# Main Parts of GAS (Based on the course and project Aura by Stephen Ulibarri)

AbilitySystemComponent

AttributeSet

GameplayAbility

AbilityTask

GameplayEffect

GameplayCue

GameplayTag

# First Steps of GAS

AbilitySystemComponent and AttributeSet can be added either on the Pawn class or the PlayerState class.

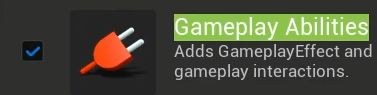
The image shows the difference.

AuraPlayerState.cpp

NetUpdateFrequency is how often the server will try to update clients.

AAuraPlayerState::AAuraPlayerState()  
{  
 NetUpdateFrequency = 100.f;

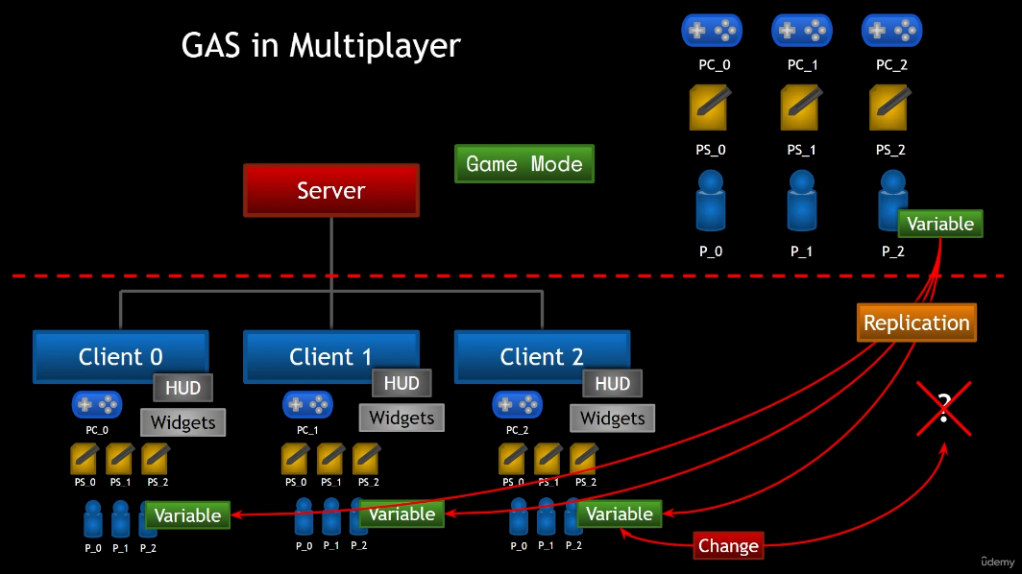
}



GameplayAbilities plugin must be enabled.

{  
 "GameplayAbilities",  
 "GameplayTags",   
 "GameplayTasks"  
}

Modules to add to the project.

Multiplayer flow in GAS.

# ASC and AS Setup (AbilitySystemComponent and AttributeSet)

We first create pointers for ASC (UAbilitySystemComponent) and AS (UAttributeSet) in the base character class AAuraCharacterBase. Then construct the actual components in the AAuraEnemy class for the enemies and in the AAuraPlayerState class for the player. Also, set ASC replicated in both AAuraEnemy and AAuraPlayerState constructors:

AuraPlayerState.cpp

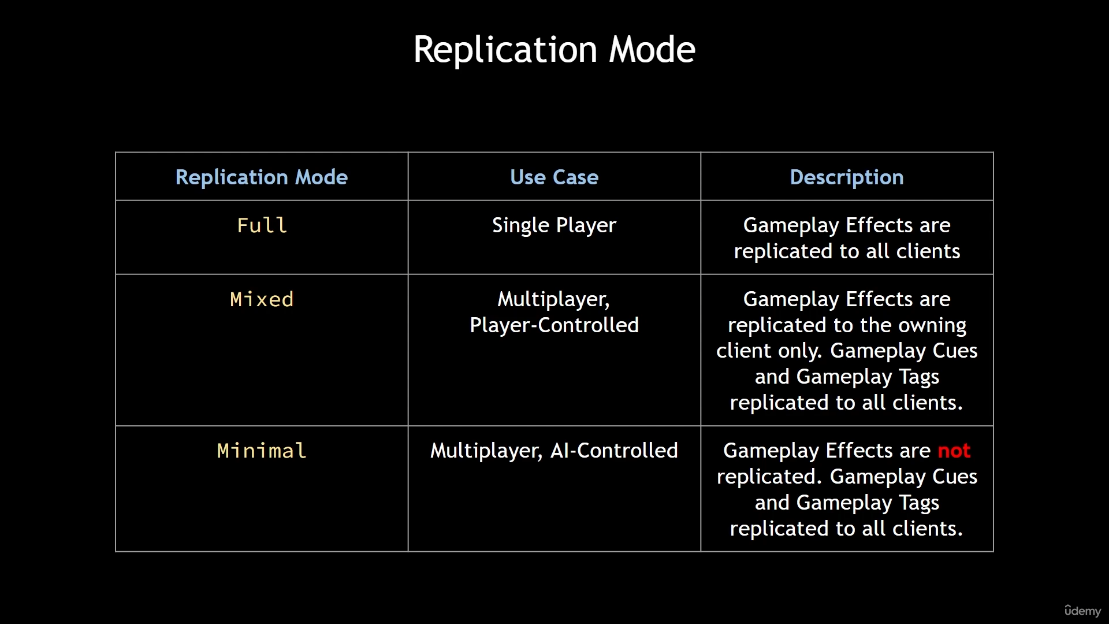
AAuraPlayerState::AAuraPlayerState()  
{  
 NetUpdateFrequency = 100.f;  
  
 AbilitySystemComponent = CreateDefaultSubobject<UAuraAbilitySystemComponent>("AbilitySystemComponent");  
 AbilitySystemComponent->SetIsReplicated(true);   
  
 AttributeSet=CreateDefaultSubobject<UAuraAttributeSet>("AttributeSet");  
}

We must also implement the interface IAbilitySystemInterface in both classes. This interface allows us to check if an actor has ASC, if it does, then get its ASC by implementing the function:

AuraPlayerState.cpp

virtual UAbilitySystemComponent\* GetAbilitySystemComponent() const override;

We also create a getter for AS (AttributeSet) for convenience.

Now, we set the replication mode of the ASC components. There are 3 modes of replication modes. According to this table, mixed is used for the player and minimal is used for the AI.

Here is the function for setting replication mode of ASC:

AuraPlayerState.cpp

AAuraPlayerState::AAuraPlayerState()  
{

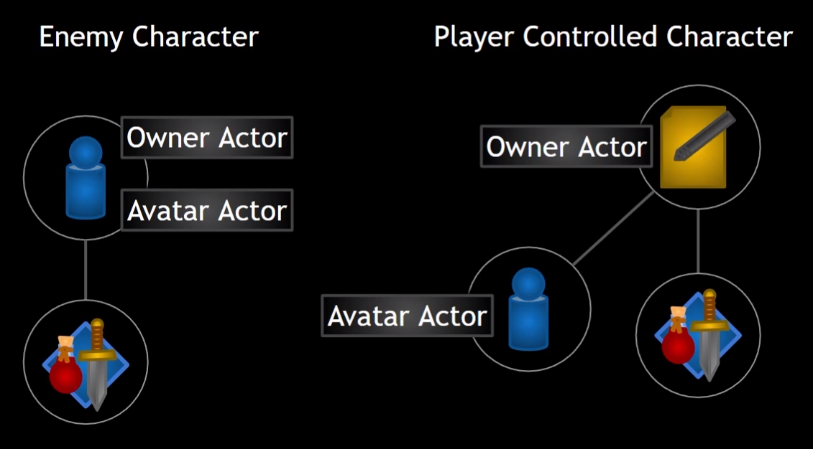
.

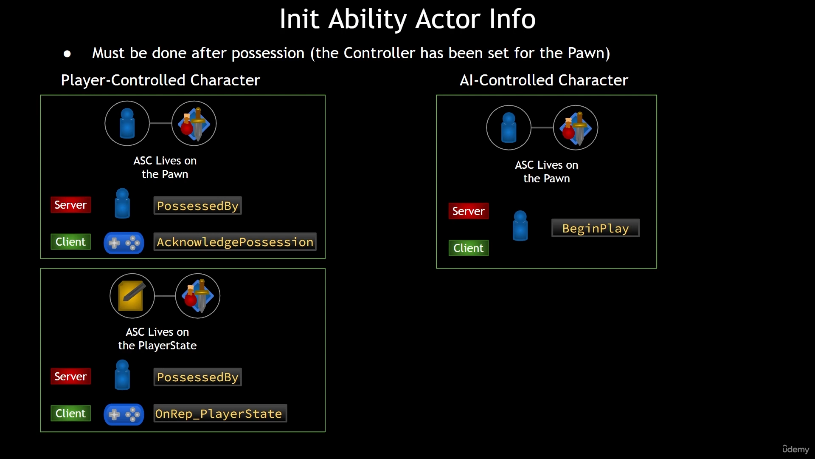
.

AbilitySystemComponent->SetReplicationMode(EGameplayEffectReplicationMode::Mixed);

.  
 .  
}

We must initialize ASCs in our project and set the pointers in PlayerCharacter to the ASC that was constructed in PlayerState. But first, we need to set the ownership of ASCs.

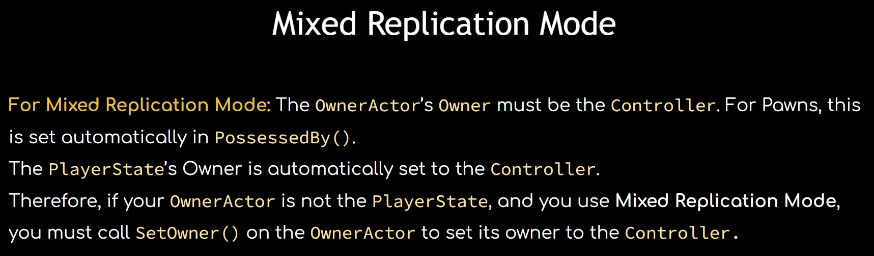
ASC has the concept of ownership. Since it may be constructed on the classes other than APawn, it has 2 separate variables, OwnerActor and AvatarActor. OwnerActor is actually the owner of ASC, AvatarActor is the representation of ASC in the world. For this reason, the ownership of ASCs in our project should be as it’s shown in the image.

InitAbilityActorInfo must be done after possession. First for the player, for the server side of the code, it will be called in the function PossessedBy. However, since our ASC for the player was constructed in PlayerState and not only the pawn must be possessed but also our PlayerState must be valid at the time we call InitAbilityActorInfo, we are going to use OnRep\_PlayerState for the client side. For the AI controlled character, since its ASC lives on the pawn, we call InitAbilityActorInfo in BeginPlay for both server and client sides.

AuraCharacter.cpp

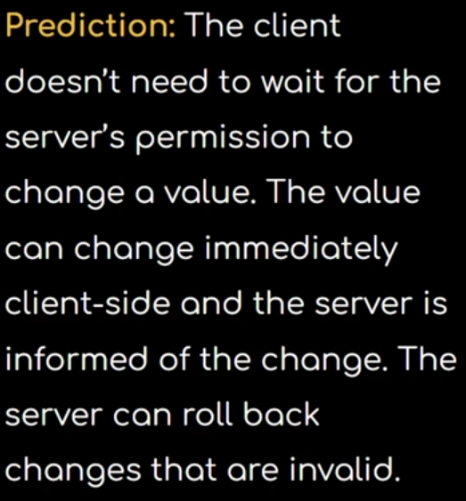
void AAuraCharacter::PossessedBy(AController\* NewController)  
{  
 Super::PossessedBy(NewController);  
  
 *// Init Ability Actor Info for the Server* InitAbilityActorInfo();  
}  
  
void AAuraCharacter::OnRep\_PlayerState()  
{  
 Super::OnRep\_PlayerState();  
  
 *// Init Ability Actor Info for the Client* InitAbilityActorInfo();  
}

void AAuraCharacter::InitAbilityActorInfo()  
{  
 AAuraPlayerState\* AuraPlayerState = GetPlayerState<AAuraPlayerState>();  
 check(AuraPlayerState);  
 AuraPlayerState->GetAbilitySystemComponent()->InitAbilityActorInfo(AuraPlayerState, this);  
 Cast<UAuraAbilitySystemComponent>(AuraPlayerState->GetAbilitySystemComponent())->AbilityActorInfoSet();  
 AbilitySystemComponent = AuraPlayerState->GetAbilitySystemComponent();  
 AttributeSet = AuraPlayerState->GetAttributeSet();  
}

Also, there is an important detail about mixed replication mode where if you decide to set a mixed replicated ASC’s owner, make sure that you set it to an actor which is controlled by a Controller.

# AttributeSet

It's better to have different attribute sets in one attribute class. All attributes are numerical quantities associated with a given entity in the game such as character. All attributes are floats. They exist in a structure called FGameplayAttributeData. Attributes are stored in the AttributeSet. Attributes can be set directly in code, but the preffered way to change them is by applying GameplayEffects.

Aside from the built-in capabilities of gameplay effects, it also allows us to predict changes to attributes.

Attributes consist of 2 variables: BaseValue and CurrentValue. BaseValue is the permanent value of an attribute. CurrentValue is the BaseValue plus any temporary modifications caused by gameplay effects.

Note: It’s a common mistake to confuse BaseValue and CurrentValue with Health and MaxHealth. MaxHealth also should be a separate attribute. BaseValue doesn’t represent the maximum value of an attribute.

Here we create an attribute called Health. Since we’d like most attributes to be replicated, we mark it and give it a function using a common naming convention OnRep\_Health. We pass OldHealth attribute for the replication function and inform the AbilitySystem for it to do necessary job. We must then override a function that classes having replicated variables must have: GetLifetimeReplicatiedProps(…). Inside that, we use DOREPLIFETIME\_CONDITION\_NOTIFY(…) macro to register our attributes. It has 4 parameters: class name, variable name, condition and REPNOTIFY. We use COND\_None with no condition and REPNOTIFY\_Always. Always means whenever the variable is set, it replicates. There is another option REPNOTIFY\_OnChange which means that it replicates only when the value of the variable is changed.

AuraAttributeSet.h

AuraAttributeSet.cpp

void UAuraAttributeSet::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const  
{  
 Super::GetLifetimeReplicatedProps(OutLifetimeProps);  
DOREPLIFETIME\_CONDITION\_NOTIFY(UAuraAttributeSet, Health, COND\_None, REPNOTIFY\_Always);  
}

void UAuraAttributeSet::OnRep\_Health(const FGameplayAttributeData& OldHealth) const  
{  
 GAMEPLAYATTRIBUTE\_REPNOTIFY(UAuraAttributeSet, Health, OldHealth);  
}

public:   
 virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const override;   
UPROPERTY(BlueprintReadOnly, ReplicatedUsing = OnRep\_Health, Category = "VitalAttributes")  
 FGameplayAttributeData Health;  
 ATTRIBUTE\_ACCESSORS(UAuraAttributeSet, Health);

UFUNCTION()  
 void OnRep\_Health(const FGameplayAttributeData& OldHealth) const;

The proper way to change an attribute is to use GameplayEffects, however, we can still have a useful macro set for getters and setters. First, define a macro like this on top of the header class.

AuraAttributeSet.h

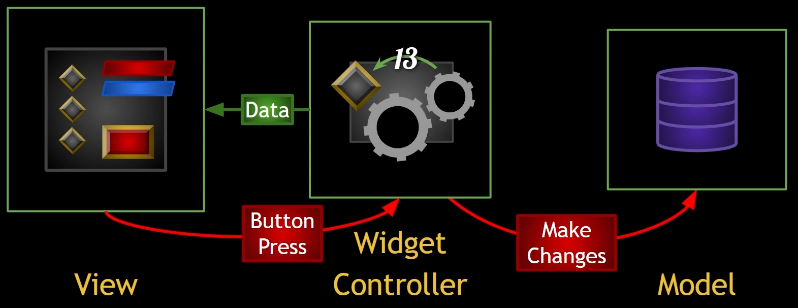
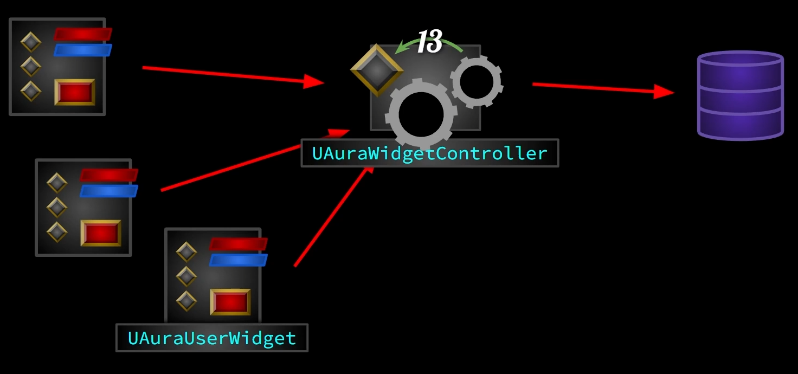
AuraAttributeSet.h

#define ATTRIBUTE\_ACCESSORS(ClassName, PropertyName) \  
 GAMEPLAYATTRIBUTE\_PROPERTY\_GETTER(ClassName, PropertyName) \  
 GAMEPLAYATTRIBUTE\_VALUE\_GETTER(PropertyName) \  
 GAMEPLAYATTRIBUTE\_VALUE\_SETTER(PropertyName) \  
 GAMEPLAYATTRIBUTE\_VALUE\_INITTER(PropertyName)

Then, add this right after declaring your variable.

ATTRIBUTE\_ACCESSORS(UAuraAttributeSet, Health);

# Game UI

In our project, we’ll create and use several WidgetControllers in order to transfer the data to the viewport and make changes when the widget is interacted. The UI system will be modular and flexible in order to handle the complexity as the project grows.

Our UAuraWidgetController will be a simple UObject that harvests data from the system and transfers to the widget. We’ll also create a custom UUserWidget, UAuraUserWidget, so that our widgets can be compatible with the system. Finally, we’ll subclass AHUD to AAuraHUD so we can construct, initialize and store these classes.

Our base widget class, UAuraUserWidget, will have a member variable for WidgetController. By this way, widgets will have pointers to their controller if necessary, and if there is, when the controller broadcast the change in the data, widget will get this change and reflect on the viewport, and we have a one-way dependency from the widgets to the controllers. The widget controller isn’t going to know what widget it’s associated with, but the widgets are going to know what their controller is.

public:UPROPERTY(BlueprintReadOnly)  
 TObjectPtr<UObject> WidgetController;  
UFUNCTION(BlueprintCallable)  
 void SetWidgetController(UObject\* InWidgetController);  
  
protected:UFUNCTION(BlueprintImplementableEvent)  
 void WidgetControllerSet();

AuraUserWidget.h

Widgets will handle the visual (i.e. how and where the data should look like etc.) kind of stuff, so we are going to use widget blueprints to set our widget controllers in widget blueprints. Since these widgets are going to be bound to the controllers, we also need a function acting like BeginPlay(), meaning that the widget should not be initialized on Construct or BeginPlay, but on a custom function such as WidgetControllerSet() which will be called when the controller is set. We create this function and call it in SetWidgetController(…) where we set the controller. Also, we want to implement WidgetControllerSet() in blueprint, because what happens after the widget controller is set will be specific to that widget.

AuraUserWidget.cpp

void UAuraUserWidget::SetWidgetController(UObject\* InWidgetController)  
{  
 WidgetController = InWidgetController;  
 WidgetControllerSet();  
}

In the widget controller side, our widget controller is responsible for getting data, meaning that it depends on the classes that are related to the data which the controller is interested in. We want our controller to have a set of variables of the classes that the controller may need to obtain the data from. These will be PlayerController, PlayerState, AbilitySystemComponent and AttributeSet.

protected:UPROPERTY(BlueprintReadOnly, Category = "WidgetController")  
TObjectPtr<APlayerController> PlayerController;  
  
 UPROPERTY(BlueprintReadOnly, Category = "WidgetController")  
TObjectPtr<APlayerState> PlayerState;  
  
 UPROPERTY(BlueprintReadOnly, Category = "WidgetController")  
TObjectPtr<UAbilitySystemComponent> AbilitySystemComponent;  
  
 UPROPERTY(BlueprintReadOnly, Category = "WidgetController")  
TObjectPtr<UAttributeSet> AttributeSet;

AuraWidgetController.h

private:

UPROPERTY()  
 TObjectPtr<UAuraUserWidget> OverlayWidget;  
  
 UPROPERTY(EditAnywhere)  
 TSubclassOf<UAuraUserWidget> OverlayWidgetClass;

Then, we create an OverlayWidget in the editor (all widgets will be child of UAuraUserWidget). In the overlay, we will show the player health and mana at the moment. So, the overlay requires a widget controller. Also, we initialize overlay in the HUD. Therefore, we will have variables for OverlayWidget and OverlayWidgetClass in order to create and add to the viewport the widget.

AuraHUD.h

Now, we create the OverlayWidgetController. In order to create a widget controller in our system, we need to set 4 variables which we created in the base widget controller class, AuraWidgetController. These 4 variables are, PS (PlayerState), PC (PlayerController), ASC (AbilitySystemComponent) and AS (AttributeSet). We’d like an easy way to set these variables. So, we create a struct, FWidgetControllerParams. We make it BlueprintType just in case we need it in blueprint. First, we create an empty constructor, then we create a second constructor setting the 4 variables. We can also define constructor in the struct declaration. And for that, we need variables for these 4 variables in the struct.

USTRUCT(BlueprintType)  
struct FWidgetControllerParams  
{  
 GENERATED\_BODY()  
  
FWidgetControllerParams() {}  
  
FWidgetControllerParams(APlayerController\* PC, APlayerState\* PS,  
 UAbilitySystemComponent\* ASC, UAttributeSet\* AS) :  
  
PlayerController(PC), PlayerState(PS),  
AbilitySystemComponent(ASC), AttributeSet(AS) {}  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite)  
TObjectPtr<APlayerController> PlayerController = nullptr;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite)  
TObjectPtr<APlayerState> PlayerState = nullptr;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite)  
TObjectPtr<UAbilitySystemComponent> AbilitySystemComponent = nullptr;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite)  
TObjectPtr<UAttributeSet> AttributeSet = nullptr;

};

AuraWidgetController.h

AuraWidgetController.h

Then, we create a function to initialize these parameters. Make it BlueprintCallable in case we need it in blueprint. We set the 4 member variables to the struct variables.

AuraWidgetController.cpp

public:UFUNCTION(BlueprintCallable)  
 void SetWidgetControllerParams(const FWidgetControllerParams& WCParams);

void UAuraWidgetController::SetWidgetControllerParams(const FWidgetControllerParams& WCParams)  
{  
 PlayerController = WCParams.PlayerController;  
 PlayerState = WCParams.PlayerState;  
 AbilitySystemComponent = WCParams.AbilitySystemComponent;  
 AttributeSet = WCParams.AttributeSet;  
}

public:UOverlayWidgetController\* GetOverlayWidgetController(const FWidgetControllerParams& WCParams);   
 void InitOverlay(APlayerController\* PC, APlayerState\* PS, UAbilitySystemComponent\* ASC, UAttributeSet\* AS);

private:

UPROPERTY()  
 TObjectPtr<UOverlayWidgetController> OverlayWidgetController;  
  
 UPROPERTY(EditAnywhere)  
 TSubclassOf<UOverlayWidgetController> OverlayWidgetControllerClass;

Now, we can create OverlayWidgetController. We want our HUD to initialize the widget controllers, we also want only 1 instance running in the game of a widget controller class. So, we initialize OverlayWidgetController in the HUD, we also create a getter for that controller so there will be only 1 instance of that class. When creating the instance of the widget controller, we pass this (HUD) as the outer object. We also create a function, InitOverlay(…), to initialize the overlay widget controller. We can’t initialize it in BeginPlay since we don’t have access to the 4 key variables in BeginPlay. In the initializer function, we pass these 4 variables so we can set FWidgetControllerParams. We check the pointers here, but we also want to give a message which describes the situation in case it can’t access the OverlayWidget class and OverlayWidgetController class, so we use checkf(…). After checking the classes, we call the function GetOverlayWidgetController(…) which will either create a new controller or return the existing one, we store the controller in a local variable. And using this local variable, we call the function SetWidgetController(…) on OverlayWidget to set its controller. Finally, we add the OverlayWidget to the viewport. This InitOverlay(…) function constructs both the overlay and its controller, and set the overlay widget’s widget controller before adding it to the viewport.

UOverlayWidgetController\* AAuraHUD::GetOverlayWidgetController(const FWidgetControllerParams& WCParams)  
{  
 if (OverlayWidgetController == nullptr)  
 {  
 OverlayWidgetController = NewObject<UOverlayWidgetController>(this, OverlayWidgetControllerClass);  
 OverlayWidgetController->SetWidgetControllerParams(WCParams);   
 }  
  
 return OverlayWidgetController;  
}

void AAuraHUD::InitOverlay(APlayerController\* PC, APlayerState\* PS, UAbilitySystemComponent\* ASC, UAttributeSet\* AS)  
{  
 checkf(OverlayWidgetClass, TEXT("Overlay Widget Class uninitialized, please fill out BP\_AuraHUD"));  
 checkf(OverlayWidgetControllerClass,  
 TEXT("Overlay Widget Controller Class uninitialized, please fill out BP\_AuraHUD"));  
  
 UUserWidget\* Widget = CreateWidget<UUserWidget>(GetWorld(), OverlayWidgetClass);  
 OverlayWidget = Cast<UAuraUserWidget>(Widget);  
  
 const FWidgetControllerParams WidgetControllerParams(PC, PS, ASC, AS);  
 UOverlayWidgetController\* WidgetController = GetOverlayWidgetController(WidgetControllerParams);  
  
 OverlayWidget->SetWidgetController(WidgetController);  
 WidgetController->BroadcastInitialValues();  
  
 Widget->AddToViewport();  
}

AuraHUD.cpp

AuraHUD.h

AuraCharacter.cpp

It's time to call this InitOverlay(…) function to initialize the overlay. A good place to call the function is InitAbilityActorInfo(…) in AuraCharacter, where the character was already possessed by the controller and we also set PlayerState, AbilitySystemComponent and AttributeSet. Also, we can always access HUD through the player controller. So, we get the player controller and get the HUD, cast it to AuraHUD and then call InitOverlay(…). The question arises in this solution is that either we should check or assert PlayerController pointer. Since it’s a character class and exists for all the players on all the machines in multiplayer, and since only the local controller exist for the player, we should null check for the controller, not assert check. If we assert check, the program will crash while trying to get other players controllers whereas their controllers don’t exist on that machine. So, we do null check and continue and only for the local player, we get the controller and call the function in the HUD. Remember to set the class pointers in blueprints.

void AAuraCharacter::InitAbilityActorInfo()  
{  
 .

.

.  
  
 if (AAuraPlayerController\* AuraPlayerController = Cast<AAuraPlayerController>(GetController()))  
 {  
 if(AAuraHUD\* AuraHUD = Cast<AAuraHUD>(AuraPlayerController->GetHUD()))  
 {  
 AuraHUD->InitOverlay(AuraPlayerController, AuraPlayerState, AbilitySystemComponent, AttributeSet);  
 }  
 }  
}

Now, we will create a delegate to broadcast the initial values to the widget. Since the widget has a handle to the widget controller, it can register the delegate in the widget controller, and when the widget controller broadcasts the initial data, the widget will know and reflect it in the viewport. We create this delegate in the base widget controller class, AuraWidgetController, so all the widget controller can broadcast their own unique initial values. After creating it in the base widget controller, we override it in OverlayWidgetController class.

DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE\_OneParam(FOnAttributeChangedSignature, float, NewValue);

OverlayWidgetController.h

public:

virtual void BroadcastInitialValues() override;

OverlayWidgetController.h

AuraWidgetController.h

public:

UFUNCTION(BlueprintCallable)  
 virtual void BroadcastInitialValues();

Now, to broadcast the values which Health, Mana, MaxHealth and MaxMana in this case, we will create a few delegates. We want them to be dynamic multicast, because we want to set them in blueprint and also, we might multiple widget blueprints to bind to these delegates so they can update.

public:UPROPERTY(BlueprintAssignable, Category = "GAS|Attributes")  
 FOnAttributeChangedSignature OnHealthChanged;

.

.

.

OverlayWidgetController.h

void UOverlayWidgetController:: BroadcastInitialValues()  
{  
 const UAuraAttributeSet\* AuraAttributeSet = CastChecked<UAuraAttributeSet>(AttributeSet);  
  
 OnHealthChanged.Broadcast(AuraAttributeSet->GetHealth());  
 OnMaxHealthChanged.Broadcast(AuraAttributeSet->GetMaxHealth());  
 OnManaChanged.Broadcast(AuraAttributeSet->GetMana());  
 OnMaxManaChanged.Broadcast(AuraAttributeSet->GetMaxMana());  
}

In order to broadcast the correct value, we need to cast AttributeSet member variable to AuraAttributeSet where we store the attributes of the character. Then we broadcast the values.

OverlayWidgetController.cpp

To broadcast the initial values, we need to have a valid attribute set at that point in time. We should also call this after any widgets have had their widget controller set.

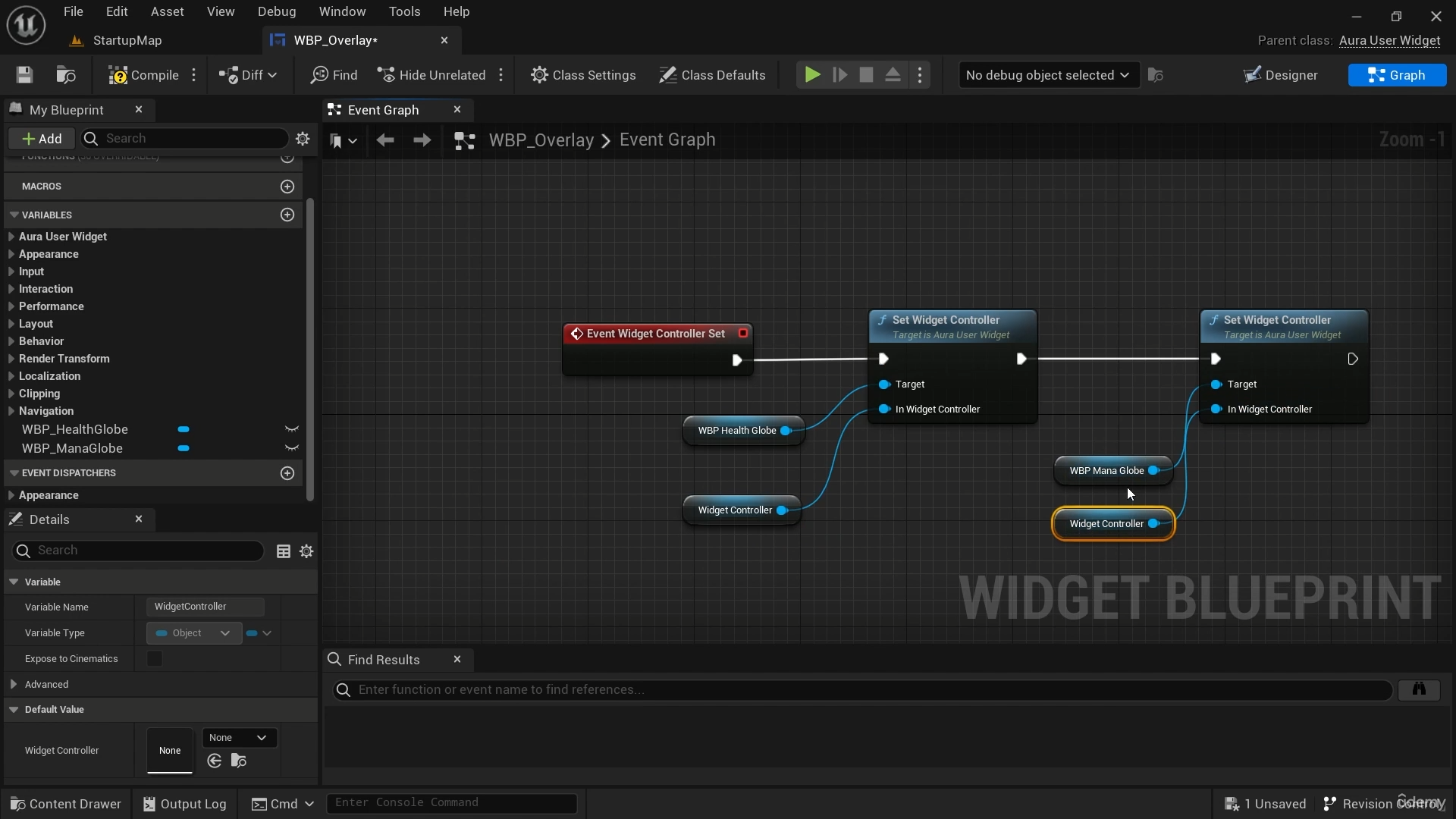
void AAuraHUD::InitOverlay(APlayerController\* PC, APlayerState\* PS, UAbilitySystemComponent\* ASC, UAttributeSet\* AS)  
{  
 .

.

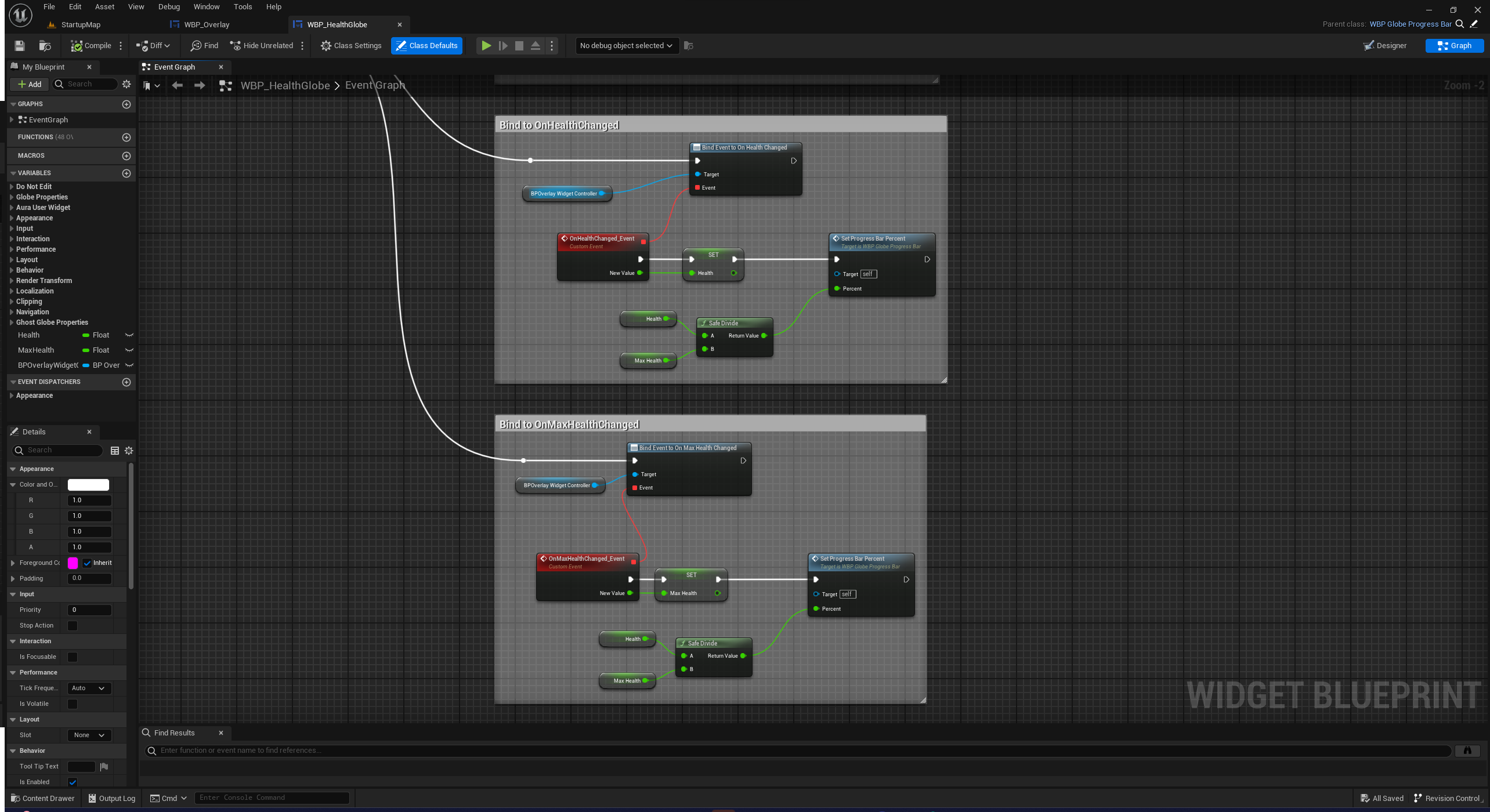
.  
  
 OverlayWidget->SetWidgetController(WidgetController);

-> WidgetController->BroadcastInitialValues(); <-  
  
 Widget->AddToViewport();  
}

AuraHUD.cpp



Now, we need to set our widgets’ (HealthGlobe and ManaGlobe) widget controllers. We set overlay’s widget controller in the code, and we have an event in AuraUserWidget, WidgetControllerSet(), which is called by HUD after the widget controller is set. This will make sure the overlay’s widget controller is set, so we can set the globe widgets’ controllers in this event.

When we set the globes’ widget controllers, their EventWidgetControllerSet() will also be triggered. So, we can bind globe widgets to those delegates. In the assigned broadcast event, we can set the health progress bar.

Note: In case you don’t see OverlayWidgetController to cast the widget controller, you should use BlueprintType and Blueprintable UCLASS specifiers in OverlayWidgetController. By this way, we will subclass OverlayWidgetController and create a blueprint and use it as the overlay’s widget controller. Remember to set the BP version of this controller in the HUD blueprint.

Now, in addition to binding widgets to initial values, we also want to respond to whenever an attribute changes. There is a function in ability system for that purpose. Since this is a functionality that all the widget controllers may need, we declare it in the base widget controller, leave it blank though. We define it in the child widget controllers. First, we bind OverlayWidgetController to the AbilitySystem for when an attribute changes using AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(…). We are going to add a lambda here, instead of a function, object etc. A lambda is an anonymous function that we don’t declare but define it in where it's called. It has square bracket, parenthesis and curly brackets. Since it’s an anonymous function, we can’t access to member variables or functions. To do that, we need to capture them in square brackets to use these member variables. We can capture variables, by pointers, by reference etc. In order to access our delegates created in OverlayWidgetController, we must capture this, referring to that widget controller. In parenthesis, we pass the signature of the function, and in curly brackets, we code the function body.

void UOverlayWidgetController::BindCallbacksToDependencies()  
{  
 const UAuraAttributeSet\* AuraAttributeSet = CastChecked<UAuraAttributeSet>(AttributeSet);  
  
 AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(AuraAttributeSet->GetHealthAttribute()).AddLambda(  
 [this](const FOnAttributeChangeData& Data)  
 {  
 OnHealthChanged.Broadcast(Data.NewValue);  
 }  
 );  
}

OverlayWidgetController.cpp

public:   
  
 virtual void BindCallbacksToDependencies();

AuraWidgetController.h

AuraHUD.cpp

Finally, we need to bind our callbacks to dependencies meaning that we need to call the function BindCallbacksToDependencies. It will be also done in the HUD. We can call this function in GetOverlayWidgetController(…) where that this widget controller is constructed for the very first time.

UOverlayWidgetController\* AAuraHUD::GetOverlayWidgetController(const FWidgetControllerParams& WCParams)  
{  
 if (OverlayWidgetController == nullptr)  
 {  
 .

.

.

OverlayWidgetController->BindCallbacksToDependencies();  
 }  
  
 return OverlayWidgetController;  
}

In this section, we basically bind our widget controller to AbilitySystem for attribute changes, and then broadcast these changes to the widgets in the lambda.

# Gameplay Effects

A GameplayEffect is an object of type UGameplayEffect. They are used to change Attributes and GameplayTags. They are data only, and we do only create blueprints based on UGameplayEffect and not subclass it. GE (GameplayEffects) changes attributes in 2 ways: modifiers and executions. Modifiers specify a type of operation to be performed on the attribute. They take a value, referred as magnitude, and uses that magnitude to change the attribute depending on the modifier operation. There are different types of operations such as Add (can have negative values to get the effect of subtraction, multiply, divide and override (replaces the attribute’s value with the value provided for the magnitude). The magnitude used in these operations is produced from the magnitude calculation. There are also several types of magnitude calculations: scalable float (can hardcode a value directly or use a table scaling the magnitude based on the GE level), attribute based (uses another attribute to change an attribute and customize even further i.e. Health = Strength x 10), custom calculation class (MMC, ModifierMagnitudeCalculations) (can create a class designed to capture other variables such as attribute or any other variables, and use them in a calculation that can be arbitrarily complex) and set by caller (key value pair, assigning a magnitude associated with a name or a gameplay tag, may be useful if need to set the modifiers magnitude based on code logic at the time the GE is created or being applied). There is an even more powerful way to change an attribute’s value in the gameplay effect is to use a custom execution (GameplayEffectExecutionCalculation, ExecCalc). These can change more than one attribute and they can do anything else that we code.

GE have a duration policy. Instant effect is applied instantly, has duration can modify an attribute for a set period of time, infinite can affect an attribute and that effect doesn’t go away until it’s removed manually.

GE can stack and have their own policies how to stack, can add gameplay tags and grant abilities. These are just a few things the gameplay effects can do.

GE can be applied directly, but usually we create a more lightweight version of them, GameplayEffectSpec. A spec typically contains the bare bones information needed to perform the modifications (still, the only actual instance of the GE class is CDO (ClassDefaultObject)). An effect spec carries information such as gameplay effects. It also has an effect context, an additional class that can store more information about the effects being applied. References to the causer and the target of the effect are easily accessed and can be used to determine what happens to the overall magnitude calculation as well as trigger gameplay mechanics to any actors involved in the effect. GE are often applied from withing gameplay abilities but they don’t have to be.   
  
It's time to change the effect actor, AuraEffectActor. Effect actor is not going to have an ASC component, however, it’s going to act as an effect actor which will apply a GE effect to a target with an ASC. We need to have a clean actor with only a scene component set as the root component, we need to add mesh and collision in the blueprint side. Then, we create a subclasss variables for a GE types (we will show instant GE only but the process is the same for all) and a function for applying GE such as ApplyEffectToTarget(…). In order to apply GE, we need target actor’s ASC. We can either cast interface to get this, but there is a more convenient way to do that which is using the static function library of GAS. The function is UAbilitySystemBlueprintLibrary::GetAbilitySystemComponent(…) which takes the target actor as argument and casts GAS interface but furthermore, if can’t cast the interface, it also looks for the ASC in any case. So, it’s better to use it to get the ASC of the target actor. To apply the effect, we have 4 options, since we use Spec and we apply effect already using Target’s ASC, we choose TargetASC->ApplyGameplayEffectSpecToSelf(…). Only the first input parameter is mandatory and it’s the GameplayEffectSpec, we will make one. Any ASC can make a spec and we’ll use TargetASC for this one, TargetASC->MakeOutgoingSpec(…). It takes the GE class, level and GE context handle. Context for GE is something we associate with the GE, specifically its effect spec by passing in an effect context or effect context handle (handle is a lightweight thing that contains the pointer to the context itself). We make context handle by using TargetASC->MakeEffectContext(). It doesn’t take any inputs. We store it in the FGameplayEffectContextHandle strutct. Context handle can be subclassed too. It stores the actual context as a pointer, Data. The handle itself has some utilites but the actual context carrying the information is Data. Now, we can set things on context, for example what caused this effect, by using EffectContextHandle.AddSourceObject(…), we can pass any UObject as the input parameter but in our case, we pass this as the source for the GE. Now, we can finally apply the effect. As we said, ApplyGameplayEffectSpect requires GameplayEffectSpec, not the spec handle. So, we need to get Data out of the spec handle. Also, since Data is a shared pointer, we need to get raw pointer by using Get(), finally, since it required the const reference and not the pointer itself, we need to dereference it. The final parameter should look something like this: TargetASC->ApplyGameplayEffectSpecToSelf(\*EffectSpecHandle.Data.Get()).

AuraEffectActor.h

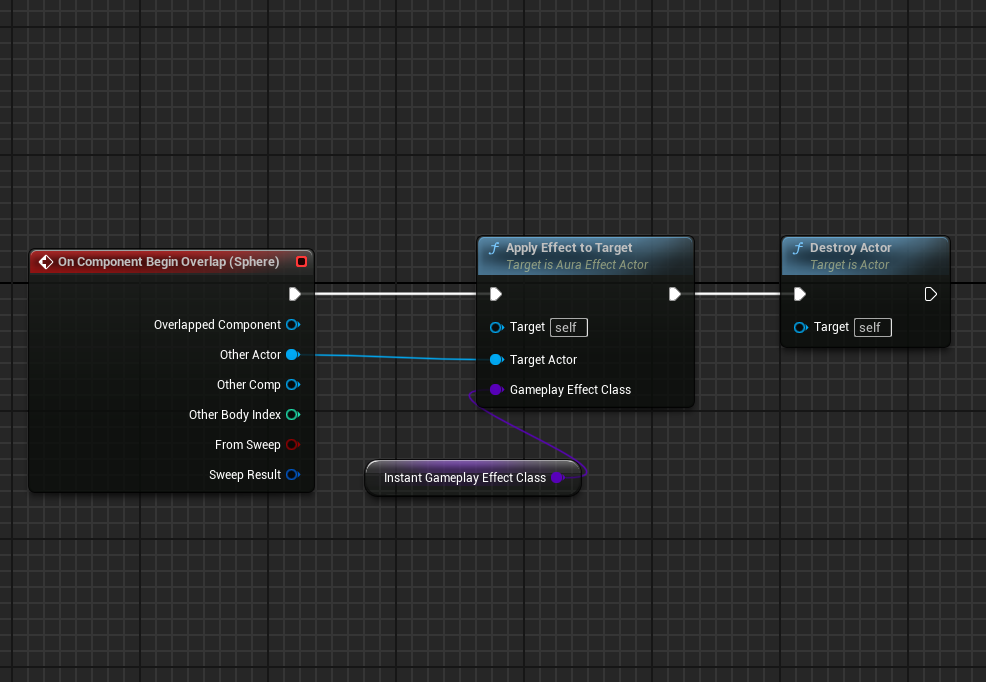
protected:

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|InstantEffect")  
 TSubclassOf<UGameplayEffect> InstantGameplayEffectClass;

UFUNCTION(BlueprintCallable)  
void ApplyEffectToTarget(AActor\* TargetActor, TSubclassOf<UGameplayEffect> GameplayEffectClass);

AuraEffectActor.cpp

void AAuraEffectActor::ApplyEffectToTarget(AActor\* TargetActor, TSubclassOf<UGameplayEffect> GameplayEffectClass)  
{  
 UAbilitySystemComponent\* TargetASC = UAbilitySystemBlueprintLibrary::GetAbilitySystemComponent(TargetActor);  
  
 if (TargetASC == nullptr) return;  
  
 check(GameplayEffectClass);  
 FGameplayEffectContextHandle EffectContextHandle = TargetASC->MakeEffectContext();  
 EffectContextHandle.AddSourceObject(this);  
 const FGameplayEffectSpecHandle EffectSpecHandle = TargetASC->MakeOutgoingSpec(  
 GameplayEffectClass, ActorLevel, EffectContextHandle);  
 const FActiveGameplayEffectHandle ActiveEffectHandle = TargetASC->ApplyGameplayEffectSpecToSelf(  
 \*EffectSpecHandle.Data.Get());  
  
 const bool bIsInfinite = EffectSpecHandle.Data.Get()->Def.Get()->DurationPolicy ==  
 EGameplayEffectDurationType::Infinite;  
 if (bIsInfinite && InfiniteEffectRemovalPolicy == EEffectRemovalPolicy::RemoveOnEndOverlap)  
 {  
 ActiveEffectHandles.Add(ActiveEffectHandle, TargetASC);  
 }  
}

Now we create a simple GE class for the health potion. It’s going to be an instant GE. Then in BP class of the health potion, we choose that GE class for our InstantGameplayEffectClass variable. Then on collision overlap, we call ApplyEffectToTarget(…) and pass InstantGameplayEffectClass. Finally, we destroy the actor so it acts like a potion and we consume it by picking it up.

As yet, we’ve discussed only instant GE. Further information about Gameplay Effect types can be found [here](https://docs.unrealengine.com/5.3/en-US/gameplay-effects-for-the-gameplay-ability-system-in-unreal-engine/#gameplayeffectlifetime). For Has Duration GE and Infinite effects, create variables for classes as we’ve done for Instant GE in AuraEffectActor class. Also, create Application Policy for all 3 effects and make them edit anywhere so we can set the application policies. First, create 2 enums for application and removal policies. Removal is only for infinite GE since it needs to be manually removed (also create a bool so we can set either it needs to be removed or not). Then create policy variables for each GE type. We will use these policies when we apply GEs and select in GE Blueprints how we should apply and remove our GEs. Also, we can convert HasDuration and Infinite GEs into Periodic GE by giving a Period value in the details tab of GEs. We can also set how GE behaves on stacking. Now since we can remove GEs, we will create 2 functions one for on begin overlapping and another for on end overlapping. Removing an infinite GE is a bit more complicated. We can store if the actor is infinite or not, and then check if the removal policy is “remove on end overlap”, then we can remove the infinite effect. To do this, we can have a pair of the two, the active effect handle and the duration policy. On removal, we first check the map for handles, compare it to the Target ASC, if it matches, remove the effect from the target, also remove it from the map. We also create a float variable for ActorLevel which can be edited later on and pass it in MakeOutgoingSpec.

UENUM(BlueprintType)  
enum class EEffectApplicationPolicy  
{  
 ApplyOnOverlap,  
 ApplyOnEndOverlap,  
 DoNotApply  
};  
  
UENUM(BlueprintType)  
enum class EEffectRemovalPolicy  
{  
 RemoveOnEndOverlap,  
 DoNotApply  
};

UCLASS()  
class AURA\_API AAuraEffectActor : public AActor  
{  
 GENERATED\_BODY()  
 .   
 .   
 .  
  
protected:

*/\*\*  
 \* Variables  
 \*/*

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects")  
 bool bDestroyOnEffectRemoval = false;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|InstantEffect")  
 TSubclassOf<UGameplayEffect> InstantGameplayEffectClass;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|InstantEffect")  
 EEffectApplicationPolicy InstantEffectApplicationPolicy = EEffectApplicationPolicy::DoNotApply;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|DurationEffect")  
 TSubclassOf<UGameplayEffect> DurationGameplayEffectClass;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|DurationEffect")  
 EEffectApplicationPolicy DurationEffectApplicationPolicy = EEffectApplicationPolicy::DoNotApply;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|InfiniteEffect")  
 TSubclassOf<UGameplayEffect> InfiniteGameplayEffectClass;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|InfiniteEffect")  
 EEffectApplicationPolicy InfiniteEffectApplicationPolicy = EEffectApplicationPolicy::DoNotApply;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects|InfiniteEffect")  
 EEffectRemovalPolicy InfiniteEffectRemovalPolicy = EEffectRemovalPolicy::RemoveOnEndOverlap;  
  
 TMap<FActiveGameplayEffectHandle, UAbilitySystemComponent\*> ActiveEffectHandles;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "AppliedEffects")  
 float ActorLevel = 1.f;  
  
 */\*\**  *\* Functions*  *\*/* UFUNCTION(BlueprintCallable)  
 void ApplyEffectToTarget(AActor\* TargetActor, TSubclassOf<UGameplayEffect> GameplayEffectClass);  
  
 UFUNCTION(BlueprintCallable)  
 void OnOverlap(AActor\* TargetActor);  
  
 UFUNCTION(BlueprintCallable)  
 void OnEndOverlap(AActor\* TargetActor);

};

AuraEffectActor.h

Final code for both header and cpp should look like this.

void AAuraEffectActor::ApplyEffectToTarget(AActor\* TargetActor, TSubclassOf<UGameplayEffect> GameplayEffectClass)  
{  
 UAbilitySystemComponent\* TargetASC = UAbilitySystemBlueprintLibrary::GetAbilitySystemComponent(TargetActor);  
  
 if (TargetASC == nullptr) return;  
  
 check(GameplayEffectClass);  
 FGameplayEffectContextHandle EffectContextHandle = TargetASC->MakeEffectContext();  
 EffectContextHandle.AddSourceObject(this);  
 const FGameplayEffectSpecHandle EffectSpecHandle = TargetASC->MakeOutgoingSpec(  
 GameplayEffectClass, ActorLevel, EffectContextHandle);  
 const FActiveGameplayEffectHandle ActiveEffectHandle = TargetASC->ApplyGameplayEffectSpecToSelf(  
 \*EffectSpecHandle.Data.Get());  
  
 const bool bIsInfinite = EffectSpecHandle.Data.Get()->Def.Get()->DurationPolicy ==  
 EGameplayEffectDurationType::Infinite;  
 if (bIsInfinite && InfiniteEffectRemovalPolicy == EEffectRemovalPolicy::RemoveOnEndOverlap)  
 {  
 ActiveEffectHandles.Add(ActiveEffectHandle, TargetASC);  
 }  
}  
  
void AAuraEffectActor::OnOverlap(AActor\* TargetActor)  
{  
 if (InstantEffectApplicationPolicy == EEffectApplicationPolicy::ApplyOnOverlap)  
 {  
 ApplyEffectToTarget(TargetActor, InstantGameplayEffectClass);  
 }  
 if (DurationEffectApplicationPolicy == EEffectApplicationPolicy::ApplyOnOverlap)  
 {  
 ApplyEffectToTarget(TargetActor, DurationGameplayEffectClass);  
 }  
 if (InfiniteEffectApplicationPolicy == EEffectApplicationPolicy::ApplyOnOverlap)  
 {  
 ApplyEffectToTarget(TargetActor, InfiniteGameplayEffectClass);  
 }  
}  
  
void AAuraEffectActor::OnEndOverlap(AActor\* TargetActor)  
{  
 if (InstantEffectApplicationPolicy == EEffectApplicationPolicy::ApplyOnEndOverlap)  
 {  
 ApplyEffectToTarget(TargetActor, InstantGameplayEffectClass);  
 }  
 if (DurationEffectApplicationPolicy == EEffectApplicationPolicy::ApplyOnEndOverlap)  
 {  
 ApplyEffectToTarget(TargetActor, DurationGameplayEffectClass);  
 }  
 if (InfiniteEffectApplicationPolicy == EEffectApplicationPolicy::ApplyOnEndOverlap)  
 {  
 ApplyEffectToTarget(TargetActor, InfiniteGameplayEffectClass);  
 }  
 if (InfiniteEffectRemovalPolicy == EEffectRemovalPolicy::RemoveOnEndOverlap)  
 {  
 UAbilitySystemComponent\* TargetASC = UAbilitySystemBlueprintLibrary::GetAbilitySystemComponent(TargetActor);  
 if (!IsValid(TargetASC)) return;  
  
 TArray<FActiveGameplayEffectHandle> HandlesToRemove;  
 for (auto HandlePair : ActiveEffectHandles)  
 {  
 if (TargetASC == HandlePair.Value)  
 {  
 TargetASC->RemoveActiveGameplayEffect(HandlePair.Key, 1);  
 HandlesToRemove.Add(HandlePair.Key);  
 }  
 }  
 for(auto& Handle : HandlesToRemove)  
 {  
 ActiveEffectHandles.FindAndRemoveChecked(Handle);  
 }  
 }  
}

AuraEffectActor.cpp

Now, we need to clamp the attribute changes for some of the attributes such as health and mana. We have a place to do that which is PostGameplayEffectExecute(…) in UAttributeSet class. There is also one function which let us do something in pre attribute change but since further calculations are not taken into account and that function is only for clamping, post execute is recommended. We can also collect many information about target and source. So, we will set these properties in a struct that we create, FEffectProperties. These properties will be useful later in the course. After setting it, we clamp the attributes.

void UAuraAttributeSet::PostGameplayEffectExecute(const FGameplayEffectModCallbackData& Data)  
{  
 Super::PostGameplayEffectExecute(Data);  
  
 FEffectProperties Properties;  
 SetEffectProperties(Data, Properties);  
  
 if (Data.EvaluatedData.Attribute == GetHealthAttribute())  
 {  
 SetHealth(FMath::Clamp(GetHealth(), 0.f, GetMaxHealth()));  
 }  
  
 if (Data.EvaluatedData.Attribute == GetManaAttribute())  
 {  
 SetMana(FMath::Clamp(GetMana(), 0.f, GetMaxMana()));  
 }  
}

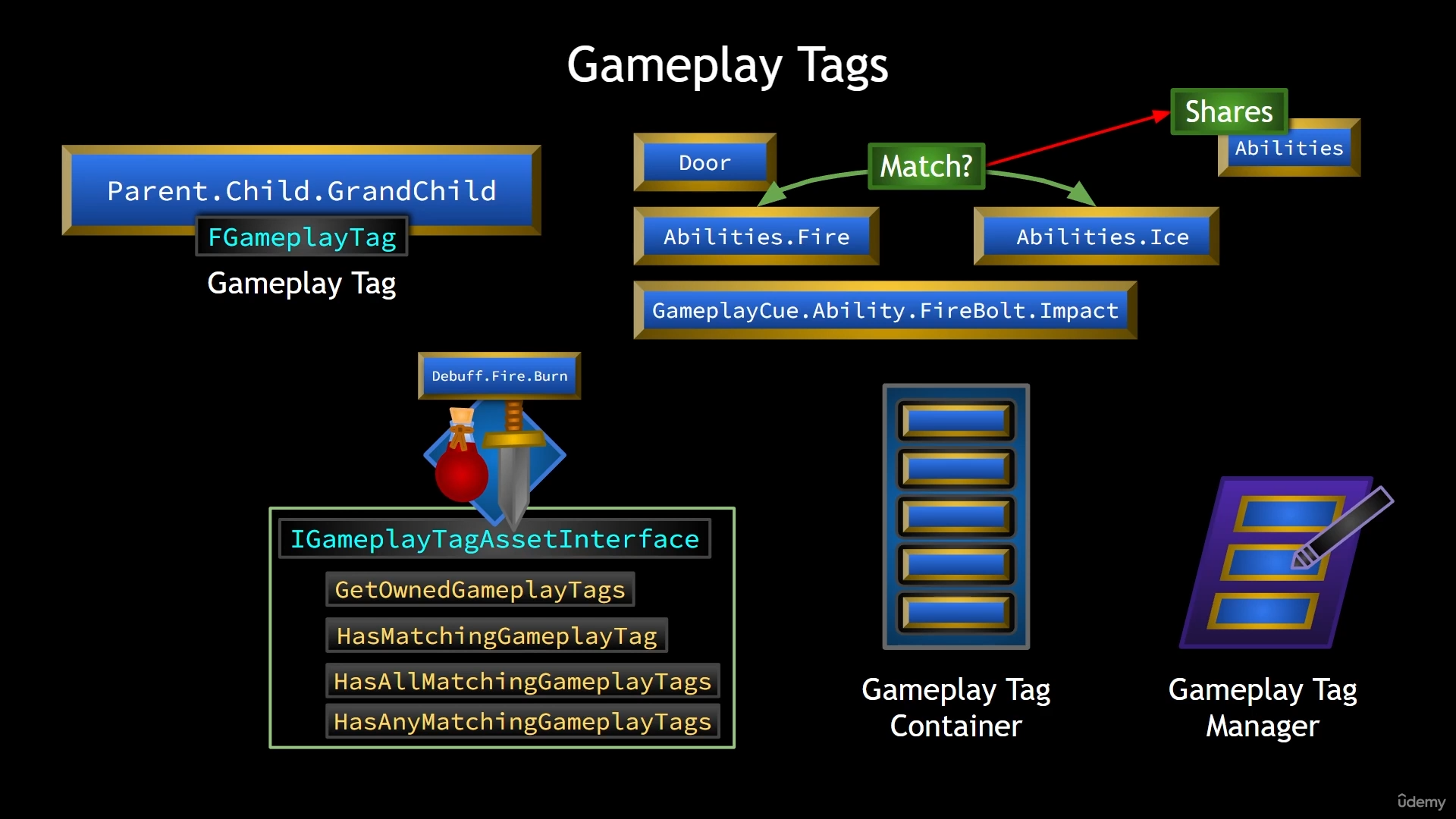
void UAuraAttributeSet::SetEffectProperties(const FGameplayEffectModCallbackData& Data,  
 FEffectProperties& Properties) const  
{  
 Properties.EffectContextHandle = Data.EffectSpec.GetContext();  
 Properties.SourceASC = Properties.EffectContextHandle.GetOriginalInstigatorAbilitySystemComponent();  
  
 if (IsValid(Properties.SourceASC) && Properties.SourceASC->AbilityActorInfo.IsValid() && Properties.SourceASC->  
 AbilityActorInfo->AvatarActor.  
 IsValid())  
 {  
 Properties.SourceAvatarActor = Properties.SourceASC->AbilityActorInfo->AvatarActor.Get();  
  
 Properties.SourceController = Properties.SourceASC->AbilityActorInfo->PlayerController.Get();  
 if (Properties.SourceController == nullptr && Properties.SourceAvatarActor != nullptr)  
 {  
 if (const APawn\* Pawn = Cast<APawn>(Properties.SourceAvatarActor))  
 {  
 Properties.SourceController = Pawn->GetController();  
 }  
 }  
 if (Properties.SourceController)  
 {  
 Properties.SourceCharacter = Cast<ACharacter>(Properties.SourceController->GetPawn());  
 }  
 }  
  
 if (Data.Target.AbilityActorInfo.IsValid() && Data.Target.AbilityActorInfo->AvatarActor.IsValid())  
 {  
 Properties.TargetAvatarActor = Data.Target.AbilityActorInfo->AvatarActor.Get();  
 Properties.TargetController = Data.Target.AbilityActorInfo->PlayerController.Get();  
 Properties.TargetCharacter = Cast<ACharacter>(Properties.TargetAvatarActor);  
 Properties.TargetASC = UAbilitySystemBlueprintLibrary::GetAbilitySystemComponent(Properties.TargetAvatarActor);  
 }  
}

USTRUCT()  
struct FEffectProperties  
{  
 GENERATED\_BODY()  
  
 FEffectProperties()  
 {  
 }  
  
 FGameplayEffectContextHandle EffectContextHandle;  
  
 UPROPERTY()  
 UAbilitySystemComponent\* SourceASC = nullptr;  
  
 UPROPERTY()  
 AActor\* SourceAvatarActor = nullptr;  
  
 UPROPERTY()  
 AController\* SourceController = nullptr;  
  
 UPROPERTY()  
 ACharacter\* SourceCharacter = nullptr;  
  
 UPROPERTY()  
 UAbilitySystemComponent\* TargetASC = nullptr;  
  
 UPROPERTY()  
 AActor\* TargetAvatarActor = nullptr;  
  
 UPROPERTY()  
 AController\* TargetController = nullptr;  
  
 UPROPERTY()  
 ACharacter\* TargetCharacter = nullptr;  
};

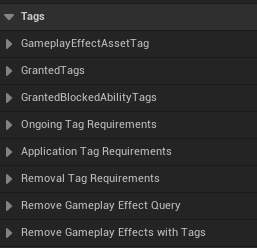
AuraAttributeSet.cpp

AuraAttributeSet.h

# Gameplay Tags

Gameplay Tags exist outside of GAS and can be used in non-GAS projects too. However in GAS, they are used extensively. Gameplay Tags are basically names. They have the type FGameplayTag and they’re registered with the Gameplay Tag Manager. They are hierarchical and each level is separated by a dot (i.e. Abilities, Abilities.Ranged, Abilities.Ranged.FireBolt). The ASC implements an interface called IGameplayTagAssetInterface. It has functionalities such as the ones in figure. Since classes can have multiple tags, they need to be stored. We don’t store them in TArrays, instead we use the GameplayTagContainer which has some gameplay tag specific functionality and some added efficiency. Gameplay tag containers have the concept of a tag map count, meaning that you can have more than one instance of a single tag in the container. It’s this map count that tells us how many of a given tag exists in the container and at times this could be zero depending on whether a tag has been added and then subsequently removed, for example.

GameplayTags can be added either in the Editor/ProjectSettings/GameplayTags or by creating a DataTable (choose GameplayTagTableRow as the row structure) and choosing it as a GameplayTagTableList in GameplayTags settings or in code as native tags. Tags will be added in DefaultGameplayTags.ini too and it can be directly added in here as well.

Tags can be added to GameplayEffects. GameplayEffectAssetTags are tags that the GE asset has them itself. GrantedTags are the ones that will be given to the actor affected by the GE (If the GE doesn’t stack, then we’ll get multiple effects applied meaning that we’ll get tags stacked. If the GE stacks, we only get one tag since there is a single effect being stacked). GrantedTags doesn’t mean anything on instant GE since it grants the tag for a duration that the actor has that effect. If we still want to know what type of effect we got (maybe for a message etc.) we use GameplayEffectAssetTag for that.

There are ways to know what type of effect has been applied. One of them is a delegate called OnGameplayEffectAppliedDelegateToSelf inherited from ASC and returns the delegate FOnGameplayEffectAppliedDelegate which has some parameters. We create a callback function. We need to bind our callback to the delegate in an appropriate place and most of the time it’s when the game starts, we can bind it in when our ability actor info is initialized, InitAbilityActorInfo. We can declare it in AuraCharacterBase and make it virtual and override it on both AuraCharacter and AuraEnemy and after that, we can bind our callback in InitAbilityActorInfo for all the characters using ASC. Since the delegate is not dynamic, we can’t use AddDynamic, we can use BindUObject and pass this as the first parameter.

We’ve created AbilityActorInfoSet(), called it in InitAbilityActorInfo on both character and enemy classes, and then bound EffectApplied(…) callback in that function.

void UAuraAbilitySystemComponent::AbilityActorInfoSet()  
{  
 OnGameplayEffectAppliedDelegateToSelf.AddUObject(this, &UAuraAbilitySystemComponent::EffectApplied);  
}  
  
void UAuraAbilitySystemComponent::EffectApplied(UAbilitySystemComponent\* AbilitySystemComponent,  
 const FGameplayEffectSpec& EffectSpec,  
 FActiveGameplayEffectHandle EffectHandle)  
{  
   
}

AuraAbilitySystemComponent.cpp

In EffectApplied(…), we want to receive asset tags and use them in UI. We are talking about GameplayEffectAssetTags, not GrantedTags. We get the reference of the EffectSpec carrying information of the GE applied in EffectApplied(…) callback function. We can use it to get its asset tags to store in a tag container. Then, we broadcast this container to the widget controllers. Then

AuraAbilitySystemComponent.h

void UAuraAbilitySystemComponent::EffectApplied(

UAbilitySystemComponent\* AbilitySystemComponent,  
const FGameplayEffectSpec& EffectSpec, FActiveGameplayEffectHandle EffectHandle)  
{  
 FGameplayTagContainer TagContainer;  
 EffectSpec.GetAllAssetTags(TagContainer);  
  
 EffectAssetTags.Broadcast(TagContainer);  
}

AuraAbilitySystemComponent.cpp

DECLARE\_MULTICAST\_DELEGATE\_OneParam(FEffectAssetTags, const FGameplayTagContainer& */\* Asset Tags \*/*);

UCLASS()  
class AURA\_API UAuraAbilitySystemComponent : public UAbilitySystemComponent  
{  
 GENERATED\_BODY()  
 .

.

.  
  
public:FEffectAssetTags EffectAssetTags;

};

We can use a DataTable to look up the tag and get some information to broadcast to the widgets. We can create the structure of that DataTable in OverlayWidgetController. Then we create a pointer for that data table. After that, we look up for the tag (we create message tags for UI and a function that finds row by tag in a data table) and if find any, broadcast it to the widgets.

void UOverlayWidgetController::BindCallbacksToDependencies()  
{  
 .

.

.  
 Cast<UAuraAbilitySystemComponent>(AbilitySystemComponent)->EffectAssetTags.AddLambda(  
 [this](const FGameplayTagContainer& AssetTags)  
 {  
 for (const FGameplayTag& Tag : AssetTags)  
 {  
 FGameplayTag MessageTag = FGameplayTag::RequestGameplayTag(FName("Message"));  
 if (Tag.MatchesTag(MessageTag))  
 {  
 const FUIWidgetRow\* Row = GetDataTableRowByTag<FUIWidgetRow>(MessageWidgetDataTable, Tag);  
 MessageWidgetRowDelegate.Broadcast(\*Row);  
 }  
 }  
 }  
 );  
}

Now in WBP\_Overlay, we can assign an event to MessageWidgetRowDelegate in EventWidgetControllerSet and create a message widget when it broadcasts.

OverlayWidgetController.h

OverlayWidgetController.cpp

USTRUCT(BlueprintType)  
struct FUIWidgetRow : public FTableRowBase  
{  
 GENERATED\_BODY()  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly)  
 FGameplayTag MessageTag = FGameplayTag();  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly)  
 FText Message = FText();  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly)  
 TSubclassOf<class UAuraUserWidget> MessageWidget;  
  
 UPROPERTY(EditAnywhere, BlueprintReadOnly)  
 UTexture2D\* Image = nullptr;  
};

DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE\_OneParam(FMessageWidgetRowSignature, FUIWidgetRow, Row);

UCLASS(BlueprintType, Blueprintable)  
class AURA\_API UOverlayWidgetController : public UAuraWidgetController  
{  
 GENERATED\_BODY()  
  
protected:UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category ="WidgetData")  
 TObjectPtr<UDataTable> MessageWidgetDataTable;template <typename T>  
 T\* GetDataTableRowByTag(UDataTable\* DataTable, const FGameplayTag& Tag);

.

.

.

}

template <typename T>  
T\* UOverlayWidgetController::GetDataTableRowByTag(UDataTable\* DataTable, const FGameplayTag& Tag)  
{  
 return DataTable->FindRow<T>(Tag.GetTagName(), TEXT(""));  
}

# RPG Attributes

We want to initialize the default attributes with gameplay effects. We create 3 gameplay effects for default values of vital, primary and secondary attributes. Then create a function to initialize the default attributes.

void AAuraCharacterBase::ApplyEffectToSelf(TSubclassOf<UGameplayEffect> GameplayEffectClass, float Level) const  
{  
 check(IsValid(GetAbilitySystemComponent()));  
 check(GameplayEffectClass);  
 FGameplayEffectContextHandle ContextHandle = GetAbilitySystemComponent()->MakeEffectContext();  
 ContextHandle.AddSourceObject(this);  
 const FGameplayEffectSpecHandle SpecHandle = GetAbilitySystemComponent()->MakeOutgoingSpec(  
 GameplayEffectClass, Level, ContextHandle);  
 GetAbilitySystemComponent()->ApplyGameplayEffectSpecToTarget(\*SpecHandle.Data.Get(), GetAbilitySystemComponent());  
}  
  
void AAuraCharacterBase::InitializeDefaultAttributes() const  
{  
 ApplyEffectToSelf(DefaultPrimaryAttributes, 1.f);  
 ApplyEffectToSelf(DefaultSecondaryAttributes, 1.f);  
 ApplyEffectToSelf(DefaultVitalAttributes, 1.f);  
}

protected:  
 .

.

.

UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Attributes")  
 TSubclassOf<UGameplayEffect> DefaultVitalAttributes;  
  
 UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Attributes")  
 TSubclassOf<UGameplayEffect> DefaultPrimaryAttributes;  
  
 UPROPERTY(BlueprintReadOnly, EditAnywhere, Category = "Attributes")  
 TSubclassOf<UGameplayEffect> DefaultSecondaryAttributes;

.

.

.

void ApplyEffectToSelf(TSubclassOf<UGameplayEffect> GameplayEffectClass, float Level) const;  
 void InitializeDefaultAttributes() const;

AuraCharacterBase.cpp

AuraCharacterBase.h

The question is where do we call it. In this case, we want to initialize the default attributes for Aura. So, we can call it at the end of InitAbilityActorInfo() in AAuraCharacter. Later on, we will initialize for enemy characters too. Initializing default attributes can be done on server only. The reason is they are marked as replicated. However, it can be called on both server and client if required.

One way we initialize our secondary attributes (which are based on primary attributes) is to create an infinite GE and modify it with the type of Attribute Based.

We want to modify MaxHealth and MaxMana in a more complicated way. For this reason, we will use Modifier Magnitude Calculation (MMC). This is a custom class that we can use in our GE to do complex calculations for our attribute changes. We want characters have level and MaxHealth and MaxMana to depend on the level. For this, we will give PlayerState for players and AuraEnemy for enemies a level integer and create a combat interface which will return the level (will do more stuff in the future). We will also make the level replicated for only players in PlayerState because we’re concerned with checking the level on the server for AI.

UMMC\_MaxHealth::UMMC\_MaxHealth()  
{  
 VigorDef.AttributeToCapture = UAuraAttributeSet::GetVigorAttribute();  
 VigorDef.AttributeSource = EGameplayEffectAttributeCaptureSource::Target;  
 VigorDef.bSnapshot = false;  
  
 RelevantAttributesToCapture.Add(VigorDef);  
}  
  
float UMMC\_MaxHealth::CalculateBaseMagnitude\_Implementation(const FGameplayEffectSpec& Spec) const  
{  
 const FGameplayTagContainer\* SourceTags = Spec.CapturedSourceTags.GetAggregatedTags();  
 const FGameplayTagContainer\* TargetTags = Spec.CapturedTargetTags.GetAggregatedTags();  
  
 FAggregatorEvaluateParameters EvaluationParameters;  
 EvaluationParameters.SourceTags = SourceTags;  
 EvaluationParameters.TargetTags = TargetTags;  
  
 float Vigor = 0.f;  
 GetCapturedAttributeMagnitude(VigorDef, Spec, EvaluationParameters, Vigor);  
 Vigor = FMath::Max<float>(Vigor, 0.f);  
  
 ICombatInterface\* CombatInterface = Cast<ICombatInterface>(Spec.GetContext().GetSourceObject());  
 const int32 PlayerLevel = CombatInterface->GetPlayerLevel();  
  
 return 80.f + (2.5f \* Vigor) + (10.f \* PlayerLevel);  
}

MMC\_MaxHealth.cpp

We initialize our vital attributes (MaxHealth and MaxMana) in the same place where we initialize other default attributes, AAuraCharacterBase::InitializeDefaultAttributes(). We set them with an instant GE.

We created our Attribute Menu in the editor. We want to get the information about our attributes or respond to when an attribute changes. We structured our system into pieces where we have widgets, widget controllers and model. So, we need a widget controller for our attribute menu widget. Instead of having one dispatcher for each attribute, we will have one dispatcher for all attribute changes. For this reason, we create a structure called FAuraAttributeMenu which contains the information of gameplay tag, name, value and a description for the attribute. We will create a DataAsset, UAttributeInfo, to look attributes up by their gameplay tags. We need references for our gameplay tags. We can do this by RequestGameplayTag(…) as we saw, but there is a better approach for this. We can create native gameplay tags in C++. Once we check the tag, we can broadcast the change to the widgets. We are going to create it as a struct, FAuraGameplayTags in AuraGameplayTags. We make it static singleton. Then we will declare the tags and also a static function to initialize them.

FAuraGameplayTags FAuraGameplayTags::GameplayTags;  
  
void FAuraGameplayTags::InitializeNativeGameplayTags()  
{  
 */\*\*  
 \* Primary Attributes  
 \*/* GameplayTags.Attributes\_Primary\_Strength = UGameplayTagsManager::Get().AddNativeGameplayTag(  
 FName("Attributes.Primary.Strength"),  
 FString("Increases Physical Damage"));  
 .

.

.  
  
 */\*\*  
 \* Secondary Attributes  
 \*/* GameplayTags.Attributes\_Secondary\_Armor = UGameplayTagsManager::Get().AddNativeGameplayTag(  
 FName("Attributes.Secondary.Armor"),  
 FString("Reduces damage taken, improves Block Chance"));  
 .

.

.  
}

AuraGameplayTags.h

struct FAuraGameplayTags  
{  
public:  
 static const FAuraGameplayTags& Get() { return GameplayTags; }  
 static void InitializeNativeGameplayTags();  
  
 */\* Primary Attributes \*/* FGameplayTag Attributes\_Primary\_Strength;  
 .

.

.  
   
 */\* Secondary Attributes \*/* FGameplayTag Attributes\_Secondary\_Armor;  
 .

.

.   
  
private:  
 static FAuraGameplayTags GameplayTags;  
};

AuraGameplayTags.h

We can initialize it in our asset manager, so we create our own asset manager, AuraAssetManager. In our asset manager, since it is a singleton too, we basically return the asset manager by casting our asset manager class. Then we need to override a function called StartInitialLoading() to initialize our data asset created for gameplay tags.

UAuraAssetManager& UAuraAssetManager::Get()  
{  
 check(GEngine);  
  
 UAuraAssetManager\* AuraAssetManager = Cast<UAuraAssetManager>(GEngine->AssetManager);  
 return \*AuraAssetManager;  
}  
  
void UAuraAssetManager::StartInitialLoading()  
{  
 Super::StartInitialLoading();  
  
 FAuraGameplayTags::InitializeNativeGameplayTags();  
}

AuraGameplayTags.h

public:  
 static UAuraAssetManager& Get();  
  
protected:  
 virtual void StartInitialLoading() override;

AuraAssetManager.h

In order to make our asset manager as default, we need to find DefaultEngine.ini placed in [ProjectName]/Config, find the section [/Script/Engine.Engine] and add the following line (change if there is one): AssetManagerClassName=/Script/Aura.AuraAssetManager

Here is how we set up our AttributeInfo data asset (FText is used to show a text in a widget):

USTRUCT(BlueprintType)  
struct FAuraAttributeInfo  
{  
 GENERATED\_BODY()  
  
 UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)  
 FGameplayTag AttributeTag = FGameplayTag();  
  
 UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)  
 FText AttributeName = FText();  
  
 UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)  
 FText AttributeDescription = FText();  
  
 UPROPERTY(BlueprintReadOnly)  
 float AttributeValue = 0.f;  
};  
  
*/\*\*  
 \*   
 \*/*UCLASS()  
class AURA\_API UAttributeInfo : public UDataAsset  
{  
 GENERATED\_BODY()  
  
public:  
 */\*\*  
 \* Variables  
 \*/* UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (TitleProperty = "{AttributeName}"))  
 TArray<FAuraAttributeInfo> AttributeInformation;  
  
 */\*\*  
 \* Functions  
 \*/* FAuraAttributeInfo FindAttributeInfoForTag(const FGameplayTag& AttributeTag, bool bLogNotFound = false) const;  
};

FAuraAttributeInfo UAttributeInfo::FindAttributeInfoForTag(const FGameplayTag& AttributeTag, bool bLogNotFound) const  
{  
 for(const FAuraAttributeInfo& Info : AttributeInformation)  
 {  
 if(Info.AttributeTag.MatchesTagExact(AttributeTag))  
 {  
 return Info;  
 }  
 }  
  
 if(bLogNotFound)  
 {  
 UE\_LOG(LogTemp, Error, TEXT("Can't find Info for Attribute Tag [%s] on AttributeInfo [%s]."), \*AttributeTag.ToString(), \*GetNameSafe(this));  
 }  
  
 return FAuraAttributeInfo();  
}

AttributeInfo.cpp

AttributeInfo.h

Now we can create a DataAsset based on AttributeInfo in the editor and add our attributes in it.

In some places, such as attribute menu, we may want to access the AttributeMenuWidgetController without getting HUD etc. In this case, we can use Blueprint Function Library. We create our own BFL for AS and create two functions that gets us the two widget controllers that we’ve created.

UAttributeMenuWidgetController\* UAuraAbilitySystemLibrary::GetAttributeMenuWidgetController(  
 const UObject\* WorldContextObject)  
{  
 if (APlayerController\* PC = UGameplayStatics::GetPlayerController(WorldContextObject, 0))  
 {  
 if (AAuraHUD\* AuraHUD = Cast<AAuraHUD>(PC->GetHUD()))  
 {  
 AAuraPlayerState\* PS = PC->GetPlayerState<AAuraPlayerState>();  
 UAbilitySystemComponent\* ASC = PS->GetAbilitySystemComponent();  
 UAttributeSet\* AS = PS->GetAttributeSet();  
 const FWidgetControllerParams WidgetControllerParams(PC, PS, ASC, AS);  
 return AuraHUD->GetAttributeMenuWidgetController(WidgetControllerParams);  
 }  
 }  
 return nullptr;  
}

AuraAbilitySystemLibrary.cpp

public:  
 UFUNCTION(BlueprintPure, Category = "AuraAbilitySystemLibrary|WidgetController")  
 static UOverlayWidgetController\* GetOverlayWidgetController(const UObject\* WorldContextObject);  
  
 UFUNCTION(BlueprintPure, Category = "AuraAbilitySystemLibrary|WidgetController")  
 static UAttributeMenuWidgetController\* GetAttributeMenuWidgetController(const UObject\* WorldContextObject);

AuraAbilitySystemLibrary.h

(Same for the OverlayWidgetController)

We can create AttributeMenuWidgetController in the same way we did OverlayWidgetController. In the attribute menu widget, we can simply get the controller by calling its static function from our BFL.

We want to see the changes in our AttributeMenuWidgetController. In our widget controller class, we get the array from AttributeInfo data asset, get it’s attribute getter and bind it to ASC’s attribute change delegate GetGameplayAttributeValueChangeDelegate(…). In lambda function,