Gameplay Abilities

Gameplay Abilities are actions or skills that an actor can perform

* Class derived from UGameplayAbility
* Defines what an ability does and the conditions under which it can be used
* Not implemented as a simple function; GAS game ability is an instanced object running asynchronously
* Can be activated at some point in time and run multi-stage tasks that may or may not span across a span of time
* These tasks could branch off into different tasks, apply their own gameplay effects, kick off **gameplay cues**
* Can be conditional, based on player or environmental interaction
* Gameplay abilities have built-in replication and prediction support
* Built in concept of cost and cooldown

A screenshot of a video game

Description automatically generated

## Ability Tasks

Gameplay abilities use ability tasks (although they don’t HAVE to)

* Ability tasks derived from UABILITYTASK
* Specialised form of UABILITYTASK
* Perform asynchronous work during a gameplay ability’s execution
* Can affect execution flow by broadcasting delegates
* Can be used in C++ or in BP through a specialised BP node
* BP node can have multiple output execution pins
* These are execute as a result of delegate broadcasts made in response to occurrences determined by developer – thing happens in game, task broadcasts delegate
* Can design the ability in the BP but make the actual work happen in code, very flexible

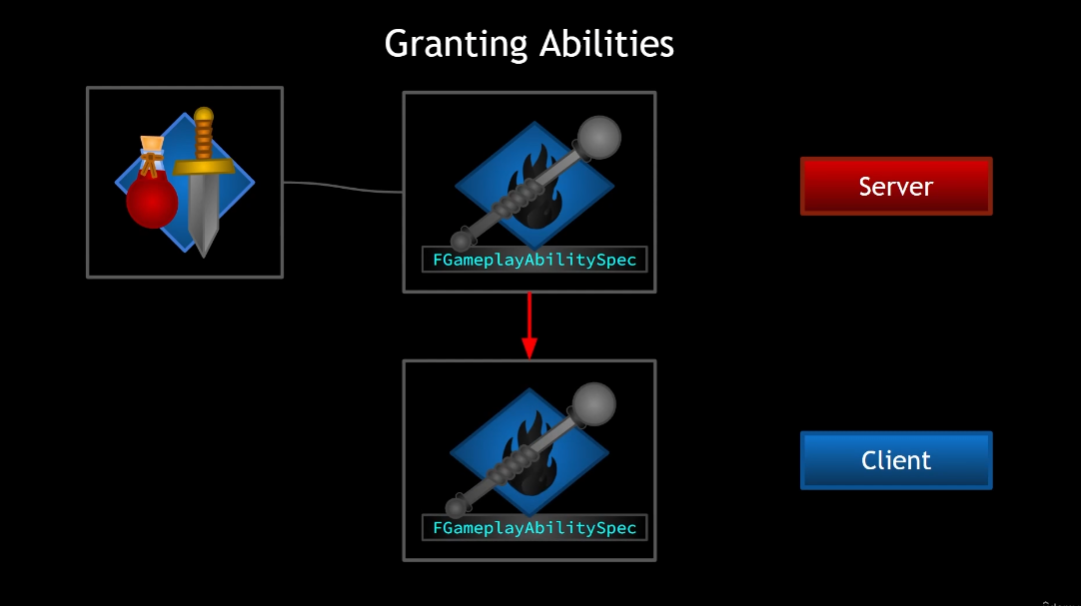
A screenshot of a computer

Description automatically generated

## Granting Abilities

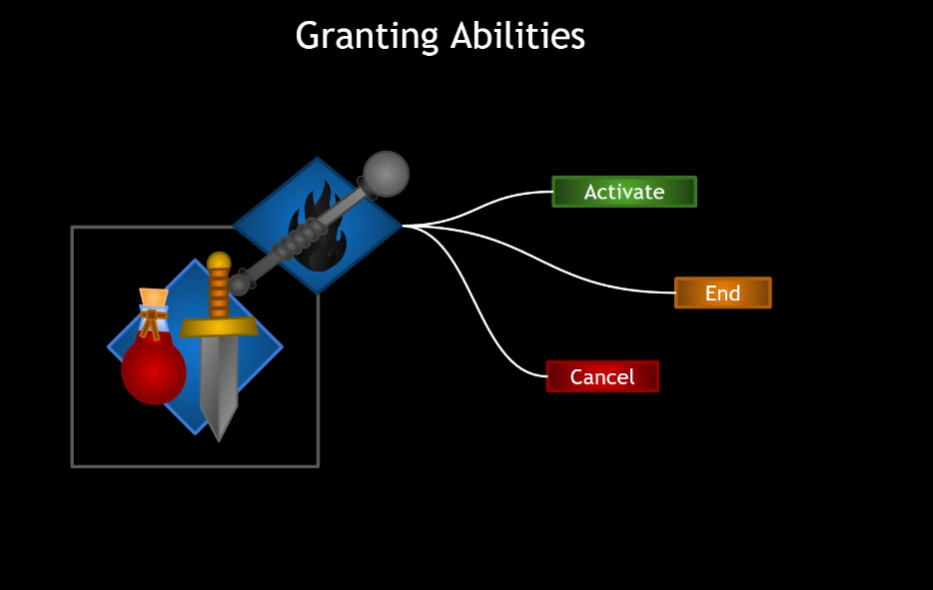
To use a gameplay ability the ASC must be *granted* the ability

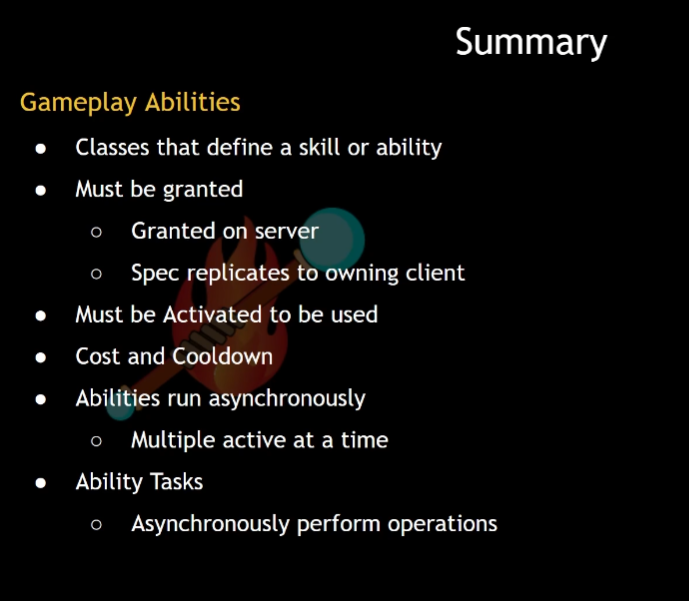
* GameplayAbilitySpec is created
* Spec defined details pertaining to that ability including
* GameplayAbility class itself
* Ability level
* Dynamic info that can be changed at run time
* Abilities granted on the server
* Ability spec then replicates to the owning client



Gameplay Abilities have the concept of *Activation*

* Once activated, gameplay abilities are *Active*
* Until Ended or Cancelled
* Abilities can end themselves or be cancelled exernally





# Granting Abilities

Possible: Make a blueprint based on the gameplay ability class

Instead: create a base class of our own! That way all GPA we create will have the option to have some baseline functionality

A screen shot of a computer

Description automatically generated

A black screen with orange and white text

Description automatically generated

Very basic template to start!

So, we want to grant abilities and this is done on the server

Let’s start by just giving the character a list of abilities that should be given at the start of the game

In AuraCharacterBase:

Add a TArray of GameplayAbility classes(type TSubclassOf)

Then we can set that TArray from within the character blueprint

A screen shot of a computer

Description automatically generated

So there will be an array of abilities, Now we need a way to \*give\* these abilities

This is done through the AbilitySystemComponent, so let’s add a function for adding those abilities

We may want to override in child classes so Protected

A screen shot of a computer

Description automatically generated

This function should only add abilities on the server, so we’ll do an auth check first

A screen shot of a computer

Description automatically generated

If we do have authority we want to:

Grant the ability; done by ASC, so let’s have the ASC actually do it; the CharacterBase will just call it

It should be public

A screen shot of a computer

Description automatically generated

The function will need to take in Gameplay Ability classes – this is the function we’ll call from CharacterBase

Since it’s a list of things we’ll loop through them

A screen shot of a computer

Description automatically generated

To add the abilities we need to create an ability spec from each of the ability classes



This is a struct that requires an ability class, so we’ll pass in AbilityClass

It also requires an ability level

A screen shot of a computer

Description automatically generated

Levels are just int32s. We’ll store the struct locally



And now we have the Spec, we can just use a function that exists on the ASC already called GiveAbility()



We could also give the ability and activate it immediately!

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

AbiltySpec \*cannot\* be const if we’re going to GiveAbilityAndActivateOnce!

A screen shot of a computer code

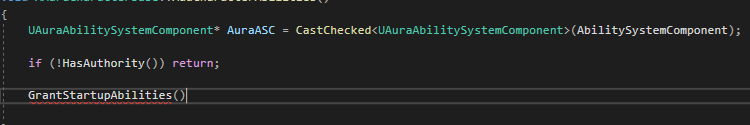
Description automatically generated

GiveAbility will accept either

For the moment we’re gong to use ActivateOnce, because as a learning exercise I want to see what happens when an ability is activated and this is the easiest option that I know so far

Last thing to do here is call the function in AuraCharacterBase

To do this we’ll need to cast to the ASC



Nb this requires an include for the AuraASC.h

A screen shot of a computer

Description automatically generated

So, where to call AddCharacterAbilities?

For the player character, a good place is in PossessedBy (worry about enemy later)

A screen shot of a computer program

Description automatically generated

A screen shot of a computer screen

Description automatically generated

Now we should see the ability added after InitAbilityActorInfo

We should expect, upon calling this function, to be granted all the abilities in the startup ability array

A screenshot of a computer

Description automatically generated

We can set the array of abilities here and there are some defaults, but I will make a new BP for the Abilities

A screenshot of a computer

Description automatically generated

It has Event Activate and Event OnEnd nodes

A screenshot of a cell phone

Description automatically generated

We can also use C++ functions, but we have events here

Testing:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

Now, pressing play should grant and activate this ability on press play

A purple and green object on a grey tile floor

Description automatically generated

Done, it’s active. But there’s no End Ability print string – Event OnEndAbility was never called!

Events can end themselves

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A close up of a purple object

Description automatically generated

# Settings on Gameplay Abilities

New GA\_Test has been created, and is based on the C++ implementation of the Gameplay Ability class

A black and white striped background

Description automatically generated

Lots of options for the class defaults here:

A screenshot of a computer

Description automatically generated

Tags:

You can add multiple tags directly to the ability from this page:

A screenshot of a computer

Description automatically generated

Each item has a mouseover that comes directly from the comments in the gameplay ability class in C++ and are self-explanatory

A black screen with green text

Description automatically generated

Tags explained in WoW terms

* Abilities have the concept of having tags, and they can be added here But why do we want tags?
* Cancel abilities with tag
* This ability will cancel other abilities that have a specified tag
* Ice Block is removed when you Blink
* Block abilities with tag
* When you use this ability, you cannot use OTHER abilities which have the specified tag
* Eg you cannot Fireball while you are in an IceBlock (wow mage)
* Activation Owned tags: Ability puts a tag on the owner, which can replicate - there’s a setting in Ability System Global to turn this on
* Icy Fingers wow? Build up secondary resources/combo points/holy power etc
* Activation Required tags: you can only use this if the owner has all the required tags
* Finishing move requires the secondary points to use/Ice lance needs icy fingers etc
* Ability is blocked if activating actor has tags
* Stealth if tagged as ‘In combat’?
* Debuff that blocks an ability like Forbearance blocks Divine Intervention – use with Activation Owned tag to put a cooldown on a target instead of the ability itself
* Source Required/Source Blocked: limitations on when an ability can be activated
* Target Required: eg can only envenom if target has poison
* Target Blocked: eg Cannot knockback target if target is rooted?

A screenshot of a computer

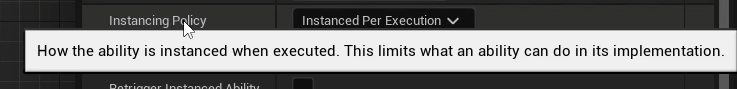
Description automatically generated

If you press input/receive input every frame that is sent to the server…not performance friendly? Encourages button mashing

A screenshot of a computer

Description automatically generated

If we’re using Local predicted do we need to use a replication policy? Its set to Do Not Replicate; should we turn it on, or not? No, gameplay abilities are automatically replicated by default, no need to change this.



A black and white screen with text

Description automatically generated

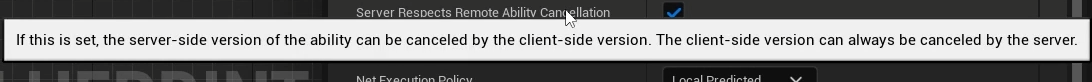
Gameplay abilities are classes that can be instanced.

Instanced per actor means we only make a single instance of the class upon activation. This one instance is reused every time it’s activated.

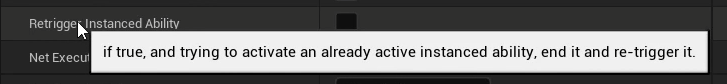
Since it’s instanced per actor, the actor is persistent, can store persistent variables and if you need to reset the variables you have to do it manually because the ability is never destroyed, it’s reused.

Instanced per execution: a new instance is created with each activation. Ability actor is created and destroyed over and over again. Does not store persistent data between activations because each activation creates a new instance of the class. Least performant option.

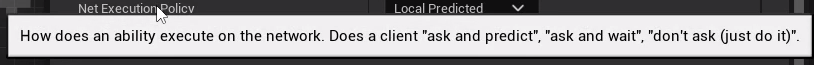
Non-instanced: class default object, no instances at all. Limited use, cannot store state, bind to delegates, never actually creates objects of this class. Like a static function library. Best performance.



“If you cancel this ability on the client, the server will be informed and will cancel it as well”



Self-explanatory, perhaps something like J-Boots from EQ?



A screenshot of a computer program

Description automatically generated

4 options

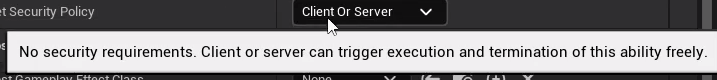
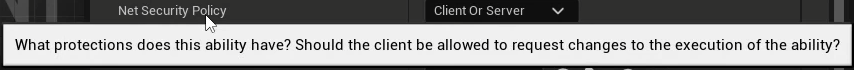
Local only: client only, server does not execute anything. Good for cosmetic only things that do not affect gameplay

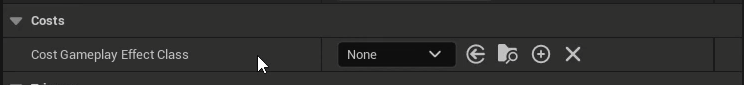
Local predicted: Takes time to travel to server, which can roll back invalid changes.

Server only: passive gameplay abilities you want running all the time/listening

Server initiated: runs on server then replicates.

In this project Local Predicted is fine for most abilities





Gameplay ability has a resource requirement directly related to attributes. Costs are implemented in the form of gameplay effects, so make a gameplay effect, select it here to connect them. Eg Mana, energy for WoW rogues, Breath for swimming under water etc

A screenshot of a computer

Description automatically generated

Allows the ability to be triggered in response to an event eg Reincarnation triggered in response to player death

Gameplay tag based:

A screenshot of a computer

Description automatically generated

Possible sources:

A screen shot of a black box

Description automatically generated

For example cleric resurrection

A black button with a black square

Description automatically generated

Gameplay effect based

A screenshot of a computer error

Description automatically generated

# Input Config Data Asset

Immediate activation works, but it would also be desirable to activate abilities via input

A screenshot of a computer program

Description automatically generated

To do: create a data asset that contains input actions but link up input actions with gameplay tags

At runtime we can assign various tags to gameplay abilities – as abilities are activated every ability is associated with the tag which can be checked and changed

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated

Target: an array of input actions, nothing too complicated, but the input actions need to be linked to gameplay tags

Easy way to do this is how Lyra does it, a struct with an input action and a gameplay tag

A screen shot of a computer program

Description automatically generated

Struct will contain an input action: will be a const and we’ll forward declare the class to avoid including the header file, and initialise as a null pointer

A screen shot of a computer

Description automatically generated

And a tag

A computer screen shot of a program code

Description automatically generated

Now that we have this struct our data asset can contain an array of these that we can set in the data asset blueprint

A screen shot of a computer

Description automatically generated

This array will be populated with input actions

A screen shot of a computer program

Description automatically generated

Specifically AuraInputActions that have a gameplay tag

If we have a pointer or reference to this input config we can look up any actions that have a specific tag

Function to do this:



Why return an InputAction, not an AuraInputAction though? Because this will be more useful if we return just the InputAction associated with a gameplay tag, and so the struct is an internal element to the input config; we don’t have to worry about breaking it open and finding the action inside; we’ll do it here



This will return the input action associated with a given tag

Loop through the Ability Input Actions by const reference

For each action we check the association is not null and also if there’s an exact match to the given tag

Then



There’s also the bool for if we don’t find anything

A black background with white text

Description automatically generated

A black background with text

Description automatically generated with medium confidence

And finally since we have to return a pointer, we’ll return a nullptr

A screenshot of a computer program

Description automatically generated

So now we have a function to find an input action associated with a given tag, which only works if we link up tags to input actions by filling out the data asset

We also need Input tags for any of the inputs we want in the game

We can add these natively to AuraGameplayTags.h

A screen shot of a computer

Description automatically generated

And initialise in AuraGameplayTags.cpp

A computer screen shot of text

Description automatically generated

A black rectangle with white text

Description automatically generated

Tags are in the editor!

Add new data asset:

A screenshot of a video game

Description automatically generated

We can add new Ability Input actions, but we need to make the input actions first. They are one-dimensional IAs (move is 2)

A black and white background

Description automatically generated with medium confidence

IMC links physical input to IA

A black and grey background

Description automatically generated with medium confidence

The Data Asset linking IA to tags

A black rectangular object with white dots

Description automatically generated

A little tedious but it means the data assets can be swapped out dynamically even at runtime

# Aura Input Component

Now that we have an input config we want a way to bind callback functions to inputs

We did this previously in the player controller



This was done using a function on the enhanced input component called BindAction



Let’s check out the definition for how this works

Here’s one possibility:

A screen shot of a computer

Description automatically generated

* It binds to an object UFUNCTION
* It takes a trigger event
* 
* A UObject
* 
* And a function name
* 

The function name is the clue that this isn’t the one we’re using for this next part! We’re passing in a function rather than a function name; this is a little bit above the definition of the one from the controller

A black background with white text

Description automatically generated with medium confidence

This looks like a very complicated definition so we should break it down

A black background with white text

Description automatically generated

There it is, a wrapper for a function pointer

Let’s look at the variatic inputs:

A computer screen shot of a blue and white box

Description automatically generated

This seems to indicate that this template function can take a variable number of arguments

It can in fact take a function that can take zero or more parameters

So what we can do is bind functions to inputs that can take input parameters eg gameplay tags

We’ll use BindAction, and loop through the input config (which has an array of input actions) and bind functions for those input actions

We’ll do this on the player controller

Currently the controller is simply calling BindAction on the inputcomponent, but we could have our own custom inputcomponent, so that the player controller doesn’t have to take care of things that the input component should be taking care of

The player controller should just call a single function and pass in any function callbacks that we want to bind for any given input. In fact, we could pass in the whole inputconfig data asset to the input component and let the input component handle it.

A screenshot of a computer

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screenshot of a computer program

Description automatically generated

This new class is an EnhancedInputComponent class



Configured correctly, we can set the project to use this as the default inputcomponent

First we want a function that can be called from the player controller that will bind inputs to callbacks, and we want those callbacks to be parameters to a function – we want the function to just handle things

A screen shot of a computer

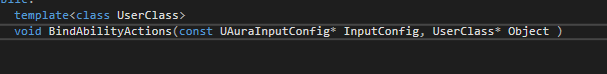
Description automatically generated

This will bind input actions for the abilities and take an input config

A screen shot of a computer program

Description automatically generated

Using an include instead of a forward declaration is because we’re going to make this into a template function. If it’s a template function we’ll define it in the header file



Using the template parameter and they class we’re using is UserClass, which will be that type (a pointer to it, because we’ll be calling BindAction which requires a UserObject

We’ll have other types as well:



This will be of the type that we’d like to bind when an input is pressed



We’ll do the same for a HeldFunction and a Released Function

A screen shot of a computer

Description automatically generated

So we can have functions that we can bind to all of the input actions when they are pressed, released or held!

A screen shot of a computer

Description automatically generated

Inputconfig should not be null

Next we loop through the InputConfig’s InputAbilityActions

A computer screen shot of a blue and white text

Description automatically generated

A screen shot of a computer

Description automatically generated

(it’s a public variable after all)

So what will we do for each item we loop through?

Bind the pressed, released and held functions for each

Why bind the SAME functions to all of the inputs?

Because we can pass along a gameplay tag!

So if we combine these functions to all the input actions, and we can identify those inputs by tags

We can pass those to the ASC, which can decide what to activate

A screenshot of a computer

Description automatically generated

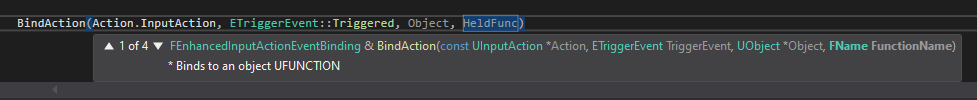
A screen shot of a computer

Description automatically generated

If we’re holding the button down we’ll call the Bind on the input action and the ETriggerEvent triggered, which operates every frame

A screenshot of a computer

Description automatically generated



Next is the Object, and finally the HeldFunc

NB: This template is able to take in a variable number of arguments, and if we pass in more than this those will be passed along to the Held Function and the Held Function – if we make one that can take a gameplay tag, will receive that tag! So we’ll also pass in Action.InputTag

Now any action, along with it’s associated input tag, will be passed along into it’s callback!

A computer screen with many colorful text

Description automatically generated

As long as the input is held, this callback will be triggered every frame

A screen shot of a computer

Description automatically generated

Started: as soon as the input is pressed, the function callback we’re going to bind to it is called only once

A black background with white text

Description automatically generated

Completed: callback will be called when we STOP holding the input

So now we have this function that, when we pass it, we pass in an inputconfig data asset, a User object and functions – callback that can receive gameplay tags, and they functions will be bound to each of the actions for pressed, released and held!

A screen shot of a computer program

Description automatically generated

Now we just need to tie it all together by going into the player controller and calling the function

# Callbacks for Ability Input

BindAbilityActions takes:

* InputConfig
* Object
* 3 functions
* Pressed
* Released
* Held
* Is a Template type
* - can handle WHATEVER is passed in provided that what BindAction does to it supports those types
* What it supports is function types
* Even functions that themselves can take input parameters

How Bindactionworks:



Takes a function (PressedFunc) and followed by that function we can pass in inputs that will be passed along into those callbacks when the input action is made (pressed)

To call this in the player controller we need 3 callback functions : An AbilityInputPressed, AbilityInputReleased and AbilityInputHeld that we can bind to these functions above

We shall also add a data asset to the player controller for the input config and forward declare the type



A black screen with white text

Description automatically generated

Now we have the input config we need the 3 callbacks

A screen shot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

We’re not using References here, for the gameplay tags, so in this scope they are an undefined type

A screenshot of a computer error

Description automatically generated

The easy fox is to include the GameplayTagContainer in the header file



Now we need to bind our 3 functions to all the inputs in the data asset, the input config. We’ll do that in the SetupInputComponent

A screen shot of a computer program

Description automatically generated

We’ve not yet set the new custom Input Component to the projects input component yet, but we will. So, rather than casting to UEnhancedInputComponent we will now cast to UAuraInputComponent

A screen shot of a computer

Description automatically generated

A black background with green and orange text

Description automatically generated

Now we’ll call the function on AuraInputComponent::BindAbilityActions

A screen shot of a computer program

Description automatically generated

The first argument is the data asset, followed by the object



Next the 3 functions, by pointer. We can actually do it like this:



Corrected some errors:



A black screen with white text

Description automatically generated

Now, provided we set our input component to AuraInput Component (we’re casting Checked so we have to or the game crashes), thanks to BindAbilityInputActions, when we press an input in the data asset the callbacks are fired.

Prove with debug message:

A screen shot of a computer program

Description automatically generated

NB: FColor::Red, not RED

In Editor:

A black and grey background

Description automatically generated

In player controller:

A black and grey background

Description automatically generated with medium confidence

A close up of text

Description automatically generated

Pressing and holding the input causes Red and Green; holding input allows the Red to fade but Green continues, releasing causes Blue.  
  
We now have Data-driven input, which is super powerful, because we can swap out or configure the data asset and everything is associated with input tags

Now that we have these three callback functions, we can use the tags to activate abilities as long as abilities have tags associated with them

Next step is therefore to associate the gameplay abilities with an input tag – swapping out the gameplay tag for a given ability allows us to remap it to a different key at runtime!

Nb: since the functions do not require any member variables or anything that requires an instance of anything to exist, we can make them statics

Also cursortrace could be refactored and made more compact