Aalto University
School of Science

Master's Programme in Computer, Communication and Information Sciences

A Study and Implementation of Turn-Based Combat Systems in Role-Playing Games

Fatemeh Amereh

Master's Thesis
2024

Aalto University
School of Science

Author Fatemeh Amereh

Title A Study and Implementation of Turn-Based Combat Systems in Role-Playing Games

Degree programme Computer, Communication and Information Sciences

Major Computer Science

Supervisor Prof. Perttu Hämäläinen

Advisor Prof. Perttu Hämäläinen

Date 18 Decemeber 2024 **Number of pages** 32 **Language** English

Abstract

Role-playing games (RPGs) have been a prominent genre that has significantly shaped the video game industry, offering a wide selection of character classes and abilities that create a rich and dynamic combat experience. The aforementioned different ability types affect characters in various ways and provide intricate interactions between their effects. Turn-based RPGs, in particular, lean more toward strategic gameplay and feature more complex ability sets and effects. A well-developed combat system should support this variety, minimizing the effort required to introduce new character classes and abilities. This could help with faster iterations in the production cycle, which is essential considering the scale of modern RPGs.

In this thesis, we create a turn-based combat system that is modular and extensible, facilitating the process of feature development for programmers. Moreover, the system prioritizes designer-friendliness, enabling game designers to easily add new classes and abilities. The system is based on the Dungeons & Dragons 5th edition ruleset and is developed using Unreal Engine 5, utilizing the Gameplay Ability System plugin. We demonstrate the workflow to define new character classes and abilities and showcase how well the system integrates them.

Keywords Role-Playing Games, Turn-Based Combat System, Dungeons & Dragons, Modular Design, Game Development, Software Engineering Principles

Preface

I want to thank my family, as I am where I am today because of their unwavering support. I also want to thank Professor Perttu Hämäläinen for his guidance throughout this journey.

Otaniemi, 18 December 2024

Fatemeh Amereh

Contents

Abbreviations

RPG	Role-Playing Games
D&D	Dungeons & Dragons
UE	Unreal Engine
GAS	Gameplay Ability System
ECS	Entity-Component-System
ASC	Ability System Component
GE	Gameplay Effect

1 Introduction

In Role-playing games (RPGs), players guide characters through narrative-rich worlds, with a strong emphasis on intricate combat systems. [1] Combat System is a fundamental part of RPGs, and players spend a considerable amount of time in battle or in preparation for one. [3, 4] Combat typically includes managing character attributes, using special abilities, and strategizing to overcome enemies. Turn-based combat, as opposed to real-time systems, emphasizes strategic decision-making. [1]

RPGs feature diverse character classes, each with unique abilities and intricate interactions. These interactions, such as status effects that provide buffs or debuffs, can significantly influence combat flow. [1] Adding new character classes and abilities requires cooperative work from programmers and designers, which impacts the development timelines.

To address this, a modular and extensible combat system is essential. This approach enables developers to integrate new character classes and abilities with minimal effort, streamlining development and enabling rapid prototyping and testing, leading to a more agile development cycle.

This thesis explores the development of a turn-based combat system designed to address the complexities of modern RPG development. By breaking down the system into modular components, the system aims to minimize the time required to implement new abilities. At the same time, our system emphasizes designer-friendliness; enabling game designers to focus on creative aspects while developers handle the technical intricacies of system integration.

The combat system is built upon the well-established Dungeons & Dragons (D&D) 5th Edition ruleset, providing a familiar framework for character abilities and mechanics. The system is implemented using Unreal Engine 5, leveraging the Gameplay Ability System (GAS) plugin, a framework for the creation, management, and execution of abilities. The system is then analyzed for its capacity to support D&D abilities and necessary enhancements for a complete D&D combat implementation is presented.

The thesis is structured as follows: Chapter 2 gives an overview of the history of RPG games, and chapter 3 examines turn-based combat systems. As our implementation is heavily inspired by Dungeons & Dragons, we will cover its basics in chapter 4. Chapter 5 sets the goal for our turn-based combat system and chapter 6 will cover Unreal Engine and GAS as our development tools. Chapter 7 will go into more detail about our implementation and presents the results. Finally, Chapter 8 and 9 will cover the discussions, conclusions, and future work.

2 Role-Playing Games: A History

RPGs have their roots in tabletop gaming like Dungeons and Dragons, which emphasizes narrative-driven adventures and turn-based strategic combat. These core elements were adapted into early computer-based games during the late 1970s and early 1980s. Early titles such as Akalabeth (1980) and Rogue (1980) featured rudimentary graphical interfaces and turn-based combat systems that simulated the decision-making processes of tabletop gameplay.

As technology improved, RPGs entered a golden age and the 1980s and 1990s, marked by the rise of franchises like Wizardry, Ultima, and The Bard's Tale. These games introduced more sophisticated combat mechanics, including grid-based positioning, party-based systems, and class-specific abilities. Players could control multiple characters with distinct roles, such as tanks, healers, and damage dealers, emphasizing teamwork and strategy.

The 1990s brought Baldur's Gate and Planescape: Torment incorporating real-time with pause mechanics. This hybrid approach allowed players to issue commands in real-time while pausing to make tactical decisions.

The transition to 3D graphics during the late 1990s revolutionized RPG combat systems. Games like Diablo popularized action RPGs with fast-paced, real-time combat that appealed to a broader audience. Simultaneously, tactical systems remained prominent in titles such as Icewind Dale, which maintained the classic party-based approach with enhancements like line-of-sight mechanics and environmental effects.

In the modern era (2000s, Present), RPGs have achieved mainstream success with titles such as The Witcher 3 and Skyrim, which blend narrative depth with visually stunning combat systems. Combat has evolved to incorporate elements of environmental interaction, resource management, and emergent gameplay. For example, games like Divinity: Original Sin allows players to manipulate terrain and combine abilities to create unique tactical advantages. Massively multiplayer online role-playing games (MMORPGs) such as World of Warcraft introduced large-scale battles and cooperative combat, adding new layers of complexity to RPG combat systems. [1]

RPGs can be broadly categorized by their combat systems:

1. Turn-Based: Players take action in turns, where each character or enemy acts sequentially. Examples: Persona, XCOM, Divinity: Original Sin, Baldur's Gate. [2]

2. Real-Time: Combat occurs continuously, with all characters acting simultaneously. Examples: The Elder Scrolls, Diablo, Dark Souls.

3. Hybrid: "Pause & Play" and "Active Time Battle (ATB)" systems are a blend of both turn-based and real-time combat. Players are allowed to pause combat anytime, or when they have accumulated enough points, to issue commands or make tactical decisions.

3 Turn-Based Combat systems

A turn-based combat system is a gameplay mechanic in which players and opponents take turns to perform actions, such as attacking, defending, or using abilities, rather than engaging in continuous real-time interactions. This allows players to carefully strategize, considering factors like positioning, resource management, and ability synergies. Combat is divided into distinct turns, during which each participant executes their moves independently. Turns often follow a specific order, determined by factors like initiative, speed stats, or a pre-defined sequence. During the combat, players have the time to evaluate the battlefield and plan their actions accordingly. This adds to the strategic depth and allows for tactical decision-making. In many games, players control a group of characters, each with unique abilities and roles. Turn-based systems provide opportunities to coordinate party members effectively. Also, players often manage limited resources, such as action points, mana, or stamina, to execute abilities or moves during their turn.

Turn-based combat systems have maintained their appeal in modern RPGs by offering players strategic depth and decision-making [5], and a nostalgic connection to classic gaming experiences. Recent titles have embraced and innovated traditional turn-based mechanics. One notable example is Baldur's Gate 3, released by Larian Studios in 2023. The game was praised by both players and critics, achieved substantial commercial success, and won the Game of the Year Award, proving the viability of turn-based combat in contemporary gaming. [6]

Baldur's Gate 3 is based on Dungeons and Dragons 5th edition and features 12 character classes and about 250 core spells. [7] In its turn-based combat, players control individual characters during their turns, deciding on positioning, abilities, and items. Characters' abilities and spells can interact in synergistic ways. Using spells like Bless can boost your party members' actions, and spells like Bane can hinder enemies. A well-timed combination of abilities can turn the tides in your favor during combat.

Turn-based combat systems are the focus of this thesis, and the game Baldur's Gate 3 has heavily inspired this work. In the next section, we will discuss D&D, a well-known table-top RPG with an intricate turn-based system.

4 Dungeons & Dragons

4.1 History

The history of Dungeons & Dragons (D&D) began in the 1960s and early 1970s with the rise of tabletop wargames, which focused on simulating battles with miniatures and complex rule sets. Playing at the World by Jon Peterson provides a comprehensive history of D&D. Peterson details the development of D&D in 1974, starting with a series of experimental rule systems designed by Gary Gygax and Dave Arneson. D&D was initially conceived as a set of rules that would allow players to simulate fantasy adventures with miniatures, but it quickly grew into a more narrative-driven game where players would assume the roles of individual characters, each with unique abilities and personalities. [8] D&D influenced the fantasy literature, film, and the games industry, inspiring many novels, movies, television shows, and the early computer RPGs. It also contributed to community building; many people would gather to embark on a campaign in a fantasy world together. [9]

D&D has contributed to game design by formalizing RPG mechanics. [8] Peterson highlights how D&D introduced the concept of character classes (such as fighter, wizard, and cleric), character sheets, leveling systems, and hit points, elements that are now standard in RPGs. The game's use of polyhedral dice added an element of chance, making gameplay unpredictable and dynamic, while the Dungeon Master (DM) took on the role of guiding the narrative, creating challenges, and acting as both storyteller and referee.

4.2 Combat

Combat is central to D&D's experience. [10] The combat in D&D is a turn-based system where players and their characters battle monsters or other opponents. Players take actions in the game and they are the core gameplay mechanic. There are three main types of actions: Action, Bonus Action, and Reaction. An Action is the most common type and allows for actions like attacking, casting a spell, or using a special ability. A Bonus Action is a less frequent action that can be used in addition to an Action, while a Reaction is a quick action that can be taken in response to a specific event. To resolve an action, players typically roll a 20-sided die (d20) and add relevant modifiers based on their attributes, abilities, and equipment. If the total roll meets or exceeds a certain target number (often called a Difficulty Class or DC), the action is successful. As an example, a spell attack roll for a wizard is resolved by rolling a d20 and adding some modifiers on top of it, such as the intelligence modifier, proficiency bonus (a modifier scaled by character level), and any possible bonus from status effects.

4.3 Main Attributes

Each character has six main attributes: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. These attributes influence various aspects of gameplay, such as attack rolls, saving throws, and skill checks:

1. Strength: Governs physical power, impacting melee attacks, carrying capacity, and saving throws against being forced prone.

2. Dexterity: Represents agility and reflexes, affecting ranged attacks, AC (Armor Class), initiative, and saving throws against traps and being surprised.

3. Constitution: Determines a character's health and stamina, influencing hit points and saving throws against poisons and diseases.

4. Intelligence: Reflects mental acuity, impacting spellcasting ability, skill checks related to knowledge, and saving throws against illusions and mind-affecting effects.

5. Wisdom: Represents perception and intuition, affecting wisdom-based skill checks, initiative, and saving throws against enchantment and charm effects.

6. Charisma: Represents personality, influence, and leadership, affecting persuasion, deception, and social interactions, as well as certain spellcasting abilities.

4.4 Character Classes

There are 13 official character classes in D&D 5th edition. Namely, Artificer, Barbarian, Bard, Cleric, Druid, Fighter, Monk, Paladin, Ranger, Rogue, Sorcerer, Warlock, and Wizard. They can be categorized as Martial, Spell-casting, or Hybrid classes:

- Martial classes: Fighter, Barbarian, Monk, Paladin, Ranger. Strength, Dexterity, or Constitution are key attributes based on class.

- Spell-casting classes: Wizard, Sorcerer, Warlock, Druid, Cleric. Intelligence, Wisdom, or Charisma are key attributes based on class.

- Hybrid Classes: Bard, Artificer. They are versatile. Can excel in both combat and social situations.

4.5 Abilities and Conditions

DnD offers a wide variety of character abilities and spells, often unique to specific character classes. As an example, Sneak Attack ability is specific to the Rouge character class, and Action Surge is unique to the Fighter class. Different spell abilities like Fireball, Healing Word, and Hold Person offer damage, buff or debuff, and utility. This variety of abilities in D&D comes from the different character classes, subclasses, races, feats, and spells. For example, the Fighter class excels in melee combat while the Wizard class focuses on spellcasting and intricate rituals; each has specific abilities to showcase these differences.

Some of the spells in D&D can inflict conditions on characters, impacting their abilities and course of combat. Some common conditions are:

- Blinded: The creature can't see and automatically fails any ability check that requires sight. Has disadvantage on attack rolls.
- Incapacitated: The creature can't take actions or reactions.
- Poisoned: The creature has disadvantage on attack rolls and ability checks.
- Prone: The creature is lying on the ground and has disadvantage on attack rolls and Dexterity saving throws.

By carefully selecting and timing your spells to inflict specific conditions on your enemies, you can create advantageous situations for your party.

As mentioned before, Baldur's Gate 3 developed by Larian Studios is based on D&D and represents a modern and interactive implementation of D&D mechanics. Players can create characters using the same class options available in D&D 5th Edition, and combat follows the same turn-based model. Players take turns based on initiative rolls, which determine the order of actions in combat.

5 Towards Implementing a D&D Combat System

Implementing a turn-based combat system like D&D's combat with its vast array of abilities, character classes, and mechanics in a digital format, is challenging. Each class in D&D has unique mechanics, strengths, weaknesses, and its own array of abilities. Many of these abilities synergize with others, affecting characters and the environment. A well-developed turn-based combat system based on D&D should support this myriad of features that have complex synergy with each other.

5.1 Software Engineering Principles

If the complexity mentioned beforehand is not handled well, it could lead to prolonged development cycles due to tightly coupled code and lack of modularity. This, in turn, makes the designers dependent on programmers for even minor updates. It could also increase the likelihood of bugs and unintended interactions between various system components. To prevent these potential issues, we should take lessons from software engineering principles.

5.1.1 Modularity

Modularity is the division of a complex system into various parts. [11] Modular software design involves breaking down a larger codebase into smaller sections, or modules, each responsible for a specific function. This strategy improves the design process by enabling easier debugging, testing, and understanding of the system. [12] As the game develops, modularity supports easier maintenance and better scalability. [13]

It should be noted that a successful implementation of modular design relies on clearly defined design rules and well-defined guidelines that dictate how individual modules (or components) should interact and fit together within the overall system. These rules ensure that modules can be developed, tested, and integrated independently without causing inconsistencies or integration issues later on in the process. [14]

5.1.2 Coupling

The root cause of "bad code" lies in the interdependence among its modules, referred to as coupling. [15] When code is tightly coupled, it becomes challenging to modify; altering one component can impact its dependents, potentially causing them to break. [16, 17] As a result, extending the software to support new features could require rework and modification of multiple modules. In contrast, a loosely coupled code improves reusability across the system which can save development time and effort. Coupling also helps with maintainability.

5.1.3 Composition and inheritance

Inheritance and Composition are two fundamental object-oriented programming (OOP) techniques that enable developers to extend and enhance the properties of

classes.

- Inheritance allows a class to inherit properties and behaviors from a parent class, helping with reusability and creating hierarchical relationships. On the negative side, it can lead to tightly coupled systems, making maintenance and scalability more challenging. [18]

- Composition involves building classes by combining objects of other classes, focusing on the "has-a" relationship rather than the "is-a" relationship emphasized by inheritance. This method advocated for loose coupling and modularity. [18, 16]

When it comes to choosing between the two techniques, the specific requirements of the system should be taken into consideration. While inheritance is suitable for representing hierarchical relationships, composition is preferable for creating flexible and modular systems.

5.1.4 Software Design goals

Drawing from these principles, to develop a maintainable and customizable combat system that can easily support the vast amount of features in D&D, we should aim for a modular and extensible system; one that reuses shared components that have well-defined functions. These principles will reflect on our development tool choices (Chapter 6) as well as our system design and implementation (Chapter 7).

5.2 Tools for Designers

Tools in game development support creativity and rapid prototyping. Technology acts as a medium between ideas and their realization during the creative process, and for games, ideas need to be implemented and tested before they can be fully understood. This highlights the need for fast iterative testing. [19, 20] To achieve this goal, our system should prioritize designer-friendliness for the tools created. This leads to less dependency on programmers and faster iterations.

In this context, providing visual tools and editors is beneficial, as they allow designers to define character classes, abilities, and stats without needing extensive programming knowledge. By using data-driven systems, these tools enable designers to work more independently and iterate quickly. Furthermore, modularity and reusable components can also help designers as they can mix and match components to reach their desired outcomes.

6 Development Tools

We have set two goals for our combat system. It should be: Modular and extensible, and designer-friendly. This will allow the easy addition of new character classes and abilities. We take these goals into consideration while choosing our tools.

In this thesis, We use Unreal Engine 5 and utilize the Gameplay Ability System plugin. In the following sections, we will talk more about these tools and how they have helped with the development process.

6.1 Unreal Engine 5

Unreal Engine 5, developed by Epic Games, is a 3D creation platform, widely used in game development. It offers tools and technologies that enhance the workflow and speed of game development. One such tool that empowers game designers, is the blueprint visual scripting. It allows designers to implement gameplay mechanics without programming expertise. This accelerates prototyping, making it easier to experiment and iterate on game ideas. This goes well with our goal of being designer-friendly. By exposing functionality to the blueprint system and editors, designers will become more independent.

Unreal Engine has many tools that facilitate industry-level game development. From its advanced graphics capabilities to scalable tools tailored for large-scale projects. We will discuss one such tool in more detail in the next section.

6.2 Gameplay Ability System

"The Gameplay Ability System (GAS) is a framework for building attributes, abilities, and interactions that an Actor can own and trigger. The system is designed to be adapted to a wide variety of Gameplay-Driven projects such as Role-Playing Games(RPGs), Action-Adventure games, and Multiplayer Online Battle Arena games(MOBA)." [21] It's a powerful plugin that is designed to simplify the creation and management of complex gameplay mechanics, particularly those involving abilities, effects, and attributes. GAS is highly modular and can handle games with numerous abilities and effects. It also comes with built-in multiplayer support.

GAS is inspired by Entity-Component-System (ECS) principles. ECS is a design pattern used in game development and other software systems to manage game objects and their behaviors in a modular, efficient, and scalable way. GAS resembles ECS, particularly in its modularity, separation of concerns, and scalability. However, it is not a pure ECS implementation, as it integrates with Unreal Engine's object-oriented architecture and leverages Actor Components for functionality.

GAS is built around several key components, discussed in the following subsections.

6.2.1 Gameplay Attributes

Gameplay Attributes represent the numerical properties of an Actor, such as health, stamina, or strength. They are stored in an Attribute Set, which is a custom UObject

class. They interact with Gameplay Effects to modify their values dynamically. For example, an attribute like 'Health' can be reduced by damage effects or increased by healing abilities.

6.2.2 Gameplay Abilities

Gameplay Abilities define specific actions or powers that an Actor can perform, such as attacking, healing, or using special skills. They Contain the logic for ability activation, execution, and cancellation and can be triggered by player input, timers, or other in-game events. For example, a "Fireball" ability could involve logic for targeting, applying damage, and triggering animations when executed. ActivateAbility() function of abilities can be overridden with custom logic. The function is also exposed to Blueprint which makes it easier to work with for designers. The workflow of abilities can be seen in figure 1.

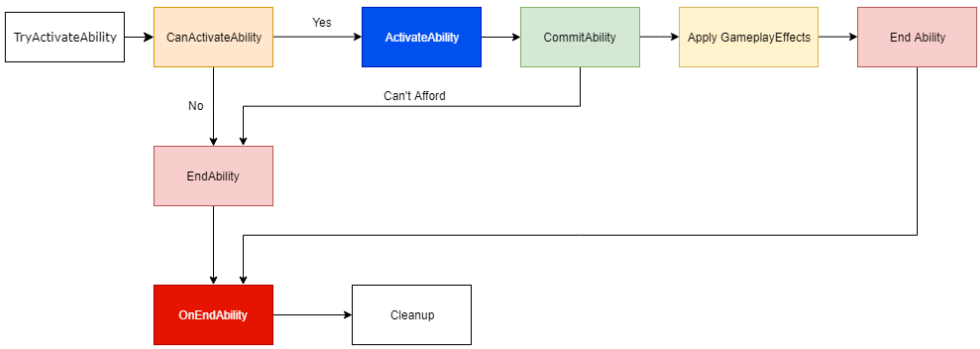

Figure 1: Flowchart of a simple gameplay ability. [22]

6.2.3 Gameplay Tags

Gameplay Tags are labels used to add conditions, filter interactions, or categorize effects. They can be used for conditional logic, such as restricting abilities or determining effect application. For example, a character with the "ImmuneToFire" tag would not take damage from fire-based abilities.

6.2.4 Gameplay Effects (GEs)

Gameplay Effects are vessels through which abilities change Attributes and Gameplay Tags on themselves and others. They define how attributes are modified, such as applying damage, buffs, or debuffs. Gameplay Effects change Attributes through Modifiers and Executions (GameplayEffectExecutionCalculation). Gameplay Effects have three types of duration: Instant, Duration, and Infinite. They can be set to stack. For example, a "Poison" effect might apply damage to health over time, while a "Shield" effect increases defense for a limited duration. The many settings this data-driven asset provides can be seen in figure 2. Designers can configure the basic settings, add different components, modifiers, and executions.

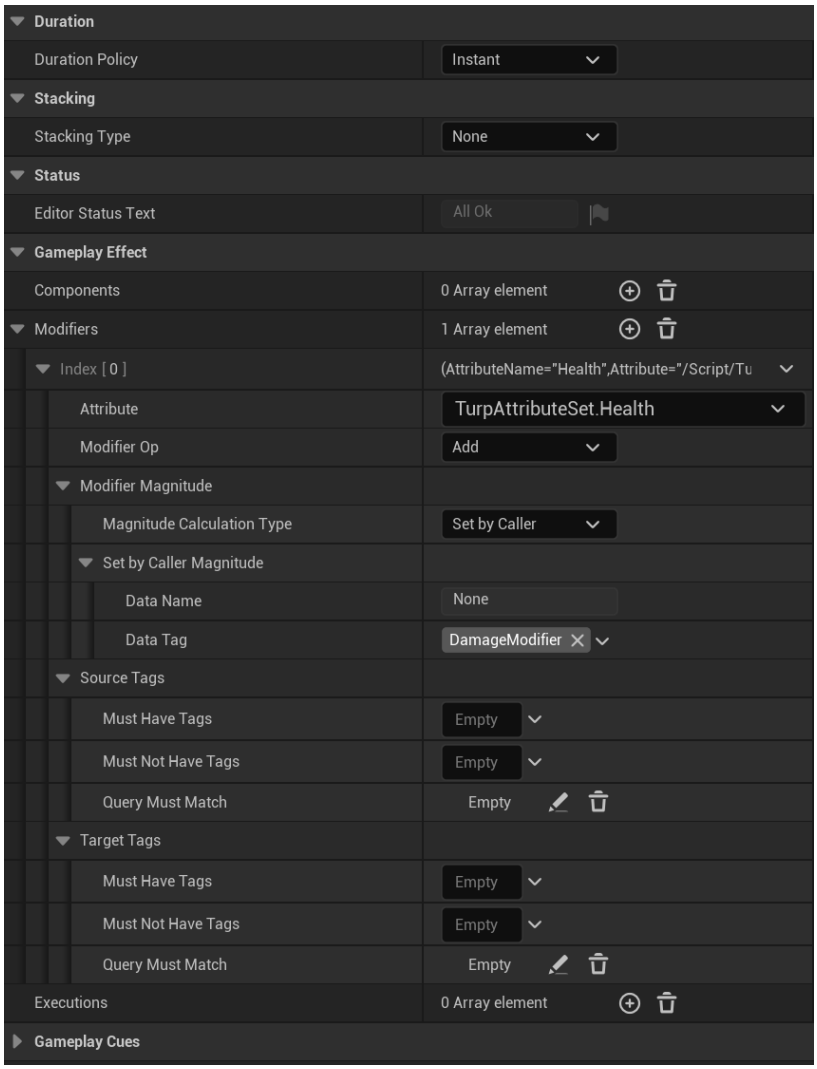

Figure 2: A Gameplay Effect asset in Unreal Engine 5.

6.2.5 Ability System Component (ASC)

The Ability System Component is the core of GAS and is responsible for managing abilities, attributes, and gameplay effects for an Actor. The component should be attached to an Actor to enable the use of GAS (Figure 3). It keeps track of the Actor's abilities, active gameplay effects, and attribute values. It also handles ability activation, execution, and cooldowns.

6.3 Considerations

While there are benefits to using GAS, there are certain limitations as well. In what follows we give an overview of GAS's strengths and limitations.

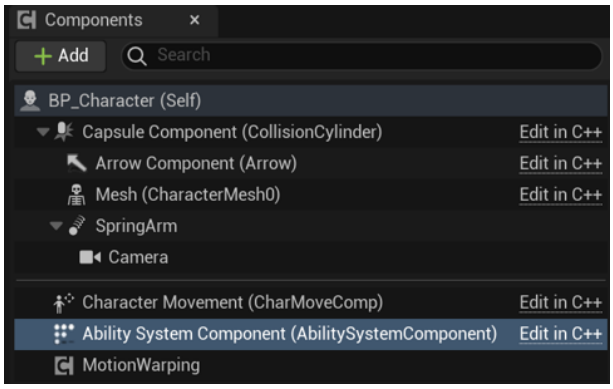

Figure 3: List of components of a character visible in Blueprint.

6.3.1 Strengths

GAS has been battle-tested in AAA commercial games such as Paragon and Fortnite among others. [22] Let's go over its main selling points.

- An out-of-the-box solution: GAS provides an extensive system for implementing abilities, attributes, and effects. This can reduce the need for custom system implementation, allowing developers to focus on higher-level gameplay design.
- Scalability: GAS handles complex interactions and large numbers of abilities and effects efficiently This makes it well-suited for games with intricate and numerous mechanics, such as RPGs.
- Modularity: The modular structure of GAS allows developers to easily reuse abilities and effects, which enhances reusability and maintainability.
- Designer-Friendliness: GAS has blueprint integration and reusable Modular Components. This enables designers to quickly prototype abilities and tweak parameters such as cooldowns, costs, and durations directly in the editor.

These points align very well with our goal of a modular and designer-friendly RPG combat system which makes us more inclined towards using GAS.

6.3.2 Limitations

There are some drawbacks to using GAS which could be important depending on your team and project's needs.

- Steep Learning Curve: GAS is a complex system with interconnected components. There are often many ways to achieve the same goal with slight differences, which requires a comprehensive understanding of the system and its inner workings. Effectively using the framework can be challenging, particularly for new users of the plugin.
- Overhead for Simple Projects: For projects with a limited number of abilities and more simple mechanics, GAS may introduce unnecessary complexity.

- Customization Challenges: Depending on the project's needs, you might still need to extend the system which often requires diving into Unreal Engine's C++ codebase.

Regarding customization challenges, GAS lacks native support for turn-based mechanics. For example, durations for Gameplay Effects are only in seconds. To achieve a turn-based system, you have to extend the system either by having your own Gameplay Effect definition or by replacing the timer which requires modifying the source code. While this complexity diminishes our gains from using GAS, the advantages mentioned before outweigh the challenges for our project.

7 Design & Implementation

Our goal is to implement an RPG turn-based combat system based on D&D 5th edition that prioritizes ease of use and flexibility. This enables programmers to efficiently extend its functionality and introduce new features while empowering designers to easily define and integrate new abilities and character classes.

We will begin by examining different classes and abilities present in D&D. Subsequently, we will present our design decisions and implementation details in the following sections.

7.1 D&D Specifications

In D&D, characters belong to various classes, each having a unique set of abilities. Characters use these abilities to deal damage, heal, and inflict various status effects. Interactions caused by status effects are one of the challenging parts to implement. For instance, if the character is "Blinded" which can be caused by the ability called "Blindness", the character's attack rolls will be adversely affected. Similarly, if an ability grants immunity to ice damage, subsequent attacks of that type will be nullified.

7.1.1 Ability Structure

We need a well-defined structure to organize D&D abilities to facilitate both the definition and programming of their logic.

Spells are often more complicated than melee attacks. Let's examine a few spells.

- "Fire Bolt" deals instant damage to the target upon hit. Requires attack roll.
- "Fireball" deals instant full damage to targets who fail a dexterity saving throw, and half as much to those who succeed.
- "Blindness" inflicts the condition "Blind" to the target for 10 turns if it fails a constitution saving throw. Target has disadvantage on attack rolls and attack rolls against it has advantage. If the target succeeds the save, condition is removed.
- "Healing word" instantly heals the target for an amount.

Based on these examples, we can categorize abilities into two primary groups: those that primarily deal damage and those that primarily inflict conditions. Conditions are represented as Gameplay Tags assigned to characters. These tags can affect actions like attack rolls, ability checks, and saving throws, or grant a temporary increase/decrease to an attribute or modifier.

In what follows, we provide the design based on this categorization.

7.2 GAS Implementation

As discussed in section 6.3.2, GAS doesn't come with native turn-based support. To address this, we implement a custom definition of the Gameplay Effect class that

understands the concept of turns. This modification allows us to have turn-based duration for our effects and abilities. Additionally, we need a mechanism to manage these custom effects for each character, which we will achieve by extending the Ability System Component.

7.2.1 Effect Class

For our custom effect class, to ensure a designer-friendly workflow, we take advantage of Data Assets in Unreal Engine as a place where Effect types are defined. Data Assets are specialized assets that store and organize game data, such as character stats, item properties, or game settings. This approach allows designers to easily manage and modify game data without requiring changes to the underlying code. Figures 4 and 5, showcase the definition of two abilities. There are fields to determine the duration of the effect in turns, and whether this effect can stack. There are Damage and Condition settings based on the categorization discussed in section 7.1.1.

7.2.2 Conditions

To manage and define conditions, we utilize another dedicated data asset. This asset specifies the actions affected by each condition. Figure 6, showcases the definition of "Poisoned" and "Blessed" conditions. The "Poisoned" condition, for instance, affects the attack roll action and imposes disadvantage on it. The "Blessed" condition, on the other hand, affects many modifiers as well as the attack roll.

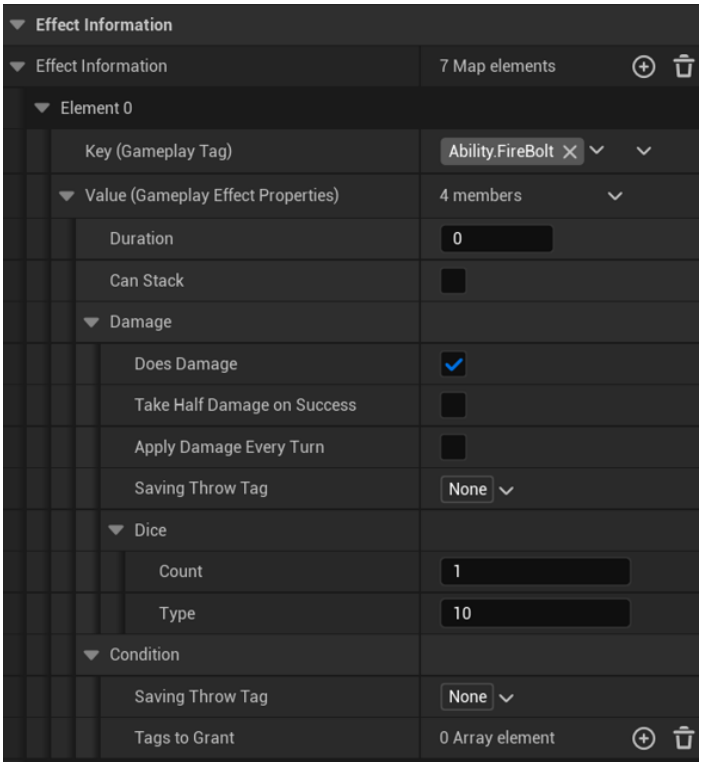

Figure 4: Fire Bolt ability definition in the custom effect data asset.

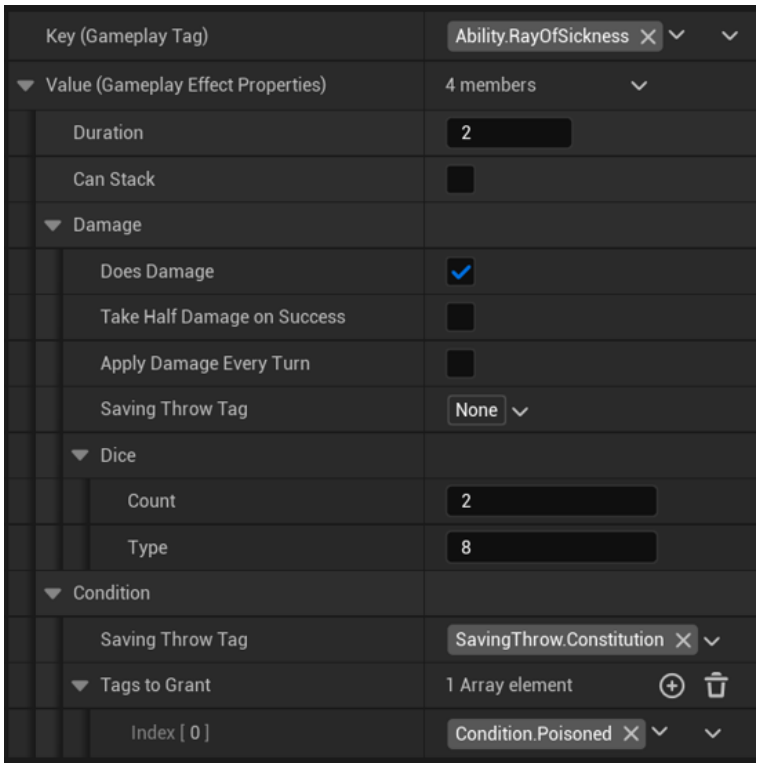

Figure 5: Ray of Sickness ability definition in the custom effect data asset.

7.2.3 Abilities

Effects are applied to the ASC of a character through Abilities, which in turn changes the attributes or applies a condition. The Gameplay Ability class is exposed to blueprints, which makes the definition of the ability's logic easier for designers. Fig 7 illustrates the Blueprint implementation of the "Fire Bolt" ability. Upon activation, the 'Activate()' event triggers the following sequence:

1. Target Acquisition: An asynchronous task is executed; the ability's execution will halt until a mouse click is detected and the target is received.

2. Animation Playback: A specific animation has to play, which is a character holding a fireball and throwing it at the target.

3. Projectile Launch: Finally, a projectile is spawned and launched towards the selected target.

7.2.4 Character Classes

Characters in D&D come with their own set of abilities and default attributes. Fighters have a higher strength modifier, while Wizards have more points in intelligence. We need a mechanism to define these values. We define another custom data asset that allows for an easy description of classes and their settings as demonstrated in figure 8.

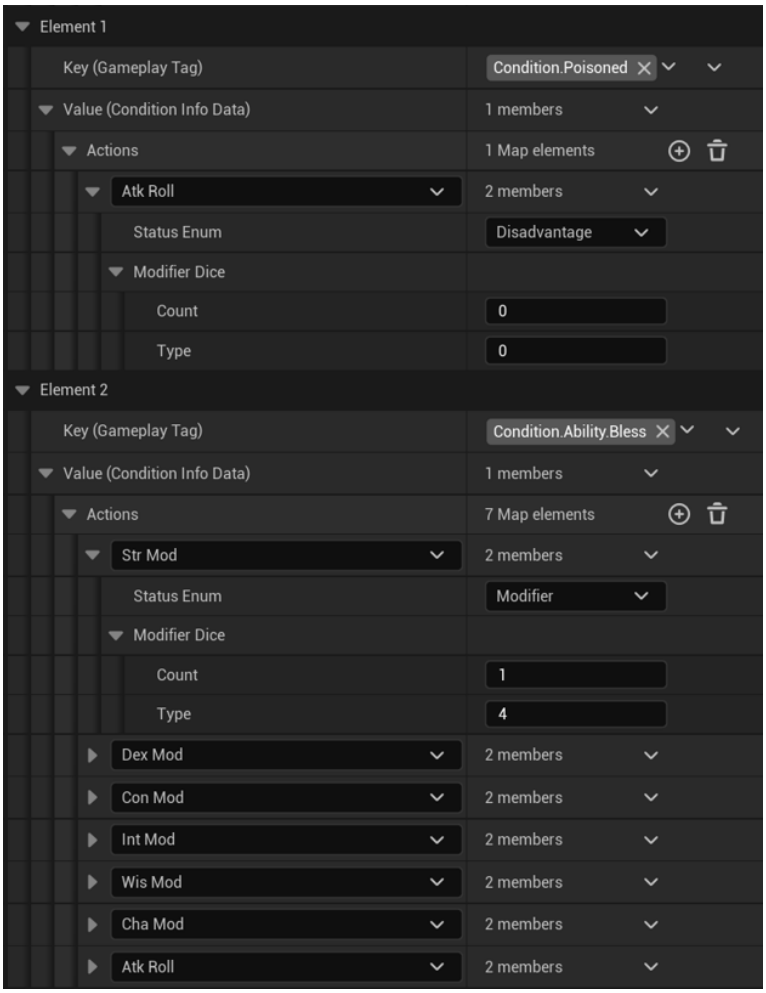

Figure 6: Conditions' data asset.

7.2.5 Extended ASC

The Ability System Component within GAS manages the tracking of default Gameplay Effects. It maintains a record of active effects on the character and raises events relative to the addition and removal of these effects. Since our implementation utilizes a custom effect definition (Refer to section 7.2.1) instead of the native Gameplay Effect class, we require a separate mechanism for managing their lifetimes. Therefore, we extend the ASC to track active custom effects and Gameplay Tags that influence ability actions. This extension involves maintaining internal maps to store and manage these elements.

7.3 Combat Packet

In a turn-based combat system, at any given time, there is only one active ability being executed, and many actors and systems within the game need access to the information about the currently active ability. Consider the ability "Counter Spell" in D&D. This

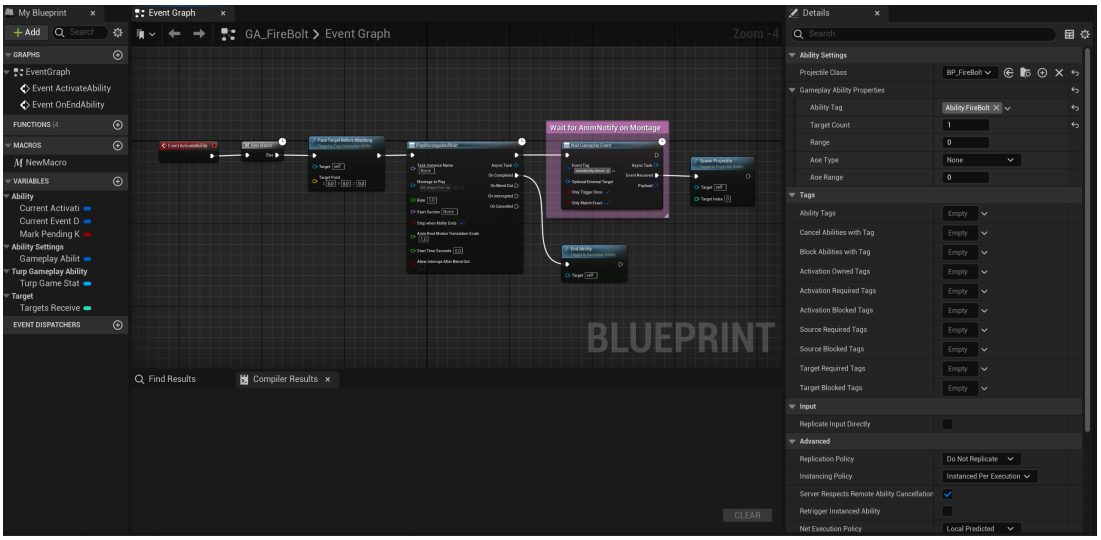

Figure 7: Fire Bolt ability blueprint.

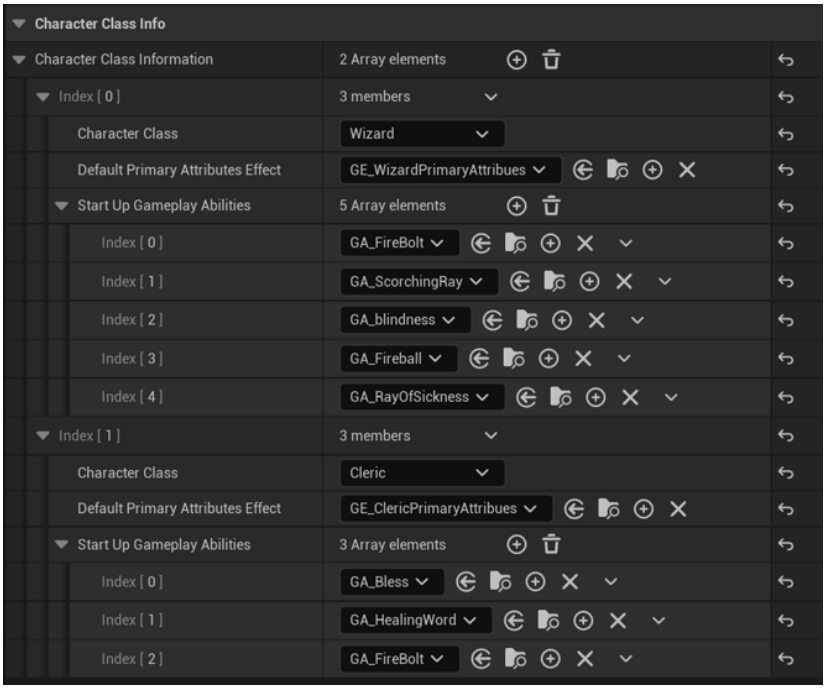

Figure 8: Character class information data asset.

spell allows a caster to nullify another caster's spell, under certain conditions. Let's imagine a scenario where my character casts a spell on Enemy A. Enemy B has the ability to counter-spell my attack, effectively nullifying it. Now, Enemy B needs a way of knowing about the type of ability my character has used and whether the enemy can counter it. To account for these scenarios and to make the information access workflow easier, we introduce a global Combat Packet, which stores relevant information about the active ability. The Combat Packet contains the following:

- Ability's Source: The actor or entity initiating the ability.
- Ability's Targets: The entities targeted by the ability, which can include enemies, allies, or the environment.
- Ability Properties: Detailed information about the ability, particularly the effect that it applies.

The workflow of the Combat Packet is as follows. When an ability is triggered during combat, the Combat Packet is initialized with the initiator actor's ASC and the ability's properties. Once the targets for the ability are received, the respective property on the Combat Packet is populated. The packet is now ready for read by any actor or system that needs the information. Using a single global Combat Packet offers several advantages:

1. It enables centralized data access, reducing redundancy, and offering performance benefits.
2. This design aligns with our modular development principles, allowing for easy modifications and extensions to the system.
3. It provides a scalable framework should the game grow in complexity, or new features become essential.

7.4 Event-driven Programming

One of the most useful paradigms used in Unreal Engine is event-driven programming. It allows for building and managing complex interactions in games. Event-driven programming revolves around reacting to events or triggers. Unreal Engine extensively uses Delegates and Multicast Delegates to implement event-driven patterns. These allow different parts of the game to subscribe to and react to events without directly depending on one another, creating a loosely-coupled architecture. Using events increases modularity, reusability, and scalability, as well as providing performance benefits. [23]

GAS utilizes Event-driven programming extensively, enabling communication between different systems (abilities, effects, and attributes). Delegates are used for various scenarios such as ability activation, attribute changes, and gameplay effect events. We take lessons from this system and incorporate event-driven programming into our system. Below are a few examples of our use cases:

- Updating status maps on ASC: When a condition is applied, the status maps get updated.
- Continuing Ability Execution: A gameplay ability should resume execution after receiving its targets.
- Spawning projectiles or applying effects: When the correct keyframe has reached an ability's animation, a projectile should be spawned (For appropriate abilities) or the ability's effect should be applied.

- UI Communication: The user interface should reflect real-time character updates.

7.4.1 Model-View-Controller Pattern

The Model-View-Controller (MVC) pattern utilizes event-driven programming to facilitate communication between its components (Model, View, and Controller) and event-driven programming ensures that changes in one component are propagated efficiently to others without creating tight coupling between them.

The MVC pattern helps with the separation of concerns, as it splits the system into 3 distinct components that have different responsibilities. [24] This pattern is encouraged to be used for developing graphical user interfaces as it offers the flexibility to develop and change views and controllers independently of the model. [25] This is aligned with our modularity principles. The pattern maps to our system as follows:

- Model: ASC and other systems that provide data like a turn-based manager.
- View: User interface classes.
- Controller: Middleware classes connecting the Model and View.

Using this pattern we can reflect the attribute and status changes of the character in the UI efficiently.

7.5 Demonstration

The combat system draws inspiration from Baldur's Gate 3. Ability icons are displayed at the bottom of the screen, allowing players to select and activate them. Upon selecting a target, the corresponding animation plays, and the effect is applied if successful. A combat log provides detailed information about each ability's outcome. Figure 9 illustrates the result of a failed "Fire Bolt" attack. The target's name is displayed, along with the attack roll, all applied modifiers, and the target's Armor Class. Figure 10 shows the successful application of the "Blindness" ability. Target has failed the saving throw and gained the Blinded condition. Relevant dice roll values are also presented.

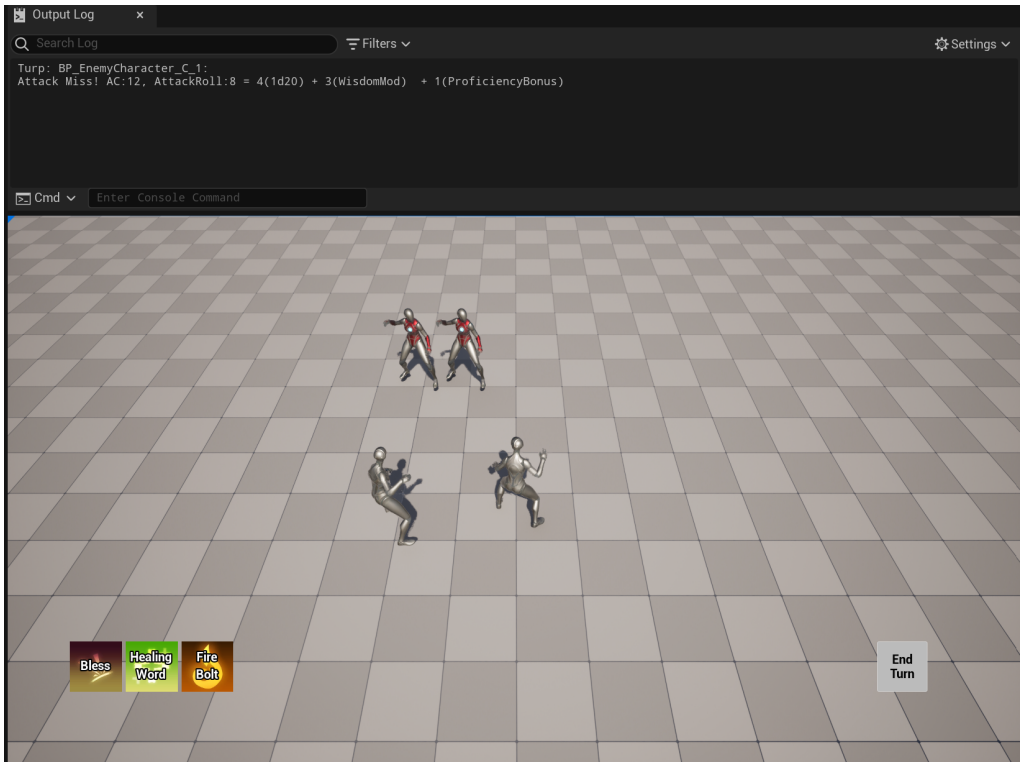

Figure 9: Fire Bolt ability result showcase.

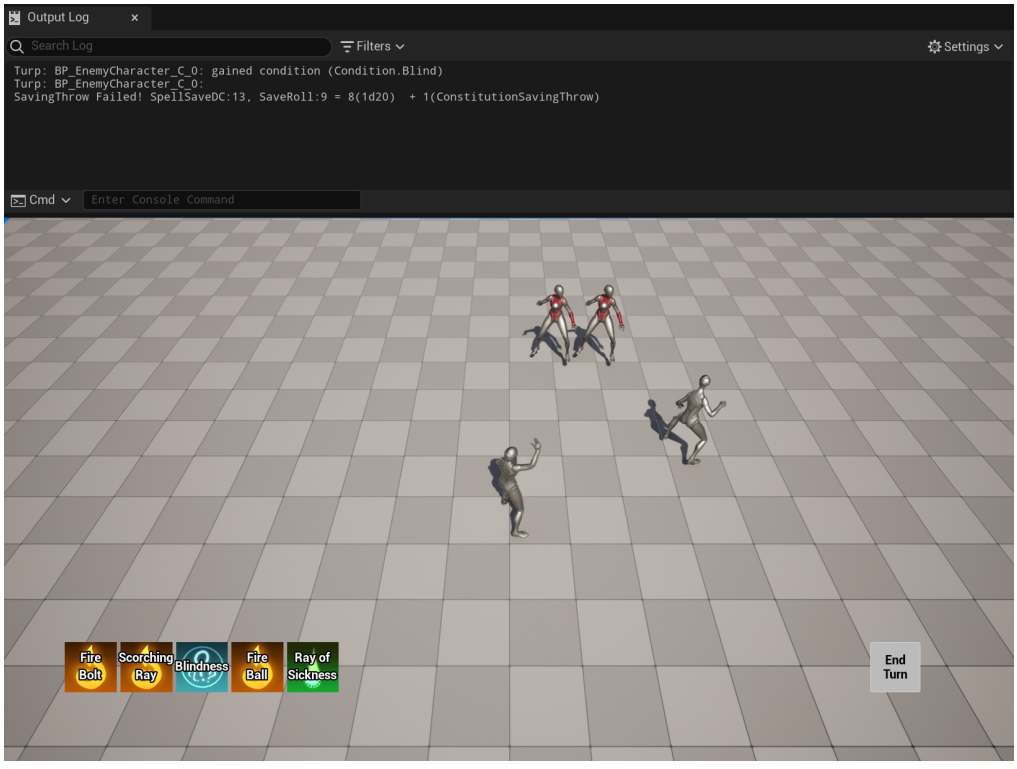

Figure 10: Blindness ability result showcase.

8 Discussions

Our turn-based combat system can support D&D abilities to a great extent and it can simulate the core functionality of most abilities. However, given the vastness of D&D, some features are yet to be implemented. These include damage types, ability costs, and condition activation at custom times (e.g., the start or end of a turn), among others. To achieve our goal of a modular and extensible system, we have leveraged software engineering principles and patterns (Sections 5.1 and 7.4), and selected tools that align with these objectives (Chapter 6); this will facilitate the process of adding new features significantly. To support damage types, a new setting can be added to our existing effect properties (Section 7.2.1). For instance, for the FireBolt ability, this setting could be represented as a gameplay tag, such as "Damage.Fire.". This tag would then be checked against the current conditions on the character as an application rule; If the character has the tag "Immunity.Fire", the effect of Fire Bolt would be nullified. The System is well capable of achieving these interactions. Similarly, ability costs can be supported by defining custom settings and utilizing the built-in cost management features in GAS.

In our implementation, we utilized Unreal Engine's Gameplay Ability System (GAS) plugin, despite its lack of native turn-based support. To account for this limitation, we developed a custom effect class tailored to our specific requirements (7.2.1). While this fulfilled our needs, it also meant giving up on Unreal's built-in, component-based Gameplay Effect class. This trade-off reduced the value GAS had to offer, particularly in its handling of effects and reusability through components. However, GAS remained an appealing choice due to its modular architecture and robust handling of the main building blocks of a combat system as mentioned in section 6.2, aligning well with our design goals for the system.

GAS is designed to support a wide variety of design paradigms, contributing to its highly modular and component-based nature. However, this flexibility comes at the cost of added complexity. Depending on the needs of a game project, modifications to the system may be inevitable. This complexity can significantly hinder development, as developers may find themselves needing to delve into the plugin's source code to implement necessary changes, replace some components, or change their game design to fit within the constraints of the system.

It can be argued that if the design rules of our project are well-defined, such as the D&D ruleset, designing and building a custom system, tailored specifically to those needs becomes a viable alternative. By applying the lessons learned from working with GAS and utilizing the software engineering principles we can develop a system both modular and extensible, tailored to the unique requirements of D&D. This will make the system more user-friendly and provide full control over all aspects of its design and functionality.

9 Conclusions & Future Work

Role-playing games have been a prominent genre in the games industry, with successful games each year. One of the most notable examples is Baldur's Gate 3 by Larian Studios based on Dungeons & Dragons ruleset, which inspired this thesis of creating a turn-based combat system, capable of supporting different character classes and abilities with ease. In chapter 5 of the thesis, we established the importance of software engineering principles and designer-friendliness, to achieve modularity and extensibility, which is essential for handling a myriad of abilities and their interactions. To this end, we leveraged Unreal Engine 5 and the Gameplay Ability System (GAS) (Chapter 6), modifying and extending the system to meet our specific requirements, and designing our structure to simplify the definition of game mechanics, as outlined in chapter 7.

As mentioned in the Discussions chapter, D&D offers a vast array of abilities and features. While our current implementation provides a strong foundation, there are still some functionalities that need to be integrated to realize the full depth of D&D combat experience. These include additional ability types, damage mechanics, and condition triggers, among others.

One potential area for improvement is the management of turns and timing. As mentioned earlier, Unreal Engine's native functionality does not support turn-based mechanics out of the box. To address this, delving into Unreal Engine's source code and declaring a new timer that understands the concept of turn could be worthwhile. However, such an approach would come with its own set of challenges, such as complicating the process of maintenance and updating the engine. Any modifications to the core engine could complicate future updates and integration with newer Unreal versions, so such a decision should be carefully considered based on the specific needs of the project and its timeline.

Alternatively, for future work, another viable direction could be to move away from GAS altogether and build a fully custom, modular system specifically designed to meet the unique needs of D&D. A system built from the ground up would provide the flexibility to tailor every aspect of combat mechanics, from turn management to ability interactions. This could offer a level of control over the system that a more generalized framework like GAS can not. However, this direction could require more development resources.

References

[1] Ville Mäkelä and Albrecht Schmidt. 2020. I Don't Care as Long as It's Good: Player Preferences for Real-Time and Turn-Based Combat Systems in Computer RPGs. In Proceedings of the Annual Symposium on Computer-Human Interaction in Play (CHI PLAY '20). Association for Computing Machinery, New York, NY, USA, 231–240. https://doi.org/10.1145/3410404.3414248 (accessed on 18.12.2024)

[2] M. S. Noorudin, S. N. K. Mohd Jamil, M. H. Alkawaz and M. G. Md Johar, "The Role Playing Games from Classic to Modern: Future Perspective," 2021 IEEE Symposium on Industrial Electronics & Applications (ISIEA), Langkawi Island, Malaysia, 2021, pp. 1-5. https://doi.org/10.1109/ISIEA51897.2021.9509982 (accessed on 18.12.2024)

[3] Barton, M., & Stacks, S. (2008). Dungeons and Desktops: The History of Computer Role-Playing Games 2e (2nd ed.). A K Peters/CRC Press. https://doi.org/10.1201/9781351273404 (accessed on 18.12.2024)

[4] Stenström, C.D., & Björk, S. (2013). Understanding Computer Role-Playing Games - A Genre Analysis Based on Gameplay Features in Combat Systems. International Conference on Foundations of Digital Games. https://api.semanticscholar.org/CorpusID:14768535 (accessed on 18.12.2024)

[5] M. Horsfall and A. Oikonomou, "A study of how different game play aspects can affect the popularity of role-playing video games," 2011 16th International Conference on Computer Games (CGAMES), Louisville, KY, USA, 2011, pp. 63-69. https://doi.org/10.1109/CGAMES.2011.6000361 (accessed on 18.12.2024)

[6] Nicole Carpenter. *Baldur's Gate 3 Wins Games of the Year Award*. Polygon, 2023. Available at: https://www.polygon.com/24218806/baldurs-gate-3-wins-hugo-award. (accessed on 18.12.2024)

[7] Baldur's Gate 3 Wiki. Available at: https://bg3.wiki/. (accessed on 18.12.2024)

[8] Loponen, M., & Särkijärvi, J. (2015). Review: Playing at the World. International Journal of Role-Playing, (5), 66–67. https://doi.org/10.33063/ijrp.vi5.237 (accessed on 18.12.2024)

[9] Ewalt, David M. (2013). Of Dice and Men: The Story of Dungeons & Dragons and the People Who Play It. Simon & Schuster.

[10] Evan Torner, Jon Peterson, 2024. "Combat in Dungeons & Dragons: A Short History of Design Trajectories", Fifty Years of Dungeons & Dragons, Premeet Sidhu, Marcus Carter, José P. Zagal. https://doi.org/10.7551/mitpress/15175.001.0001 (accessed on 18.12.2024)

[11] Prayaga, Lakshmi. (2010). Mapping software engineering principles to stages in game development. Journal of Computing Sciences in Colleges. 26. 208-214. (accessed on 18.12.2024)

[12] Jacob Brenkus, Alex Fausnaugh, Kayla Welton, Cleveland State University Advisor Prof. David G. Aloi, Cleveland State University. The Advantages of Modular Design in Software Engineering. Cleveland State University. (accessed on 18.12.2024)

[13] Serrano Zanetti, M. (2013). A complex systems approach to software engineering (Doctoral thesis, ETH Zurich). ETH Zurich Research Collection. https://doi.org/10.3929/ethz-a-010007378 (accessed on 18.12.2024)

[14] Boekhoud, P. S. (2015). The impacts of a modular design strategy on modern video game development: An explorative study (Master's thesis, University of Twente). https://essay.utwente.nl/67724/ (accessed on 18.12.2024)

[15] R. C. Martin, "The dependency inversion principle", Jun. 1996. [Online]. Available: https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf (accessed on 18.12.2024)

[16] Liukku, Oskari. Clean Code: Analysing game code using agile programming practices. (Master's thesis, Aalto University, Espoo, Finland). Available: https://urn.fi/URN:NBN:fi:aalto-202006213882 (accessed on 18.12.2024)

[17] R. C. Martin, Agile software development: principles, patterns, and practices, ser. Alan Apt series. Upper Saddle River, N.J: Prentice Hall, 2003, 529 pp., isbn: 978-0-13-597444-5. (accessed on 18.12.2024)

[18] Omar Alam and Jörg Kienzle. 2012. Designing with inheritance and composition. In Proceedings of the 3rd international workshop on Variability & Composition (VariComp '12). Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/2161996.2162002 (accessed on 18.12.2024)

[19] Lê, Patrick Masse, David Paris, Thomas. (2013). Technological Change at the Heart of the Creative Process: Insights From the Videogame Industry. International Journal of Arts Management. 15. 45-59.

[20] Engström, Henrik Brusk, Jenny Erlandsson, Patrik. (2018). Prototyping Tools for Game Writers. The Computer Games Journal. 7. 10.1007/s40869-018-0062-y.

[21] Epic Games. Unreal Engine: Documentation, Gameplay Ability System,[Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/gameplay-ability-system-for-unreal-engine?application_version=5.5 (accessed on 18.12.2024)

[22] Tranek. GAS Documentation, [Online]. Available: https://github.com/tranek/GASDocumentation (accessed on 18.12.2024)

[23] Butler, S., & Oliver, T. (2024). Game Development Patterns with Unreal Engine 5: Build maintainable and scalable systems with C++ and Blueprint. Packt Publishing Ltd.

[24] Necula, Sabina. (2024). Exploring The Model-View-Controller (MVC) Architecture: A Broad Analysis of Market and Technological Applications. https://doi.org/10.20944/preprints202404.1860.v1.

[25] T. Ollsson, D. Toll, A. Wingkvist and M. Ericsson, "Evolution and Evaluation of the Model-View-Controller Architecture in Games," 2015 IEEE/ACM 4th International Workshop on Games and Software Engineering, Florence, Italy, 2015, pp. 8-14. https://doi.org/10.1109/GAS.2015.10 (accessed on 18.12.2024)