Unreal Engine's Gameplay Ability System, from programming framework to designer's tool

Guillaume David
gjd.david@gmail.com
CNAM-ENJMIN
Angoulême, France

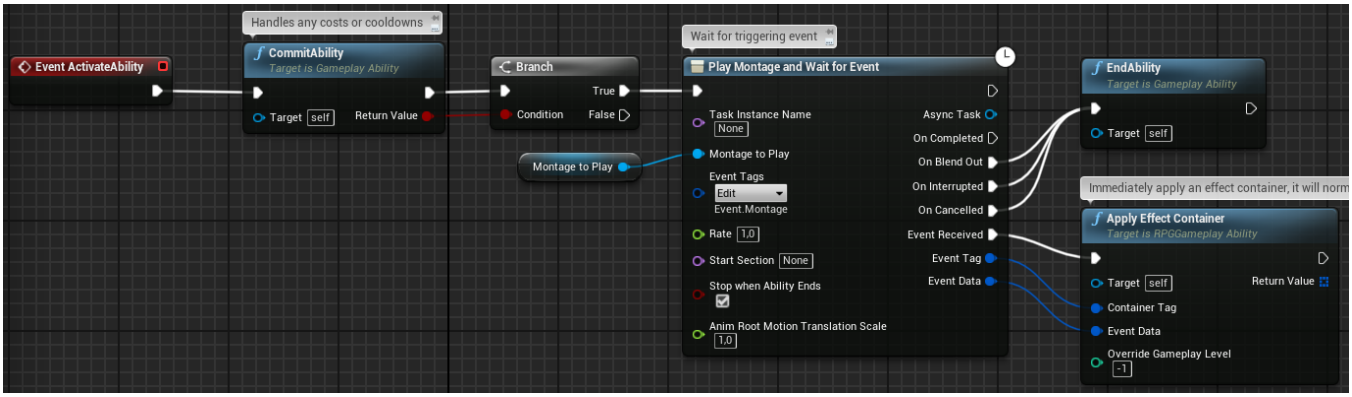

Figure 1: Melee attack Gameplay Ability in the Action RPG sample project

ABSTRACT

In this article I investigate how Unreal Engine's Gameplay Ability System plugin works and what steps should be taken in order to make it usable by Game Designers in a team project. I will go from a general overview of the System's architecture to thoughts about how specific game mechanics could be implemented with it.

CCS CONCEPTS

• **Software and its engineering → Interactive games**.

KEYWORDS

Unreal Engine

ACM Reference Format:
Guillaume David . 2021. Unreal Engine's Gameplay Ability System, from programming framework to designer's tool. In ,. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

While readily-available game engines like Unity and Unreal Engine come with extensive tool sets that cover the most common needs of game developers, they can't provide for the very specific needs of such a diverse industry and even small-scale projects like students games can benefit from building custom tools. Games like MMORPGs and MOBAs typically have a large number of character abilities like skills and spells so building a custom editor and underlying framework makes sense and allows game designers to iterate quickly and be more independent from the programmers while designing such abilities [1]. Unreal Engine 4 (UE4) aims to provide such a framework with the Gameplay Ability System (GAS), although it is hardly usable as is and needs to be tailored to each project's specific usage. This article will give a tour of the GAS's architecture and investigate the steps needed in order to turn it into a tool for game designers.

1.1 For Whom: Relevant Team Roles

Aside from hobby projects and a select few professional solo developers, making video games is a team effort and makes use of very different skills. Of these different fields of expertise, the relationship between *game designers* and *programmers* is of particular interest to this article. Although the specifics differ from studio to studio, the following roles usually exist in some form and perform at the interface between the fields of programming and game design:

- *Gameplay Programmers* translate the game designers' rules and mechanics into code
- *Tool Programmers* provide tools for other team members' use
- *Technical Game Designers* are the interface on the design side and are familiar with the technical implications of each game design decision

The subject of this article sits at the intersection of these three disciplines and should be of interest to all of these roles.

1.2 For What: Game Genres and Use Cases

I originally began my reflection about what I would call *evolving systemic games*, games like *League of Legends* or *Magic the Gathering* which have an *ever increasing* (from new content being released regularly) number of abilities, effects and conditions that can interact with one another. It obviously makes perfect sense to have a dedicated tool for designing abilities when developing such a game,

, 2021. https://doi.org/10.1145/nnnnnnn.nnnnnnn

it would also make sense to use it in games with a large, albeit fixed, number of abilities such as role playing or strategy games.

Then I read about UE4's GAS and found out that *any game* could use it extensively. Basically anything that can happen in a game can be implemented using GAS, although maybe not everything should. It may make sense to make a character's jump be a Gameplay Ability if we want to be able to give it a different jumping mechanics like a double-jump, likewise a sprint action that depletes a stamina attribute makes a good candidate for a Gameplay Ability. Item use, power-ups, different weapons are almost as obvious use cases as RPG spell casting. The system is designed for *gameplay* though so things like UI interaction may be better implemented in other ways.

2 GAS BASICS

The Gameplay Ability System was originally developed for Epic Game's *Paragon* and is in use in their highly successful game *Fortnite*, both of which are online multiplayer games. The GAS is indeed designed to handle network replication of Ability use and Effects, this has implications in the system's architecture and in the work needed to make use of it in such a context. The GAS can of course be used in an offline or single-player environment and, as my interest lies in the designer-facing part of the system, I will not concern myself with its networking capabilities and simply trust that they will not hinder any attempt to build a designer-friendly layer on top of it.

Official documentation[4] of the GAS is fairly recent and is predated by several unofficial sources in the form of tutorials, wiki articles and Dan Kestranek's documentation effort on GitHub [6]. These different unofficial sources[6][7] contain different information pertaining to their author's specific interests while the official documentation is little more than an introduction. All of these sources also seem to make assumptions about what kind of game is being made and I found all of them unsatisfying to some degree. On the other hand, gaining a deeper understanding of the GAS by reading the source code is a daunting task for someone who is not familiar with Unreal's inner workings. In my experiments I feel like I barely scratched the surface.

Producing a comprehensive documentation of the Gameplay Ability System is well out of the scope of this work so I will only summarize the various concepts and classes involved in this system. A rather restrictive summary of the relationships between these elements goes as follow:

Gameplay Abilities run *Ability Tasks* and apply *Gameplay Effects* which in turn can modify *Attributes* and *Gameplay Tags*.

2.1 Gameplay Tags

While not strictly speaking part of the GAS, Gameplay Tags are tightly integrated in it. According to Unreal Engine's documentation

> Gameplay Tags are conceptual, hierarchical labels with user-defined names.[5]

The GAS uses Gameplay Tags as conditions: Effects or Abilities can be blocked based on their tags, or have tag requirements. The hierarchical capability of Gameplay Tags can be very useful for subcategories, so that a damage effect with the tag `damagetype.magic.fire` would be recognized as an effect with `damagetype.magic`.

Figure 2: Unreal's Gameplay Tag editor

2.2 Ability System Component

This is the component that allows Actors to use the Ability System. This component should be present on any actor that either:

- Has Attributes
- Should have or use Gameplay Abilities
- Can be subject to Gameplay Effects

The Ability System Component also handles network replication of these elements and features several callbacks for responding to their changes.

2.3 Attributes

Attributes are floating-point numerical values used to represent anything about an Actor, most commonly things like a character's health or movement speed. Attributes have a concept of a *base value* versus a *current value* to represent temporary changes to them (buffs and debuffs) and come in *Attribute Sets*. An Ability System Component can have multiple different Attribute Sets and Attribute Sets can be added at runtime. Also the Ability System Component invokes delegates on Attribute change so we can respond accordingly.

2.4 Gameplay Abilities

Gameplay Abilities are what game designers want to make, these are the actions that characters (or other Actors) can *do* in the game. Gameplay Abilities are usually implemented in Blueprint by stringing together Ability Tasks. Gameplay Abilities can be granted to (or removed from) an Ability System Component and can then be *activated* either by player input or by other events, they can also be *cancelled* before completion.

2.5 Ability Tasks

Ability Tasks are used for any asynchronous step of an Ability, like playing an animation (and waiting for a specific point in it), moving an actor or generally waiting for a condition to be met. In Blueprint, Ability Tasks appear as *latent nodes* that can resume execution at a later point in time under specific conditions.

Figure 3: Example of an Ability Task

2.6 Gameplay Effects

Gameplay Effects represent changes to be made to an Ability System Component : modifying Attribute, granting or removing Gameplay Tags, granting or removing Gameplay Abilities. Gameplay Effects can be either Instant, Permanent or Duration-based, in the latter case they either change the Ability System Component temporarily or at regular interval like in the case of a damage-over-time effect.

2.7 Gameplay Events

Gameplay Events are more or less what the name suggests: they are data structures that represent activity within the GAS, or whatever we can think of. They can be used to trigger the activation of Gameplay Abilities, or be waited for inside a running Ability with a *Wait Gameplay Event* Ability Task, and hold data regarding their *Instigator* and *Target*.

Of these concepts, I have found fewer mentions of Gameplay Events in online sources. I don't know is this is because Gameplay Events are a more recent addition to the GAS or because they were not of interest to the various authors.

3 MORE ABOUT GAMEPLAY EFFECTS

Gameplay Effects are designed to be used as data-only Blueprints, as such a Gameplay Effect subclass should not hold any behaviour. The data they hold is rather involved, containing multiple arrays of complex types, and can contain references to other classes that provide custom calculations.

3.1 Effect Modifiers

Modifiers are the main mechanism used by Game Effects, they describe modifications to be made to a target's Attributes. Modifiers have a lot of options regarding how they are applied such as how their Magnitude is computed and how this Magnitude affects the target Attribute: one modifier can be added to the Attribute while another can be multiplied. The default calculation comes from Paragon and makes assumptions about how Modifiers should be combined, for example multiplicative Modifiers are added together before being multiplied whereas a different game would just multiply them together.

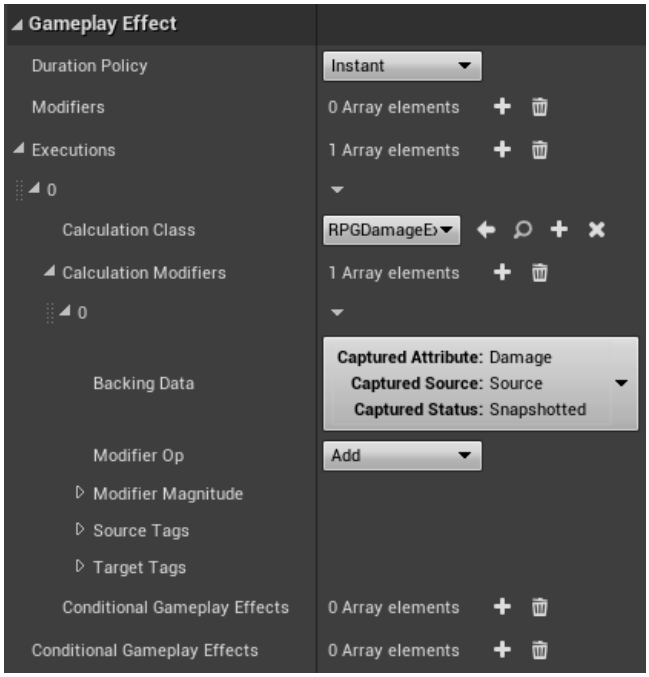

Figure 4: A Gameplay Effect using an Execution Calculation

3.2 Magnitude Modifier Calculations and Execution Calculations

These are two options to run custom code when applying and effect. Magnitude Modifier Calculations are assigned to a modifier and essentially encapsulate a function that returns a new Magnitude for the modifier, while Execution Calculations are a property of the effect itself and essentially generate new Modifiers on the fly.

3.3 Gameplay Effect Specs

Gameplay Effects themselves are not directly applied to their targets, instead a Gameplay Effect Spec instance is created first. The Gameplay Effect Spec can be modified before being applied, and can also be passed around, for example by giving it to a projectile Actor as a variable. Effect Modifier definition, including fixed Magnitudes, Magnitude Modifier Calculations and Executions Calculations cannot be switched in a Gameplay Effect Specs, but we can make use of dynamic "SetByCaller" values, Gameplay Tags, and multiple Specs can also be linked together.

4 DESIGN CONSIDERATIONS

The GAS's architecture can be a bit confusing and feel overcomplicated at times, in part because of its nature as a network-ready framework, in part for its goal of being generic enough to be usable

for all kinds of games. As such, there are often multiple ways of using the GAS to implement each part of a game's design therefore each project should establish how they will use it. While the decisions rest on each team's members, I will investigate a few of the choices that have to be made.

4.1 What Systems Use GAS?

Gameplay Abilities are just a layer on top of regular Blueprints so there is no reason anything that is doable with Blueprint could not be done as an Ability. What *should* be done that way, though? Special powers of a game's characters are the obvious use case, basic attacks, item use, environment interactions, power-ups, these are also commonly implemented as Gameplay Abilities. Is basic movement an Ability? It could be handy if we should have multiple movement modes. Is clicking a unit to select it an Ability? What about some AI decision-making code? The game's UI should respond to Attribute change (e.g. a Health bar), does it trigger with Gameplay Events?

4.2 Who Owns an Ability System Component?

Let's consider the very common case of a player's character switching weapons, attacking with the weapon would be a Gameplay Ability, as would other uses of it like reloading or parrying. Does each weapon Actor come with its own Ability System Component or does it only grant Abilities to the Character? The Ability System Component holds Attribute Set so what about the weapon's state, things like durability or ammo, are they Attributes of the weapon itself, Attributes granted to the Character, or just plain variables on the weapon Actor?

What about non-player Characters? Do they all have their own Ability System Component? What about inanimate Actors like a destructible wall with a "health" attribute? What if we have *lots* of candidate Actors, maybe performance will become an issue.

4.3 Who Knows about What?

The question boils down to what in the game is a *generic* mechanics versus what is a *special* case. The answers to this question would be more implementation *guidelines* than strict rules.

Let's say we have characters with a Health Attribute, and a Damage Gameplay Effect that should lower this Attribute. Does the Effect directly change the Attribute? Then can something without Health still be damaged? Or does the Attribute Set take care of applying damage to itself? If so then different Actors can have different ways of tracking damage done to them. Then what about Armor or other forms of damage mitigation? If everyone has Armor then it could be an Attribute and the damage calculation (as decided above) would account for it, in the case of more exotic form of damage mitigation, though, Armor could be a Gameplay Ability that triggers upon receiving a Damage Effect and edits the Game Effect Spec.

I have seen advice on implementing a *stun* effect by making a `stunned` Gameplay Tag and setting the Gameplay Ability base class to be blocked if the owner has this tag.[3] In this case *all* the Abilities know that being *stunned* is a possibility, the stun effect is a generic mechanics of this game. Should every possible status effect have its own tag and every other relevant part of the game check for it?

4.4 Triggering Abilities

Gameplay Abilities can be Activated by several means. The most direct way is to bind input directly to the Ability so that the player can directly activate it, while this method is simple and may be suitable for many games, it may lack flexibility for others. The second way is to call one of the Activate methods, in that way we can activate and Ability by code whenever we need, and we can also Activate all Abilities that share a specific tag, allowing us to group Abilities together.

Gameplay Abilities can also be triggered by the reception of a Gameplay Event, this way the Event's Instigator doesn't need to know about the Ability for it to be activated. Another way of tying Ability execution to Gameplay Events is by using an Ability Task like the built-in Wait Gameplay Event, in this way the Ability stays activated but is put on hold until the event is received.

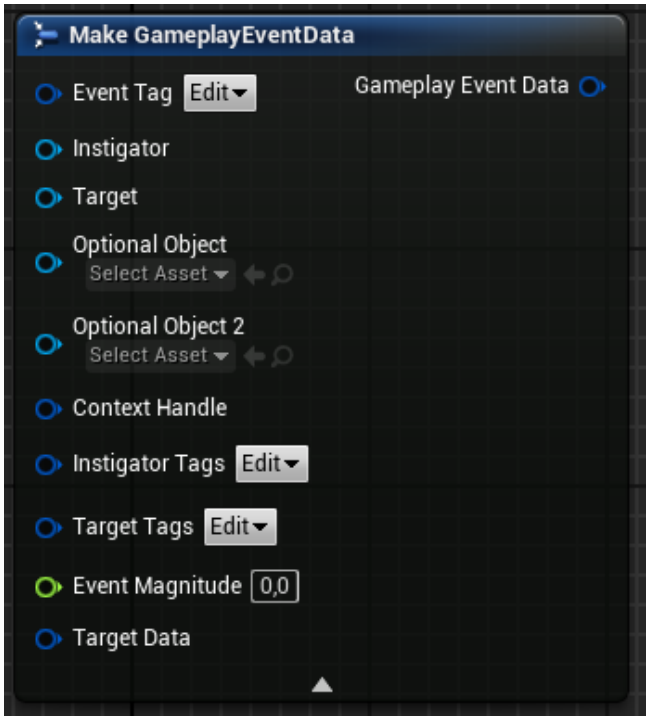

Figure 5: The data structure ferried by Gameplay Events

4.5 Sending Gameplay Events

What use does our game make of Gameplay Events? If we are to use it extensively, if triggered Gameplay Abilities are commonplace for example, we then have to decide what events are fired and when.

4.5.1 Event Routing and Non-local Triggers. Gameplay Events are sent to specific Actors, but there may be cases when we want to trigger an Ability when something happens *anywhere*. *Magic the Gathering* has a lot of cards with "when X happens" triggered

abilities, and if we already make extensive use of Gameplay Events it would make sense to use them for things like game statistics (e.g. how much damage has been dealt in one session), there are several ways of making sure events get to where they are needed that coule be implemented by overriding the SendGameplayEvent method in our game's GameplayAbility class.

5 WHAT TO EXPOSE

The object of this work is to investigate how the Gameplay Ability System can be used by a team with varying degrees of technical familiarity with its workings. Some parts of the GAS can only be accessed through C++, so naturally a programmer would set it up for the rest of the team. The overall complexity of the system also highlight the need to adapt it to our project, so what tools should we give to the (technical-) game designers, which building blocks should we provide ready-made and which should be left for them to design?

5.1 Making new Gameplay Abilities

When a game designer wants an element of the game to be able to *do something*, this could be implemented as a Gameplay Ability so making new Abilities would probably be a common occurrence. As they are essentially bits of Blueprint code, making a new Gameplay Ability is actually programming, so different designers will feel differently about it, but it can also be made to feel less like coding. One way is to provide custom nodes tailored to our games: Blueprint Macros, Functions, or Ability Tasks. Another way is to make reusable GameplayAbility subclasses. A given project will usually have one such subclass that serve as the basis for all of this game's abilities, then additional classes deriving from that one, implementing the most common forms of specific abilities in our game, to be derived as data-only Blueprints so that the code is completely reused.

5.2 Making new Gameplay Effects and Attribute Sets

Making a new Gameplay Effect or Attribute Set would probably not be needed unless we are implementing a whole new *generic* mechanics. It looks fairly easy to make similar effects *just different enough* to make debugging difficult so if a game is going to need lots of them I suggest that Gameplay Effects be *semantically distinguishable*, for example if a game has a *Healing* mechanics that is different from *Damage* then we don't implement Healing using a negative Damage effect. Similarly if such a game has mechanically different Damage sources then two separate Damage Gameplay Effects would be understandable. However, making a "10 damage" effect and a "damaged based on Character Strength" effect might be confusing and increase the risk of errors and bugs. Effects that are similar can be differentiated by giving different Gameplay Tags and SetByCaller values to their GameplayEffectSpec.

5.3 Handling Effect Specs

If we are to allow Gameplay Effect Specs to be applied (rather than Gameplay Effects themselves) after they are properly set up by the Gameplay Ability or other delivery mechanism, we ought to make helper functions or Blueprint Macros to make sense of it. As we would have a limited number of Gameplay Effects, each could come with a function to make a Gameplay Effect Spec from it with the right parameters. For instance, creating a Damage Effect Spec could ask for damage amount and a Damagetype Gameplay Tag.

5.4 Delivering Effect Specs

Gameplay Effect Specs (as well as Gameplay Effect themselves) can be applied to any Ability System Component from any script, not just from Gameplay Abilities. In fact, it makes perfect sense for a Health Pickup in a FPS game to not have an Ability System Component and just apply a Healing effect to whatever picks it up. Likewise, a projectile created by a Gameplay Ability would apply its Damage effect on hitting a suitable target. The GAS also allows applying Effects to any entity or group of entities represented by a TargetData structure.

We should establish guidelines regarding Gameplay Effect delivery from Gameplay Abilities, especially for those mediated by a spawned Actor like our projectile above. Maybe the Actor delivers the effect itself in a script, or maybe it passes its target back to the Ability, either via TargetData or via a direct reference. Barring more complex multi-effect Abilities, there should be few (ideally one) well identified, places to look for Effect delivery logic.

5.5 Spawning Actors Through Gameplay Abilities

Gameplay Abilities can use Spawn Actor Ability Task, however the Ability can't track what happens to that actor afterwards. If the Actor is a delivery system for a Game Effect Spec then the Gameplay Ability could still be responsible for making said Spec, the Actor class would then have a Gameplay Ability Spec public variable so that it could be "loaded" on spawn, this is the way used in Epic's Action RPG example project [2]. Alternatively, the Actor could send Gameplay Events that the Ability would wait for, or we could create a new Ability Task that has output execution pins for when the Actor does something like OnBeginOverlap.

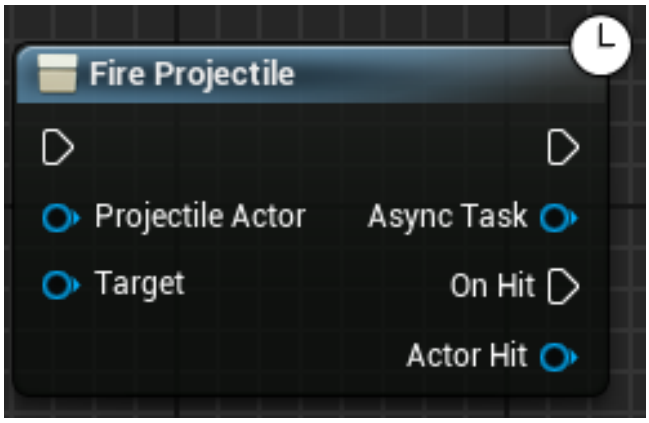

Figure 6: A proposed Fire Projectile Ability Task with On Hit output delegate

5.6 Ability Tasks

Ability Tasks can only be written in C++, so making new ones should arise from the designer's needs and what conventions have been settled upon. To continue on the question of spawning Actors, maybe the team has decided that using the Spawn Actor Task is good enough, or maybe we think that it would be better to have a more specialised Fire Projectile Task.

5.7 Events and Triggered Abilities

If our game makes heavy use of triggered abilities, we will have to ensure that corresponding Gameplay Events are sent accordingly. In most cases, Ability designers should be concerned with *receiving* Gameplay Events, not with sending them. Indeed any "listenable" event (lowercase 'e') should implicitly send Gameplay Events to whomever it may concern. This can be achieved by plugging into the many delegates that are already defined in the various GAS classes, or by overriding methods of these classes. In addition to specifying which Game Events should be fired when, we have to convene about which Gameplay Tag (singular) they will carry as these tags are used as keys in the Ability triggering process.

5.8 Adding and Editing Gameplay Tags

Gameplay Tags can be used for much more than identifying Ability-triggering Gameplay Events: by default in the GAS they can govern things like Ability Activation and Cancellation, Gameplay Effect conditional application (required on the source or target of an Effect), Effect immunity, ... and they can of course be used in many other creative ways. The hierarchical nature of Gameplay Tags allows us to classify them in categories and subcategories so each project would likely have its own Tag nomenclature. Some categories could contain a more or less fixed list of tags and the creation of new categories should maybe be limited, for that matter the Unreal Editor allows the restriction of some parts of the Gameplay Tag hierarchy, showing a warning asking if you have permission from the Lead Designer (or whatever role of you set as owner of the category) before editing tags. [5]

5.9 Emergent Behaviour Through Ability Combination

In the case of a complex game we may want to allow Abilities or Effects to interact with each other in order to create emergent combinations. If the game is supposed to evolve a lot, a given Ability or Effect doesn't have to know how and by what it can be modified so we don't end up with tons of special cases that may or may not be relevant in the end product. This is a reason for my previous suggestions: properly set up Gameplay Events allow Abilities to be trigger when, say, damage is just about to be dealt and that damage is a Gameplay Effect Spec that can then be modified by the triggered Ability. If our game designers are a bit too fond of such interactions, though, we are likely to hit a point where these guidelines are not enough, and we would have to either implement their ideas as special cases or maybe build new subsystems to accommodate them. Such a case that I can foresee is if an ability should add behaviour to a spawned actor, like changing *all* projectiles to not be destroyed on impact and instead bounce towards an additional target.

6 IMPLEMENTATION SPECULATIONS

In this section I will look at a few Abilities and how I would consider implementing them using the Gameplay Ability System in different contexts. I will use examples from well-known games and will try to consider different scenarios regarding project-wide architecture and design decisions. These are all game mechanics that I kept in mind while gaining an understanding of the GAS and its capabilities.

6.1 Quake's *Quad Damage*

The effect of this power-up is simple: anytime you would deal damage, you deal four times as much instead. Critical hits in a RPG would be very similar, with an element of chance added.

6.1.1 Damage Attribute. In a game with a restrained set of abilities such as a *Quake* clone, switching weapons could change a Damage Attribute on the player's Character. In that case the Quad Damage pickup is very simple: it applies a timed Effect with a fixed 4-magnitude multiplicative Modifier on the player's Damage Attribute.

6.1.2 GameEffectSpec Manipulation. In cases where damage amount is not tied to an explicit character Attribute, the pickup's effect can instead give a Gameplay Ability that would be triggered each time a Damage Effect Spec is created and alter it.

6.2 Damage Mitigation

The opposite of the previous mechanics: how do we lower the amount of damage dealt to you?

6.2.1 Attribute Based. In *League of Legends* for instance, every damageable entity has Armor. In that case the damage calculation, be it an Effect Magnitude Modifier Calculation or Execution Calculation, or done on the receiver with a Damage Received Attribute, can take care of applying whatever mitigation formula is appropriate.

6.2.2 Effect Spec Manipulation. If it doesn't make sense to have such an Attribute, or if we want to grant an entity some exotic damage mitigation mechanics, we can still use triggered Ability to modify the Effect Spec when you should receive damage.

6.3 Penetrating Damage Mitigation

In *League of Legends* Armor Penetration, though not as ubiquitous as Armor, is also a stat a character has, so it would make sense to have it taken into account by the damage calculation. In a game in which these are not Attributes, though, we hit a point where we have to choose how we bypass the limitations of the GAS: we can't apply Effects to Effect Specs, modifying an Effect Spec is not an Effect in itself, ... Also, while we're at it, in addition to Armor and Armor Penetration we could imagine Hardened Armor that counters Penetration, and super-penetrating damage that would counter Hardened Armor, and so on. While the rabbit hole has to end somewhere, we have to consider how far we could go down it, theoretically and practically.

6.3.1 Treat it as a Special Case. This rabbit hole ends *right here* and not any deeper, Alice. Penetration, or Hardening, is the final depth of this otherwise potentially endless stack of modifiers, and each of these effects know about those that can counter it. We use

Attributes for their parameters and its fine because there's only a couple of them.

6.3.2 Build a new Subsystem. If modifying Effect Specs on the fly is a common occurrence, we might want to build a class to reify those changes, some kind of Effect Spec Upon Effect Spec, that can stack or alter each other. We should make sure that the project really *needs* it, and beware of digging ourselves into another rabbit hole by re-making a GAS inside the GAS.

6.3.3 Subvert the GAS. Or maybe we could extend Gameplay Effects and use them for more things that they are intended to, so that modifying an Effect Spec *is* done via applying Gameplay Effects. We could also have Abilities each give custom Attribute Sets to their instigators and targets. Again beware of the slippery slope, what if Character movement was done through Gameplay Effects as well...

6.4 League of Legend's *Luden* Items

Luden's Tempest (and its variants and predecessors) grand a player's Character a new passive ability that can be roughly described as:

> When one of your abilities deals magic damage, fires a few projectiles to new different targets.

This is a clear-cut case of a triggered Ability. In order to implement such an Ability in our game, we should send "I just dealt damage" Gameplay Events to characters, these Events would trigger the Ability that would then filter based on type of damage dealt and spawn the new projectiles.

7 CONCLUSION

I had not ever heard of the Gameplay Ability System when I set out to investigate the idea of using Blueprint to create character abilities. While at first it seemed to me that the GAS was not designed to do what I envisioned, I came to realize that it is more than powerful enough for anything I can think of given only a few adjustments and ensuring that the team adheres to some guidelines. Now that I have a better grasp of the GAS's architecture and capabilities, I look forward to using it in a team environment and I hope I will be able to do so during next semester's project.

REFERENCES

[1] Thomas Blair. [n.d.]. *Crowfall - Crowfall Live! Building Myrmidon Powers.* https://youtu.be/ojOPgctlxCk

[2] Epic. [n.d.]. *Unreal Engine Documentation: Action RPG Game.* https://docs.unrealengine.com/4.26/en-US/Resources/SampleGames/ARPG/

[3] Epic. [n.d.]. *Unreal Engine Documentation: Gameplay Abilities.* https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/GameplayAbilitySystem/GameplayAbility/

[4] Epic. [n.d.]. *Unreal Engine Documentation: Gameplay Ability System.* https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/GameplayAbilitySystem/

[5] Epic. [n.d.]. *Unreal Engine Documentation: Gameplay Tags.* https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Tags/

[6] Dan Kestranek. [n.d.]. *GASDocumentation.* https://github.com/tranek/GASDocumentation

[7] KJZ. [n.d.]. *Unreal Community Wiki: Gameplay Abilities and You.* https://unreal.gg-labs.com/wiki-archives/networking/gameplay-abilities-and-you