of the body. This process is then repeated until the bone is found on the list. This makes it possible to create sounds for specific parts of the body but also allows simply setting one sound for the whole body by associating a sound with the most central bone of the

character. This can for example be used to make hitting the armor of a character feel different from hitting the character's skin. All of these sound effects are played spatially. This means that the engine creates the impression of a sound as you would expect. This means that less of the sound is heard on higher distances from it and the sound is played with a slight delay and difference of volume on the two cups of a headphone if worn.

These effects are generally played as a reaction to all attacks. But after an especially heavy attack, there should be an even stronger reaction. This is why the `stagger` state exists. Being hit by some especially heavy attacks sets the character into a state where they cannot move for a short time. During this time the character plays an animation which gives a visual indication of the stagger state and about how long it will last. A stagger event can only be triggered when the system managing the allowed inputs allows the input **`Stagger`** or **`HeavyStagger`**. The appropriate stagger type is then applied to the input limits system. The time for the limit to persist is then defined as slightly less than the time needed for the animation to fully play. This gives the player already the option to act slightly before the animation ends which gives the feeling of being able to act exactly when the character recovered. Counter intuitively, actually having the animation and the input limit end at the same time has the effect of feeling like the input is limited for longer than the animation, which is why this slight offset is necessary.

Death

The death function implemented inside the general character also has to be overridden here. The addition is simple: When dying, the character becomes immune to all damage and cannot be targeted by attacks. In the technical implementation, there is one small difficulty: the death function as defined in the general character can fail if:

- the death input is not allowed
- the character can explicitly not be killed or is currently playing the death animation (as indicated through the disabled collision of the main character model)
- there is no death animation

In these cases, executing the code mentioned at the beginning of this section is at best redundant and may at worst lead to unexpected or seemingly inexplicable results. This can be circumvented by making the **`TriggerDeath`** function return whether it could be successfully executed or not. The overriding function will then

simply propagate the result and only execute the added code when death could be triggered in the original function.

3.2.3. Target Information

Furthermore, all fighter characters have a `UTargetInformationComponent`. This component has two main functions. One of more general use and one more focused on opponents. Generally, it stores whether the owning character can be targeted or not. There are two functionalities more focused on opponents. The first one allows registering and unregistering an entity that targets the character. The other one provides a numerical value for the priority with which the character should be targeted by others. Reading this information lets both the targeted and the targeting character know all the entities with the same goal. In the current version of the game, both of these systems are only used for one purpose each. For the former system this is to tell opponents that they are currently targeted by the player. The latter's use ist to tell the player's targeting assistance feature which opponent to target. When adding opponent versus opponent combat both of these functions could be used to even greater effect.

4. The Player Character

Now that the template classes for characters are created, the most important character can be added: the player character. Different to most other characters, the player character is spawned automatically when the game starts. This means that placing this character in the level will simply add the model to the world but not make it the character the playing person is controlling inside of the game. To set a new player character, the `player controller` setting has to be changed inside the level's world settings. The chosen player controller will be automatically spawned when the game is started. Its task is generally to manage the player character. This means it is responsible for communicating the player's input to the actual character and also has the option to change the character controlled by the player at any time.

4.1. The Player Controller

The player controller used in this game does two main things. These are helping the player to control the camera and creating the player character.

4.1.1. Camera Assist

The first function of the player controller in this game is controlling the player's camera. In most games such a feature is implemented to help players using controllers instead of a keyboard and a mouse. Controlling the camera separately is somewhat tedious when using a controller, as this function is controlled by the same hand that has to operate most of the interaction buttons. These buttons are heavily used in most combat games, which is why many implement a camera assistance feature to help the player focus on the combat. This is no problem when using a mouse and a keyboard. As the mouse, which normally controls the camera is also used often for other interactions by clicking its buttons. This game however also benefits hugely from a camera assistance feature in another way. As the game is a close combat focused action game with an animal as a main character, it often happens that the body of the player character covers the opponent. If the camera automatically rotates to a sideways view, the player can see both their own character and the opponent better and thus can focus even better on combat.

The most important part of such a camera assistance feature is that it never takes control away from the player as this would be annoying to players most of the time. Allowing the player to override the camera assistance at any time, also makes it less important for the system to be perfect as a player can simply change the camera's angle if they do not like the current one. Whenever the player moves the camera, this is registered with the player controller. It is then checked whether there is a character that the player seems to want to focus on. This intention is determined by checking if the most likely target of the player's focus is inside of the current view. This allows the player to turn the camera to look away from a target to stop the combat camera assistance and look at the target it when they want. The assistance feature itself only gets activated after the player did not change the camera's position for over a second. The system then first tries to focus on the camera's combat target. If there is no current target, the controller assistance feature is used. The controller assistance feature is designed to only have an effect when the player is truing slowly and moving more or less in the direction the camera faces. In these cases, the camera will rotate to face the direction the player is walking in. To rotate the camera smoothly in any case, a maximal angular acceleration can be defined. This ensures that the switch from user controlled camera to using the assistance isn't too noticeable. To ensure a smooth deceleration when close to the desired rotation, a asymptotic interpolation behavior is used to interpolate the rotation.

4.1.2. Creating the Player Character

The other important task of the player controller is creating and setting the player character. Not all of this work is done inside of the player character itself to allow for dynamic switching of characters. This results in the character being first configured and then spawned using a custom function inside the controller. This configuration process includes determining the player's spawn location and calculating the player's stats which then will be saved inside the controller and accessed by the player only by reference. While this is never directly used in the project, this enables switching to a different character before returning to the first one easily without loosing all the information of the first one. This also simplifies saving the character's data in a file. A system for storing data in this game has been experimented with and is therefore implemented but it is neither fully functional nor relevant for any other functions described in this work. This is the reason why it will not be discussed further. Selecting the soon to be created player character inside of this controller will therefore spawn it at the right time and make it player controllable.

The player character can now be created. First the different possible player inputs have to be bound to functions inside the character's code. As most of the relevant code for this can be simply copied from the engines' default third person character, this will not further be explained here. After adding the features explained in this chapter, a custom blueprint of the player character can be created and edited to have the right model and values. This final player character can then be selected inside the player controller to spawn the new character when the game starts.

4.2. Player Intent

As the playing person is not directly connected to the character in any way but controls it through a keyboard or some sort of controller the game can only interpret the input and not execute the player's actually intended action. For example, the player sometimes knows what they should do in the current situation but fail to communicate it to the computer in time. This might happen because the player simply has forgotten which key they have to press or by clicking the wrong key by accident. There can also be hardware limitations like input latency that prevent the player from communicating their intention at the right moment.

When implemented on purpose, tight timings and difficult to remember key combinations can be used to set the player in a certain state of mind. This can be used for example to make the player's journey mirror the one of the played character. As the character learns new skills in the game's story they might be confused and only slowly learn to use their new move. The game can set the player into a similar state as the character by introducing them to a complicated sequence of controls they will only slowly get used to and be able to execute correctly. This will make the player feel more connected to the character they play.

On the other hand, playing as an established super hero like Batman in the Arhkam games, most players expect to immediately play as the cover hero without having to learn Batman's skills from scratch. As pointed out in Mark Brown's video "Who Gets to be Awesome in Games?" (Brown 2020), this would make the player feel quite disconnected from their played character as the story automatically supposes the character to already know all the moves, but the player is not yet able to execute them satisfactorily.

Therefore, the code inside the player character has to take the feeling that should be given to the player into account. The systems should help the player to execute the intended action at the right time in a way that fits the difficulty intended for the action's execution and not simply the difficulty resulting through the indirect

```
void APlayerCharacter::TryJump()
{
    //check if the input system allows the jump action
    if (!AcceptedInputs.IsAllowedInput(EInputType::Jump))
    {
        //store time, input type and the action itself
        //(MaximalInputWindowTime is also passed for technical reasons)
        LastInput.TryUpdateStoredInput(GetWorld()->GetRealTimeSeconds(),
            MaximalInputWindowTime, EInputType::Jump, [this]{ TryJump(); });
        return;
    }
    //if the input is not allowed, we can remove the last saved input as it is no
    longer needed
    LastInput.Invalidate();
    Jump();
}
```

Figure 4.1.: Example of how a currently invalid input is stored (not fully representative of the function used in-game)

communication over keyboard or other controllers.

4.2.1. Input Windows

One of the simplest ways to reduce unwanted difficulty of a game system is to give the player an extended amount of time where they can input their action. This can mean saving too early inputs and executing them at the first moment they could be executed as mentioned in "Dreamscaper: Killer Combat on an Indie Budget" (Cofino 2022). Using this trick makes the player feel more in control over the actions as their input does not get dropped when just slightly off. This is achieved by storing the relevant information about a function that could not execute as shown with the code in figure 4.1. This includes the time of the attempted execution in seconds since the game started, the input type that has to be allowed for the action to execute and the action itself. When the current input limit ends, the system lets the prior limit's follow up limit (if there is any) execute and then tries to execute the saved action. If this fails, the same process will be repeated after the newly queued up limit ends. To prevent executing inputs with a context that has become irrelevant due to the passed time, only the most recent input is saved and only for a set amount of time. This makes the game feel more responsive without making it confusing. There is however one exception to newer inputs overriding older ones: normal walking movement cannot override an already queued input. This is done because players tend to continue holding the move keys in the direction of an opponent when fighting. This would lead to every other input being overridden.

Extending the input window on the other side is more difficult though. While saving the player's action for later is simple, giving the player the option to react later is much harder. In most cases it is simply not possible without compromising input latency. It would for example be possible to save incoming damage and execute it 200ms later only if the player did not react. This would give the player more options to dodge the attack. This however results automatically in significant delays between the damaging action and the damage actually applying. Testing it out, the result felt pretty bad and did not really add much to the player's satisfaction. This meant that for the player to benefit noticeably from the change, the input window would have to be longer but this would make the delay feel even worse. The better solution to this proved to be adding clear sound queues to the attacks and making the attacks only start to cause damage a bit later. This gives the player a better chance to dodge an attack without sacrificing input latency.

4.2.2. Who is the Player Targeting?

Interpreting the player's intent only through code can often be difficult. In the GDC talk "Evolving Combat in 'God of War' for a New Perspective" Mihir Seth mentioned the usage of the controller's right analog stick as an effective way of determining where the player is aiming to hit. The work implements this idea by saving movement inputs as an in-world unit vectors pointing in the direction of the movement accompanied by a timestamp of the time the input occurred. This allows the game to remember player inputs for a short duration. This can then be used to predict the player's intended action more accurately. When the player movement input happened too long ago, the remembered direction is unlikely to be the player's current target anymore, so the direction the camera faces towards is used in such situations.

Games frequently use the same key on the keyboard for different actions depending on the situation. When climbing, many games switch the jump button to accelerate the climbing and disable the run button for example. Using the same key for similar actions like jumping and faster climbing even though their implementation might be quite different makes it easier for the player to remember the correct key to press. This concept is used in this project with the player's blink ability. This ability works both as a fun mechanic and as a cheap way to implement a dodge without having an according animation. The player teleports to the other side of the currently targeted opponent becoming invincible for 500ms. To reduce the mental load for the player the blink ability shares a button with the effectively similar dash. The dash ability accelerates the player for 100ms leaving the player at higher speed than their simple walk for the duration of the key press. Similarly to the blink, the first 500ms of the dash also have the invincible effect. Furthermore

the blink only makes sense to execute in combat, while the dash makes little sense to execute in combat as the player mostly pushes against the opponent without actually being displaced. Using the same key for both abilities therefore makes a lot of sense. The context sensitive trigger executes the blink ability when the player has a combat target and is no more than 100cm away from it. If the blink cannot execute as the location to blink to is obscured, or any of the above conditions for the blink are not met, the dash will be executed.

For both accessibility and design reasons many games help the player to hit their intended target when executing an attack. The goal of this is to make combat more intuitive and allow the player to focus on the more interesting parts of combat. Using such a system has however also the potential of making combat less intuitive when often misinterpreting the player's intention. To determine which opponent in a combat situation should be targeted by the player, the game uses an auto targeting function. In every frame, the player character generates a score for each opponent in the relevant radius. The score is generated based on:

- how centered the opponent is in the camera (max. 1pt)
- how close the opponent is to the player (max. 1pt)
- how much the player's given input direction points towards the opponent (max. 3pt)
- if the opponent is already the current target (max. 0.5pt)
- the individual targeting priority as set in the `UTargetInformationComponent` for each opponent type (as a multiplier)

The weight values used were found simply by trial and error and tuned to make the system feel about right. Different games may well have different sweet spots. In addition to these weighted criterias, opponents that are not visible at all to either the camera or the player model due to an object being in the way can not be chosen as the new target. This is done to prevent the character from automatically focusing on a target that the player most likely does not even want to. Technically this is done by checking if a collision with an object marked as visible occurs along the straight line from both the player's location and the camera's location to the center of the opponent. If a collision occurs, it means that there is no way of seeing the character as it is blocked by an object. In this case the same process will be repeated for each corner of the smallest box the character fully fits in. If the center and all corners of the character are obscured, this normally means that it is actually not visible. Thus the opponent cannot become the new target. Using the `UTargetInformationComponent`, the player then communicates to the highest

ranking opponent, that this opponent has become the new target and removes itself from the list of characters targeting its last targeted opponent. The information about the current target is then used inside the system helping the player to hit their target and in the attack priority calculation for opponents as explained in chapter 5.2.1.

4.2.3. Hitting the Target

Helping the player can be done in many ways, but one of the most used in combat games is moving the player towards their intended target dynamically, so they are more likely land an attack.

A 3D game displayed on the 2D monitor plane naturally loose some of the depth information. This is what often makes it quite difficult for the player to estimate the actual distance between them and their target. As a result, it will happen often that the distance is slightly too short and the attack will miss. Similarly, when the game simply makes the player attack in the direction of the camera facing, it is highly likely that the attack will hit slightly to the right or left, missing the target. This happens mainly as the character in a 3rd person game is normally in the center of the screen, often taking up a large amount of screen space. As such the character's own model will often prevent the player aiming correctly at opponents in front of them.

Missing an attack in such a way is infuriating for most players as the system does not recognize their intent correctly and then punishes them for their perceivedly wrong action. From a design perspective the skill to be tested in such a situation is often timing and the correct choice of attacks. "Move the camera correctly and find out whether the 3d distance projected onto the screen is 5cm or 6cm" is rarely one of the skills originally intended to be used.

To ensure that the player can actually focus on the timing and choice of attacks, the game will automatically determine the intended target of the player and move the character towards it. This is done by selecting a target location at the moment of the attack's execution. This target location is either the currently targeted opponent (if one does exist) or straight in the direction the player was walking before. If the player is not targeting an opponent and did not walk for an extended amount of time, this would be practically random. So, in these situations, the target location is calculated as straight in the direction of the camera facing. This information is then given to the `USuckToTargetComponent`, which stores it. The component then waits until the animation of the attack communicates the be-

ginning of the process of moving the player towards their target and turning the player to face it. The time window where the sucking to target finds place within is defined inside every animation using an `UAnimNotifyState` objects derived from this Unreal Engine class can be placed within an animation and set a start and end time. This enables matching the movement baked into the animation to the one created by the `USuckToTargetComponent`.

Helping the player even a bit more, the suck-to-target component can adapt the target location when targeting a moving object. This ensures that the attack still hits its target even when some movement of the target takes place. To prevent this helping hand from being more than just that, the `USuckToTargetComponent` only follows such a moving object as long as it is inside of a defined radius around the starting position. If the object moves outside of that radius, the player character will simply move to the closest point still inside the radius.

In many ways, this `USuckToTargetComponent` is similar to the `UMotionWarpingComponent` provided by the Unreal Engine. There are two reasons for not using the provided standard component in this project: firstly, while both systems allow for moving objects to be followed, the `UMotionWarpingComponent` does not allow setting a maximal following radius; secondly and more importantly, the `UMotionWarpingComponent` showed some severe bugs when used with the character animations used in this game. In the last seconds of the animation, there is a slight forward movement followed by a small backwards movement. The `UMotionWarpingComponent` can normally deal with such movements, but in Unreal Engine 5.2 the characteristics of these animations lead to the character slightly overshooting. If the `UMotionWarpingComponent` is both repositioning and rotating to face the target, the character now has to rotate 180 degrees to face the target point. As the animation then plays the slight backwards movement, the character overshoots once again and has to rotate back 180 degrees before the process is finished. This means that the last split second of the attack was an extremely short 360 degrees rotation that was noticeable as a strong stutter. To circumvent this issue, the custom `USuckToTargetComponent` uses the actual position of the character instead of the position relative to the animation as its reference point.

While the suck-to-target system is most relevant and noticeable on the player character, it is also used on opponent characters to ensure that their attacks connect, as it would otherwise often happen that they try to attack but completely miss. It can therefore not be said that this is a player exclusive feature, but it still is a very important way of helping the player execute attacks in line with to their intentions.

4.3. Visualisation

Another player specific function is the visual indication of in-game values. Depending on the type of game created, the number of values presented as well as which ones are presented are wildly different. Games can add a lot of uncertainty by simply not displaying certain values and thus add to a realistic or mysterious feeling. On the other hand, showing values in a very detailed way adds to the strategy element of a videogame as the player has to make decisions based on the given information. Many games choose to display a selected number of properties about both player and opponents to give the player enough data for a solid decision while also hiding many properties to avoid confusing the player. In this game, every `AFighterCharacter` has a display associated to them showing their health and toughness. Changes in these values are communicated to the display using the delegates associated with health and toughness changes. In addition to these values, the player character displays the remaining cooldown time of the player's skill and ultimate as well as small timers under the player's health bar which represent the remaining time of their current status effects. This can also be seen on image 4.2. All of the time related displays are driven by the Unreal Engine included `FTimerHandle` which is simply the value associated with an in-game timer. The timer handle can be used to get the remaining time of the timer, pause it, stop the timer and other related functions. As both the attack cooldowns and the status effects use a timer anyways to end the cooldown or remove the effect after the time is up. When a status effect timer shows zero, its icon is removed from the screen, as the effect is not applied anymore. To avoid empty spaces between the displayed status effect timers, the timers to the right of the just removed timer get moved one place to their right, filling the gap.

Figure 4.2.: This is how the character's in-game user interface looks; the small icon under the health bar represents the current status effect's remaining time, while the filled white circles show that both skill and ultimate are not on cooldown.

5. Non-Player Characters

After having created and implemented the player character, the game is now a lot more customized but there is still no combat. For this, the non-player characters (NPCs) have to be added next. NPCs behave like an inverse of the player character. They have to be placed inside of the level and then automatically spawn their controller when the game begins. The non-player characters are related even more closely to their `AAIController` than the player character to their controller. Most of the NPCs' behavior related functionality makes more sense to implement inside of the controller while the action related functionality is most often inside of the character. This is mainly due to where the Unreal Engine exposes the relevant functionality.

When creating NPCs, one of the most fundamental rules is to make them behave the way the player would intuitively expect them to. NPCs that cannot participate in combat are expected to behave like living creatures, meaning they should normally move around or be visibly focused on doing some task. Characters standing around doing nothing and simply waiting for the player to interact with them will inevitably feel wrong and break the game's immersion. In combat this is even more important. Almost all players have an image in their mind of how characters normally behave in combat even before playing the game. If the game manages to pick up on that image and expand on it, the combat fundamentals will be easy and intuitive to learn. This enables the designers to add more interesting and fun mechanics earlier in the game. On the other hand, having a counter intuitive combat system will result in the players needing more time to learn the basics which in turn means that the more interesting mechanics can only be added later on. Furthermore such a combat experience will frequently lead to frustrated players with the impression that the game is actively trying to sabotage them instead of being a fun experience for them. Therefore, a large part of the work done on NPCs is related to making their behavior intuitive and not forcing the player to re-learn established patterns.

Characters' behavior, meaning if and how they react to inputs, is a crucial part in making a game easy accessible and allowing the player to focus on challenging content rather then the system. To achieve state based reactions, Unreal Engine

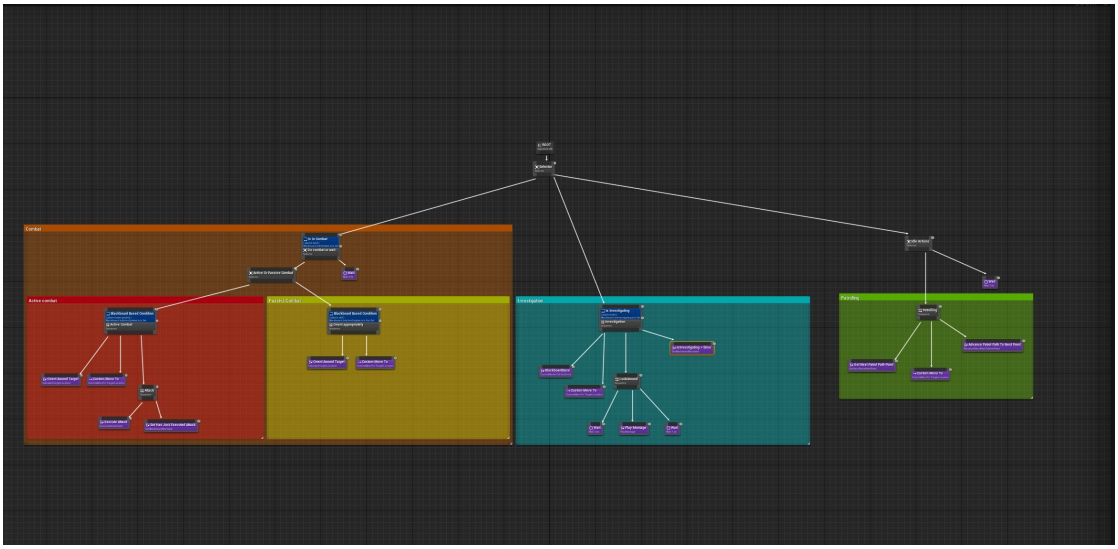


Figure 5.1.: Screenshot of an opponent NPC's behavior tree as used in the project. The behavior tree can largely be divided into combat (orange) and non-combat behavior (blue and green)

provides `behavior trees`. In editor, these constructs are displayed as a tree like structure with one `root node` which is connected by one directional links to the lower nodes. When activated, the tree keeps executing until it is actively stopped. Whenever multiple connections from one node to the lower ones are present, the leftmost of these connections is executed first. The first path is followed until an executed node results in a failure, or all nodes contained in the executed branches have returned success. In the first case, the failed result is communicated to the prior node. If the node is a `select` node, the leftmost not yet executed branch is followed. If the node is a `sequence` node, it requires all of its branches to return success for itself to return success so the failure is then communicated the `sequence`'s parent node. This system allows for complex state based actions while still being relatively easy to read.

For the purposes of this work, characters' behavior as seen on image 5.1 can be subdivided into their non-combat and combat behavior.

5.1. Non-Combat Behavior

NPC's non-combat behavior describes what the character does before and after combat situations and how they react to inputs during this time. Non-combat behavior aims to give the player the impression of credible living beings. Minimally,

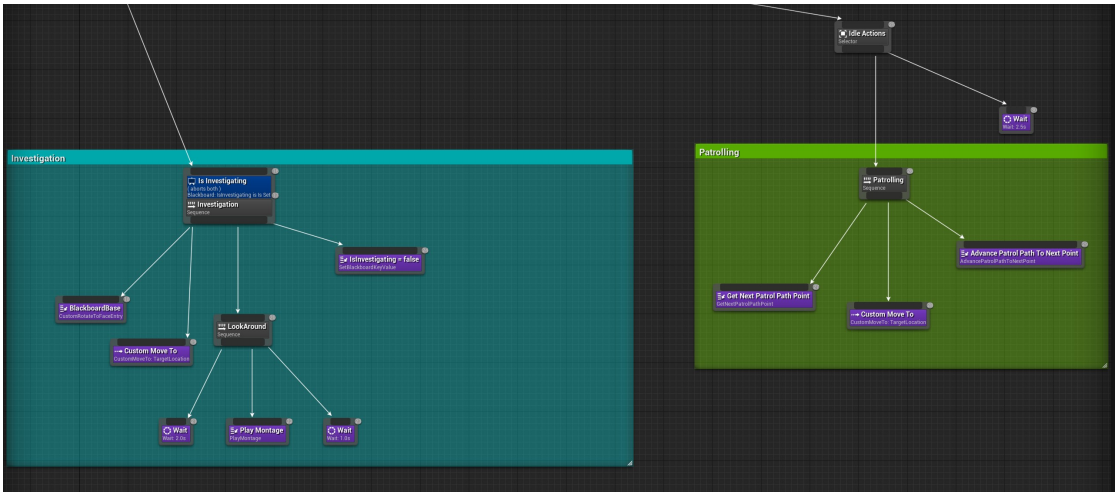


Figure 5.2.: Screenshot of the non-combat logic from an opponent's behavior tree used in the project. The tree consists of investigation behavior (blue) and patrol behavior (green)

this means that the NPC should move around or do some other feasible action. They should also perceive the player's presence in some way. The behavior used in this game is pretty simple, as the focus of the work is on the combat behavior. The characters normally follow a patrol path when not in combat. If they perceive an opponent but do not see it, they investigate the source location of their perception as seen on image 5.2. This means, walking to the location and looking around, before returning.

While this chapter is focused on combat NPCs, it's contents can also be applied to non-combat NPCs.

5.1.1. Movement

The easiest way to make the NPCs feel alive is to make them walk around. To make a character walk, some things have to be done first. If any creature walks from point A to point B, they normally take more or less the most efficient route. However, only in a few cases this most efficient route is the straight connection between A and B. Often, the floor is not even and there might be objects obscuring the straight line between the two points. The natural route to walk to a destination is the shortest path over the floor that does not intersect with any walls or other obscuring objects. Unreal Engine provides an advanced system for path finding that does just that. Firstly, you have to place a **`NavMeshBounds`** actor inside of the level and adjust its size to match the one of the world in all dimensions.

The engine will then generate a `navmesh` that represents the walkable area in the level. Using the NPC's `AAIController`, move requests can be generated, which will move the character along the navmesh using the shortest path to reach the given target. Using this standard features works well for most scenarios and even supports moving towards a movable object. When walking, the character will face the same direction as their forward movement, as intuitively expected. This makes the system work well for most scenarios.

Patrol Paths

To give the NPC a simple path to walk along, the game uses a system inspired by Ryan Laley's video tutorial "Unreal Engine 5 Tutorial - AI Part 6: Patrol Path" (Laley 2022). This method has `APatrolPath`s placed in the world, each of which has a variable number of path points associated with it. These path points can be moved around in the editor and placed at various locations. Characters can then be assigned such a path to follow. Differently from the system implemented by Ryan Laley in his tutorial, a `UPatrolManagerComponent` is used in this game to keep track of where a character is on its patrol path. Using a component to let characters follow exactly a path instead of implementing fully in the behavior tree makes it easier as all the code related to the path follwoing is in one place rather than distributed over different nodes in the behavior tree. Furthermore, this also makes it easier to add more complex functionality to the path following behavior. One of the more complex behaviors this component easily allows is walking to the closest path point instead of the next path point in some situations. When starting to walk along the patrol path for example, the patrol manager searches all path points of the given patrol path and chooses the closest one to be the first point of the character's patrol. This makes dynamic placing characters in the world around the same patrol path easier. This calculation is also done when an NPC ends combat and goes back to patrolling around the area. Picking the closest patrol path point prevents the NPC from weirdly walking paths while stupidly following the given order. As proposed by Ryan Laley, the patrol path gives both the option of making the path linear (A-B-C-B-A) or circular (A-B-C-A-B-C). This can be set per patrol path. The patrol manager then reads the setting from the patrol path and iterates through the path points dependently. Inside the behavior tree as shown in image 5.3, the character is assigned the target location as determined by the `UPatrolManagerComponent` ("Get Next Patrol Path Point" node) and moves towards it ("Custom Move To" node). When reaching the location, this is communicated to the patrol manager, which then changes the result of the next target location assignment ("Advance Patrol Path To Next Point" node).

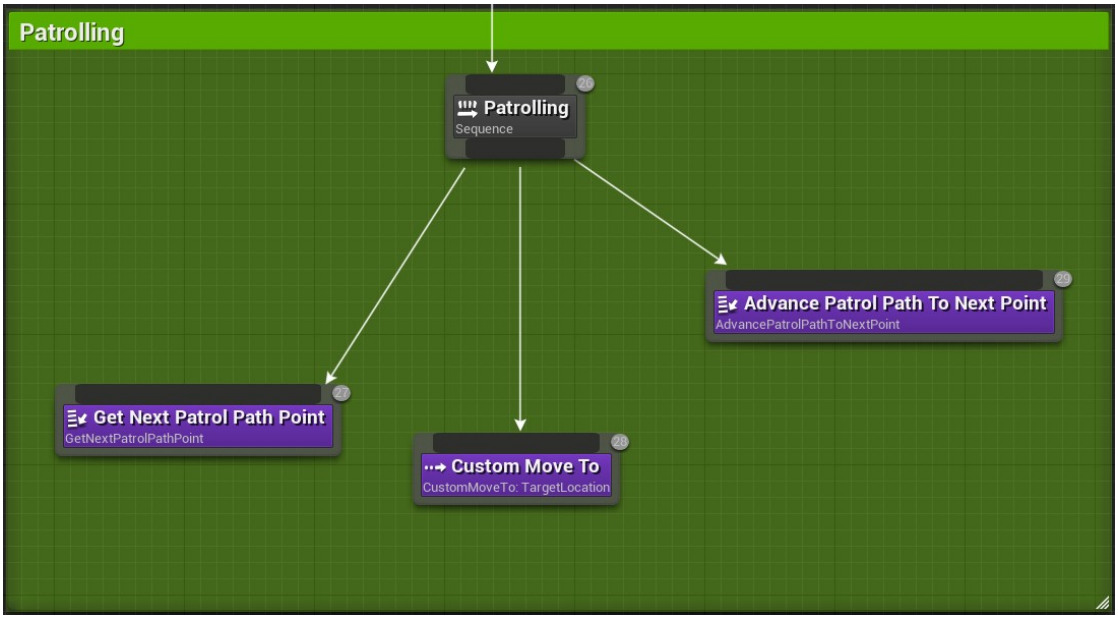


Figure 5.3.: Screenshot of an opponent character's patrolling behavior; The purple instruction nodes are executed from lowest instruction index to highest (here from left to right)

Switching Target Locations

While using the engine's navigation system works well with simple walking and the custom implemented path following system, it often leads to bad results when combining it with the combat positioning system described in chapter 5.2.2. One reason is that the engine's system is not optimized for switching the location the character is moving towards during this process. It does work but when triggering such an override in Unreal Engine 5.2, the character's walk speed quickly becomes zero, before returning to its normal value again. This causes the character's animation to quickly switch from walking to standing and back to walking. This visual glitch can be prevented using the built in object following option. When a followed object changes its position, this is handled absolutely smoothly and without visible deceleration. To make use of that option, NPCs can be assigned an invisible, non-physical object each. When requesting a move to a certain location, the object is smoothly moved to the actual target location and the given movement request is modified to walk towards the assigned object instead of its original target. This movement target is automatically created as soon as the controller is assigned to a character. To reduce unneeded calculations, this target is only used in combat. Besides of the disadvantage of unneeded calculations, this system also has a flaw making it significantly less useful outside of combat. In combat,

the terrain is mostly flat and the obstacles are rather tree-like in size and therefore relatively small. Outside of combat, the traversed distances are often longer and the objects blocking the direct route are frequently bigger. When the system interpolates the followed object's location between the old and the new target location, the target object is allowed to move through solid objects and float many meters above ground as it would otherwise require much more calculation time. In such occasions, the movable object cannot reasonably be mapped to a point on the navmesh and the character therefore stops moving. This introduces a similar glitch in non-combat movement as the Unreal Engine provided variant does when directly used for combat positioning.

Where to look

The second reason why the built-in movement system has to be slightly adjusted in some situations is the automatic forward facing of the character. While this behavior is optimal in most scenarios, combat encounters are generally not one of them. Such situations should be tense moments where both the player and the NPC have to focus. Even though the NPC can not truly focus on battle, such an impression can be created by making the NPC look continuously towards the player. Using this to create a tense atmosphere, it is not helpful if the attacking NPC turns sideways to adjust its position slightly, resulting in a right angle between the player and the NPC. Unreal Engine lets the developers manually set if the rotation of an NPC should always face a certain object or point or if the NPC should rotate in the direction of its movement. Using the first option leads to the NPC always facing the player which is better than before but also leads to counter intuitive behavior when an NPC has to walk a long distance away from the player. In such a case, the character would simply walk backwards for an extended amount of time. While pretty good looking and simple if the character has a backwards walking animation, this feels like a weird parody of the moonwalk when the character has no such animation. As none of the characters provided in the pack of creatures used in the practical part of this work (*Quadruped Fantasy Creatures* 2018) contain such animations, a slightly more complicated approach had to be used. While the implementation of such an approach requires some time, this is still a lot faster than accurately animating a backwards walk without much experience in animation. A dedicated `UCharacterRotationManagerComponent` is used to switch between the two provided orientation modes. Valid move requests during combat trigger a calculation deciding which of the two rotation modes given by the engine is more optimal for the given request. Generally, the NPCs closer than their maximal distance allowed for their passive position always face the player. One exception to this are NPCs that walk to a position outside of that range. In these cases, the NPC is hardly focusing on combat and therefore it

makes sense for the character to not look at the player. Other paths containing at least one traversal of the maximal distance line, will trigger a continuous check so that the NPC will face the player within the line and does not do so outside the line.

5.1.2. Detecting the Player

Now that the NPCs can walk around and investigate, some rules have to be defined to switch between patrolling, investigating and combat related behavior. The most intuitive way to toggle between these states is using the character's perception of the player. Simulating senses for a computer controlled character is not required for all games. For games like Nintendo's Super Smash Brothers series, this is pretty unnecessary. In such games, the player has no possibility to hide and NPCs' are therefore expected to always know the player's location. But for games that provide obstacles where players can hide, an NPC is only expected to react when they can realistically perceive the player. This means the NPC has to either hear, see or otherwise feel the effect of the player's action to be able to react. It is therefore important to find a way to detect the player only when it makes sense but still achieve reliable detection.

Seeing the Player

Unreal Engine's `perception system` can do a lot of that detection work for the developer. It provides a generic interface for handling detected characters and objects. The system supports using normal senses like sight, hearing, touch and damage sensations. However, it can also use more abstract senses like a `prediction sense` and a `team sense`. All of these options can be freely selected and combined.

The most important sense in this game is the sight sense. It allows the detection of any object or character inside its field of view and within the maximal view distance of the detecting character. Detecting the player character, the NPC will enter combat mode and attack the player.

Having detected an object or character, the engine calls a custom overrideable function which contains a reference to the detected object and information about the sensation of that detected object. This information is saved inside the engine defined `FAIStimulus` structure and can mainly be used to get information about the sense that detected the object and the location from where the sense was triggered. This provides three options of how saving the position of a sighted object can be handled:

- Saving the location the object was sensed at
- Saving the actual location of the object sensed
- Saving the sensed object itself so its exact location can be determined at any time

All of these options are quite similar in many ways, but also different in others. As the perception system only tracks the first detection of a sighted object, using one of the first two options will make it impossible for the NPC to follow the target character. The third option would allow the NPC to always know the location of its target. This is the best option for most cases. It can also benefit from small performance optimizations. To optimize this behavior this game stores both the reference to the detected object and the information contained in the engine's `FAIStimulus` when the object is detected at first. Every frame, the distance between the object's current location and its last detected location is calculated. If this distance exceeds a certain thresh hold, the game then calls the same function the engine uses for newly detected objects but with the object's new location. This effectively enables sight detection to be treated like any other sense as it is updated whenever a relevant change to the seen object's position occurs while still reducing calculation time as the process is not repeated every frame.

Another difficulty with the specific function the engine uses for the sight sense is its reaction to a detected object leaving the field of view. When the seen object moves to a location where it cannot be seen anymore by the character, the character immediately looses track of it. This is problematic for many reasons. The most disturbing one is that some characters (e.g. Barghest) move their head strongly up and down while walking or running. This strong movement of their eyes frequently leads the Barghest loosing the target from sight for a split second. Thus, the Barghest immediately forgets the player's location as discussed before. Then, having nowhere to go, the Barghest stops walking. This adjusts the Barghest's head in a way that it sees the target again. Therefore the character starts walking and repeats the cycle. To prevent such weird stuttering behavior, a useful extension is to make NPCs remember sighted characters for a short time.

Theoretically, the Unreal Engine provides a `stimulus age` system. This age information is passed to the NPC through the `FAIStimulus` structure whenever something gets sensed. The system should remember the stimulus for a customizable amount of time, before it removes them from the character's memory. Using Unreal Engine 5.2, this behavior did not seem to work. Even when setting a maximal age for the stimulus, the NPC called the function to forget the object at the moment the object left the character's view. Since the `FAIStimulus` information

is only provided as a copy in the functions that are easily accessible and there is thus no built-in possibility to manually wait for the stimulus to exceed its maximal age. Using a custom implementation of that concept however lead to the expected result. When the engine registers an object as having left the viewable area of a character, the object is saved to a list along with the time when the character lost sight of it. The character then waits until one second has passed before it forgets the character. If the object gets seen again before, it is removed from the list. This implementation is quite simple to create and solves the problem of loosing sight to a character too quickly.

Sight detection can be very tricky if you want it to feel right. The main reason for this is that sensing and reacting to visual impulses are things we do millions of times a day and we expect very particular responses in certain situations. The problem about this is, that it is often difficult to find rules describing these intuitively expected reactions.

Seeing a movable object, you of course know exactly where it is in relation to you. If that object now moves left and out of your sight, you'd intuitively know where you'd expect that object to be as you turn your head to follow it. This is mostly based on the object's trajectory when still in sight. Some part of this can be reproduced using Unreal Engine's prediction sense. The moment where the object leaves the leftmost part of the NPC's field of view, the NPC will request a prediction event. This will predict the object's position in a given amount of time using its current velocity. In most cases, this will lead to the expected behavior.

But say, you accidentally kick a ball and see it vanish behind a wall. You know, of course, that this ball has fallen to the floor soon after it vanished from your sight. However, an NPC that looses sight of that ball at the wrong moment, will expect it to continually fly upwards, as it only takes the object's velocity into account and ignores other effects like gravity. Similar situations to the one with the ball described above can also occur in the game. This can also be seen on image 5.4. After having lost the player from sight at the location of the green sphere, the NPC predicts the player position in one second to be at the purple sphere, which it is obviously not.

Because this unwanted behavior does not occur often in normal gameplay, I did not yet fix it in my work. I focused instead in solving other more problematic issues as described in different chapters.

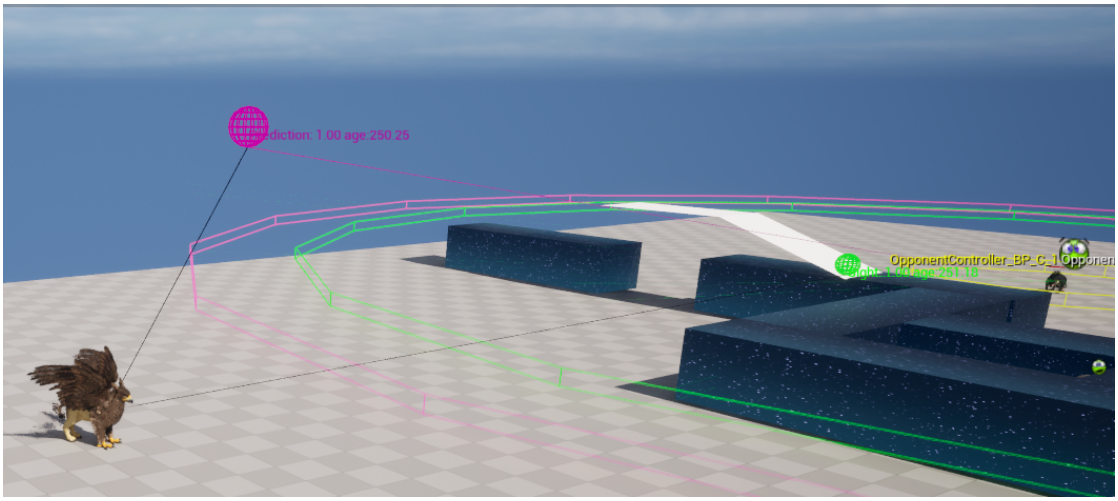


Figure 5.4.: Screenshot from a test level of the game using the engine's sense debug function; The player character can be seen to the left, while the NPC is seen behind the wall on the upper right.

Other Senses

The other senses implemented in this work are the hearing sense, as well as the damage and touch senses. When any of these other senses detects an object, the investigation process is started. This produces the intuitively expected behavior even for the damage and touch senses where this is not immediately evident. If an opponent gets hit from the back by a stealth attack or an arrow, the expected reaction is for the NPC to rapidly turn around to see where the danger is coming from. In these cases the NPC cannot know where exactly the danger came from but it knows where it felt the contact. This can be done by checking the type of the sense triggering the detection inside the program.

If the systems as described before were used, the NPCs would not only sense the player as intended but also detect all other characters including the other NPCs. This can be prevented by using another system included in the engine. Unreal Engine's team system provides a simple interface for determining the relation of two characters or even objects to another. Inside of every sense's configuration settings, there is also an option to decide which type of relation to the NPC another character has to have for the sense to register it. By default, the system supports three different relationships between entities (characters and objects) which are:

- friendlies (both entities have the same team identifier)
- neutrals (all entities that are neither friendlies nor opponents)
- opponents (the two entities have a different team identifier)

The implementation of the system is generally pretty simple, since you only have to add the `IGenericTeamAgentInterface`. Interfaces are a special kind of Unreal Engine constructs that provide a set of fully overrideable functions and cannot store any variables. Any Unreal Engine object can implement such an interface by adding the interface to its inheritance list. A team can be assigned to the character implementing the interface by overriding the function getting the team's index. Problematically, the documentation is not entirely clear whether the team data should be implemented by characters or controllers. In most cases implementing the system on the character works fine but the perception system in Unreal Engine 5.2 seems to have problems with this implementation. Assigning a team to every character, the perception system still sees all characters as neutral. This would suggest that the team information should be implemented inside of the controller. But this only enabled the perception system to differentiate between opponents and neutrals, it somehow still saw all friendlies as neutrals. Assigning both all characters as well as all controllers to a team produced the expected behavior. To ensure that a character and its controller are always on the same team, simple functions have been implemented which only allow setting the team in one of the two while the other one copies the team information when spawned to match the first one's.

5.2. Combat Behavior

The behavior of NPCs when out of combat as well as how they transition to combat is now defined. The next step is defining how the NPCs should behave in combat itself. The general principles of how a NPC should behave in combat are pretty intuitive. Often the NPC is expected to execute a melee attack. As such an attack requires physical contact with the target, the most intuitive way to achieve this is to make characters in combat walk towards the player. While this simple approach can work for scenarios with only a few NPCs, it fails when a larger number of characters are attacking the player simultaneously. The first of the two main problems in this situation is that there is simply not enough space around the player and thus, some of the NPCs will not be able to finish their move request and instead be stuttering in place while trying to do so. The other problem is less of a visual but a mechanical nature. The goal of most combat systems is to make the player feel powerful and give them tactical choices. When opponents simply

bunch together around the player, the player might feel overwhelmed. Furthermore mechanics like dodging, countering or blocking are more dependent on luck than on skill in such situations, as the chaotic executions of attacks and the number of possible damage sources is too high for the player. This will therefore result in a bad player experience.

5.2.1. Who gets to attack?

To prevent all NPCs from attacking at the same time, the characters attacking the player are divided into active and passive combat participants. Only a selected number of NPCs is given the opportunity to become active and advance towards the player to attack while the other NPCs passively stand back, waiting for their turn. This system is inspired by the explanations of Mihir Seth in his talk "Evolving Combat in 'God of War' for a New Perspective" at the game developer's conference (GDC) (Seth 2021). Along with many other things, Mihir Seth explains the advantages of such a division in 'God of War' and the parameters based on which this division is based on in the game. The idea is to generate a score for each participating opponent based on their distance to the player, their priority setting and their position on the screen. So called `aggression tokens` are then distributed to the top scoring NPCs. As there is only a limited number of those tokens, this results in a limited number of attackers at every time. Another advantage of this system is that the dangerousness of the opponents attacking at a time can be balanced by making more dangerous characters require more aggression tokens. As not much more detail on the implementation was given in the talk, the system used in this work is most likely quite different from the one in 'God of War' in its execution.

Most of the decision process of who gets to attack in this work is done through the `ACombatManager`. The combat manager is a non-physical object that has to be placed inside of the world. When characters enter combat, they are registered within the combat manager and become a passive combat participant. As a part of the registration process, the system tries to grant the newly registered participant the required amount of aggression tokens. If a new participant requires more tokens than currently available the participant remains passive, otherwise it becomes active. After attacking the player or the attack being interrupted by the player, the claimed aggression tokens are released. This also happens when the NPC exits the combat state for some reason. Every time aggression tokens are released, the `ACombatManager` attempts to redistribute the now available tokens. For this, a sorted list of all passive combat participants with a score larger than zero is created. Slightly different from the system explained by Mihir Seth, the score calculation is divided into two main score generation systems. The first score

is the aggression score of the character, which is heavily based on the GDC talk. If the character can explicitly not become aggressive the score is -1, otherwise it is dependent on:

- The character's distance to the player (max. +AggressionPriority)
- Whether the character is on screen (max. +1.5)
- How centered the character is on the screen (max. +2)
- If the character is the player's current target (max. $3.4 * 10^{38}$)

$3.4 * 10^{38}$ is used as it is slightly less than $3.40282 * 10^{38}$, the maximal value allowed in the variable.

Plus, if the character cannot currently attack, its score decreases by 20% for every second it is expected not to be able to. This ensures that NPCs mostly attack the player when they are able to and that characters that cannot attack soon (even if they are the player's current target) will not block aggression tokens from characters that would be able to attack sooner. The second part of the score is the calculated attack value. This has been added to account for the unequal strength of attacks and ensures that some attacks are valued more. This can be relevant if one character's strongest attack is still on cooldown and only have weak ones that can be executed and another character has a stronger attack that can be executed immediately. In this case, the second character should be preferred over the first one, which does not happen in the aggression score generation. To make decisions from the NPC more interesting for the player, the attack to value inside the attack score calculation can even be chosen at random. This way, the NPCs will not simply cycle through their attacks from strongest to weakest. To still give some design control for which attacks an NPC should use often, the attack is weighted with their set attack priority before being chosen at random. If the character is granted their attack tokens, they reposition in a way that they can execute this attack specifically in an optimal way. This also reduces the complexity of positioning and is therefore even more useful.

It is possible that one character has a higher score than another but also requires more tokens. If there are now enough tokens currently available for the second character but not for the first, two options are theoretically possible. The character requiring more tokens could be omitted, allowing the character with lower score to receive the token. Alternatively, none of the characters could be granted a token. The advantage of the first decision would be that the player always has about the same difficulty in combat. But it could also mean in some situations

that characters requiring more tokens will never be able to attack as the NPCs requiring less tokens always fill the free space up, never leaving enough space for a high token character. I have chosen the second option for this game with one addition: a character which didn't receive aggression tokens because there were not enough of them left will be remembered for the next token redistribution. Until this anticipated character is given a token, no tokens are distributed to other characters. Such an anticipated character will therefore certainly attack at some point, but in the time between, the characters attacking the player will gradually become less. While this can be seen as bad for the difficulty of combat for the player, it also gives a sense of anticipation when the weaker attacking opponents step back one by one until the strong opponent finally attacks.

5.2.2. Positioning

When in combat, both active and passive characters have to position themselves according to the situation. When a character becomes aggressive and is given a target to attack, it then has to reposition itself to execute the attack. Multiple factors have to be considered then to find out where exactly the character should go to. Connecting attacks require to be closer to the player than a certain threshold for example. A simple system would be to lay a grid over the world and calculate a score for each point on the grid. To reduce computational load, the scores would only be calculated inside a certain radius. So every character would determine their best location first and then move towards it. In his already previously mentioned GDC talk, Mihir Seth (Seth 2021) explained that they did not find much success using such a system. This was mainly due to the reason that there are many moving parts within combat, including both the player and opponents. This would lead to frequently shifting weights in the calculation and frequently changing best locations. This then would lead to the characters repositioning far to frequently. The team did find success using a much simpler system where they would simply check if the character's current position is valid and simply move the NPC to the nearest valid point otherwise. This reduced the frequency of repositioning opponents significantly. This approach was the inspiration of the implementation for the repositioning algorithm used in this work.

The `Environmental Query System (EQS)` as provided by the Unreal Engine would be an option for implementing this repositioning system. There are however some key points that made a fully custom implementation of the system more useful. Firstly, the EQS is designed to weight point against each other. This requires analyzing all target points to find the best one(s). The system proposed by God of War's combat team is however exactly designed to not weigh points but find valid points. This proposed system can therefore be optimized so that the first valid

location found ends the search process and thus, significantly reduces computation time. Furthermore, the Unreal Engine's EQS is easy to use with non-dynamic search queries but less practical when search parameters dynamically change. This again makes optimization for high performance high accuracy queries as needed in combat more difficult. The custom algorithm used in this game can be divided into three parts:

1. Determining the limiting factors
2. Checking if a recalculation is required
3. Finding a good location

Determining the Limiting Factors

First, the factors influencing if the position is valid or not have to be determined. These factors differ slightly depending on the combat state of the character. The first factor is the distance from the player. It is defined by a minimal and a maximal value as well as an optimal minimum and an optimal maximum. This makes the character's distance to the player have three states:

- `Invalid` - for distances between minimum and maximum
- `Valid` - for distances between the minimum and maximum but outside of optimal min and max
- `Preferable` - for distances between the optimal minimum and maximum

Dividing points into these three categories can be used to make the character only reposition when at an invalid location and then move to a preferable location. This significantly reduces the frequency at which repositioning characters is required. The division into these three categories is also used for all other constraints. This also allows the system to always choose locations that return preferable for all constraints while still allowing the character to move to a less preferable position if no other exists.

The distance constraint for passive NPCs is directly defined in the character's settings. Active characters on the other hand, aim to be at a distance to the player defined by the attack they want to execute. Since the combat manager also allows characters without any attacks available at that moment to become aggressive, such characters then calculate the average of their now available attacks and if they do not have any, they calculate the overall average of all their attacks.

To prevent multiple NPCs from trying to stand in the same spot, reserved space

constraints are created next. One of these will be created for each NPC that can realistically stand in the way. In most cases, this will simply be all characters participating in combat. Characters that are currently executing an attack animation using suck-to-target or walking towards a location will not block space according to their current location but their target location. This is more relevant for the newly calculated target location.

Is a recalculation required?

The next step is checking if all the constraints determined in the first part of the calculation are satisfied with the current relevant position of the character. The position relevant to this calculation is different depending on the current state of the character. If the character is already moving towards a target location, the relevant position is its target location. Otherwise, if the character is standing, its current location is taken as the relevant position. Next, the relevant position's match level is generated. The generator then projects the point onto the navmesh before calculating the position's match level with each of the constraints. The match level of each constraint is zero if the given position does not satisfy the constraint and 1 or more otherwise. If any of the constraints return a zero value or the relevant position cannot be projected onto the navmesh, the search is stopped and a match value of zero is returned. If all the constraints are satisfied, the sum of all the matches will be returned. Only if the position's returned match value is zero, a new position has to be calculated. Otherwise the NPC will not change their movement.

Finding a good location

Before determining the new target location, one last constraint will be considered. The last constraint divides the world into four rectangular zones with one common intersection point at the player's location. As proposed by Mihir Seth, this will make each character stay in the same area in combat even when out of the player's sight. This will give the player a more intuitive overview of the location of all combat participants. God of War's combat team found that players loosing track of where in relation to them opponents were would become confused and play cautiously, which directly contradicted their goal of an aggressive offensive main character. Giving the player the option to more easily keep track of their opponents was also in the sense of this game, which is the reason why such a constraint was added. Most likely, this constraint is slightly different from the implementation proposed by Mihir Seth. The implemented system only specifically disallows positions in the one zone opposite the NPC's current zone. The character's current zone is treated as preferable and the two remaining zones are

simply valid. This was changed to avoid frequent repositioning failures when allowing only the character's current zone. The single zone would often not allow for a location with all constraints satisfied. This modified variant keeps most of the original's intended points but allows more points to be valid leading to less frequent failures of the algorithm.

To determine the target location, an object of the custom point generator structure has to be defined first. Such a generator can be iterated through to produce a number of points through a predefined algorithm. These samples can then each be analyzed using a similar algorithm as the one used in step 2. Children of the `FPointGenerator` structure can be created to implement any desired way of point generation. In this case the `FCircularPointsGenerator` is an efficient option for both active and passive characters. This generator produces points in a circular pattern, adding points on a larger radius with the defined density after completing the prior circle. Similarly to the above mentioned distance constraints, this point generator only places points on a radius between a set maximum and minimum radius. The settings of the already calculated requirement for the distance to the player can be used to define a point generator that only produces valid results within this constraint. This eliminates the need to generate a high number of points outside this range that are predestined to be invalid. Using this simple optimization allows the system to either run faster or analyze points at a higher density in the relevant area to give a more accurate result. The density of the generator can also be set to a smaller value when generating points for passive combat participants, as they do not require the same precision as the active combat participants which are given significantly more attention by the player.

Next, every position produced by the point generator is analyzed with the same function used in step 2. The allowed maximal distance for a point to the navmesh is increased by 400cm here to allow more accurate analysis of even more uneven terrain and this time the exact value returned by the **`GetMatchLevel`** function is considered for the final result. As soon as a point achieves a preferable rating from all constraints, the position is selected and the calculation can be terminated. Analyzing all points generated without finding such an overall preferable point, the highest scoring point with all valid ratings will be chosen. This avoids NPCs standing around uselessly in combat due not finding a good enough position.

Part III.

The Product

6. Discussion

Having talked about many of the contents of the practical part of the work, the next part will focus on discussing the result and my personal experience with working on this Maturaarbeit.

6.1. Results

The current implementation state of the game satisfies all of the discussed targets. Getting into the basics of the combat system should be easily possible for averagely experienced players as the game uses mostly standard control schemes and supports both controller and keyboard inputs. Many systems were included to support the player and help them maintain control over the game and confidence in their abilities. These systems include attack token management system for opponents, the suck-to-target system, the camera assistance feature, forgiving input windows and long maximal combo durations. Opponent behavior in and out of combat, as well as all the character's reaction to hits are tuned to feel as intuitively expected, even though they might not always be physically accurate. This is achieved by providing various supporting systems like the movement system, the perception system, the combat repositioning system and how both player and opponent characters react to being hit by an attack. Hits are specially emphasized by hit effects which together with the suck-to-target system make the player feel powerful when fighting. To allow players to grow with improving skills, the player character has a blink ability that allows avoiding damage when used at the correct time. The audio and visual clues used in opponent's attacks combined with this ability give the player differently long but feasible windows for dodging, giving players with a correct timing an advantage. The player character has also been given different attacks and a combo tree. Stronger attacks can be executed when correctly remembering the sequence of the attacks inside of a combo, which enables the player to get better. The different player attacks and the ability to execute combos also enable varied gameplay by allowing the player to execute different actions. The game also includes two different kinds of opponent characters (Barghest and Griffon). The difference between the characters is also amplified by each of them also having a distinct attack style. This means that the Barghest characters mainly use quick, aggressive attacks causing an average amount of

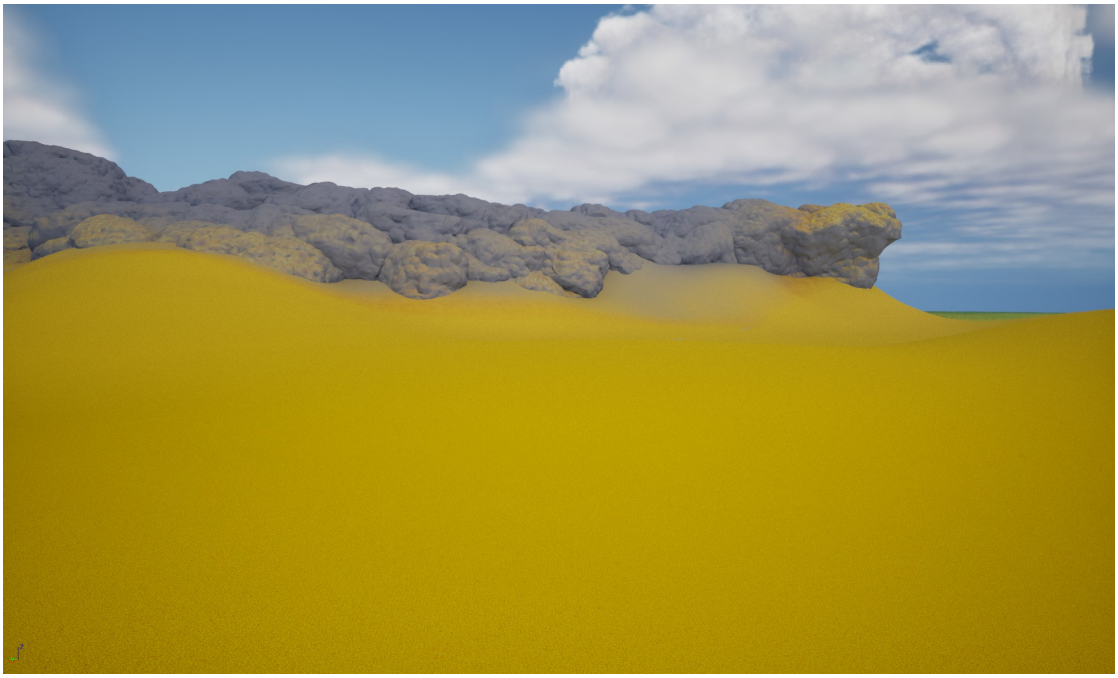


Figure 6.1.: Screenshot of the game's world rendered in the editor showing the desert biome in the lower half and the mountains biome in most of the upper half while the grasslands biome can be made out on the upper right

damage, while the Griffon characters use clearly noticeable wide swinging attacks causing significantly more damage. Both kinds of characters also have a variety of possible attacks. The random execution of the attacks further makes combat feel interesting. The game also provides a small world, which includes three separate biomes: desert, mountain and grasslands. All of them have a different terrain to make the game more interesting: The desert as seen in the lower part of image 6.1 has a number of dunes while the mountains seen on image 6.2 and 6.1 have a lot of rocky terrain run through by a flat path and the grasslands seen mostly on image 6.2 are mostly flat and planted with grass and trees. These different terrains also supported robust testing of the systems and now provide an interesting environment for anyone to play through.

Still, there are many systems which could be added, improved or optimized. A system to save the current game status should be looked at in more depth to allow the player to stop the game and continue at another time. Furthermore, leveling of any kind is still not implemented inside of the game meaning there is no way to actually progress inside the game. Currently, attacks are often detected even when

Figure 6.2.: Screenshot of the game's world rendered in the editor showing the mountain biome to the left and the grasslands biome to the right

the attack did not actually hit as the system used is quite inaccurate as explained in chapter 3.2.2. By adding a larger variety of attacks to the player character or by adding a rudimentary elemental system (e.g. fire, water etc.) with type advantages and disadvantages, the game would also gain significantly more strategical depth. Furthermore, the performance of the game is currently substandard due to little graphical optimization. Also, no extensive play tests have been carried out to verify that the systems have the intended effect on the player or to get a representative player feedback.

6.2. Reflexion

In general, the execution of the work went smoothly. Of course, there were some difficulties as sometimes the mechanic or the code simply did not work as expected. In some of these cases, it was a simple mistake in the code that could be easily corrected when found, in other cases large parts of a system had to be reworked to get the desired behavior. In rare cases I found some dysfunctional Unreal Engine features. The large scope of the project however always allowed me to work on something else and come back to a problem later with new ideas or newly gained knowledge. It turned out that my project plan was of great help: I started before

the summer holidays and consequently worked at least three hours per week. This prevented me from ending up with a huge stress at the end. I always have written down the most important systems that are still missing or not completed and used a dynamic plan so I could work on another system in periods when I was not able to make progress with the first one. Over the course of the work my debugging skills and my experience with the tools improved significantly, which enabled my to find bugs and errors much faster. Working on this project became a true passion of mine, which resulted in a significantly higher work-amount than the "officially" reserved lessons for the Maturaarbeit. I worked more on the practical part of the work at the beginning and invested more time for the written part at the end of the semester. Summing up, this work was really fun for me as I this project also became my hobby. Due to this work I could also learn a lot about C++ programming and debugging and an immense amount about the mechanics of games. The first two of which will certainly become useful when studying computer science.

Appendix

Material used for the Practical Work

Practical Work

Scagnetti, Raffael (January 2, 2024). *Maturaarbeit_R.Scagnetti_2023-24*. URL: https://github.com/PhoenixPhantom/Maturaarbeit_R.Scagnetti_2023-24 (visited on January 2, 2024).

Engine

Unreal Engine 5 (UE5) (May 11, 2023). Version 5.0-5.2. Epic Games Inc. URL: https://www.unrealengine.com/en-US/.

Used Assets

CGI Moon Kit (September 6, 2019). URL: https://svs.gsfc.nasa.gov/cgi-bin/details.cgi?aid=4720 (visited on December 4, 2023).

Deep Star Maps (May 3, 2023). URL: https://svs.gsfc.nasa.gov/3895 (visited on December 4, 2023).

jinyuliao (July 15, 2023). *GenericGraph*. URL: https://github.com/jinyuliao/GenericGraph/ (visited on October 5, 2023).

M.E. O'Neill (imneme) (April 8, 2022). *pcg-cpp*. URL: https://github.com/imneme/pcg-cpp (visited on October 18, 2023).

PeriTune (2015a). *Labyrinth(Royalty Free Music)*. URL: https://soundcloud.com/sei_peridot/labyrinth?in=sei_peridot/sets/peritunematerial (visited on December 15, 2023).

– (2015b). *WildField(Royalty Free Music)*. URL: https://soundcloud.com/sei_peridot/wildfield?in=sei_peridot/sets/peritunematerial (visited on December 15, 2023).

– (2019). *Desert3(Royalty Free Music))*. URL: https://soundcloud.com/sei_peridot/desert3?in=sei_peridot/sets/peritunematerial (visited on December 15, 2023).

– (2021). *Desert6(Royalty Free Music)*. URL: https://soundcloud.com/sei_peridot/desert6?in=sei_peridot/sets/peritunematerial (visited on December 15, 2023).

Quadruped Fantasy Creatures (October 30, 2018). Protofactor Inc. URL: https://www.unrealengine.com/marketplace/en-US/product/7f7775996f7442b187f6fa510ec9d289 (visited on September 6, 2023).

Scagnetti, Matteo (2023a). *Dunkler Wald*.

– (2023b). *Leichter Wald*.

– (2023c). *Wald Wind*.

Unreal Engine 5 (UE5) - Starter Content (May 11, 2023). Epic Games Inc. Included in the Engine.
Wolf's Howl Gives You Goosebumps (the good kind) (October 26, 2022). URL: https://www.youtube.com/watch?v=JQUHDWHa7WQ (visited on November 16, 2023).

Combat

Cofino, Ian (August 26, 2022). *Dreamscaper: Killer Combat on an Indie Budget*. URL: https://www.youtube.com/watch?v=3Omb5exWpd4 (visited on September 6, 2023).
How to make combo attacks - Beginner tutorial Unreal Engine 5 (September 28, 2022). URL: https://www.youtube.com/watch?v=rKbb4PBmFUA (visited on October 3, 2023).
Seth, Mihir (November 5, 2021). *Evolving Combat in 'God of War' for a New Perspective*. URL: https://www.youtube.com/watch?v=hE5tWF-Ou2k (visited on September 6, 2023).

Artificial Intelligence

All about BTTasks in C++ (January 27, 2021). URL: https://www.thegames.dev/?p=70 (visited on September 6, 2023).
Laley, Ryan (December 29, 2022). *Unreal Engine 5 Tutorial - AI Part 6: Patrol Path*. URL: https://www.youtube.com/watch?v=z8VJhDmAyx4 (visited on September 6, 2023).
– (February 17, 2023). *Unreal Engine 5 Tutorial - AI Part 9: Prediction Sense*. URL: https://www.youtube.com/watch?v=-EU8EedeVi4 (visited on September 6, 2023).

Visual FX

Ali, Ashif (June 20, 2019). *Unreal Engine Impact Effect in Niagara Tutorial*. URL: https://www.youtube.com/watch?v=8UThnwfEwng (visited on September 6, 2023).
– (February 28, 2022). *Animal Bite FX in UE4 Niagara Tutorial*. URL: https://www.youtube.com/watch?v=geHrMbvA5HA (visited on September 6, 2023).
Deplanque, Oliver (2023). *Animal bite*. URL: https://www.artstation.com/artwork/b5XEod (visited on September 6, 2023).
How to Add Camera Shake in Unreal Engine 5 (August 16, 2022). URL: https://www.youtube.com/watch?v=rTVb7nEhKv8 (visited on October 18, 2023).

Unreal Enngine Tutorial [UE5]: Aura Effect From Scratch - PART 1 [NIAGARA] (May 17, 2022). URL: https://www.youtube.com/watch?v=S_pA2uLhZWc&list=PLvYJUGYt8abFzLte-jehECt2LHWGLfM1X&pp=iAQB (visited on November 13, 2023). (Parts 1 - 3).

Sound FX

Unreal Engine 5 | Dynamic Footsteps with MetaSounds (July 24, 2021). URL: https://www.youtube.com/watch?v=pYxocPdtHBw (visited on September 6, 2023).

Unreal Engine 5 | MetaSound Footsteps (May 31, 2021). URL: https://www.youtube.com/watch?v=SXBEbs69sZ8 (visited on September 6, 2023).

Unreal Engine 5.1 | Creating Soundscapes (October 12, 2022). URL: https://www.youtube.com/watch?v=nS3leRwZbNk (visited on December 12, 2023).

Various Topics

Humpherys, Ben (July 10, 2016). *UPROPERTY(EditAnywhere)*. URL: https://benui.ca/unreal/uproperty/ (visited on August 6, 2023).

Świerad, Oskar (May 28, 2016). *UE4: How to fix translucent materials*. URL: https://www.youtube.com/watch?v=ieHpTG_P8Q0 (visited on August 6, 2023).

Wadstein, Mathew (July 10, 2016). *Unreal Concepts - Dynamic Circular Progress Bars (UE4)*. URL: https://www.youtube.com/watch?v=9NtSfPq95fQ (visited on September 6, 2023).

Texts Cited

Brown, Mark (October 29, 2020). *Who Gets to be Awesome in Games?* URL: https://www.youtube.com/watch?v=XSvclSkmdyY (visited on January 3, 2024).

Godot Engine 4.1 documentation in English (October 8, 2023). URL: https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_languages.html (visited on October 10, 2023).

Tayler, Dustin (September 14, 2023). *Video Game Mechanics for Beginners*. URL: https://www.gamedesigning.org/learn/basic-game-mechanics/ (visited on October 11, 2023).