GD3P01 - ExpBoost plugin

Technical Design Document

Aiden Storey

Table of Contents

1. Project Summary

1.1. Introduction

This document describes how the Unreal Engine plugin ExpBoost was implemented to provide an experience and attribute system for developers. It will discuss the specifics of the logic used to create the systems.

1.2. Core Aim

The ExpBoost plugin aims to give developers access to a simple yet elegant set of tools that can be harnessed to setup an experience and attribute system for a wide range of game types. It utilises the functionality that is provided by Unreal Engine to form a set of components and helper classes to make such systems possible. The primary use of the plugin is aimed at developers that are utilising the blueprint features of Unreal Engine, with underlying support for those using C++ / scripting.

1.3. Key Technical Challenges

A concern when developing for an engine would be ensuring that the solution fits the requirements laid out by its design, whilst not compromising the usefulness of the plugin. This applies to both the formatting standards of the engine as well as the intended restrictions it may have in place.

1.4. Goals

The main goal is to create a plugin that gives the developer all of the intended features without introducing any bugs to their program, or having a great deal of processing overhead.

2. Requirement Analysis

2.1. Overview

The list of requirements has been created after analysis of both the Unreal Engine's architecture and basic principles of software engineering

2.2. Specific Requirements

- **Must utilise the Unreal class, function, and property macros.**

  Due to the plugin being aimed at developers who are using the blueprint functionality within the Unreal Engine, it is essential that the where possible the appropriate macros are used.

- **Simple to understand and use**

  To ensure that any developer is able to easily utilise the functionality of the plugin, it must be designed with usability in mind. This doesn't just apply to the functionality provided, naming conventions must be easy to understand with tool tips where available.

- **Exp Handling**

  The plugin must be able to handle the distribution and receiving of experience between the actors within a scene, as to simulate an RPG style game.

- **Attribute Definition**

  Attributes that represent various elements required in a game such as health or mana must be able to be defined dynamically so that it is not limiting the game.

- **Attribute Modification**

  Attributes must be able to have modifiers applied to them to change their values so that things such as items and skill buffs are a possibility.

3. Implementation

3.1. Overview

The implementation follows the standards set for the Unreal Engine, utilising its common practices and development methodology.

3.2. Experience Handling

One of the key features of the ExpBoost plugin is the handling of experience being passed between objects within the scene. Since any object in the scene can dispatch or receive experience or both, each part of the experience system needs to be its own type. After the split there are three main parts which are split into Unreal Engine Actor Components to best be utilised within a game.

3.2.1. EBDispatcher

This component dictates how the experience is distributed to the receivers. Since there are different types of experience distribution, it is driven by a switch statement which provides the right function to be called for the given type. Within the scope set out for this plugin there are three types of distribution given to the EBDispatcher:

- World: In this method of dispatching the EBDispatcher iterates over all of the EBReceivers in the scene and gives them the amount of experience set for the dispatcher.
- Individual: This method requires an EBComponent to be present so that it can pass the experience to the EBReceiver of the Actor that last dealt damage to the component. Within the context of the plugin this is used for when the EBAttriubute representing health has been reduced to zero.
- Other: Due to the scope constraints of this plugin this method is an intentionally empty method that is able to be overridden by a developer using the plugin to extend its functionality.

3.2.2. EBReceiver

This component is a buffer between the EBDispatcher and the EBComponent. Since not all objects with an EBComponent will need to receive experience, this acts as a middleman that can be attached when needed. When the EBDispatcher dispatches experience it is passed into the EBReceivers one function which then looks for an EBComponent owned by the same Actor and passes it off. This also acts as a way for developers to change how incoming experience occurs since the ReceiveExp function is able to be overridden in blueprints.

3.2.3. EBComponent

This component is fundamentally the core of the entire plugin. It holds the current information about levels, experience, and manages how they change throughout a game. It also handles taking damage and distributing it to the appropriate health representing EBAttribute.

When the EBAttribute representing health reaches zero this signals the death of the component, at which time it will look for an EBDispatcher attached to the same Actor and if found will use it to dispatch experience accordingly.

The current information about levels and experience is defined by a current variable (i.e. level and exp) and a max of the variable (i.e. level max and exp max). The exp max variable is taken from a CurveFloat type within the Unreal Engine. This means that the developer using the plugin is able to setup an experience curve and then the exp max will update based on the level accessing it.

The EBComponent is where the experience calculations actually take place. This happens when an attached EBReceiver passes experience onto the component. The logic behind the calculation run on this passed in experience is shown in the following pseudo code.

```
function receiveexp param amount
        add amount to current exp
        while exp is greater than exp max
                subtract exp max from exp
                increment level // level up
                update exp max

                if level is equal to level max
                        set exp to 0
                        break loop

                if not able to multilevel
                        set exp to the lesser of exp or exp minus 1
                        break loop


        end
end
```

3.3. Attribute Definition

The other main aspect of the ExpBoost plugin is its attribute system. This is built around a similar method to the experience related aspects in that it is built as a component that can be attached to an Actor.

3.3.1. EBAttribute

The EBAttribute is how an attribute for use in a game is defined. One of the main things that was kept in mind is that attributes often get stronger the greater the level of the player. This meant that like the experience, the attribute was going to be defined by a CurveFloat type so that it would be able to increase as the player leveled up.

This also required the EBAttribute to be able to take the level from the main component, since that is where the main leveling of the Actor would occur. To achieve this the component can lock all of its own leveling methods and give control to an EBComponent by binding to it, which will mean the component is now in control of the level. There is still basic control over the variable in that it is able to reset its value to max when the main component updates its level.

3.3.2. EBComponent

One of the main features of the EBComponent is that it can take damage and give it to the appropriate EBAttribute that represents health. To do this, the name of the EBAttribute must be given to the component so that it can search for it attached to the same Actor. If it is present it will reduce the health attribute by the incoming damage amount and if that amount reaches 0, it will notify the death of the component.

3.4. Attribute Modification

In a lot of game scenarios it is important that external factors are able to increase or decrease the value of the attributes of an Actor. Within the ExpBoost plugin this is answered by having modifiers that can be applied to any given attribute.

3.4.1 EBAttributeModifier

This is set up as a simple struct that has the name of the attribute to modify, the type of modification, and the amount to modify by. There are three types of modifications that can be applied:

- Static: This is where the amount is directly applied to the base value of the attribute.
- Multiplier: This is where the result of the static plus base value is multiplied by the amount then added to the number.
- Percentage: This is where the result of the static plus base value is multiplied by the amount divided by 100 then added to the number.

3.4.2 EBItem

To be able to have multiple attributes being modified at once, the EBItem acts as a way for the developer to collate a bunch of modifiers. It has the ability to be attached/detached from an actor at which point it will iterate over the EBAttributes attached to the actor and then apply the modifiers that are also in the EBItem.

4. Future Improvements

4.1. Overview

The underlying motivation for the ExpBoost plugin is to provide easy to setup and use RPG style elements. This leads into a world of possibilities for future improvements that could be made to the plugin. The more dominant aspects are as listed below.

4.2. More Dispatcher Types

Early on during the development of the ExpBoost plugin, it became evident that the number of dispatcher types would need to be limited as to not enter to territory outside of the project's scope. A specific example of this was the desire to implement party exp distribution. However without the knowledge of how a party system would be defined it was going to be difficult to design a dispatching method that could be widely utilised.

As to not limit any developers requiring more from the ExpBoost plugin, it has been designed with the ability to create extra dispatch methods by overriding its DispatchExpOther function which is currently empty.

4.3. Party System

As stated in the previous section the lack of knowledge of the party system for the dispatcher was the main cause for not implementing party experience dispatching. A very viable solution to this problem would be to provide a party system within the plugin which integrated nicely with the currently existing components.

So that party experience is still possible for a developer, the receiver component has been designed in a way that it’s ReceiveExp function acts as a buffer between experience being dispatched and a component receiving it. This means that it can be overridden and altered to improve the functionality, not only for party experience distribution but a wide variety of options.

4.4. Experience Modification

Another issue of scope was devising a system of which incoming experience modification could occur on a per player basis. Having the ability to apply modifiers to increase or decrease the amount of experience gained is a case by case basis, but a definitely useful feature to have.

In the interim a CalculateModifiedExp function has been added to the plugin which currently returns the input experience amount. When overridden this will be able to be utilised to perform a per player modification when receiving experience.