Unreal Engine 5 Gameplay Ability System

A screenshot of a video game

Description automatically generated

## Framework for building abilities and attributes

* Actions
* Passives
* Status effects
  + Build-up
  + Wear down
* Cooldowns
* Resource costs
* Ability level
* Bonus effects
* Activate particle, SFX etc

And more

## Eg RPG:

* Attributes
  + Life
  + Damage
  + Mana
  + Effects of Magic armor/weapons/spells
* Attributes might depend on others
* Crit chance
  + Add chance to cause bonus damage
  + Maybe derived?
    - Resilience increases and therefore crit dows as well
* Abilities
  + Attack
    - Play Animation
    - Cause Damage
      * Hit target
      * Check values of attributes on attacker and target
      * Calculate based on the attributes
      * Damage fatal?
        + Health to zero?

Rewards: XP

Level up?

Increase in power

Attribute values

New abilities

More damage

Better rewards

Money

Drops

A screenshot of a video game

Description automatically generated

## Mechanics and Effects on players

* Buffs/Debuffs
  + Potion?
  + Spells
  + Items
  + Hit by abilities
  + Receive damage of type
  + Attributes ++ or --
* Attributes have own values
  + Associated animation or particle system
  + Associated audio effects
* Used in other genres
  + Progression system in many games
* More complex game becomes, the more attributes may interact
  + Added complexity in managing interaction and in design balance
* GAS is system to help organise the interactions
  + Not \*needed\* but is super helpful in implementation design!
  + Designed for Paragon (MOBA)
    - MOBAs are very complex and need a manageable framework
      * Multiplayer
    - Paragon cancelled but GAS still present
  + Fully replicated
    - (Multiplayer and lag compensated)

A screenshot of a video game

Description automatically generated

## Main Components of GAS

* Ability system component
  + Grant ability
  + Activating ability
  + Handle notifs for activation or effects applied
* Attributes set
  + Life + mana
  + Stored on attributes set
  + Attributes can be associated with other parts of GAS
* Abilities
  + Gameplay Abilities
    - Classes that encapsulate things you can do
      * Attack
      * Cast spells
      * Gather resources
    - Allows code and functionality to be kept inside a specific Gameplay ability class
* Ability Task
  + Asynchronous tasks
  + Allow code to be executed asynchronous
  + They do the work for the ability itself
* Gameplay Effects
  + Used to change values of attributes
  + Change attribute directly
    - Overtime
    - Increse/decrease periodically
* Gameplay Cues
  + Handle sounds, particles, and effects
  + Handle replication of effects
* Gameplay Tags
  + Not unique to GAS
  + Used heavily in GAS
  + Hierarchical
    - More versatile than variables

A screenshot of a game menu

Description automatically generated

## Ability system Component

Adding:

* Pawn
  + Eg a character
    - Add direct to pawn class
    - Same attribute set
* Associated class
  + Eg Player State
    - Add ability system component and attribute set to that class

A screenshot of a computer game

Description automatically generated

Why to add to Pawn directly:

* When pawn destroyed
  + Attributes and abilities are destroyed and respawn resets them to defaults

Why add to player state:

* Pawn destroyed does not destroy the attribute and abilities so respawn allows values to be associated with the newly created pawn
* Or you can change player state on the pawn and modify attributes and abilities to be swapped out for another character
* Maybe keep some things associated with the player regardless of pawns
* For NPCs it might be appropriate to keep GAS directly on the pawn or it parent

A screenshot of a computer

Description automatically generated

For this project:

NPC GAS on the pawns, but Player GAS on the player state

A screenshot of a computer game

Description automatically generated

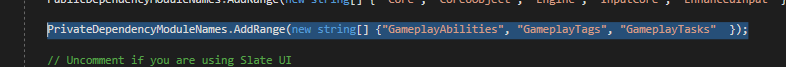
(Allows experience with both approaches)

## First Steps:

* Player State Class
* Ability System Component
* Attribute Set

Player State Class

Turn on GAS module

Add in build file: 

Create classes for ability and attributes

## GAS in multiplayer

Multiplayer always has a server

Server is an instance of the game running separately and independently of the other machines running their own instances of the game (clients)

A computer screen with text and a diagram

Description automatically generated with medium confidence

Server holds:

Game Mode

Player controllers for all players

Player states for all players

Clients hold:

Player controller for their local player only

Player states for all players

A screen shot of a computer

Description automatically generated

Pawns

A screen shot of a computer

Description automatically generated

Clients need to see other player’s pawns, and the pawns need to be able to access the player state to function. However, the server is still the authority

A computer screen shot of a computer

Description automatically generated

HUDs and widgets: each client has it’s own interface classes and the classes exist only on that client machine

Dedicated server has not HUD at all

Listen server has only the HUD for the player who’s machine is the listen server

Player controller has been set to be replicated:

A screen shot of a computer

Description automatically generated

Player state has been set to have a net update frequency of 100:

A screen shot of a computer

Description automatically generated

As changes are made on the server, the server sends data to the client 100x a second – replication

Replication:

Pawn has a variable Variable, so Variable exists everywhere the pawn exists:

A computer screen shot of a computer

Description automatically generated

If it is a replicated variable, then any changes to the variable on the server will be sent down to the clients at the next net update so the client version can be updated to new value:

A computer screen shot of a computer

Description automatically generated

Variable is changed at the client side:

A computer screen shot of a computer

Description automatically generated

The update from the client is \*NOT\* replicated to the server, replication works only one way! Server to client

Data is updated on the server from the client in the form of RPCs, Remote Procedure Calls

Workflow:

Turn on GAS module and reboot

Check that “GameplayAbilities", "GameplayTags", "GameplayTasks" are added to the build

Create classes for attribute set and ability system component

For character classes that will have abilities:

Cpp:

UAbilitySystemComponent\* AAuraCharacterBase::GetAbilitySystemComponent() const

{

return AbilitySystemComponent;

}

For header:

#include "AbilitySystemInterface.h"

#include "AuraCharacterBase.generated.h"

class UAbilitySystemComponent;

class UAttributeSet;

UCLASS(Abstract)

class AURA\_API AAuraCharacterBase : public ACharacter, public IAbilitySystemInterface

{

GENERATED\_BODY()

public:

AAuraCharacterBase();

virtual UAbilitySystemComponent\* GetAbilitySystemComponent() const override;

UAttributeSet\* GetAttributeSet() const { return AttributeSet; }

protected:

UPROPERTY()

TObjectPtr<UAbilitySystemComponent> AbilitySystemComponent;

UPROPERTY()

TObjectPtr<UAttributeSet> AttributeSet;

};

For child classes constructor:

AAuraEnemy::AAuraEnemy()

{

GetMesh()->SetCollisionResponseToChannel(ECC\_Visibility, ECR\_Block);

AbilitySystemComponent = CreateDefaultSubobject<UAuraAbilitySystemComponent>("AbilitySystemComponent");

AbilitySystemComponent->SetIsReplicated(true);

AttributeSet = CreateDefaultSubobject<UAuraAttributeSet>("AttributeSet");

}

For player state that requires ability system (header):

#include "AbilitySystemInterface.h"

class UAbilitySystemComponent;

class UAttributeSet;

/\*\*

\*

\*/

UCLASS()

class AURA\_API AAuraPlayerState : public APlayerState, public IAbilitySystemInterface

{

GENERATED\_BODY()

public:

AAuraPlayerState();

virtual UAbilitySystemComponent\* GetAbilitySystemComponent() const override;

UAttributeSet\* GetAttributeSet() const { return AttributeSet; }

protected:

UPROPERTY()

TObjectPtr<UAbilitySystemComponent> AbilitySystemComponent;

UPROPERTY()

TObjectPtr<UAttributeSet> AttributeSet;

};

Cpp:

#include "AbilitySystem/AuraAttributeSet.h"

#include "AbilitySystem/AuraAbilitySystemComponent.h"

AAuraPlayerState::AAuraPlayerState()

{

AbilitySystemComponent = CreateDefaultSubobject<UAuraAbilitySystemComponent>("AbilitySystemComponent");

AbilitySystemComponent->SetIsReplicated(true);

AttributeSet = CreateDefaultSubobject<UAuraAttributeSet>("AttributeSet");

}

UAbilitySystemComponent\* AAuraPlayerState::GetAbilitySystemComponent() const

{

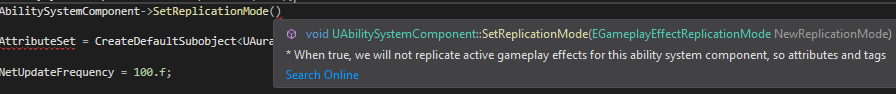
return AbilitySystemComponent;

}

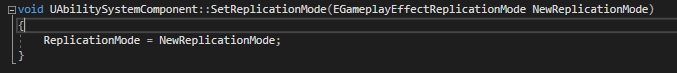
## Replication mode

AbiltySystemComponent needs to be set to replicated = true

AbilitySystemComponent->SetReplicationMode()



SetReplicationMode takes an Enum



A black screen with green text

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Replication Mode | Use Case | Description |
| Full | Single Player | Gameplay Effects are replicated to all clients |
| Mixed | Multiplayer, Player-controlled | Gameplay Effects are replicated to the owning client only. Gameplay Cues and Tags are replicated to all clients |
| Minimal | Multiplayer, AI Controlled | Gameplay Effects are NOT replicated, Gameplay Cues and Tages are replicated to all clients |

Mixed: Gameplay effects are replicated to owning client only; Effects are used to change attribute values. That kind of change happening on server SHOULD be passed to the client eg health loss.

Minimal: good for AI controlled characters

Programming for multiplayer works just as well for single player, the reverse is not true

In Enemy.cpp:

A screen shot of a computer

Description automatically generated

In Playerstate:



NB: at this point the pointers in the Aura character base class still has uninitialized pointers! These are set with valid values on Enemy, but the character does not have these initialised, because they are given values in the player state instead. This should be fixed!

A screen shot of a computer code

Description automatically generated

Also, the ability system component needs to know the identity of it’s owner – could be very different for enemy and character. For enemy, it’s obviously enemy, for character, not necessarily the case because the player state constructs the component!

AbilitySystemComponent has an ability : Ability ActorInfo

The component can always know info like owner

But component also understands it can be owned by an actor pawn or another entity like a playerstate

Therefore AbilitySystemComponent has 2 variables:

Owner Actor and Avatar Actor (both of type AActor)

Owner Actor is whatever class owns the component

The Avatar is the representation in world associated with the component. For Enemy actor, Owner and Avatar are the same

For player controlled character, ASC is constructed by the state and is therefore the owner, and avatar is the character.

A screenshot of a computer game

Description automatically generated

ASC distinguishes between the two!

We decide this: by calling function InitAbilityActorInfo(AActor\* InOwnerActor, AActor\* InAvatarActor); from UAbilitySystemComponent namespace

When is this called?

## Init Ability Actor Info:

* Must be done AFTER controller is set (after possession)
* For player: If ASC lived on pawn (it does NOT in Aura)
* On PossessedBy fn
* Ensures ASC initialised with Avatar and Owner on the server
* Only applies on server
* On client there’s a fn called AcknowledgePossession
* We know pawn has valid controller

A screenshot of a computer

Description automatically generated

* For player: if ASC is on Playerstate (Aura):
* On PossessedBy for the server
* For client : OnRep\_PlayerState
* (on replicated player state)
* ASC lives on playerstate so we need to validate controller has been set AND that player state is valid
* OnRep\_PlayerState is a “rep notify”
* Fn called as a result of something being replicated
* Player state is set on server
* Player state is replicated entity
* Therefore as soon as state set for pawn, it’ll be replicated down, triggering Rep Notif
* Ref Notif called in response to replication happening
* Pointer is now valid
* Therefore THIS is where we initialise the pointer for ASC
* Set Owner actor to playerstate
* Set Avatar to the aura character (pawn)

A screenshot of a computer

Description automatically generated

AI-characters:

* Since the AI characters have ASC on pawn
* We know AI has valid component and controller
* Call InitAbilityActorInfo can be called in BeginPlay

So, for Enemy:

In header:

protected:

virtual void BeginPlay() override;

in cpp:

void AAuraEnemy::BeginPlay()

{

Super::BeginPlay();

AbilitySystemComponent->InitAbilityActorInfo(this, this);

}

Pointer to AbilitySystemComponent already exists, so we call the Init from that with self-references to this class

For player:

In public: override PossessedBy()

From character.cpp:

void ACharacter::PossessedBy(AController\* NewController)

{

Super::PossessedBy(NewController);

// If we are controlled remotely, set animation timing to be driven by client's network updates. So timing and events remain in sync.

if (Mesh && IsReplicatingMovement() && (GetRemoteRole() == ROLE\_AutonomousProxy && GetNetConnection() != nullptr))

{

Mesh->bOnlyAllowAutonomousTickPose = true;

}

}

In header:

public:

/\*begin interface\*/

virtual void PossessedBy(AController\* NewController) override;

virtual void OnRep\_PlayerState() override;

/\*end interface\*/

In cpp:

In void AAuraCharacter:: PossessedBy(AController\* NewController)

Super::PossessedBy(NewController);

Called GetPlayerState

To determine which instance, it can be templated, so

GetPlayerState<AAuraPlayerState>();

This will return the appropriate type

We can make a local AuraPlayerState variable

AAuraPlayerState\* AuraPlayerState = GetPlayerState<AAuraPlayerState>();

So, pointer to the templated state call, named AuraPlayerState, of type AuraPlayerState

Test for validity:

AAuraPlayerState\* AuraPlayerState = GetPlayerState<AAuraPlayerState>();

check(AuraPlayerState);

Now we have the local variable we can use it:

AuraPlayerState->GetAbilitySystemComponent()->InitAbilityActorInfo(AActor\* InOwnerActor, AActor\* InAvatarActor);

For the owner actor we pass in AuraPlayerState, for the Avatar it’s this

Therefore, the fn is now:

void AAuraCharacter::PossessedBy(AController\* NewController)

{

Super::PossessedBy(NewController);

//init ability actor info for server

AAuraPlayerState\* AuraPlayerState = GetPlayerState<AAuraPlayerState>();

check(AuraPlayerState);

AuraPlayerState->GetAbilitySystemComponent()->InitAbilityActorInfo(AuraPlayerState, this);

}

We set this in possessedby so we can be sure that we have a controller set already and that the playerstate is valid and accessible and we can call the function.

Now we have the player state in memory, good opportunity to sent the abilitysystemcomponent pointer locally!

AbilitySystemComponent = AuraPlayerState->GetAbilitySystemComponent();

void AAuraCharacter::PossessedBy(AController\* NewController)

{

Super::PossessedBy(NewController);

//init ability actor info for server

AAuraPlayerState\* AuraPlayerState = GetPlayerState<AAuraPlayerState>();

check(AuraPlayerState);

AuraPlayerState->GetAbilitySystemComponent()->InitAbilityActorInfo(AuraPlayerState, this);

AbilitySystemComponent = AuraPlayerState->GetAbilitySystemComponent();

AttributeSet = AuraPlayerState->GetAttributeSet();

}

This takes care of the server side, but the client needs to be done as well!

void AAuraCharacter::OnRep\_PlayerState()

{

Super::OnRep\_PlayerState();

}

It’s the same thing, get playerstate, check it call initability actor info and set pointers

However, better not to repeat code! Create a function to reuse in both!

## Mixed Replication Mode

OwnerActor’s Owner must be the controller. For pawns, this is set automatically in PossessedBy()

The PlayerStates Owner is automatically set to the controller

Therefore, if your OwnerActor is NOT the playerstate, and you used Mixed replication mode, you must call SetOwner() on the OwnerActor to set it owner to the controller



# Attributes

When the attribute set is constructed alongside the ability system component, inside the owner actors constructor, it’s auto-registered with the ability system, and the ability system has access to it, and to any other attribute sets are registered with it. Multiple attribute sets are possible!

Eg, if you want attributes distributed among sets based on category eg Primary, secondary, tertiary like in WoW, but each set must be separate classes to avoid ambiguity when retrieving the set from the system component,

A screenshot of a computer

Description automatically generated

OTOH it’s OK to include all attributes in the same set, simpler if the attributes need to know the values of other attributes when making calculations

Attributes take up negligible memory so it’s OK to share a single attribute set among all classes and only use what you need in a given class

A screenshot of a computer

Description automatically generated

## What are attributes

* Numerical quantities associated with a given entity, like a character
* All are floats
* Exist within a structure called FGameplayAttributeData
* Stored on attribute set
* We can trace changes and respond with functionality
* Can be set directly in code
* PREFERRED way to change attributes is by applying Gameplay Effects
* Gameplay effects allow prediction of changes to attributes
* Prediction means the client does not need to wait for server permission to change a value
* Value can change immediately on client and server is informed.
* Server can rollback invalid changes if necessary

A screenshot of a video game

Description automatically generated

### Clientside change without prediction

* Clientside value needs to change
* Server should be in charge of important changes (otherwise clients could cheat)
* Client request value change to server
* Server receives request, checks validity, send confirmation if valid, client updates
* Causes noticeable delay between client needing to change value and actually changing it!
* Affected significantly by lag
* Bad gameplay experience!

A screenshot of a computer

Description automatically generated

### Clientside change with prediction

* Gameplay effect modifies value clientside: value change is instant, no lag
* Client sends to server
* Server validates, approves and informs all other clients
* If change is INVALID, for example cheating, server can reject change and rollback, setting value to correct on clientside at the next netupdate
* Having prediction built-in to GAS saves a lot of work, because prediction is complicated!

A screenshot of a computer

Description automatically generated

Attributes:

* Object of type FGameplayAttributeData, stored on Attribute Set
* Consist of 2 values, Base and Current
* Base is permanent value
* Current is Base + temp effects from Gameplay Effects
* Base value is not the max value!
* Max value is separate from the base
* Max value is it’s own attribute, allowing effects to be applied to both the current and max values separately
* Healthbar would not be Current Value/Base value
* It would be Health/MaxHealth

Creating an attribute in code:

In h create a public sector and a constructor.

Declare attributes as objects of type FGameplayAttributeData in .h public

Attributes should be a UPROPERTY and replicatable – add Replicated specifier in the UPROPERTY



We also need to use a Rep\_Notify – called automatically when server notif arrives

This is done by changing the UPROPERTY to ReplicatedUsing = <name of rep\_notif>

Convention is to name the notifier functions OnRep\_<variable name>



Type is void, but for it to be a Rep\_notif it needs to be a UFUNCTION

A black screen with white text

Description automatically generated

Can take no arguments, or one argument. Argument has to be of type of the replicated variable, in this case of type FGameplayAttributeData

Can be a constreference of the variable: when it is called it will get the \*old\* value passed in as an input parameter – useful for comparing old to new value

A black background with white text

Description automatically generated A screen shot of a computer

Description automatically generated

When Health replicates it will be equal to the new replicated value, but the old value is still known because it is passed into the function

When we set upa rep notif for an attribute the ability system has to be informed of the camge so it can perform book keeping to maintain functionality

Use macro GAMEPLAYATTRIBUTE\_REPNOTIFY in the rep\_notif function

This requires the class name as the first argument, then the name of replcated attribute, followed by old value

Requires #include "AbilitySystemComponent.h"

This cintion can be made a const, and that’s useful for what it is, so const the declaration and the definition

A black background with white text

Description automatically generated

A screen shot of a computer

Description automatically generated

Knowing the OldHealth allows the server to rollback if there is an invalid change

To make a variable replicated there’s one more function required:

GetLifetimeReplicatedProps

Signature:



Definition:

It’s an override, so we need to Super the parent:

A screen shot of a computer

Description automatically generated

This function is where we register variables for replication

To register the variables we use fn DOREPLIFETIME functions, in this case DOREPLIFETIME\_CONDITION\_NOTIFY();

Requires #include "Net/UnrealNetwork.h"

Take classname, variable, a condition under which the variable will replicate, and

DOREPLIFETIME\_CONDITION\_NOTIFY(c, v, cond, rn\_cond);



This line registers Health to be replicated (required for anything you want to replicate) without any conditions (there are other options, like you can only replicate to the owner for example) and Rep\_notify always means if the value is set on the server replicate it and on the client the value will be replicated and set.

For comparison, REPNOTIFY\_OnChanged is the default, and if the value of health is set on server and the value has not changed there’s no replication.

For this project, we’ll replicate anyway, because if we set value, we may want to always respond to the act of setting the value, whether it’s set to it’s own value or its own same value even if the numerical value has not changed eg healing at full health

There can be additional specifiers for the attributes, for example



A computer screen shot of a program code

Description automatically generated

# Attribute Accessor functions

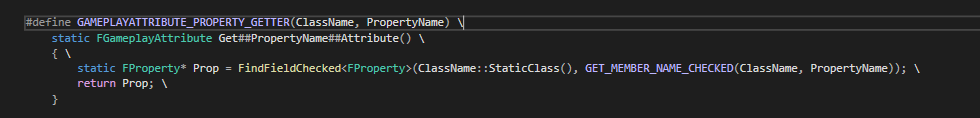
Accessor functions retrieve and set values directly from code (normally this is done by gameplay effects however)

Macros exist to mutate and access attributes.

Eg GAMEPLAYATTRIBUTE\_PROPERTY\_GETTER(<class>, <attribute>);



Definition in the AttributeSet.h:



A screenshot of a computer program

Description automatically generated

So, you can use the macro #define ATTRIBUTE\_ACCESSORS(ClassName, PropertyName) to trigger macros that create specific functions for your attribute, examples for Health below:

static FGameplayAttribute UMyHealthSet::GetHealthAttribute();

FORCEINLINE float UMyHealthSet::GetHealth() const;

FORCEINLINE void UMyHealthSet::SetHealth(float NewVal);

FORCEINLINE void UMyHealthSet::InitHealth(float NewVal);

The reason that we see GetHealthAttribute()and GetHealth() used is that GetHealthAttribute is it’s own type (FGameplayAttribute); GetHealth() returns a float. Therefore GetHealthAttribute() returns the object itself, the other returns the numerical value of health

Difference between SET and INIT macros:

A computer screen shot of a program

Description automatically generated

Set will set the numerical attribute base value; Init sets the base value AND the current value

Bringing in the definition of the macros, we now have

A computer screen shot of a program

Description automatically generated

We also need to move the #include "AbilitySystemComponent.h" from the cpp to the header

So, we can now use the accessor functions in code, for example to initialise Health at 100 in the constructor

A screenshot of a computer

Description automatically generated

No InitHealth function was created, but we used the attribute accessor macro to call the 4 macros, one of which creates the initor, which by default is InitHealth

In action:

A screenshot of a video game

Description automatically generated

A screenshot of a video game

Description automatically generated

In the above screenshot we can see the Avatar and Owner are set to the character and the PlayerState, correctly and that this is the server (authority) because there are no other client machines. This is from calling InitAbilityActorInfo

In Red we see the info is for the player character; we can cycle to other targets:

A screenshot of a video game

Description automatically generated

We also see Owned Tags and the 4 attributes from the AS; in this case all characters have Health 100 from the Init health but no other values have been set! Useful for debugging

We can use the Accessor functions now, but in other places; for example the constructor is too early to use Set

## Actors that change attributes

Created new Actor class

Standard includes, in the private section creating a collision sphere and a static mesh

UPROPERTY(VisibleAnywhere)

TObjectPtr<USphereComponent> Sphere;

UPROPERTY(VisibleAnywhere)

TObjectPtr<UStaticMeshComponent> Mesh;

The sphere requires a forward declaration and they are added to the constructor with the Mesh as the root component with the sphere attached

Mesh = CreateDefaultSubobject<UStaticMeshComponent>("Mesh");

SetRootComponent(Mesh);

Sphere = CreateDefaultSubobject<USphereComponent>("Sphere");

Sphere->SetupAttachment(GetRootComponent());

An overlap function is added for the sphere

UFUNCTION()

virtual void OnOverlap(UPrimitiveComponent\* OverlappedComponent, AActor\* OtherActor, UPrimitiveComponent\* OtherComp, int32 OtherBodyIndex, bool bFromSweep,

const FHitResult& SweepResult);

and this is called in the BeginPlay

Sphere->OnComponentBeginOverlap.AddDynamic(this, &AAuraEffectActor::OnOverlap);

NB this has the AddDynamic as this is a dynamic multicast delegate

Note also the callback function reference &AAuraEffectActor::OnOverlap

This callback fn is fired in response to an overlap of the sphere. However, OnComponentBeginOverlap requires arguments to be passed from OnOverlap, here is how to find the required arguments:

OnComponentBeginOverlap Signature from owner PrimitiveComponents.h:

/\*\*

\* Event called when something starts to overlaps this component, for example a player walking into a trigger.

\* For events when objects have a blocking collision, for example a player hitting a wall, see 'Hit' events.

\*

\* @note Both this component and the other one must have GetGenerateOverlapEvents() set to true to generate overlap events.

\* @note When receiving an overlap from another object's movement, the directions of 'Hit.Normal' and 'Hit.ImpactNormal'

\* will be adjusted to indicate force from the other object against this object.

\*/

UPROPERTY(BlueprintAssignable, Category="Collision")

FComponentBeginOverlapSignature OnComponentBeginOverlap;

So class of type FComponentBeginOverlapSignature

The delegate definition for FComponentBeginOverlapSignature is defined by macro in PrimitiveComponents.h:

/\*\* Delegate for notification of start of overlap with a specific component \*/

DECLARE\_DYNAMIC\_MULTICAST\_SPARSE\_DELEGATE\_SixParams( FComponentBeginOverlapSignature, UPrimitiveComponent, OnComponentBeginOverlap, UPrimitiveComponent\*, OverlappedComponent, AActor\*, OtherActor, UPrimitiveComponent\*, OtherComp, int32, OtherBodyIndex, bool, bFromSweep, const FHitResult &, SweepResult);

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

Now to define what happens when overlap occurs. Change the value of an attribute. Ideally, this will be done with a Gameplay Effect, but for the sake of learning, we’re doing it in code.

Doing this is code is restrictive compared to Effects.

So, in the arguments passed into our overlaps we have the OtherActor, we can access the OtherActor’s Attributes.

Reminder: we included an interface in the classes earlier



PlayerState here is derived from IAbilityInterface, which includes the pure virtual function GetAbilitySystemComponent()

This means that we can take any actor, check if it implements this interface and if so we can get the ability system component that belongs to that actor regardless of the actor type. If it implements the interface we can call GetAbilitySystemComponent on it!

So, we create a local pointer equal to a cast to the other actor’s interface, then wrap it in an if

If(IAbilitySystemInterface\* ASCInterface = Cast<IAbilitySystemInterface>(OtherActor))

(required #include "AbilitySystemInterface.h )

And then define the attribute set

So, on ASCInterface call GetAbilitySystemComponent So far we have not implemented this on any actors that return a nullptr so will skip check

From here we call GetAttributeSet() (requires include of AbilitySystemComponent.h) and this function takes a TSubclassOf

ASCInterface->GetAbilitySystemComponent()->GetAttributeSet(UAuraAttributeSet::StaticClass());

This expression will return object of type UAttributeSet, therefore we cast to UAuraAttributeSet

Cast<UAuraAttributeSet>(ASCInterface->GetAbilitySystemComponent()->GetAttributeSet(UAuraAttributeSet::StaticClass()))

Then we store it in a local UAuraAttributeSet pointer called AuraAttributeSet

The pointer needs to be a const because the cast returns a const

const UAuraAttributeSet\* AuraAttributeSet = Cast<UAuraAttributeSet>(ASCInterface->GetAbilitySystemComponent()->GetAttributeSet(UAuraAttributeSet::StaticClass()));

We now have the other actors AuraAttributeSet and can use its accessor functions

A black screen with white text

Description automatically generated

Note that this errors:

A screen shot of a computer program

Description automatically generated

This is because the object is a const; it should either set it’s own values or change in response to gameplay effects!

For now, using a const\_cast to break encapsulation, This is dangerous!

A screenshot of a computer program

Description automatically generated

if (IAbilitySystemInterface\* ASCInterface = Cast<IAbilitySystemInterface>(OtherActor))

{

const UAuraAttributeSet\* AuraAttributeSet = Cast<UAuraAttributeSet>(ASCInterface->GetAbilitySystemComponent()->GetAttributeSet(UAuraAttributeSet::StaticClass()));

//here is the const cast! This is dangerous and should be changed!

UAuraAttributeSet\* MutableAuraAttributeSet = const\_cast<UAuraAttributeSet\*>(AuraAttributeSet);

MutableAuraAttributeSet->SetHealth(AuraAttributeSet->GetHealth() + 25.f);

//that was the const cast! Change it later!

Destroy();

}

So, this code creates a local pointer and casts to it the other actors AbilitySystemInterface, and then stores that ASI if it has one. It then creates a new local pointer to store the Attribute System of that other actor. The const\_cast strips encapsulation away from the Attribute set, allowing the const to be changed (even though it shouldn’t be because it’s a const!) and changes the value by +25

Then the actor destroys itself.

This is in effect a health potion

This code isn’t great, because it’s very very specific and only affects health, it would be better if this was more versatile!