Futuregames Skellefteå

Using Unreal Engine's Gameplay Ability System for a Roguelike Auto-Shooter

Péter Gulyás

May 12, 2025

Abstract

This project report describes the development of a single-player rougelike auto-shooter game using Unreal Engine's Gameplay Ability System (GAS). The objective was to leverage GAS's data-driven framework to manage character attributes, abilities, and effects in a modular, scalable manner. The demo features a player character fighting continuous waves of enemies, leveling up and choosing random upgrades to evolve gameplay. I am going to detail how GAS was used to implement core mechanics (attributes, abilities, and effects), and how the project plan (goals, timeline, and challenges) guided the development. I will also discuss new skills gained (such as Unreal Engine C++ and GAS expertise), technical implementation details, and future work. This work demonstrates industry-relevant skills in modern game development and GAS architecture.

Contents

1 Introduction **3**

2 Project Goals and Timeline **4**

3 Completed Game Project Overview **5**

4 Technical Implementation **6**
4.1 Core GAS Components 6
4.2 Game Architecture and Additional Systems 7

5 Learnings and Future Directions **8**
5.1 New Skills and Knowledge 8
5.2 Future Improvements 9

6 Conclusion and Career Impact **10**

1 Introduction

The Unreal Engine **Gameplay Ability System (GAS)** is a highly flexible framework for implementing character abilities, attributes, and effects [1]. Originally developed for Epic's MOBA *Paragon* and used in AAA titles like *Fortnite*, GAS provides a data-driven system to define gameplay actions, status effects, and attribute modifications [2]. Because it standardizes complex ability logic and supports built-in multiplayer prediction, many developers now consider ability systems (like GAS) an "industry standard" for character-centric games [5] [4]. GAS is used in diverse genres (RPGs, action games, MOBAs) because it encapsulates everything from simple actions (jumping) to complex spell systems, handling cooldowns, costs, and networking seamlessly [2].

My project set out to build a roguelike auto-shooter demo that fully employs GAS. The goals were to create a playable demo with core GAS-based mechanics within the nine-week timeline, focusing on learning GAS architecture and building a polished gameplay loop. Key milestones included implementing character abilities, enemy AI waves, leveling up, upgrades to enhance character power, and simple UI. The plan anticipated challenges (GAS's steep learning curve, performance, balancing) and scheduled tasks accordingly. I have successfully reached most planned goals on time, though some scope adjustments were made (for example, I initially planned to add a boss, as well as some visual effects). The final demo demonstrates that GAS can easily handle the complex systems that are present in a roguelike, and it proved to be worthwhile to implement even without enabling networking.

2 Project Goals and Timeline

The original project objectives were: (1) to become proficient with GAS by using it to build game systems, and (2) to deliver a cohesive roguelike demo. The nine-week plan outlined weekly milestones such as initial setup (week 1), core GAS framework initialization (week 2), enemy AI and spawning (week 3), combat mechanics (week 4), visual polish (week 5), finalizing weapons and power-ups (week 6), and bug fixing/documentation (weeks 7–9). The roguelike elements – randomized upgrades and a leveling system – were chosen because GAS excels at data-driven, upgradeable systems.

In practice, the development largely followed this plan. By mid-project, core GAS components were in place (Attribute Sets, Ability classes, Effects, and Tags), and enemys were functional. The scheduling was a bit unrealistic: initial setup and GAS integration fit the first 3-4 weeks, and weapon/ability prototyping extended into week 5. In the rest of week 5 the focus on polish resulted in adding some visual effects, a player character and UI layouts. The tentative boss implementation in week 7 was replaced by further iteration on the core gameplay loop. In summary, most set goals were reached by week 9 (core gameplay loop, upgrade system, UI), though a few optional features were simplified to meet the deadline.

Challenges Encountered: The biggest hurdle was the GAS learning curve. GAS requires substantial setup code (especially in C++ for custom tasks) which initially slowed progress. Balancing abilities and stats proved tricky: designing abilities that use attributes like health and armor without conflicts demanded careful configurations. Keeping the scope under control was also challenging; some advanced ideas (e.g. network replication) were deferred. Nonetheless, focusing on reusable, data-driven design (as GAS encourages) mitigated many issues. By leveraging GAS's modularity, adding a new damage type or upgrading system only required creating new Ability or Effect assets, without rewriting core logic – illustrating one of GAS's strengths

3 Completed Game Project Overview

The final game is a single-player roguelike auto-shooter. The player is a dwarf mage wielding magical weapons, battling infinite waves of enemies in a confined arena. Movement and positioning are central to survival, as enemies spawn around the player. The roguelike element is implemented via a leveling and upgrade system: at the start of each run the player chooses one of three random special abilities (e.g. a spinning flame attack, a homing missile, etc.). As enemies are defeated and experience is gained, the player levels up and on each level-up a menu appears with randomized upgrades (e.g. +10% stamina, +5% critical strike chance, or a new passive effect). Additionally, every 10 levels the player gains a new weapon/skill slot, expanding their arsenal and enabling powerful ability combinations. This creates an evolving playstyle each run. These features were implemented using GAS: the upgrade buffs are applied via Gameplay Effects to the player's Attribute Set (modifying health, damage, or other stats), and each ability is a Gameplay Ability that can spawn projectiles, cause status effects, or trigger area attacks.

This genre fit GAS well – its data-driven design makes it easy to build complex, upgradeable systems like randomized perks and elemental damage effects, matching a roguelike's evolving nature. For instance, adding a new power-up simply involved creating a new Gameplay Effect class with the desired modifier. Similarly, an additional enemy type can be implemented by defining a new Ability Set and attributes for that actor, without changing existing code.

The game's UI includes a HUD showing health and mana (attributes managed by GAS) and an upgrade selection screen (triggered by the Ability System when leveling). These interfaces were created using Unreal's UMG with C++-driven updates from the GAS state. For example, the upgrade selection widget listens for a GAS event on level up, then displays buttons corresponding to preset Gameplay Effects; clicking a button applies that effect to the player via the GAS component.

4 Technical Implementation

4.1 Core GAS Components

My implementation closely followed GAS's component structure. Each character (player and enemies) has an Ability System Component (ASC) attached, which owns that actor's abilities, attribute data, and active effects. I defined a custom UAttributeSet subclass to hold stats such as Health, Armor, and Elemental Damage Resistances, etc. These attributes are mutable and replicated by GAS if needed. Abilities (in C++) inherit from UGameplayAbility. For example, the basic AI attack ability applies damage gameplay effect when in range of target.

Attributes and Attribute Sets: Attributes are represented using floating-point values encapsulated in `FGameplayAttributeData` struct. GAS uses `UAttributeSet` subclasses to group related attributes. Each attribute contains a base value (for scaling and reset) and a current value (which changes during gameplay). For instance, buffs can increase Armor temporarily by modifying the current value, while the base stays unchanged.

I used Meta Attributes like `IncomingDamage` to support layered logic. When a damaging effect is applied, it first populates `IncomingDamage`. Then, in the overridden `PostGameplayEffectExecute()` function, I determine if the hit was critical or missed, and only then apply the result to `Health`. This approach allows me to display the damage on the UI differently whenever the damage was a crit, a miss, or just a regular attack.

Gameplay Abilities: Each skill or weapon in my game is implemented as a `UGameplayAbility`. These classes encapsulate all gameplay logic tied to specific actions—spawning actors, playing animations, applying effects, or triggering sounds. Each ability is modular and reusable, based on a shared base class I created to handle ability-level logic. This includes setting gameplay tags, linking gameplay effects, and handling cooldown application.

One system I implemented using tag events is auto-reloading. Each ability adds a tag like `Cooldown.Fireball` on use. When this tag is removed, the ASC's delegate system triggers a callback. I use this to instantly retrigger the ability, simulating a seamless reload loop.

Gameplay Effects: Gameplay Effects define how stats are modified or tags are granted. In my game, I use instant effects for most attacks and stat upgrades, and duration-based effects for cooldowns. Upgrades are all implemented as effects applied during the level-up event, which lets me easily modify them via the editor.

For more complex logic I use **Execution Calculations**. These are custom classes derived from `UGameplayEffectExecutionCalculation`. An example would be dealing damage that factors in resistances, dodge chance, and crits.

The beauty of this approach is that it's completely modular and reusable. Any ability that deals damage can use this execution class without duplicating logic. And if I ever want a new damage type with its own rules, I can just create another execution class, no need to touch the abilities or gameplay effects themselves. This decoupling and flexibility is what makes GAS such a robust and scalable framework for gameplay mechanics.

Gameplay Tags: Gameplay Tags are hierarchical labels that define state, categories, or conditions. Tags let me filter, block, or activate gameplay logic declaratively.

I also use the ASC's tag event system to respond to tag changes. When a cooldown tag is removed, I check if it matches any queued ability and retrigger it if appropriate. This enables a clean and scalable way to manage cooldown timers and reactivation logic.

4.2 Game Architecture and Additional Systems

The rest of the game uses standard Unreal components integrated with GAS. The GameMode controls wave spawning, but all combat logic flows through GAS. Enemy AI controllers use simple behavior (move-to and attack) and trigger GAS abilities when in range.

The Upgrade System was built on top of GAS: on level-up, the game selects three random upgrade effects (GameplayEffects), displays them, and upon player choice applies the chosen effect to the player's ASC. Since effects can modify any attribute, I could implement varied upgrades purely through data. This highlights GAS's scalability: to introduce a new upgrade, the only work needed was designing a new Effect asset, the game loop code stayed unchanged.

To manage game states such as the main menu, gameplay, and upgrade screens, I implemented the Event Handler pattern - that we learned about during our specailization course - using a GameInstanceSubsystem. This subsystem persists across levels and exposes a lazy getter that spawns the main EventHandler actor on first call. I also created an 'IEvent' interface which each event (like menus or gameplay states) implements, allowing me to queue and transition between events generically.

A key example is the upgrade menu, which opens upon each level-up. If the player gains multiple levels at once, the system queues multiple upgrade events. This ensures that each upgrade menu is displayed in sequence without overlapping or skipping choices.

For UI integration, I used a Model-View-Controller (MVC) pattern. Each major interface, such as the in-game HUD and upgrade screen, has a controller that mediates communication between the view (widgets) and the model (the player character). This approach enforces separation of concerns and aligns with principles like Open/Closed and Single Responsibility. For example, the upgrade controller listens for UI input and applies stat changes by calling into the character's attribute logic—without embedding any gameplay code in the UI itself.

5 Learnings and Future Directions

5.1 New Skills and Knowledge

This project significantly deepened my technical skills in Unreal Engine and GAS. Key learnings include:

- **GAS Framework Concepts:** I learned the core GAS architecture - how the Ability System Component ties together Abilities, Attributes, and Effects. Implementing a damage execution class taught me how to use `GameplayEffectExecutionCalculation` for complex custom logic. I also explored asynchronous Ability Tasks.
- **Data-Driven Design:** Using GAS reinforced data-driven design pattern. For example, creating new gameplay content (enemy types, upgrades, damage types, etc.) involves mostly configuring assets, rather than hardcoding. This highlights the "modular and reusable" nature of GAS.

- **UI and UX Implementation:** I built simple UI screens and integrated them with the GAS flow (for example, listening to ASC level-up events). I also learned about general UI design principles.

I have gained proficiency in writing Unreal C++ code in general, and overall, the project advanced my engineering skills (coding and software architecture). This aligns with career goals: expertise in GAS and Unreal Engine is valuable in the industry, especially for any title requiring complex ability systems.

5.2 Future Improvements

Given more time, several enhancements could be implemented:

- **Multiplayer / Networking:** GAS was built with networking in mind (client prediction, replication). Future work would enable co-op play: enabling ASC replication, handling possession and network prediction for abilities. This would require testing gameplay under network latency.
- **Expanded Abilities and Skill Trees:** I could add more abilities and multi-tiered ability levels. For example, an ability might have multiple levels (Level 1: one missile, Level 2: two missiles, Level 3: homing, etc.) controlled by ability level variables. This would showcase GAS's ability-level scaling.
- **Gameplay Cues and Visuals:** Implementing Gameplay Cues would improve visual feedback. For each Ability and Effect, I would assign particle or sound cues triggered by tags/events, decoupling code from VFX. This would create a more polished presentation.
- **More Complex AI:** Enhance enemy behaviors (different movement patterns, special enemy abilities) using GAS for enemy skills. Additional enemy classes could be introduced simply by assigning them new attributes and ability sets.
- **UI Improvements:** The upgrade system UI could be expanded (more info, animations) and made more accessible.

6 Conclusion and Career Impact

This project successfully met its goals: a functional roguelike auto-shooter demo was built around Unreal's GAS. I followed the planned milestones and overcame challenges inherent in learning and applying a complex engine subsystem. Key outcomes include a clear understanding of GAS's core components (Ability System Component, Attributes, Abilities, Effects, Tags) and practical experience using them to craft game systems. The completed game clearly showcases GAS in action: from the leveling-based upgrade loop to the enemy AI using GAS abilities, all systems are unified under GAS architecture.

From a career perspective, this project demonstrates familiarity with industry-standard tools. Mastery of GAS is particularly marketable, as many studios adopting Unreal Engine rely on it for complex ability and combat systems. The depth of technical implementation (custom C++ ability classes, data-driven design) and the polished gameplay loop will be valuable talking points in a portfolio. In sum, the skills and knowledge gained (Unreal C++, GAS, systems design) directly support future opportunities in game development.

References

[1] Epic Games, “Gameplay Ability System Documentation”. https://dev.epicgames.com/documentation/en-us/unreal-engine/gameplay-ability-system-for-unreal-engine?application_version=5.4

[2] Tranek, "GASDocumentation", Github Repository. https://github.com/tranek/GASDocumentation

[3] Landelare, "Simple GAS Tutorial" https://landelare.github.io/2024/01/15/simple-gas-tutorial.html

[4] U. Kustra, "Why you should use the Gameplay Ability System in basically every Unreal Engine game" https://ukustra.medium.com/why-you-should-use-the-gameplay-ability-system-in-basically-every-unreal-engi

[5] R. Kavert, "Fueling your Mechanics with Gameplay Ability Systems" https://www.gamebreaking.com/posts/ability-systems

[6] S. Ulibarri "Unreal Engine 5 - Gameplay Ability System - Top Down RPG" https://www.udemy.com/course/unreal-engine-5-gas-top-down-rpg/