DRG Mods: Blueprint Modding

Buckminsterfullerene#6666

(Last updated: **23/01/22**)

Contents

Introduction

Tools

Before you can even get started with your mod, you need to install a few tools:

- DRGPacker
- UAssetGUI
- EmptyContentHeirachy (not really a tool but very useful)
- Unreal Engine 4.25.X (or whatever version DRG is currently using)
- An IDE (such as Visual Studio 2019, Rider for Unreal Engine or CLion)
- The most up-to-date game dumps (if you are using the BP dummy method explained later)
- The FSD template project (if you are using the C++ dummy method explained later)

By the time you are reading this guide, you should already have these tools and have at least the minimum required knowledge to use them (more on UE4 later). If not, I refer you to Rauliken's more general guide.

You should know the basics of Blueprinting in Unreal Engine. There are plenty of tutorials on YouTube. You should also know how Blueprinting associates with C++; here is a really good video on that. **If you haven't already, watching this video is basically a requirement to understand much of what is going on here.**

If you are into other forms of UE modding, please don't hesitate to join the UE modding Discord server.

Reading this guide

Make sure that you read through every detail of this guide thoroughly as missing something may result in many problems down the line. Of course, you can always refer to this if you need assistance on anything. Critically important details are highlighted in red, and optional but useful information is highlighted in blue.

Final note

Please be aware that as BP modding becomes increasingly advanced, this guide may become out of date until I update it. Since I'm really busy all the time this may not happen for a few days or weeks.

How blueprints work

How BP mods are loaded into DRG

Native spawning is something that was added by the developers when the modding update dropped. This allows you to load your blueprint from the BeginPlay event node. If you want your BP mod to have user-interactable UI (like a settings menu), you could either spend many hours creating your own, or use a framework like DRGLib.

Framework Mods

Usually, you will only want to use a framework mod if you want the user to be able to change settings for your mod in game. If you don't need this for your mod, you do not need to use a framework. Framework mods so far have always come in two parts:

- The devkit tools that are put into your mod's UE project
- The mod dependency that runs all of the framework's functions and processes

DRGLib

Samamster has been developing this framework as a more feature-rich and thus complex settings menu and BP modding library. The great thing about this library, is that it provides helper functions and DRG-like UI objects that makes BP modding just that little bit easier. This framework is being actively developed and maintained so I recommend using this one over BPMM, if your mod requires a settings menu.

Blueprint Mod Manager (BPMM)

When BP modding first started out, ArcticEcho created the BPMM. This is a simple framework tool for modders to use for their mod settings. Users also have the ability to enable/disable your mod from the in-game mod menu. While this is the most popular framework mod, it is no longer maintained and so has issues and new mods should not use it as a dependency if they can help it. The only times that using BPMM should be considered is if native spawning is causing issues for the mod (it sometimes has a bug or two in specific edge cases), so the mod is forced to hook BPs using the legacy BPMM method.

Methods of BP modding

There are two methods, in a way that you can use both at the same time or not if you wish:

- No-dummy method. This doesn't require any knowledge in C++ to use. You also won't need the game dumps. This is limited to built-in UE BP functions and events or those you have from a framework devkit, or you created yourself. You can still achieve a fair bit from this but are limited.
- Dummy method. You can manipulate the functions, variables and events that are running in the game. You can figure out what you need from looking at the game's dumps files. You can find the most up-to-date dumps versions here (be aware that GitHub only displays the first thousand classes on its website version).

Useful acronyms to know for BP modding

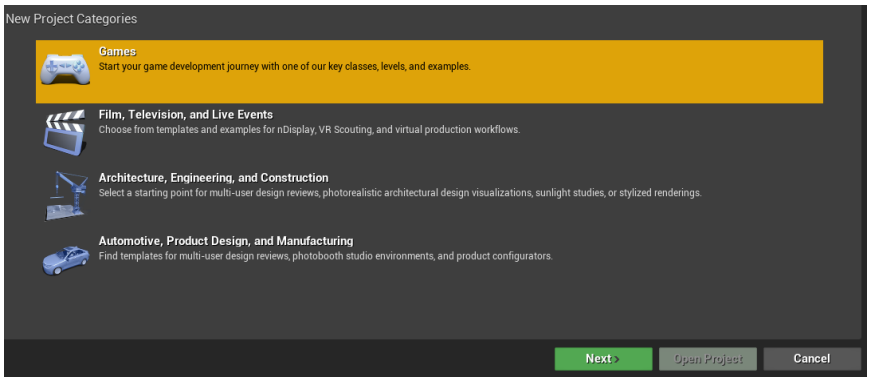
- ABP – Animated BluePrint
- BP – BluePrint
- GD – Game Data
- GM – GaMe
- ID – IDentifier
- ITM – ITeM
- LIB – LIBrary
- LVL – LeVeL
- MUT – MUTators
- OC – OverClocks
- PRJ – PRoJectile
- UI – User Interface
- UPC – UPgrade Category
- UPG – UPgrade Group
- W – Widget
- WND – WiNdow Widget
- WPN - WeaPoN

Setting up your UEE workspace

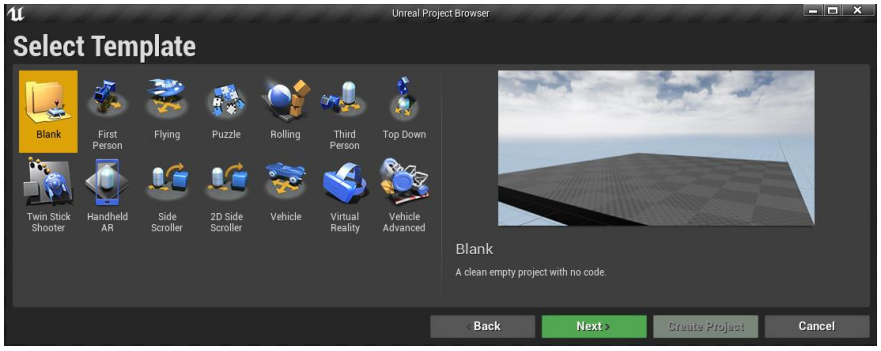
For ANY Blueprint mods, you MUST name your project FSD. Therefore, I make all my mods inside the same project and then delete the mods I don't want to pak before I pak them. The reason the name must be FSD, is because that is what the original game's UE project is called. "FSD" is probably the code-word for DRG (most games have these for various reasons).

If you haven't created your FSD project yet and are unsure of what settings you should make your project with, do the following.

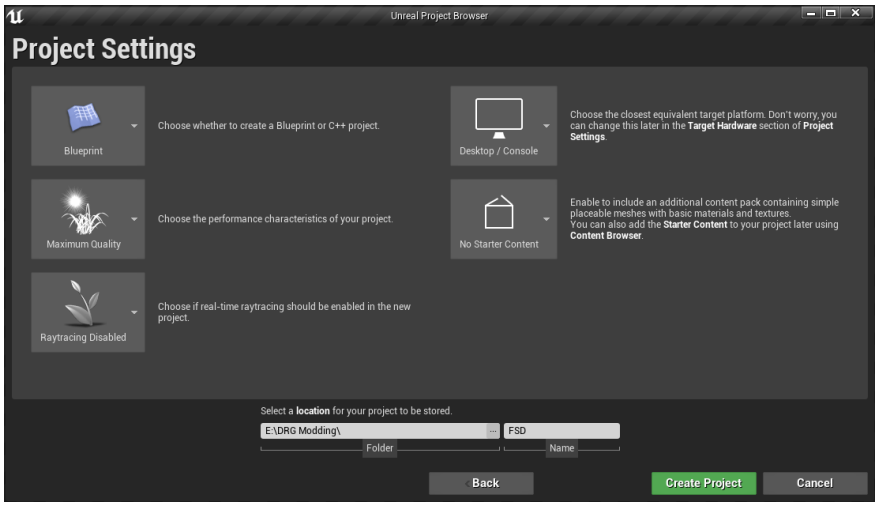
First, select the games category, then click next:



Then click on blank and click next:



Select the following options on the project settings:



Native spawning

To natively spawn a blueprint into the game, first you need to create a folder with the same name as your mod inside the Content folder. Then create a blueprint inside of that called one of two things:

- InitSpacerig – this will load your blueprint when the player is spawned in the spacerig.
- InitCave – this will load your blueprint when the player is spawned in the drop pod at the start of a mission.

For these BPs to be registered in the game you need to pack the AssetRegistry.bin file (found in your project's cooked files) into the pak as well. If you are using the DRGPacker you simply paste the AssetRegistry.bin file of your mod next to the Content folder (NOT inside!!). Then you need to go to DeepRockGalactic\FSD\Mods and make a folder with the same name as your mod, then inside that, put your .pak also the same name as your mod (without the _P). So, if your mod's name is "Test", you'd make a folder called Test and inside that put Test.pak.

And that's it!

Using DRGLib

Samamstar has a guide on how to interface your native BP.

Using BPMM

If you wish to use the BPMM, use ArcticEcho's guide.

No-dummy method

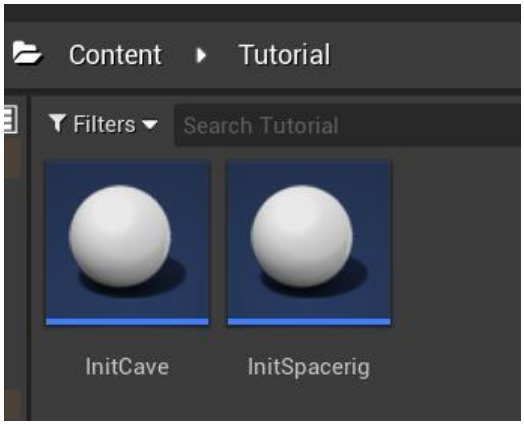
I'm going to run through the creation of a super simple mod which then outputs text to the screen, and finally saves and loads mod data. Note that my UE might look different to yours – don't worry, I'm just using a couple of plugins that makes it looks nicer.

Creating your first DRG mod

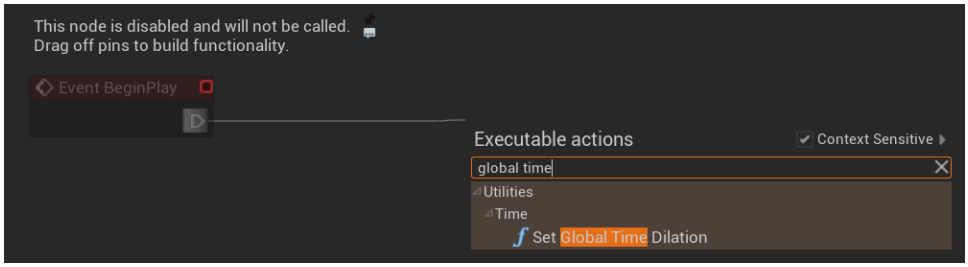
This mod will simply set the global time dilation of the game to 5. The purpose of this mod is to just run through the process of making a mod – the time dilation bit will validate that we know the mod is loaded.

First, make your mod folder inside of the Content folder in UE. I've called mine Tutorial. This will then be the name of your mod .pak file.

Now, you need to make the blueprints that the game will read and thus spawn – InitSpacerig and InitCave. To make a blueprint, right click inside the content browser and hit create blueprint class. Then inherit it from Actor.

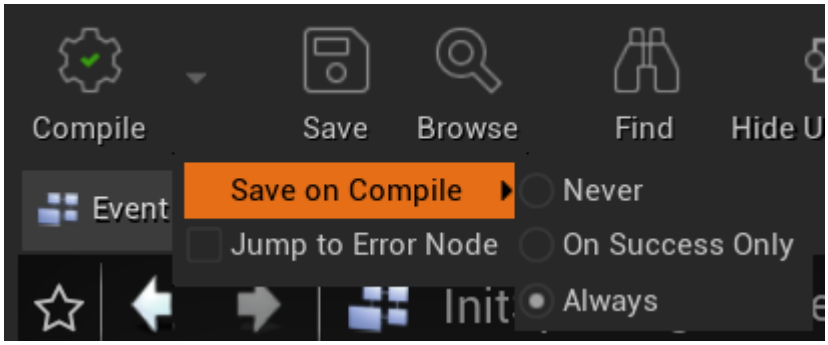


Double click on InitSpacerig first, and navigate to the Event Graph tab. We can ignore Event ActorBeginOverlap and EventTick. Drag off Event BeginPlay and find the set global time dilation node:



Now set this to 5.

To save, hit the Compile button on the toolbar. Remember to compile and save your project every now and again. Tip: you can compile and save at the same time when you click compile, by clicking the little arrow dropdown to the right of compile button and setting save to “always”:



Now we are ready to package and test this simple mod. Refer to the packaging your mod section. Remember that this method is using native spawning.

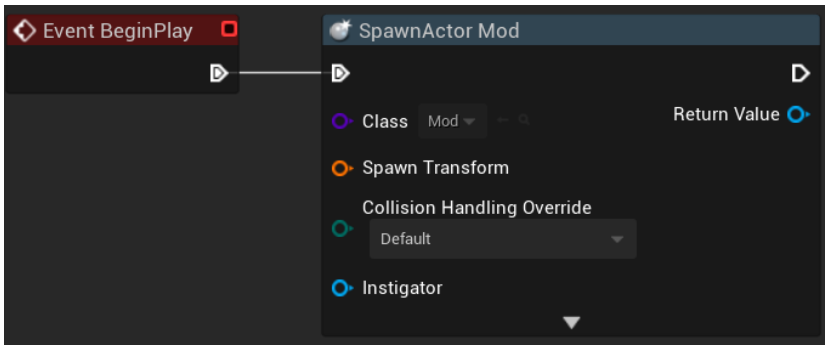
When you load into the game, everything should be moving 5x as fast! More importantly, now you know how to make native mods.

A bit more complex version

This time around, the mod will change the time dilation based on a key press.

First though, let’s streamline the same mod to be loaded by both InitCave and InitSpacerig, since we don’t want to copy and paste everything into both BPs.

Make a new BP called something like Mod (it doesn’t matter). Now in InitSpacerig and InitCave, we want to spawn this actor immediately. Get out a Spawn Actor From Class node, and set the class to whatever you called your mod BP. Do this in both InitSpacerig and InitCave:

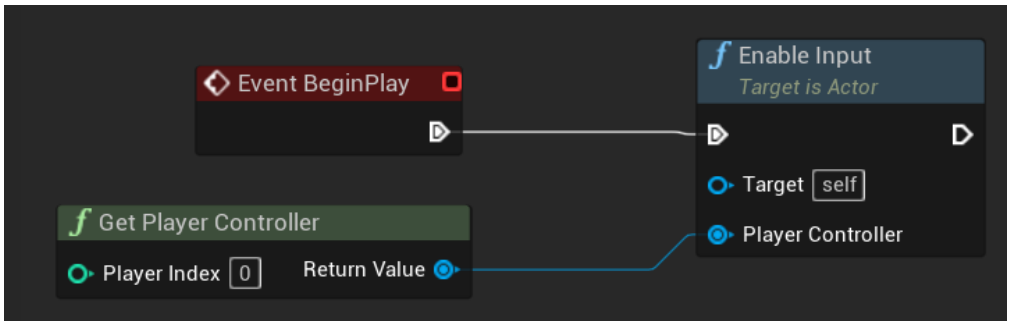


We’re not quite done yet, we need to set collision handling override to always spawn, ignore collisions. Also, right click on that orange Spawn Transform pin and hit split struct pin. If you don’t do this, it won’t know where to spawn the actor. Later on, if you’re spawning a physical actor but don’t want it to be seen, spawn it at like Z=99999.

Make sure to compile and save when you’re done in both BPs!

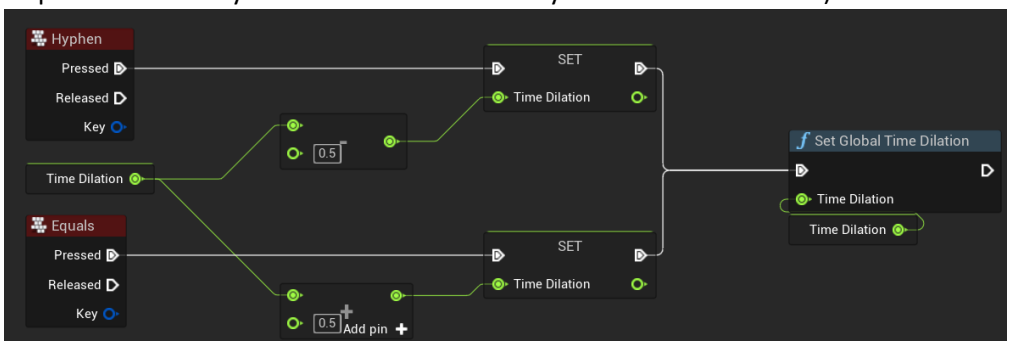
Now inside of your new mod BP, we want to add the functionality that increments or decrements the global time dilation based on what key you press. Let’s say that hyphen decrements it by 0.5 and equals increments it by 0.5.

First, let's make a variable that stores the time dilation. Set its default value to 1. Now we need to get the key events. Right click on the graph and type Hyphen and get out that node, and then do the same for Equals. Make sure that you enable input when the mod is spawned, otherwise this won't work:

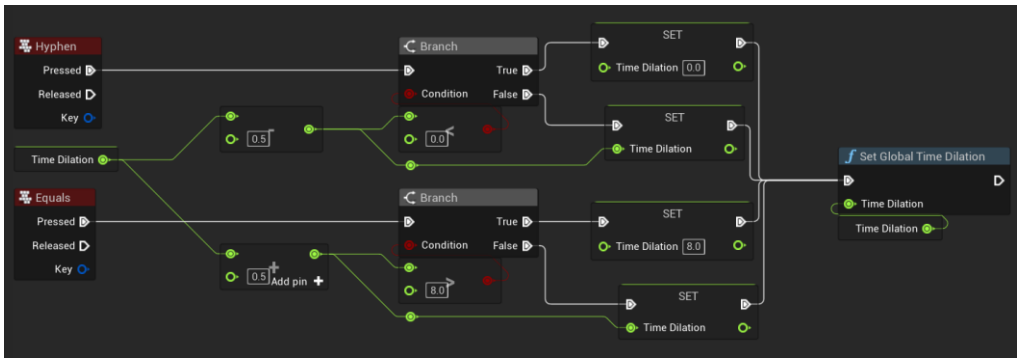


Player index 0 is always host, i.e. your player if you are the only one in the server or hosting a server.

Now make your mod functionality like this (you should have done basic blueprinting before as a pre-requisite to this so you should be able to easily follow what this does).



Now let's make sure that the dilation never goes below 0 (otherwise the game crashes) or higher than 8 (otherwise FPS is reduced the rubble).

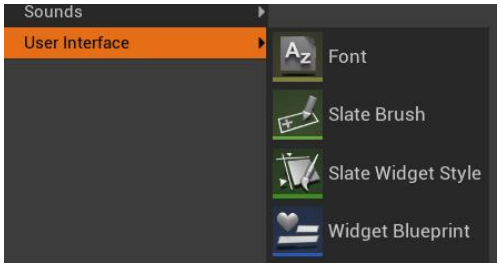


Before we go any further, let's compile, package and test the mod.

Outputting time dilation to the HUD

Now we want to output the current dilation to the HUD so we can see the value in-game.

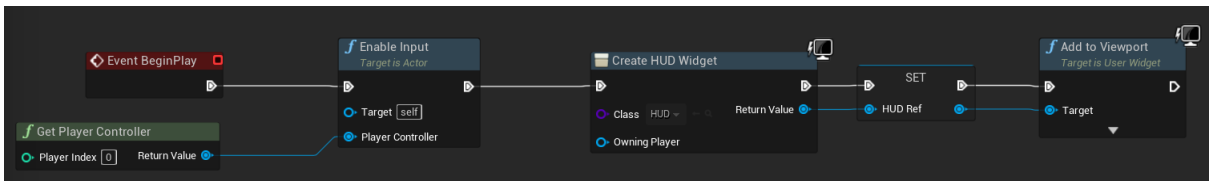
In our Tutorial folder, make a new widget blueprint called something like HUD:



Inside of the widget designer, find the text box widget in the palette and drag it out. Place it anywhere on the window – I recommend putting it about 2/3 the way up the left or right side of the window, which is where we know there is empty space on the DRG HUD. This text will just be some text like “Time Dilation:”. Next to it, put another text box with the text “1”. This is where we will change the value. The default dilation in the game is 1 so we set the default text to match that.

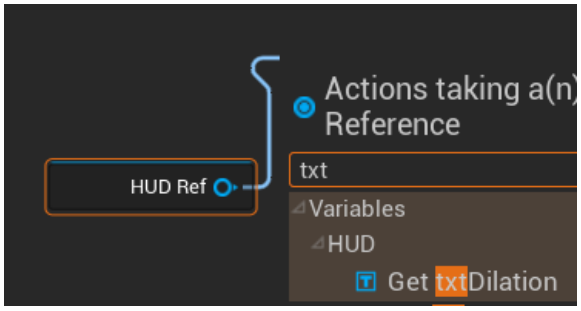
Now name that second text box to something sensible. You also need to check the Is Variable box, otherwise we won’t be able to change it from our mod BP.

Hit Compile and switch back to the mod BP. To access and set our text box, we need to first construct the widget and get an object reference from it:

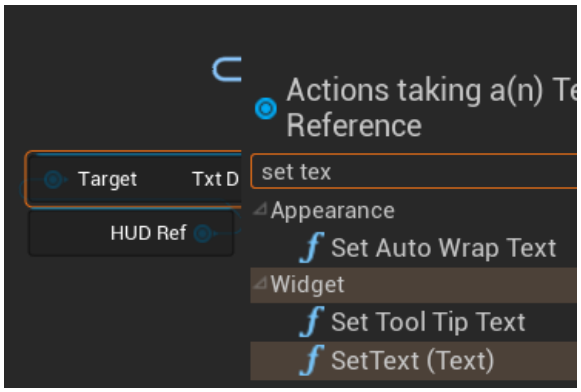


This HUD Ref variable is out object reference. The variable type is that HUD widget:

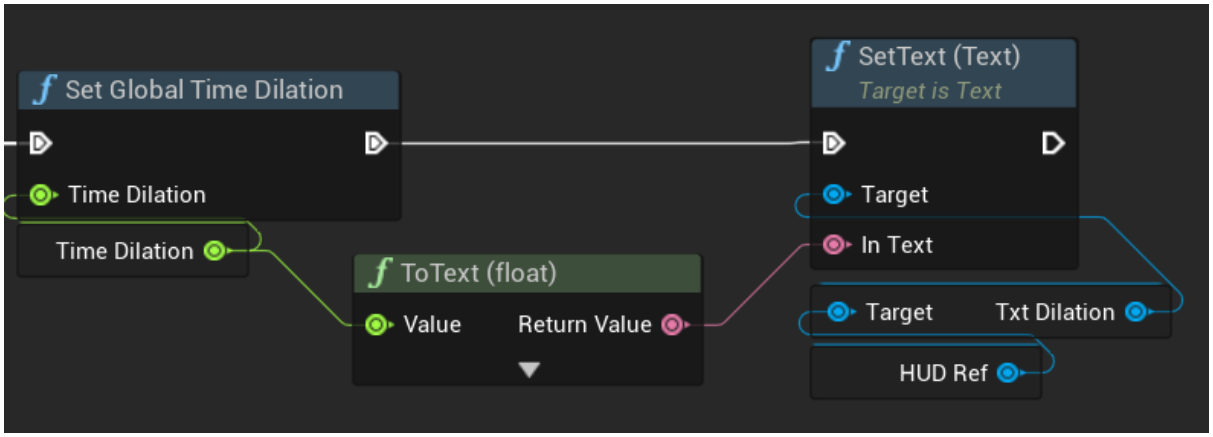
Now drag out the HUD Ref variable and we want to get the object reference to the txtDilation:



Now to set its text, we drag out the Set Text (Text) node:



Now we want to set the text to the value of time dilation:



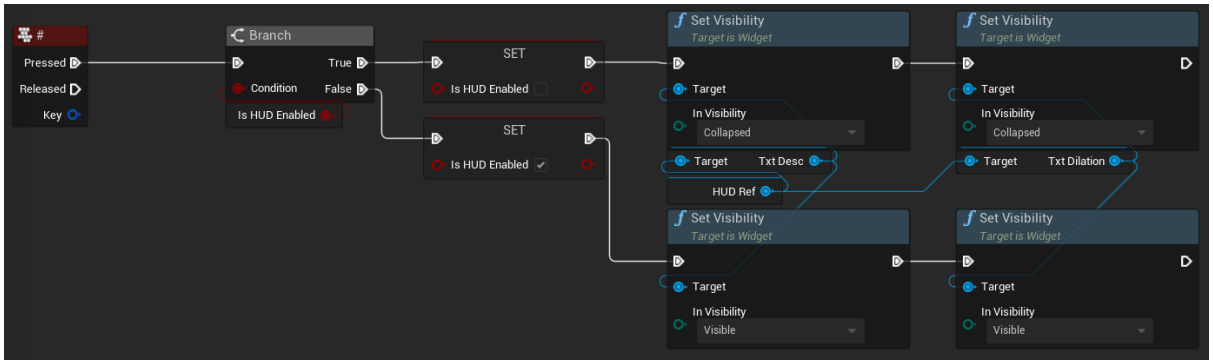
Now we can see our time dilation and that it changes when we hit our key binds. But what if we want to toggle the HUD on or off?

Toggle HUD button

First, we need to go back into our HUD designer and set that description text box to a variable so we can control it from our mod:

Now back in our mod, make a variable that will store the boolean value of whether or not the HUD is enabled. Make sure the default value is true.

When a key, say, '#', is pressed, we want to switch the visibily of the text boxes between Visible and Collapsed (you can read the tooltip regarding the differences between Collapsed and Hidden):

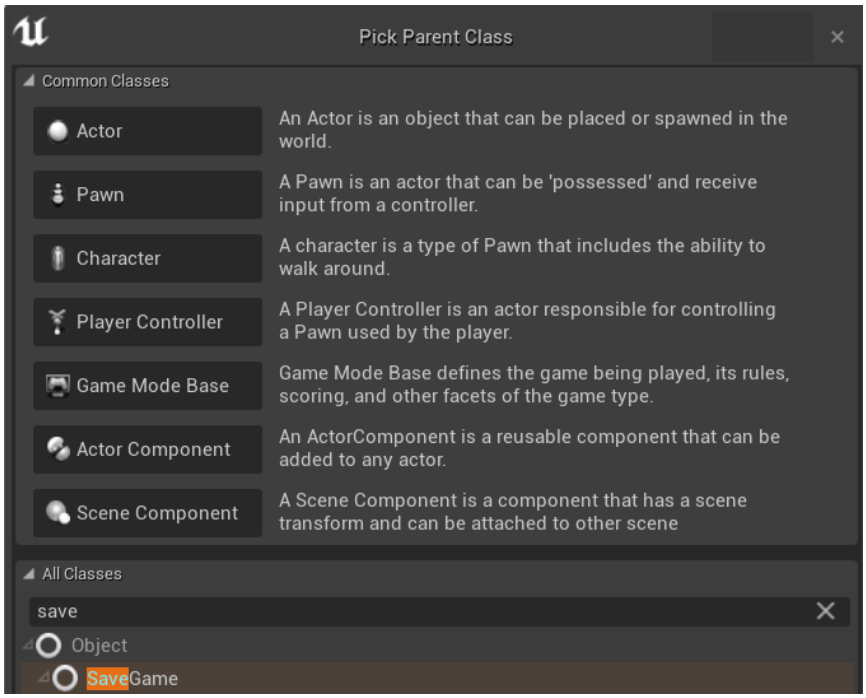


Let's check to see if this works!

Now… what if we want to save the dilation and if the HUD is enabled so that the next time the mod is loaded we save our settings?

Saving and loading settings

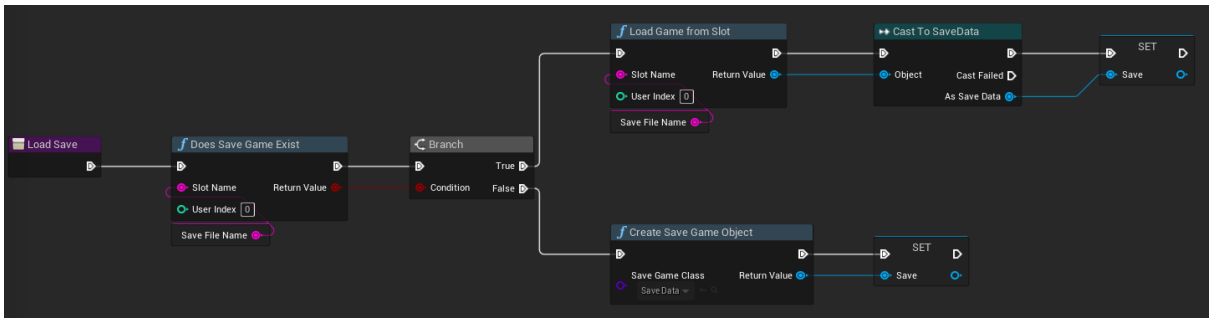
We first need to make a new blueprint that inherits the SaveGame class:



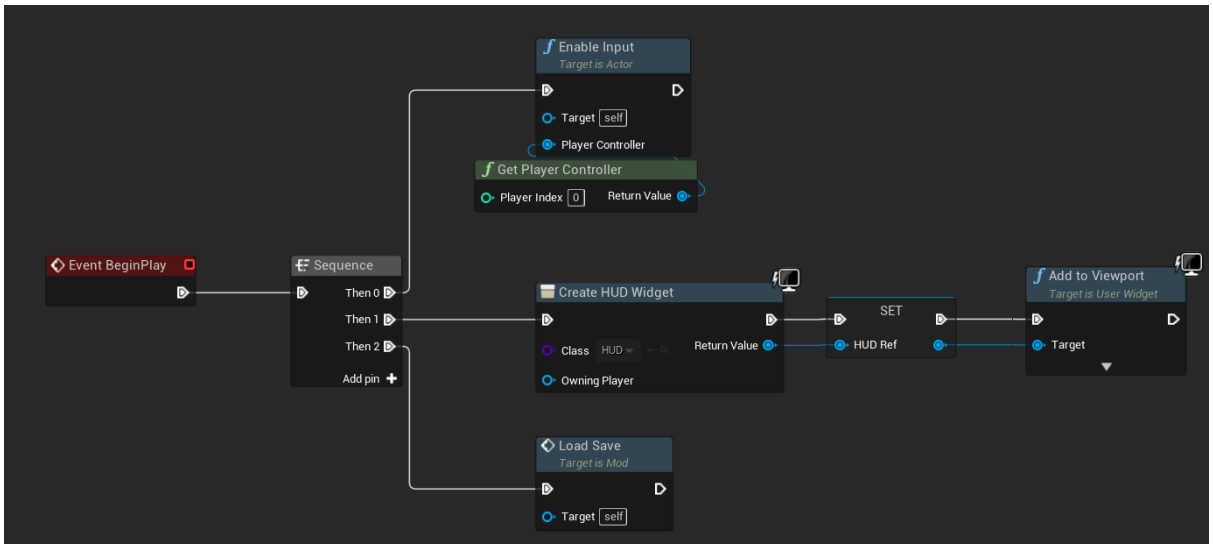
Inside of it, simply make the two variables that we want to save, and make sure that they are set to public:

Back in our mod, make a variable that is an object reference to this SaveData blueprint:

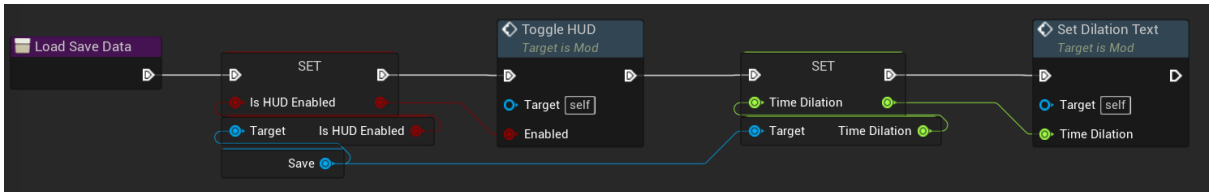
Now let's make a function that will load the save. This checks if the save game exists, and if it does, load the game from the slot. If it does not, we create an empty save game object. The save file name is Mods/<your mod name>/<your name name>. The saves are stored at FSD/Saved/SaveGames. Here my save name is Mods/Tutorial/Settings:



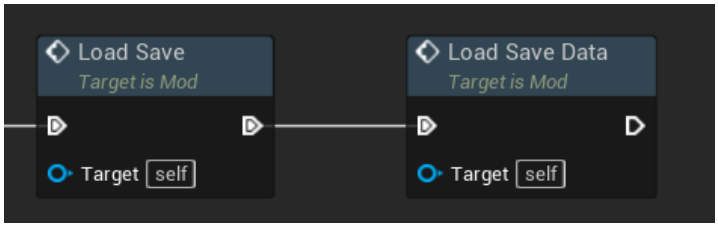
Call this function after we construct our widget (I've added a sequence node to tidy up a bit).



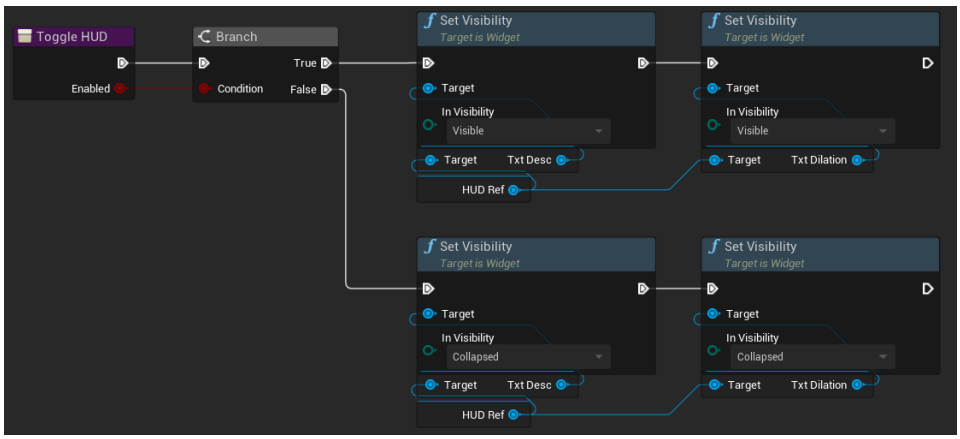
Now we want to make another function that loads the save data into our mod variables. We also want it to change the HUD values, i.e. if the HUD is disabled in the save, we want to toggle the HUD off when we load it in:

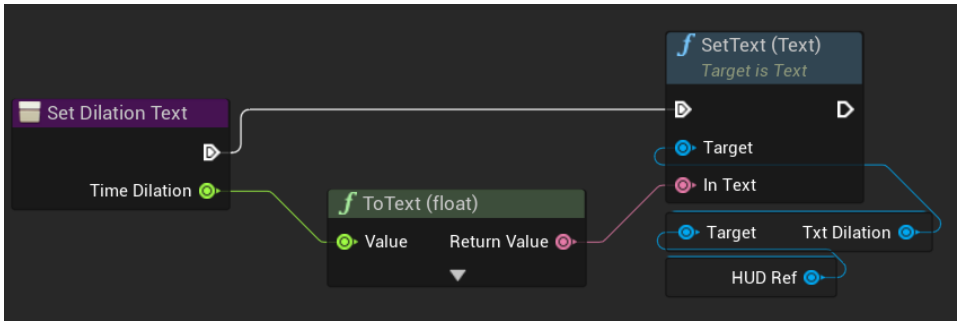


Chuck it after load save:

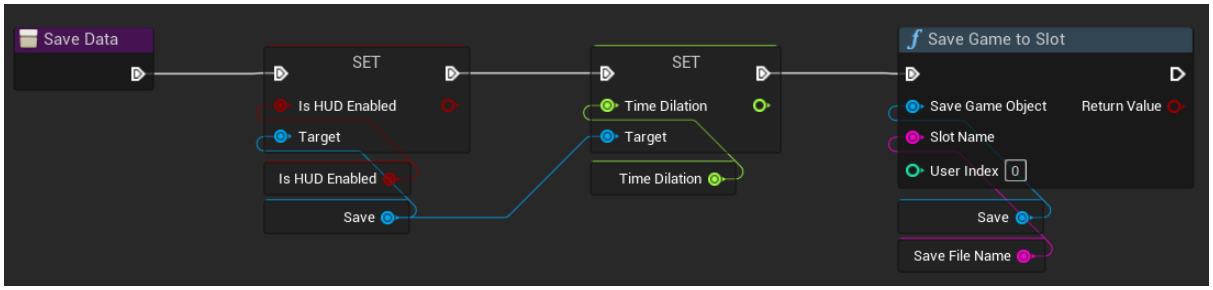


The Toggle HUD and Set Dilation Text functions are just what we have already done but put into functions so that we don't have duplicated code.

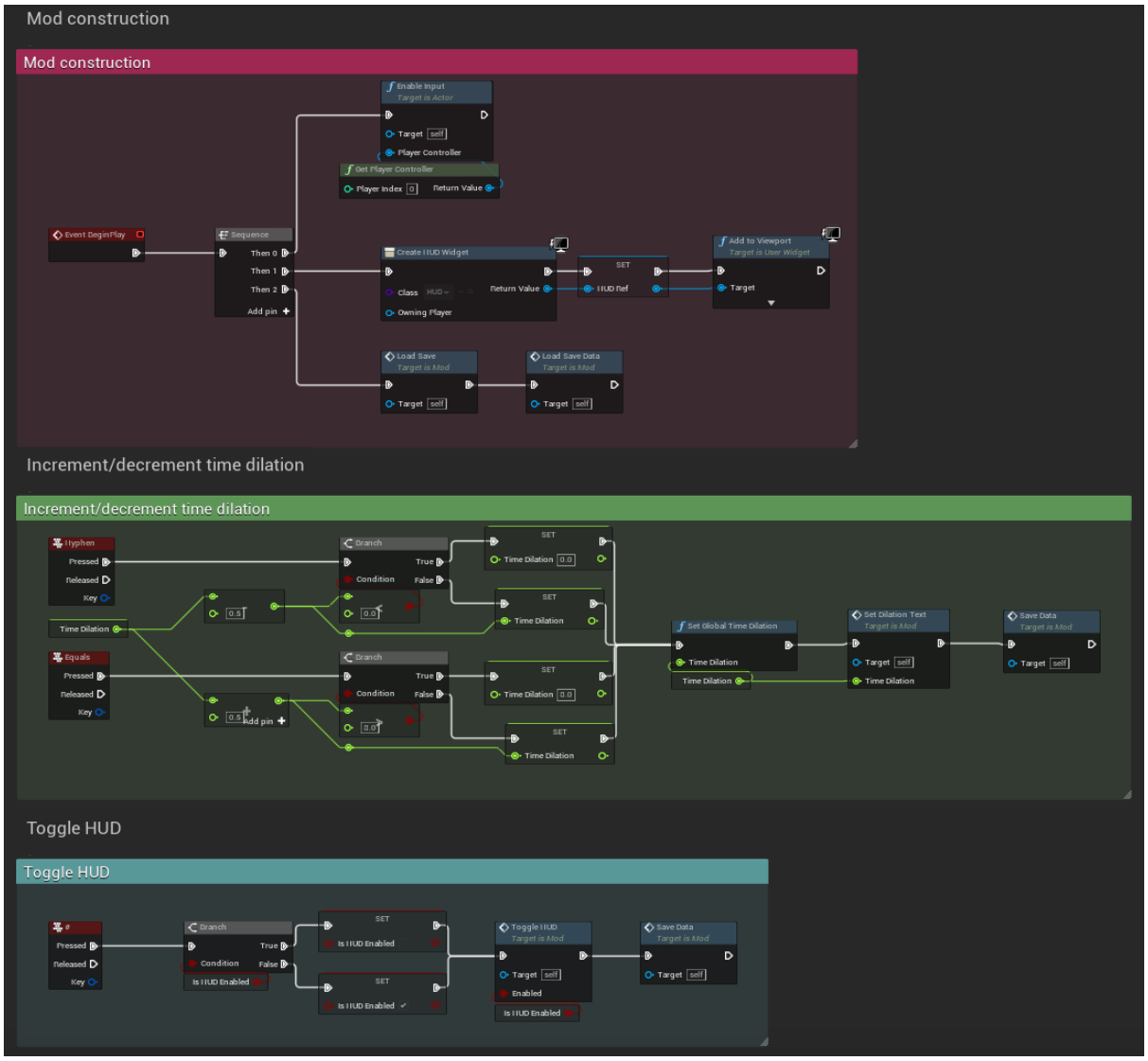




The last thing we want to do is save our values every time we change them. So let's make a new function that does this:



Now we call this function when we change time dilation and the HUD value. Now the mod looks like this (I've tidied it up with comments):



Now, compile, package and test. If you toggle your HUD off, and set dilation to like 5, then restart the game, your settings should be saved!

Dummy method

I'm going to run through making a couple of mods which utilise some of the C++ and dummy methods. Once you know how the basics work, you are then set to much more easily be able to experiment, test, and implement any of your own features that use any of the game's functions and BPs for your mods. The possibilities here are endless. Overview of the two example mods I will step-through in this tutorial:

1. Kill player on button press.
2. Self-destruct Bosco from a button press.

Both examples will still feature info on how to look at the C++ header dumps.

The very first thing you need to do is download and build the FSD Template project. This has all of the C++ classes automatically reflected in the project source. This saves modders a LOT of time, as there are thousands of classes!

Then you will also need the latest version's dumps from here. You will need to download an editor or code viewer such as Visual Studio Code to view and navigate the dumps (I use CLion).

Kill player on button press

Scouting the C++ header dumps

Now, you will notice that there are a LOT of C++ classes in the dumps. Don't be overwhelmed – most of the main functions you will need are in FSD_classes.h (although saying that, the file alone is 20k lines). If you need C++ from another class you know at the time what you are looking for – and if you don't, you can always search for key words that you are looking for and look through those files. You will be doing a lot of searching through files anyway, so you'll get good at this pretty quickly. Don't hesitate to ask in #mod-chat in the Discord where stuff might be located though.

So, back to this mod example. We need to find the function in the game that deals with killing the player. As you should always do, start by looking in FSD.h as it is most likely to be there. Do Ctrl + F to open the search function (this is the same for all IDEs and code editors), and type Kill. You may see about 100 results. You could look through these manually, OR, you could try toggling on "match by word" option that most IDEs and code editors have. In CLion, it is a square button with a "W" on it.

```
void Kill(AActor* DamageCauser);
```

So, here is a function that just kills the player when it is called. An AActor object is passed through but you can just ignore that as we don't need it. The important thing to do here is to check what class this function is inside. This is again relevant for any other functions you will find for your mods. So here, Kill() is inside struct UHealthComponentBase, which is a child class of UActorComponent, as denoted by the :.

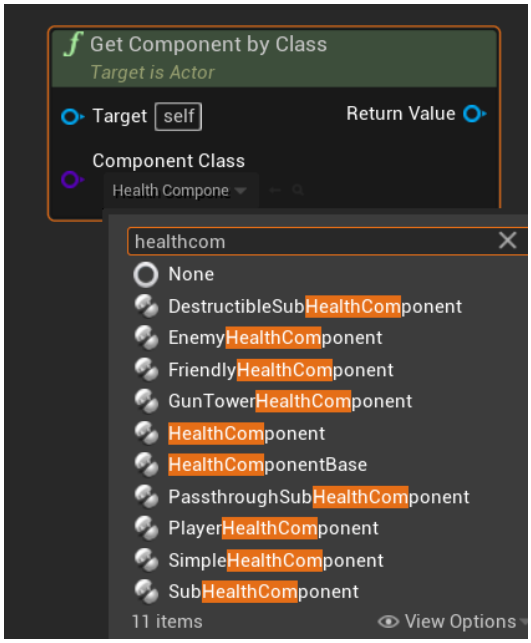
```
5624  class UHealthComponentBase : UActorComponent
5625  {
5626      FHealthComponentBaseOnHealthChanged OnHealthChanged;
5627      void HealthChangedSig(float Health);
5628      FHealthComponentBaseOnDamageHealed OnDamageHealed;
5629      void DamageSig(float Amount);
5630      FHealthComponentBaseOnDamageTaken OnDamageTaken;
5631      void DamageSig(float Amount);
5632      FHealthComponentBaseOnHit OnHit;
5633      void HitSig(float Damage, UDamageClass* DamageClass, AActor* DamageCauser, bool anyHealth
5634      FHealthComponentBaseOnBodypartHit OnBodypartHit;
5635      void BodypartHitSig(float Amount, float BaseAmount, UPrimitiveComponent* Component, UFSDP
5636      FHealthComponentBaseOnDeath OnDeath;
5637      void DeathSig(UHealthComponentBase* HealthComponent);
5638      FHealthComponentBaseOnRadialDamage OnRadialDamage;
5639      void OnRadialDamage(float Damage, float BaseDamage, const FVector& Position, float Radius
5640      FHealthComponentBaseOnCanTakeDamageChanged OnCanTakeDamageChanged;
5641      void CanTakeDamageDelegate(bool OutCanTakeDamage);
5642      bool ShowLaserPointMarkerWhenDead;
5643      bool canTakeDamage;
5644      bool PassthroughTemperatureDamage;
5645
5646      float TakeRadialDamage(float damageAmount, FVector BlastCenter, float BlastRadius, float
5647      void TakeDamageSimple(float damageAmount, AActor* DamageCauser, UDamageClass* DamageClass
5648      void SetHealthDirectly(float newHealthValue);
5649      void SetCanTakeDamage(bool canTakeDamage);
5650      void Kill(AActor* DamageCauser);
5651      bool IsDead();
5652      bool IsAlive();
5653      float Heal(float Amount);
5654      bool GetShowHealthBar();
5655      float GetHealthPct();
5656      TScriptInterface<IHealth> GetHealthComponentForCollider(UPrimitiveComponent* Primitive);
5657      FVector GetHealthBarWorldOffset();
5658      float GetHealth();
5659      UParticleSystem* GetGenericImpactParticles();
5660      bool GetCanTakeDamage();
5661      bool CanTakeDamageFrom(UDamageClass* DamageClass);
5662      void CanTakeDamageDelegate__DelegateSignature(bool OutCanTakeDamage);
5663  }
```

When you are looking for functions and variables for your mod, you will come across some really interesting looking functions and variables. I have regularly come up with entire mod ideas just from being distracted when looking through the code. Every single function and variable in all these header files can be dummied… which is a lot of possibilities. Feel free to go wild!

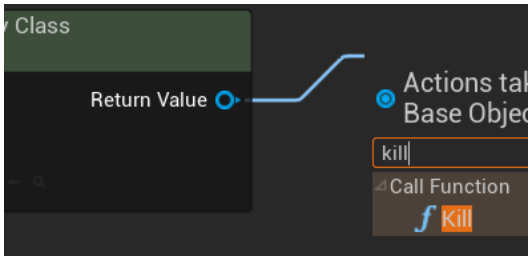
Accessing the C++

Now we know where our Kill function is, we can go about accessing it from our BP mod. Make a new mod (I just made a new one in my tutorial folder and changed the InitSpacerig and InitCave to spawn that instead).

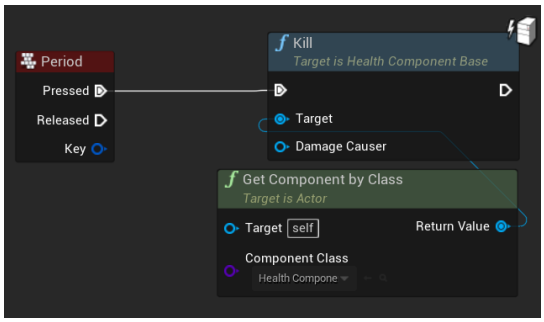
Inside your mod, right click and get a node called “Get Component by Class”. In the dropdown, you should be able to find the HealthComponentBase class:



Then if you drag off the return value pin and type Kill, you should see a function that pops up called Kill. This is your function that you created inside the HealthComponentBase C++.



Now when we press a key, say, the period, we want to kill the player:



There is a problem with this though! You cannot just call the Kill player function, as you will notice that the component by class node requires a target, which will be the player to kill.

To get the player you want to kill, we need to get the game state player array, loop through that player array then cast the array elements to BP_PlayerCharacter pawns, which you can then use as the targets.

Dummying player character BP

But what is BP_PlayerCharacter? This is a blueprint class that the game has that controls all the logic for the player. We need to dummy this in order for our killing player to work.

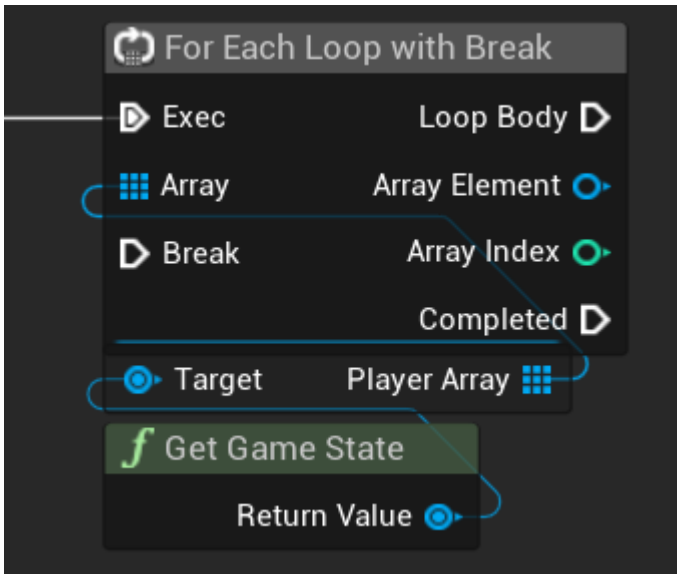
In the dumps, navigate to APlayerCharacter : ACharacter. You will see that this is where all the info about the player is stored. ACharacter inherits from APawn, which inherits from AActor. This is important to know because when you dummy blueprints, you can set the parent to be super parent (i.e. the highest in the hierarchy).

Create a folder inside Content called Character. Here we are recreating the exact file path of the asset (you can find BP_PlayerCharacter inside of your unpacked files at this location). Now you need to create a new blueprint class called BP_PlayerCharacter. Inherit it from PlayerCharacter.

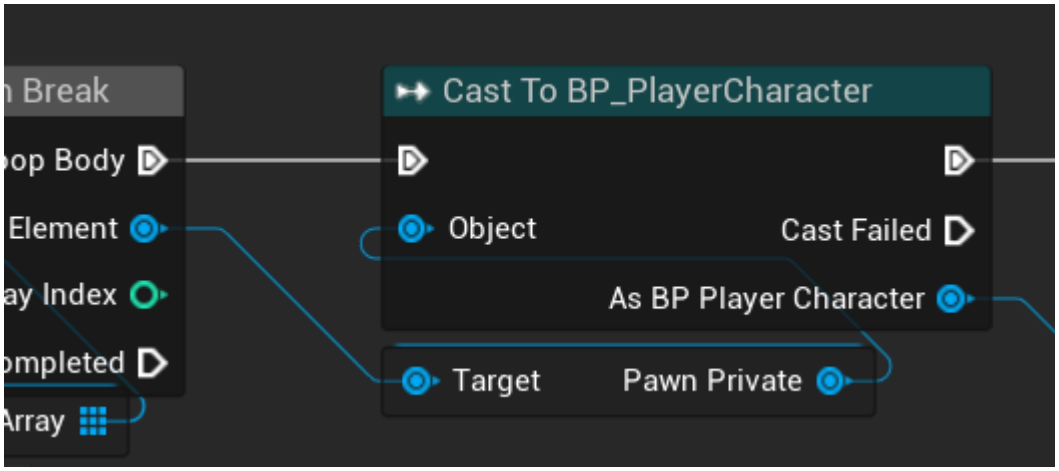
Once you create this blueprint, you don't need to do anything with it. Since we aren't dummying this file, we don't want to overwrite anything in the game with it so that is why we don't change any values. All we are doing is accessing the REFERENCE to the blueprint for the mod.

Now, back in your mod, let's get to work on this logic.

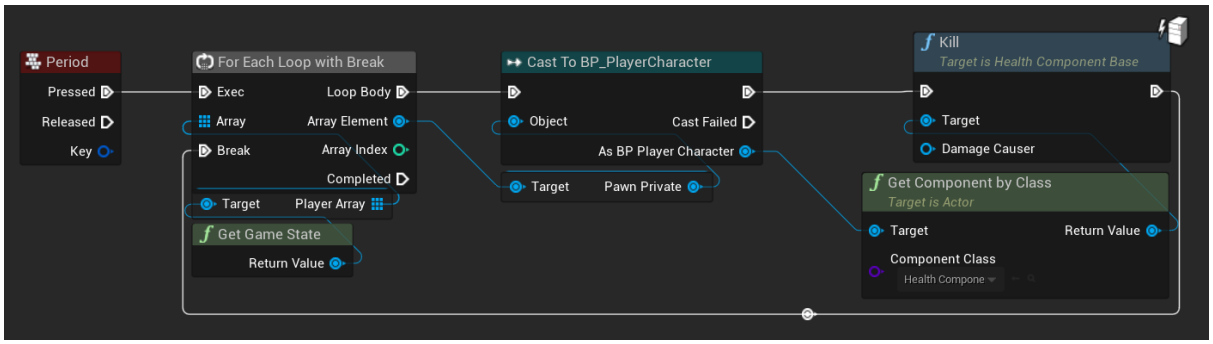
First, there is a node from the UE base Gameplay Statistics library called Get Game State. Drag off the return value and type Player Array, we want to get that. Then drag off player array and create For Each Loop with Break node. This will loop through all players in the player array and stop when it receives the break call. If you are in a multiplayer server, all the players in the game will be registered in this player array. So, if you just use a for each without a break, you could kill all the players in the game, if you wanted. This cluster of nodes will be very useful for your mods.



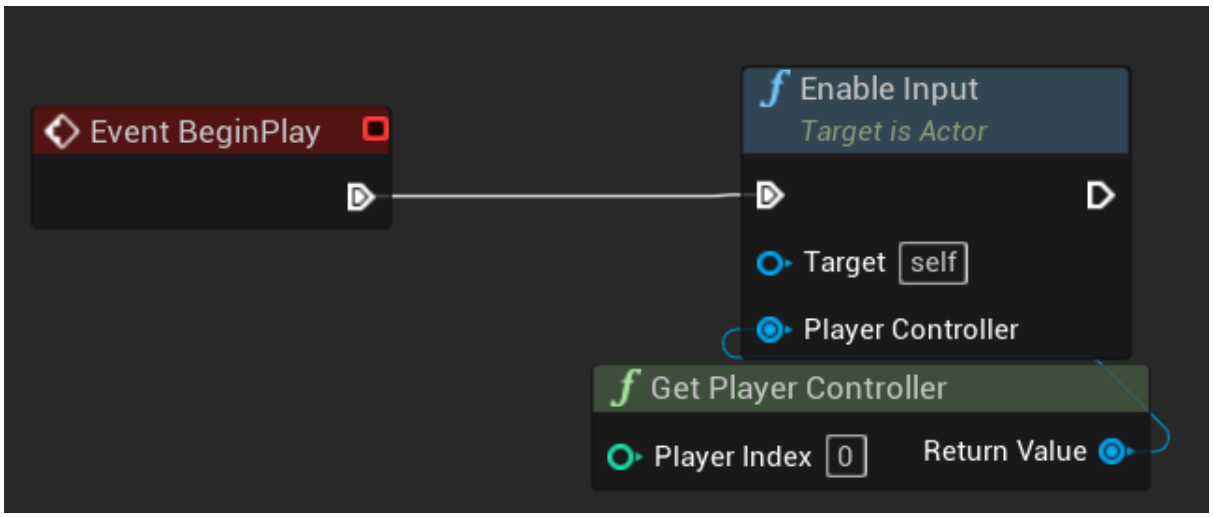
Now drag off the loop's array element and type Pawn Private. Then from the output of that read node, type Cast To BP_PlayerCharacter. Connect the execution node for that cast into the loop body.



Now plug in your Kill function execution node, and your target from As BP Player Character from the cast. Then, if you want to kill just the first player in the player array, hook up the end of the kill function execution to the break for the for each loop with break node. You don't have to do that though, as mentioned above.



Make sure that you enable input on BeginPlay!

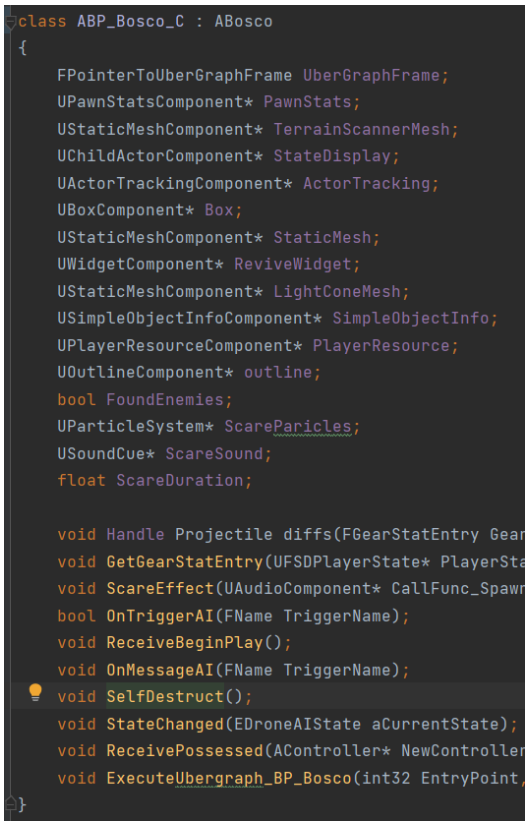


Before you package your mod, you have to make sure that you set your directories to never cook to include the Character folder. This is because if we pack our dummied BP_PlayerCharacter, the game will crash.

Self-destruct Bosco on button press

Scouting the C++ header dumps

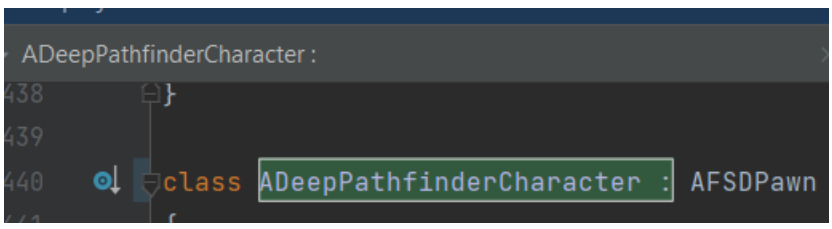
First off, we want to find BP_Bosco in the header dumps. It is actually called “BP_Bosco.h”. You will see a function called SelfDestruct() inside of it.



If you go into FSD.h, and search for “ABosco” (which is what this class inherits from), you will also see SelfDestruct and a bunch of other functions. Let’s say that I also want Bosco to salute before he self-destructs. When writing this tutorial I was just planning on him self-destructing but saw that PlaySalute() function and just knew I had to include it. That’ll happen a lot when you are looking through the dumps :)

Before we make the dummy blueprint, we need to figure out what class it needs to inherit from. So, if you look at the “struct ABosco…” line, you will see a ‘:’ which means “inherits from”, then the class it inherits from to the right. So here we see that ABosco inherits from ADeepPathfinderCharacter.

If your IDE has the feature, you should be able to just hover over the name and it will tell you what that class inherits from and click to go to the location of it. If not, don’t worry, as you can just search the file for that classname manually.



So you will see now that this class is inherited from AFSDPawn. Now we search for what AFSDPawn is inherited from, and we will see APawn.

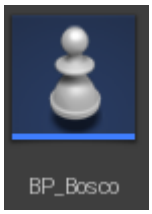
```
401    class AFSDPawn : APawn
```

APawn is a base UE class so that's as far as we need to go – but now we know, that ABosco is a child class of APawn, even though it is a bit down the inheritance tree.

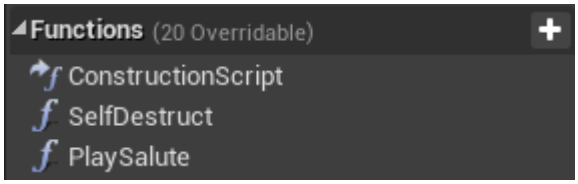
Creating the dummy BP

Now, we need to recreate a dummy blueprint in the same location where this guy is in the game files, like we do for hex mods. So in your unpacked files, search for BP_Bosco. It should be inside Content\GameElements\Drone.

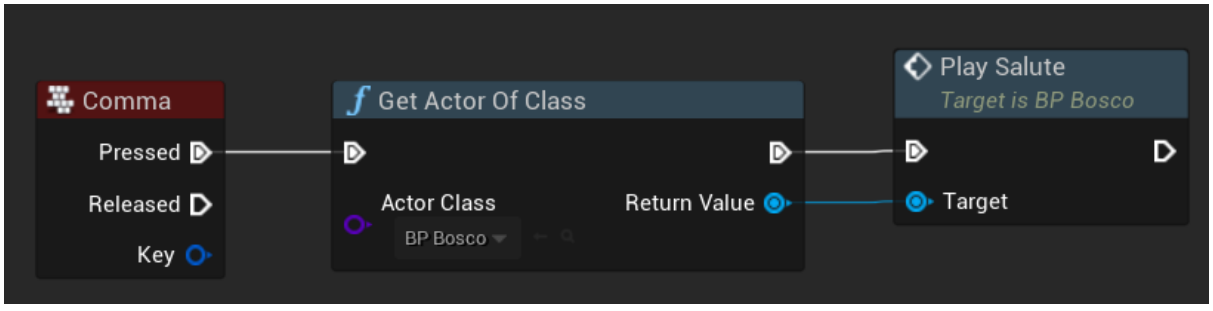
Inside your project files, navigate to that file location and create a new blueprint class. Since we discovered that ABosco is a child class of Pawn, we can select Pawn class to inherit from. We name the blueprint BP_Bosco. It should look like this:



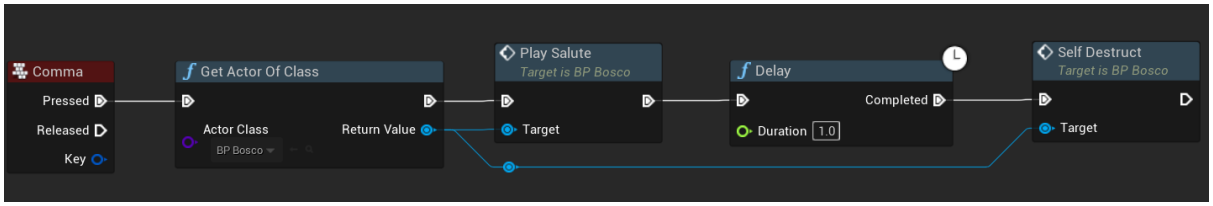
Inside of this, all we have to do, is create two functions, one called "SelfDestruct", and another called "PlaySalute". That is literally all you have to do. This now allows us to call these functions without having to touch any C++.



Now, to call on our blueprint, all we have to do is create a node "Get Actor of Class" and select BP_Bosco in the little dropdown. Then drag off the return value and type "Play Salute".



Now, let's put a short delay between saluting and self-destructing, and then call the self-destruct function.


Comma
Pressed
Released
Key
Get Actor Of Class
Actor Class
BP Bosco
Return Value
Play Salute
Target is BP Bosco
Target
Delay
Duration
1.0
Completed
Self Destruct
Target is BP Bosco
Target

Using BPMM Legacy (Don't follow unless you intend on using this method!)

BPMM Legacy does not use native spawning so is a special case when making mods.

I'm going to run through creating a small but cool mod that does uses the no-dummy method. If you've never seen nor used my *Better Time Control* mod, it allows the user to move a slider that changes the global time dilation of the game. We're going to make this, step by step.

So, the mod will teach you how to make basic widgets, how to change widget values from Blueprints and how basic DRG BP Modding works.

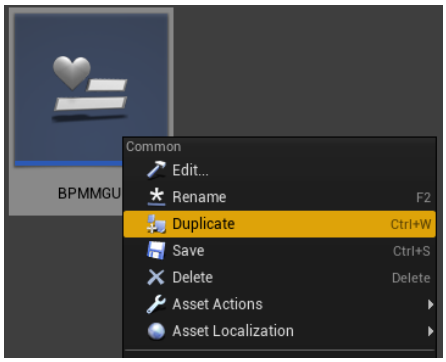
No-dummy method

Setting up the mod widget

First, if you haven't done so already, create a widget for your mod somewhere sensible.

To make your widget for the BPMM, you need some specific parameters in your canvas to be 1:1 ratio of size (UE : in-game menu). So, what I suggest you do, is download this BPMMGUI file and put it into your Widgets folder.

Then, when you make a new GUI widget, just right click it in content browser and hit duplicate:

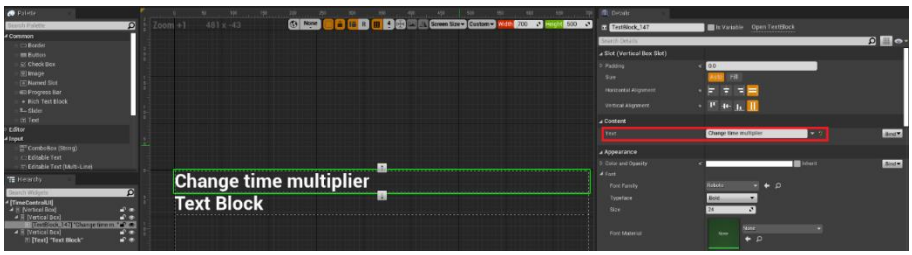


Then rename that duplicated widget to something appropriate. E.g. in this example, let's call this TimeControlUI.

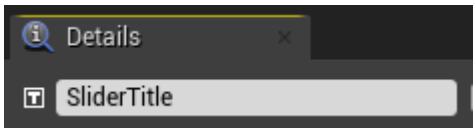
So, for this mod widget, we will need 3 main components:

1. Slider title text box. This basically "explains" what the slider does
2. Slider
3. Slider value text box. This shows what the current value of the slider is.

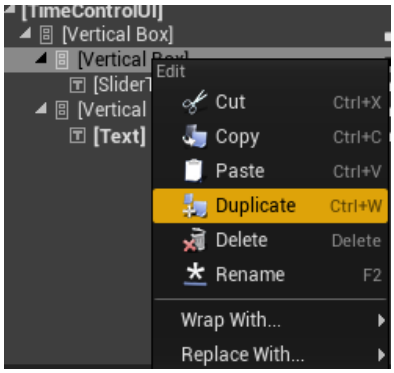
First, change the top text box that says "Sample text" to an appropriate message.



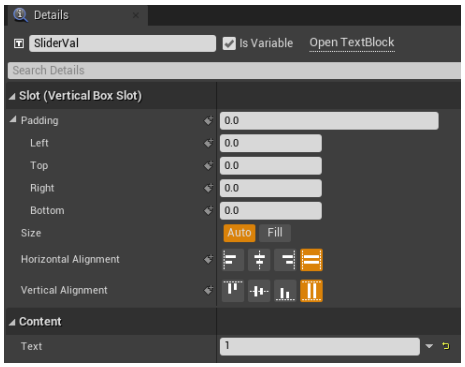
You should also make a habit of renaming your widget objects as although you will never need to access some as variables, you will for others so it's good to get into that.



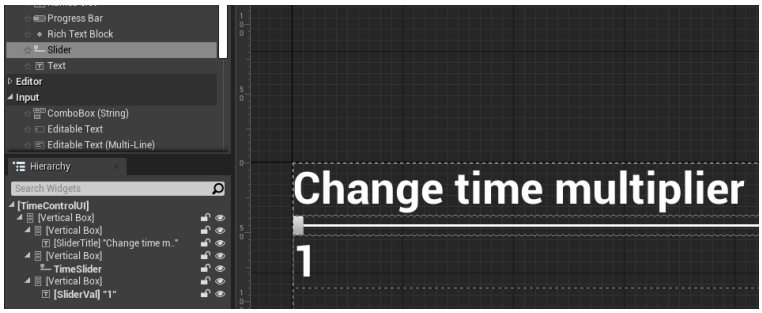
Next, in the hierarchy, right click this vertical box and duplicate it. The vertical boxes here are used to constrain elements so that they are much easier to pad and such.



Rename the newly created SliderTitle_1 to something like SliderVal and set its text to 1. Setting default text is still important because when you first load up the mod UI it will show these values. You also need to check the "Is Variable" checkbox next to the widget object name, as we will need to change this from BPs later.

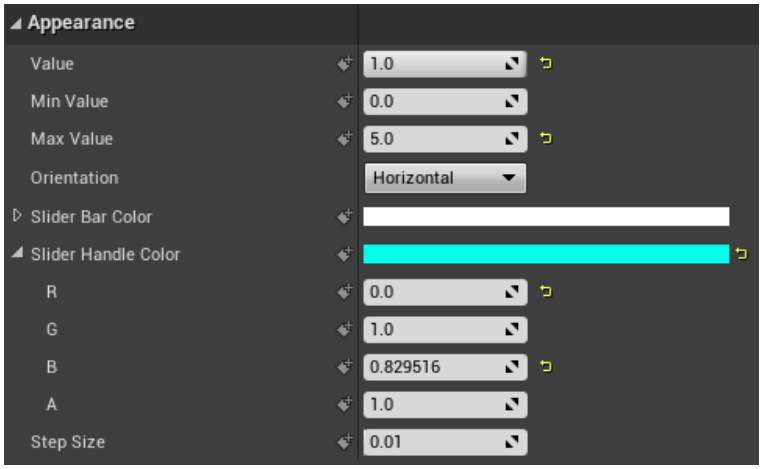


Delete the "text block" text box in the middle vertical box, as we will be replacing it with the slider. Then go into the palette and drag the slider into the now empty vertical box, either in the hierarchy or the editor. Like this:

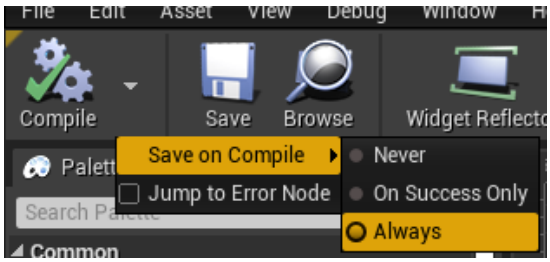


Now rename the slider to TimeSlider and make sure that the Is Variable is checked (it should be by default). Inside the appearance category for the slider, change the value to 1.0 (this is its default

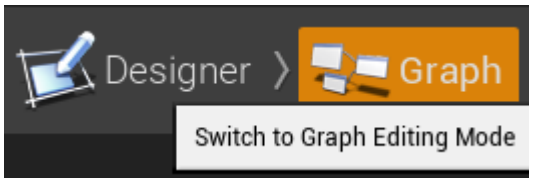
value), min value to 0 and max value to 5.0. Step size should be 0.01 as we want fine control over our time dilation values. I also like to set my handle colour to something other than just plain gray, but you can change any styles to whatever you want.



Remember to compile and save your project every now and again. Tip: you can compile and save at the same time when you click compile, by clicking the little arrow dropdown to the right of compile button and setting save to “always”:

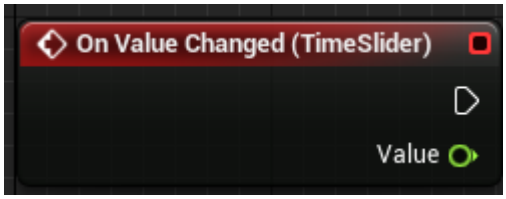


Now, go into the graph view of the widget, accessed from the top right corner of UE:



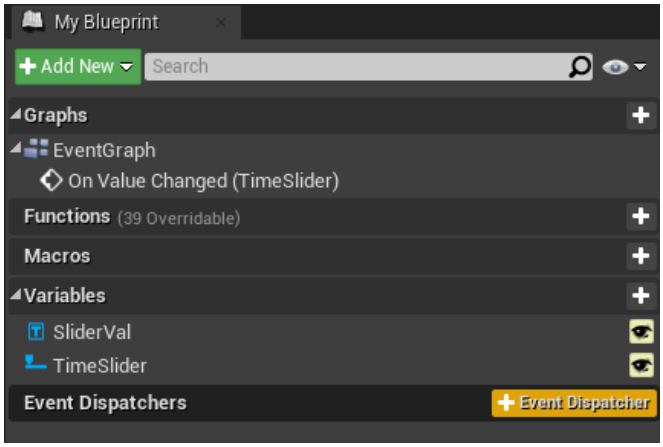
Now, you may be wondering, how do we get the current value of the slider? Well, widget objects have different events depending on what they do. You can then hook up something from that event so that it does something every time it is run. So, if you click on TimeSlider on the left pane, you will see that in details there are a list of events.

The one we want, is On Value Changed. So, press the green + button next to it to bring in the event.

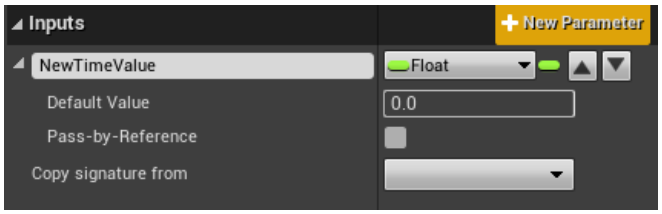


You can see that it is outputting a float called Value.

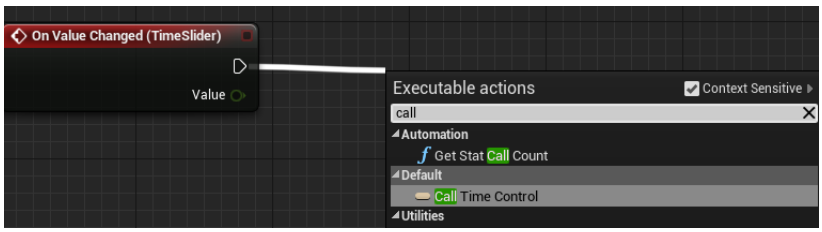
But, you may now be wondering, how will we bring this value into our mod blueprint? Well, you can create then call a new event! To do this, click create event dispatcher (in the event dispatchers section) to make one, and name it something sensible like TimeControl.



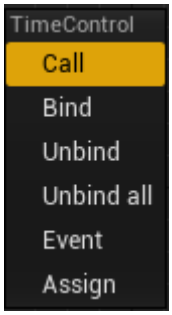
Now click on this new event in this panel, so that it comes up in the details panel. We need to create an input variable for this so that we can pass through the slider value. Click the + next to inputs and create a float variable called something appropriate such as NewTimeValue. The default value does not matter.



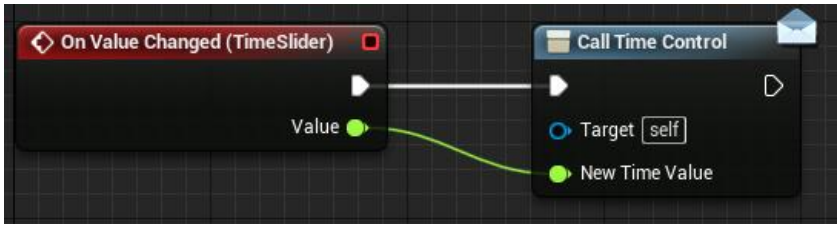
Now, drag off the On Value Changed event, type “call” and select to call the event TimeControl.



Alternately, you could drag out from TimeControl in the event dispatchers panel and click “call”.



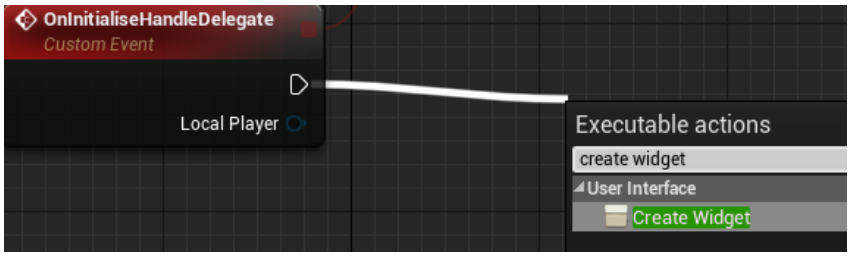
Then simply just plug it in like this:



And you’re done for this widget for now! Remember to compile and save!

Acessing and manipulating the widget objects from BP

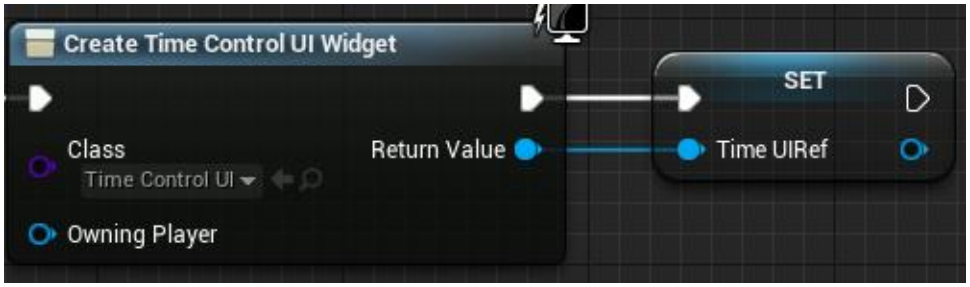
Now go back into your mod blueprint. First, we need to create the time control UI widget from the mod. You can do this by dragging off the oninitialisehandle event and typing “create widget”.



In the purple “Class” pin select class dropdown, select your TimeControlUI widget class. When you do that, your create widget node should not look like this:

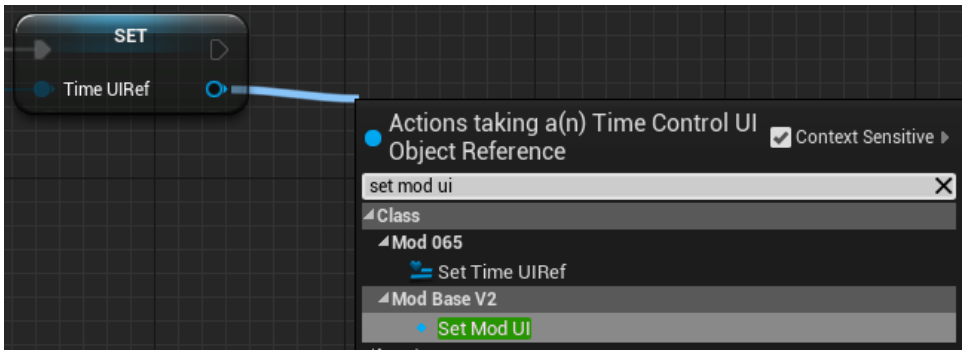
Next, we need to create a reference variable, inside our mod BP, that is of the TimeControlUI widget object type. So, go to the side, click + on the variables and call it something like TimeUIRef. Click on it, and change its type from (typically Boolean is default) to the right type.

Now, you need to use the return value from the create time control UI widget node to set the value of the TimeUIRef variable. To do this, drag out the TimeUIRef variable from the variables panel and click “set”. Then just connect the nodes, like this:

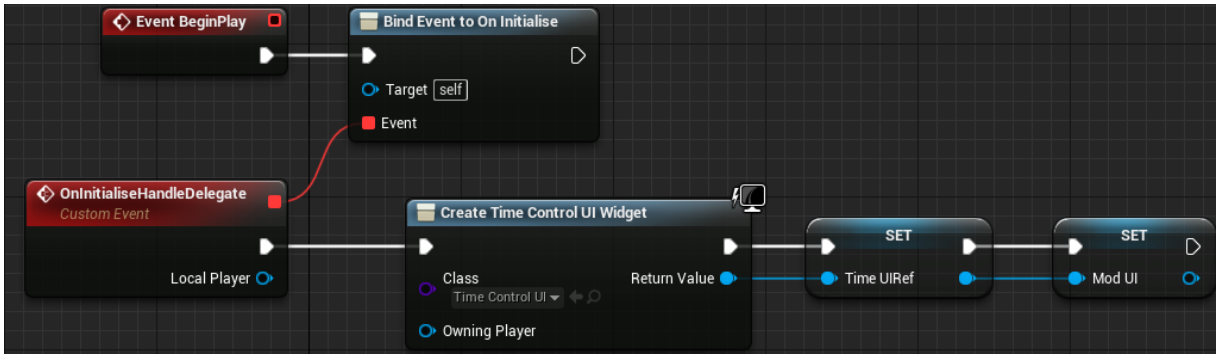


Tip: when you drag out your variables to get or set them, you can hold alt whilst dragging it out to instantly create a set, and ctrl whilst dragging it out to create a get.

If you are using the BPMM “alternate method” like I am, you need to set the mod UI variable to the same value, as this tells the BPMM that this UI is for loading into the menu. The ModUI variable is from ModBase.

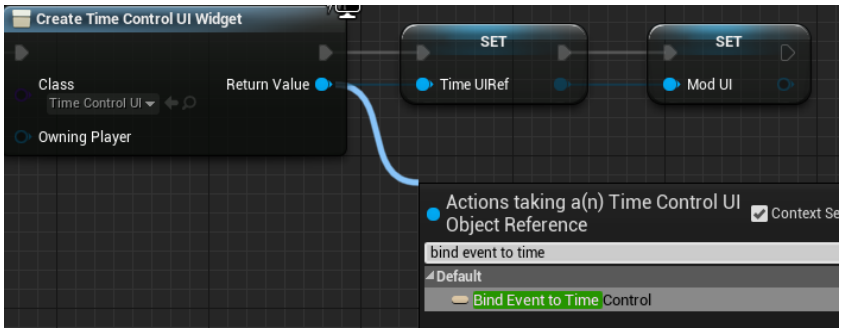


Connect the execution node and your entire BP should look a bit like this:

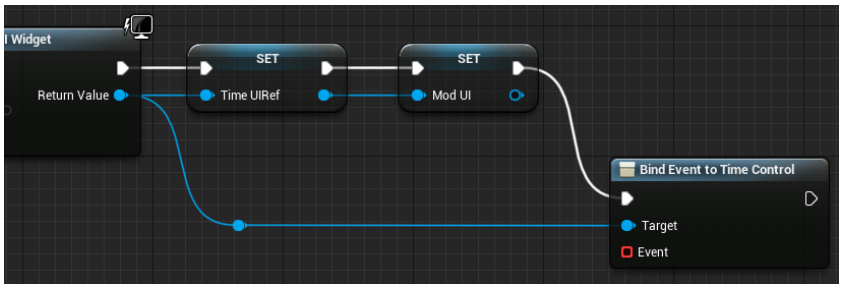


Now that you have got the widget into your mod BP, you can go about actually calling your Time Control event.

First, drag off the return value for the create time control UI widget node and type “bind event to” and select the time control one:



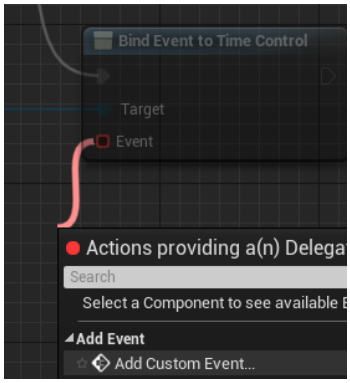
Connect the node up like this:



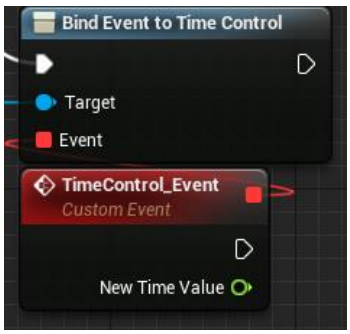
Tip: to create a reroute node, double click anywhere on the line.

Tip: to align all the nodes onto the same level, hold ctrl and select the nodes you want to align, then press ‘Q’

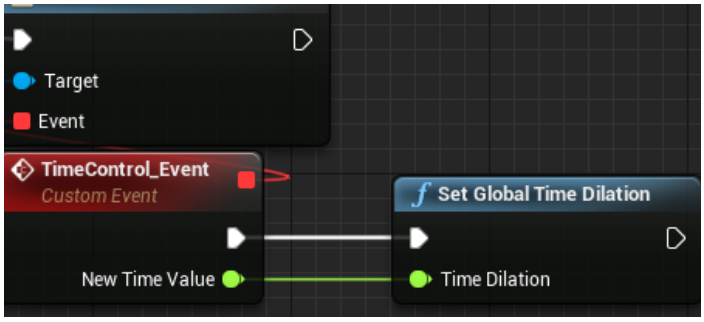
Next, drag off the red event pin and click “Add Custom Event”.



Name it something sensible like TimeControl_Event.



You will see that the new time value variable is passed through here! Now, all you need to do is drag off the execution node on this custom event and get the set global time dilation node. Hook up the output new time value pin to the input time dilation pin:



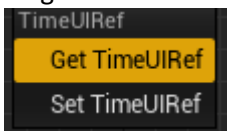
Now, the most basic of basic functionalities of this mod is done! Compile and save. There are still a few more basic features to add in this section, but let's test it in-game first. Refer to the packaging your mod section to do this. Then come back here.

Assuming that you have followed this correctly, changing the slider should change the game's time dilation. So, setting the bar to maximum increases it to 5x speed, and minimum to 0x speed, or frozen in time.

However, you may notice a couple of things:

- If you close the UI then reopen it, the slider's value resets to 1 (but does not call the on value changed event as your dilation should remain the same, at least until you move the slider again)
- The text box underneath the slider doesn't do anything.

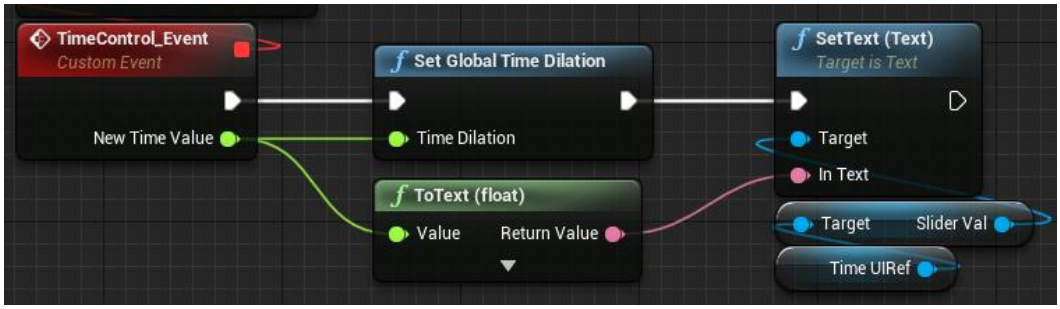
First, let's get the slider value text box to show the slider value. To do this, we need to be able to access the SliderVal text box object in our mod BP. Remember we had to set this text box to "Is Variable" earlier? This option allows us to get and set parameters for this object. There are two steps needed to get the reference for the text box object:

1. Drag out TimeUIRef and click get.


2. Then, drag off that pin and type "slider". Click get sliderVal (it should have a T in a box icon next to its name).


So you should have two nodes like this:

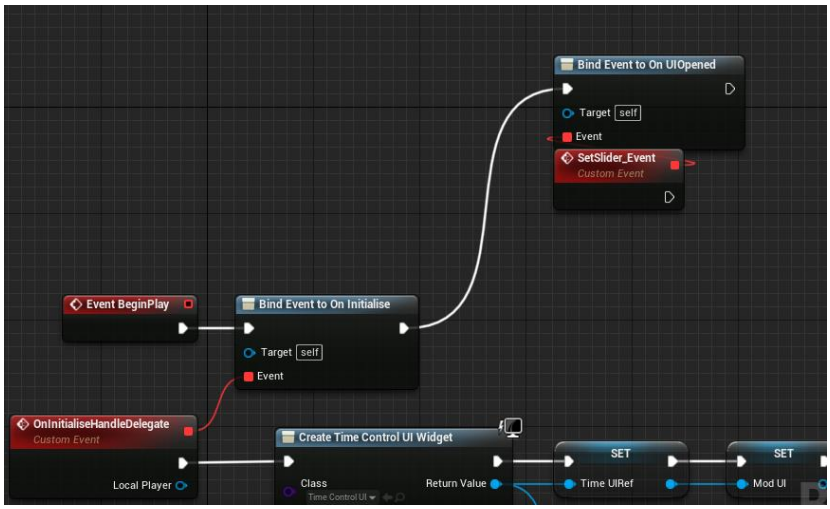
Now drag off Slider Val and type “set text” and click the one that says “Set Text (Text)”. Don’t get mixed up with the other one. Then, drag the execution node from set global time dilation to the set text (text) function. To put in the value, just drag from the TimeControl_Event New Time Value pin into the In Text pin. This should automagically create a ToText (float) node that converts float to a string. So your nodes should look like this:



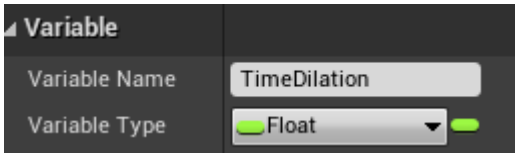
Now, everytime you change the slider value, the text box under it will show its value!

If you are using native spawning, you don’t need to bother with any of the following OnUIOpened business. If you are usng the BPMM alternate method, then you can continue onwards.

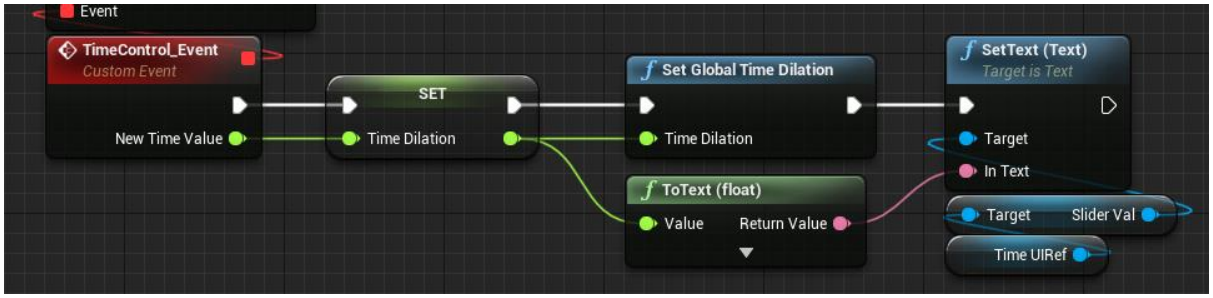
Next, we need to solve the bug. To help us with this issue, the BPMM gives us a handy event called OnUIOpened. Bind a new custom event to this and call it something like SetSlider_Event. Make sure the bind event execution node is connected from the bind event on initialise.



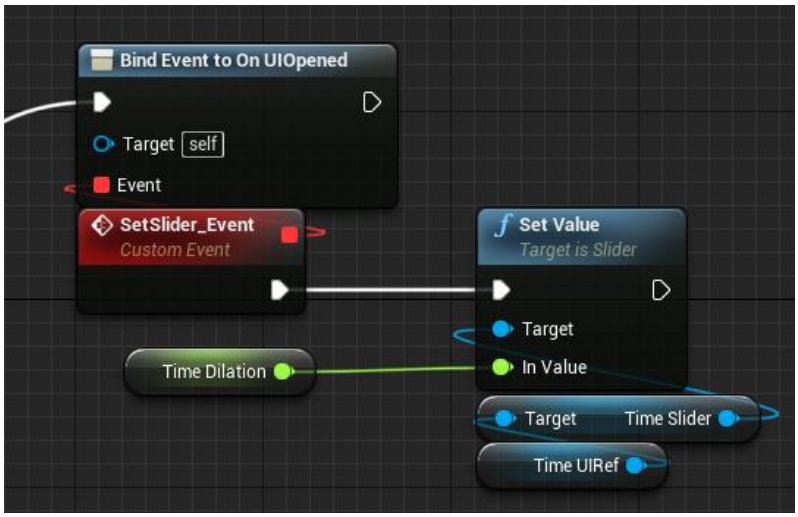
What we want here, is to set the value of the slider to whatever it was when you closed the UI. The problem, is that we don’t have any variable that is locally storing this value. So, we need to create a variable called something like TimeDilation, that is a float. Also set its default value to 1, as that is what we want our starting value to be.



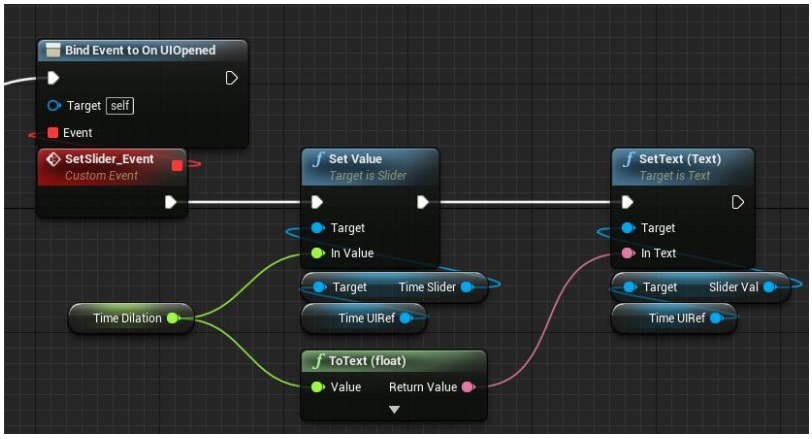
We also now need to set this variable whenever the slider value is changed. So let’s set the variable in the TimeControl_Event execution flow. So now your nodes should look a bit like this:



Now, back in your SetSlider_Event execution flow, you can get the value from this TimeDilation variable and set your slider value to it. To do this, you need to get reference from TimeUIRef and TimeSlider like you did with the set text (text) for the Slider Val. Then drag out of Time Slider and type “set value”. Then just plugin that node’s execution and time dilation. It should look like this:



Now, you may realise that we are also forgetting to set the SliderVal text to the right value when the UI is opened too. Just copy and paste this node group from where you do that in TimeControl_Event, into SetSlider_Event, plug the nodes up and you are done! It should look like this:



You can finish there if you want for this basic functionality and go onto packing you mod, but I will also explain how to display values to your player’s HUD which is very useful when debugging your mods or even just giving the player info about the mod.

So, what we will do, is display the time control value to the HUD, which will change as you change the slider.

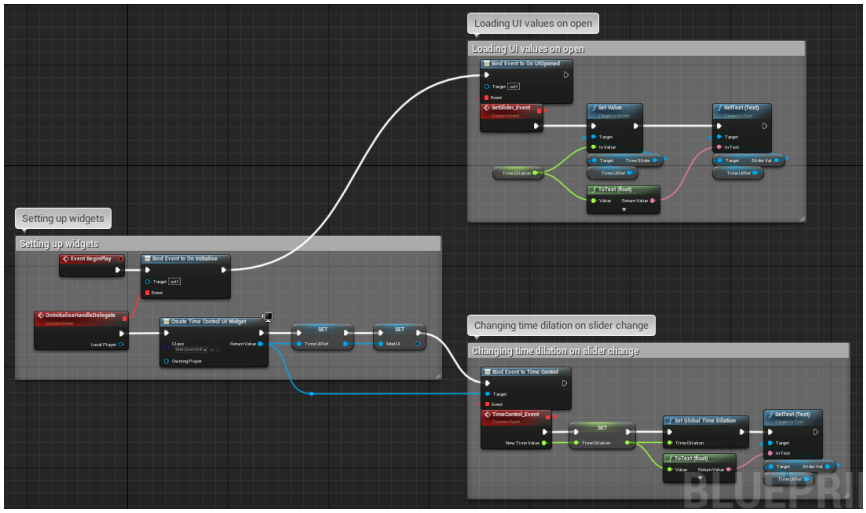
First, create a new widget and call it something like “TimeHUD”. This doesn’t need to be duplicated from the BPMMGUI file as the default canvas will be scaled to whatever your game’s resolution is set to.

Now, you will want to make a text box which says “Time Dilation:”. This won’t do anything – it just displays the label for the actual time dilation output. Now, you can put this wherever you want, although remember where your game’s other HUD elements will be, i.e. all the corners are taken up by something or other so you probably want to put them on one of the sides. I just put it about 2/3 up the left side.

You will also want to make another text box just to the right of the label box which will actually display the time dilation. Make sure you tick the Is Variable box if it isn’t already. Also you can set the default text to “1” or whatever your default slider value is. This way the player knows exactly where it is from the get-go, and it won’t just appear when you first change the slider.

So, your TimeHUD widget should look like this:

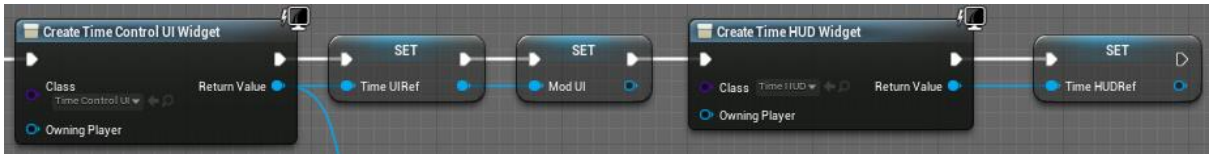
Before we get into this little part, I just want to clean up the blueprint a bit. If you highlight a bunch of nodes and press ‘C’, a comment box appears around the highlighted nodes. Then you can edit what the comment says. So now my BP looks like this:



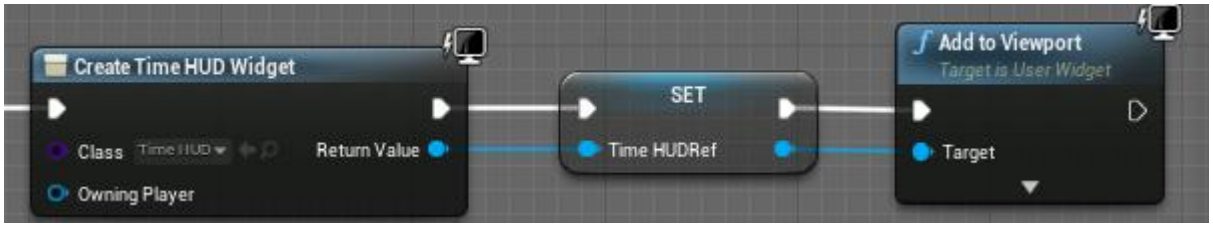
You can also put comment boxes inside comment boxes, as recursively as you want! Although at some point it may get a bit silly so you should probably be making functions for some of your stuff, although don't worry about that now.

Now, just like before when you created the TimeControlUI widget in your mod blueprint, you need to make the reference variable and set it in blueprint.

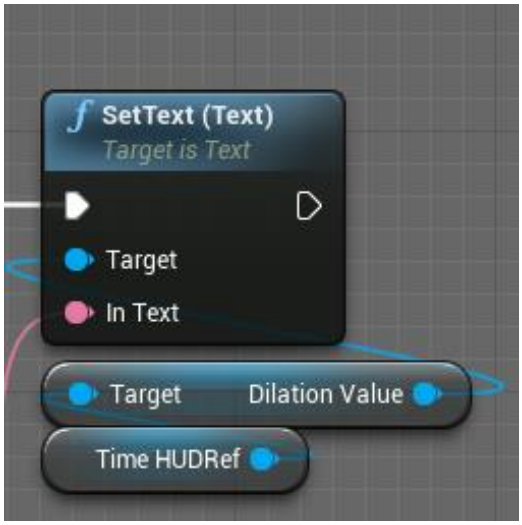
Drag it off the set Mod UI node from the previous widget creation:



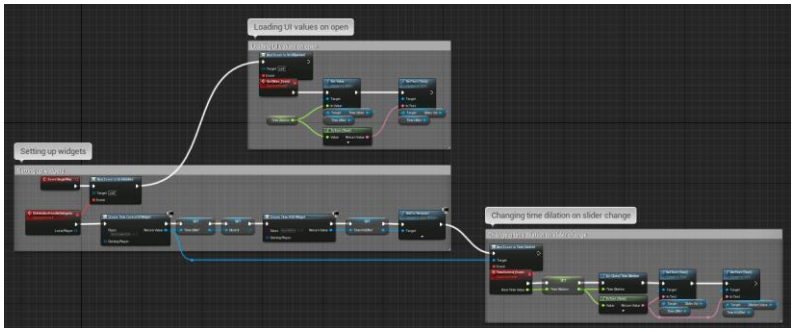
Now, since you aren't making this widget part of the BPMM mod UI, you don't want to set the variable here. Instead, you need to add this to viewport. Conveniently, there is a node called Add to Viewport!



To change the text of the dilation text box, again we use the SetText (Text) node, in the exact same we did it for the other widget. You will of course want to use TimeHUDRef instead, but it's the same thing. So, since I called my time dilation value text box "DilationValue", I just need to add this on the end of the slider change event:



So that overall, your mod blueprint should look like this:



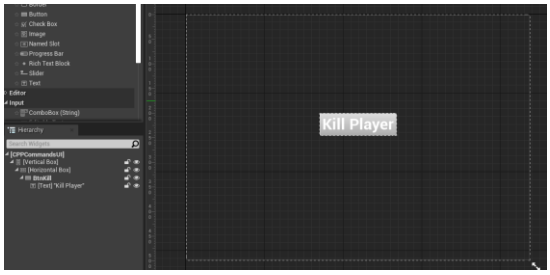
And you should be done with the basic functionalities! Check this by packaging your mod and loading it into the game.

Dummy method

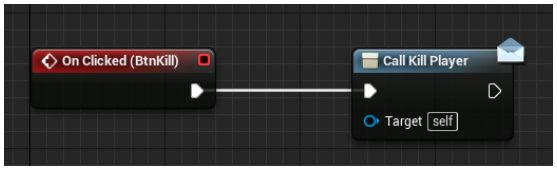
I'm going to run through making a couple of mods which utilise some of the C++ and dummy methods. You should have followed the dummy method section already.

Kill player on button press

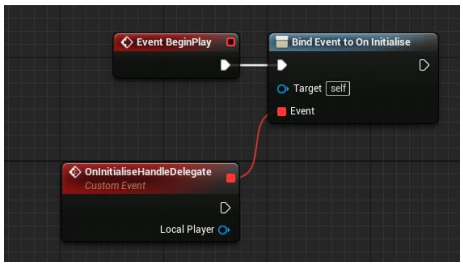
Let's create the mod UI which will have the button in, which kills the player. Name your widget something like KillPlayerUI. Now setup your UI with a button that has text inside that says something like "Kill Player". Since this is a tutorial mod, as usual my widgets look like rubbish but here's how I have mine setup:



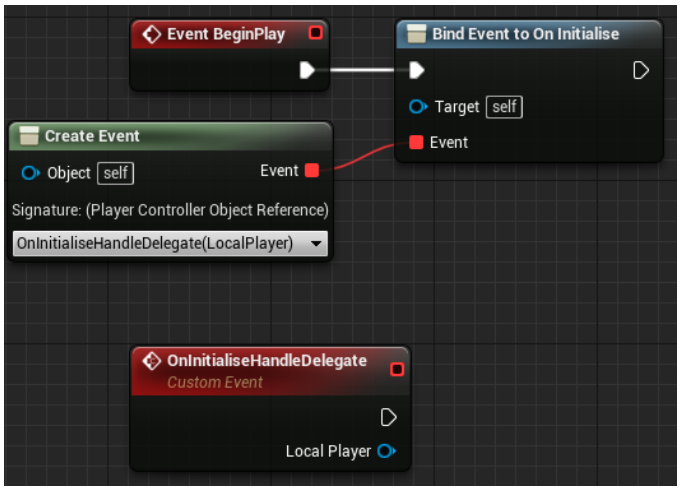
Then inside UI graph, click on the BtnKill variable and click the green plus next to OnClicked event so to create the event. Then create an event dispatcher and call it something sensible, then call it from the On Clicked event. This is nice and easy, but this is what mine looks like:



Now back in your mod, I'm going to show you a slightly neater way of binding events. Instead of binding a new custom event directly onto the bind like this:



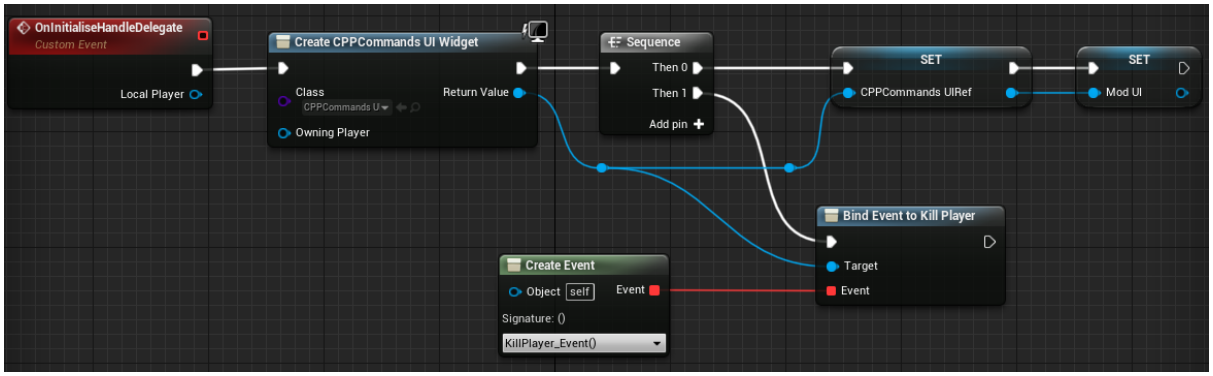
You can also drag off the red pin and type “Create Event”. Note that you should first create the event by dragging off the red pin and typing “Add Custom Event”, naming your custom event, disconnecting the red pin and then moving your custom event node somewhere else. So, the above image will now look like this:



The point of doing this, is that now your custom event execution line can be anywhere in your BP, not just from that bind event node. So, when you get to making more complex mods, this will be super useful for organisation.

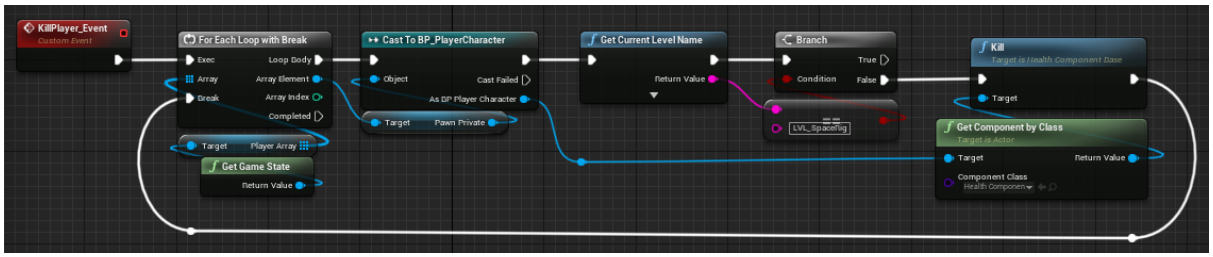
Now from your custom event, whether or not you used the optional tip above, create your KillPlayerUI widget, its reference variable, just like you should have done before. I called my UI “CPPCommandsUI” for some reason, but just to check you should be doing this for your UI name:

Now we want to bind event on that KillPlayer event dispatcher from your UI. I used the tip above again to create an event and have the event execution graph somewhere else, but here’s what it looks like now:

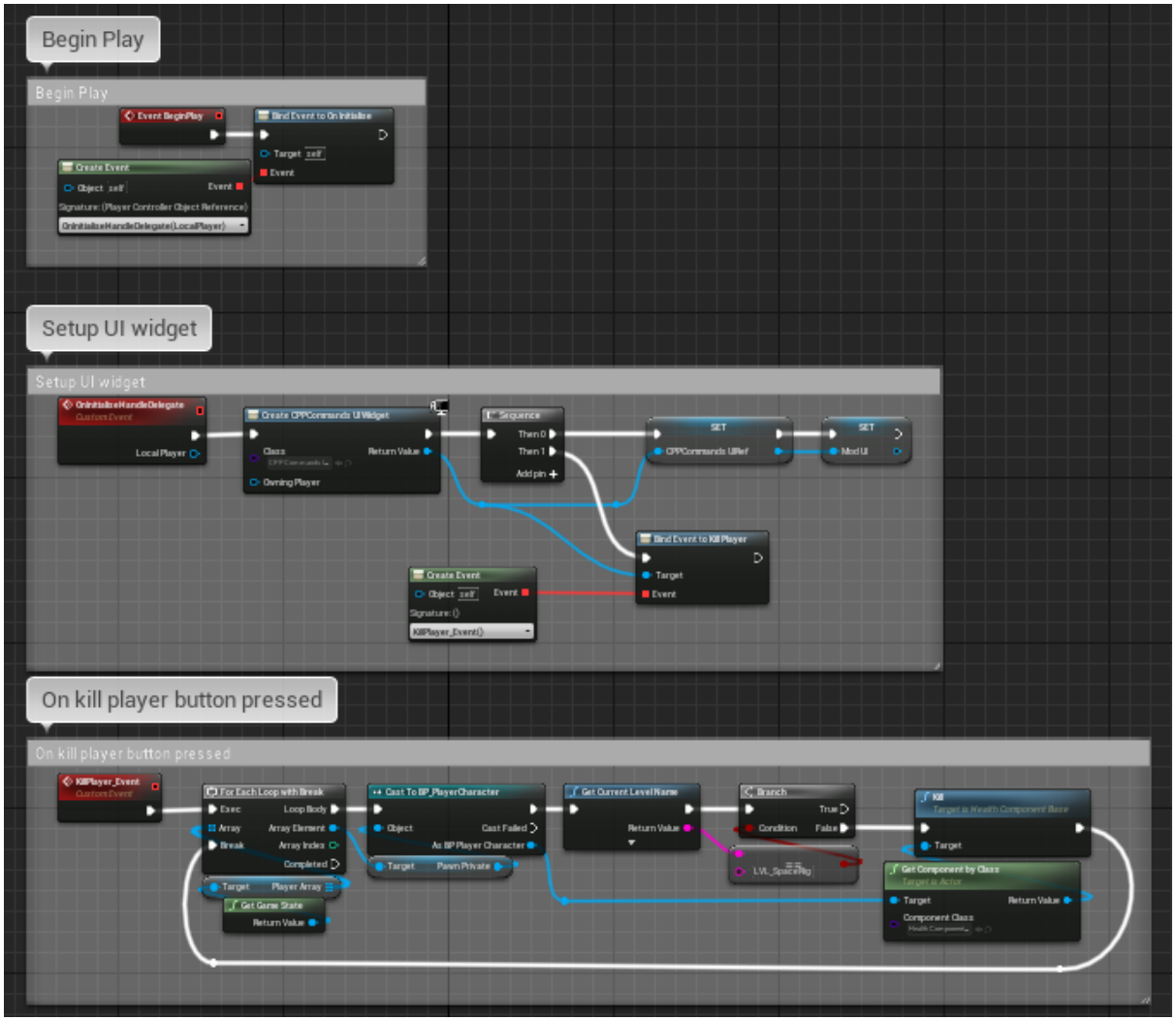


Now, you can package and test this mod, however, let’s add just a tiny new feature that will be useful for your future mods. Let’s say that we don’t want the kill player button to work inside the spacerig.

There is a handy base function (again, from Gameplay Statistics), called “Get Current Level Name”. Use a “==” node from that return value and check it against the string “LVL_SpaceRig”, which is the spacerig’s level name. Plug it into a branch and then if the condition is false, kill the player. So now your event looks like this:



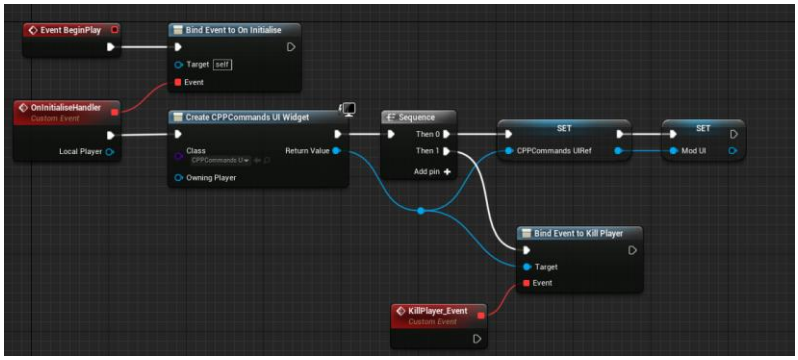
Your mod should be very simple, with a bit of tidy up:



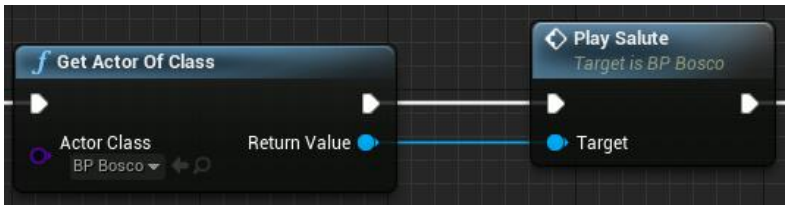
And that’s it done! Test it in-game, and if it works, congratulations on making your first C++ mod!

Self-destruct Bosco on button press

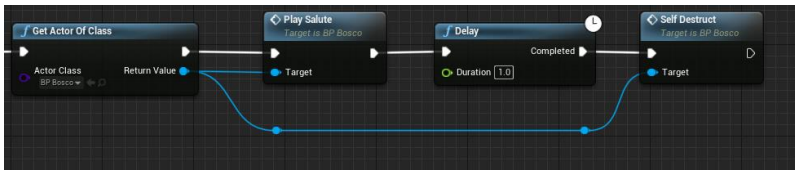
Now, let's create the mod. Make a mod in the same way you have before. If you followed the kill player on button press tutorial, you could just use that same KillPlayerUI widget as it is already setup for the same mod. So basically, copy and paste what you had done before inside the other tutorial mod (remember that I called my KillPlayer widget "CPPCommandsUI" for some reason).



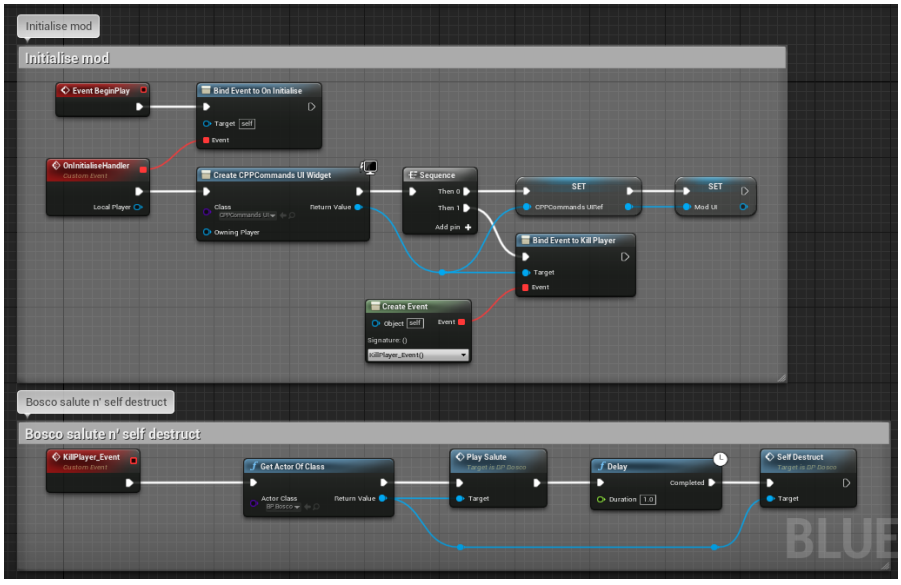
Now, to call on our blueprint, all we have to do is create a node "Get Actor of Class" and select BP_Bosco in the little dropdown. Then drag off the return value and type "Play Salute".



Now, let's put a short delay between saluting and self-destructing, and then call the self-destruct function.



And that's it! A bit of tidy up and your mod should look like this:



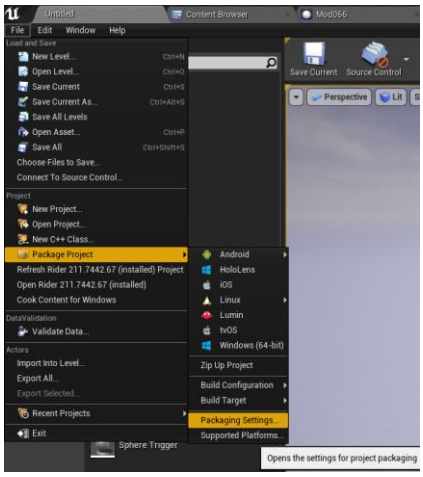
Packaging your mod

When you want to test your mod, you need to package it twice:

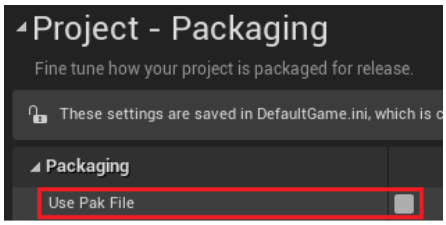
1. From UE. This will turn your assets and classes into the packaged format of .uasset and .uexp files.
2. Then, using the DRGPacker.

From UE

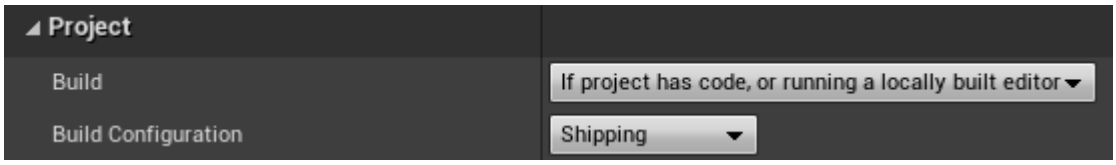
Go into the Untitled tab. Click File -> Package Project -> Packaging Settings.



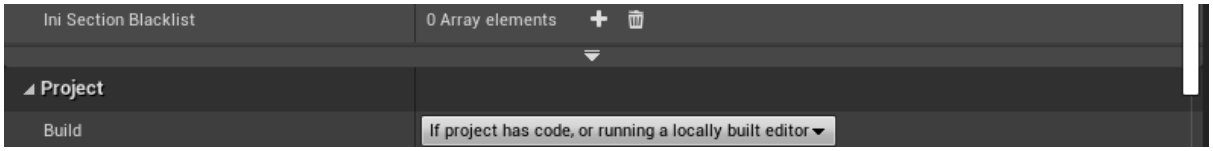
Make sure that this “Use Pak File” option is off.



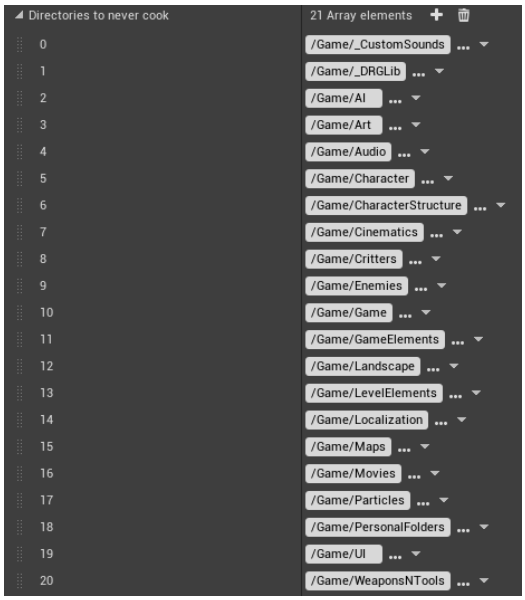
Also make sure that build configuration is set to “Shipping”.



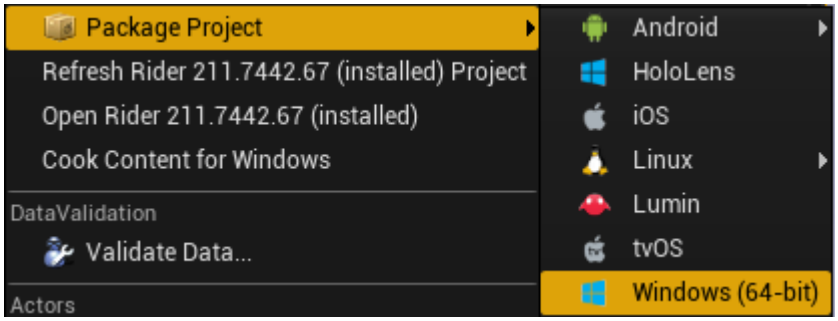
Tip: you can set directories to never package, which is useful when you don’t want to pack say, your _CustomSounds, _DRGLib or any of the dummy BP folders for whatever reason. Although you don’t have to do this, it does mean that you don’t have to delete these files manually within the cooked files, every time. To do this, click the little dropdown arrow in package settings, just above the Project tab



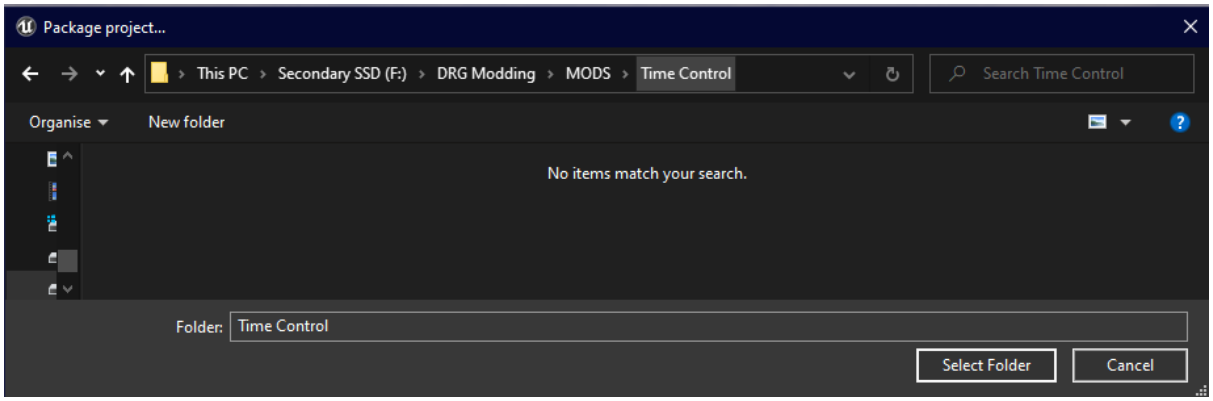
Then in directories to never cook, press the + button to add directories to the array. This is what mine currently looks like:



Now, package your project by clicking File -> Package Project -> Windows (64-bit).



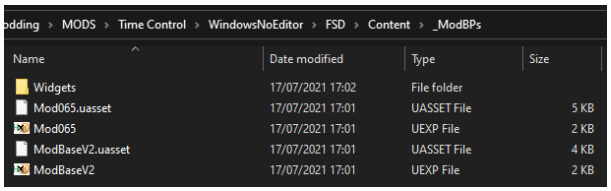
Then selecting a folder to package to:



Tip: if you already have an old cooked folder in there, delete it then click on the parent folder again in the top bar. Because otherwise, since you selected the old folder, it will cook into WindowsNoEditor again, even though you deleted the old one. This sounds confusing so if you just try it yourself you'll understand what I mean.

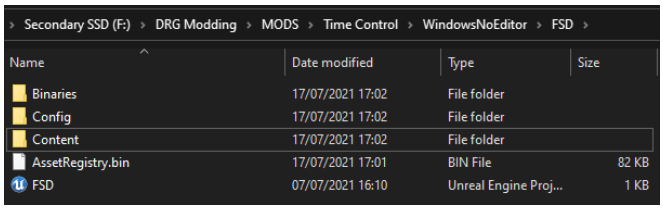
Assuming you didn't do anything wrong, your project should package after about 30 seconds (differs on how large your project is).

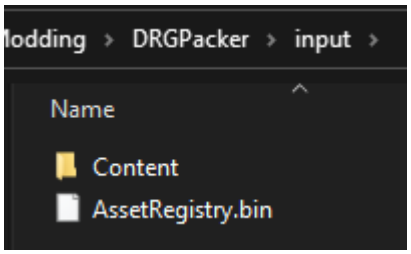
Now, navigate to the packaged files and delete any stuff you don't want to pak (other mods etc.).



Using DRGPacker

If your mod is using native spawning, copy both the Content AND AssetRegistry.bin files, and put them into the input folder inside your DRGPacker. If you are using BPMM legacy method, you only need to copy the Content folder.





Then drag the input folder into _Repack.bat. If you are using native spawning, first rename the .pak to your mod name. Then navigate to DeepRockGalactic\FSD\Mods\ and make a new folder with the same name as your mod. Then put the .pak into that folder.

If you are using BPMM legacy method, put the .pak into DeepRockGalactic\FSD\Content\Paks folder. Remember that the pak name has to end with _P to be loaded by the game from this location.

Depreciated sections that may still be helpful

This section is for depreciated tutorials that used to be main sections in this guide. I suggest that you don't follow these to complete detail, but skim reading them may be helpful for you.

The following sections are depreciated:

- No-dummy BPMM Legacy advanced functionalities
  - This section should not be followed anymore because the system I built here has a fundamental flaw that means that keys have to be remapped every time you load into the game, even though they are saved
  - If you want to have custom keybinds, use DRGLib's keybinding system that handles all of it for you
- Dummying C++ class into project
  - This section is now mostly redundant because of the FSD Template project that has all of the game's C++ classes automatically reflected inside of it, meaning that we can skip this step entirely
  - I am leaving this in because it's still got useful information in like how the reflection is done and how it works

No-dummy BPMM Legacy advanced functionalities

For advanced functionality, what I will show you to do, is:

1. Create your own input action mappings
2. Use those input action mappings to add changeable key-binds for your mod
3. Use these key-binds to change the slider value (so that user doesn't have to open the menu when they want to change the time dilation)

Setting up input action mappings

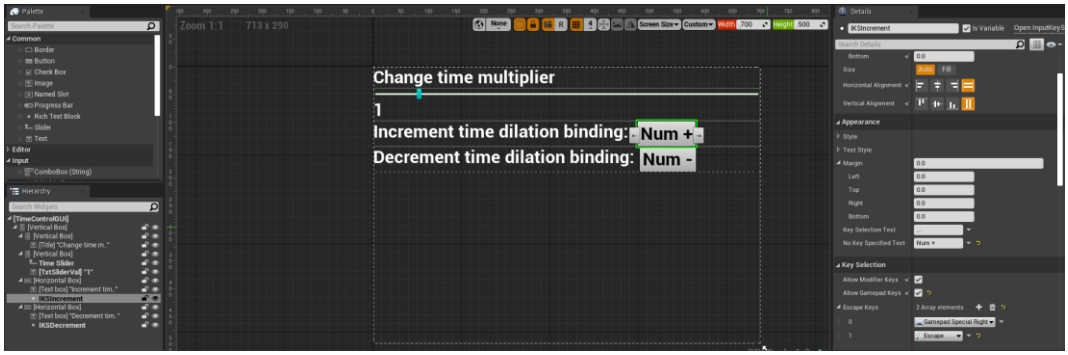
So, let's say that you want to use Numpad + and Numpad - as your mod's default keybinds. You can change this to whatever you want to use (although make sure it doesn't conflict with any of the game's default controls). First, you need to create these bindings as action mappings in your UE project.

Action mappings allow you to map keys to input behaviours by inserting a layer between the input behaviour and the keys that invoke it. Action mappings are fired for key presses and releases. Axis mappings also exist; however these only allow for inputs that have continuous range, sort of like analog controls like the thumb stick on a controller.

Go to your project settings and type "bindings" in the search bar to more easily find them. Under bindings you should see a plus next to Action Mappings. Click this, name it something like "IncreaseTimeControl" and set key to None (or just don't change it). Make another called "DecreaseTimeControl" for the other way. The reason we don't set the key value here is because you'll set your defaults somewhere else.

Adding keybind inputs to the mod widget

Now, we need to add the option to increment and decrement the time dilation keys in your mod UI. Inside TimeControlUI, use text boxes as labels and the nice widget object called Input Key Selector, to make add this feature. Like is what mine looks like:



Inside my IKSIncrement (IKS stands for Input Key Selector), I set No Key Specified Text to the default keys I will be using, Num +. I also added tool tip text to them in the behaviour section. You can do whatever you want here, my widgets are put together quickly so look awful.

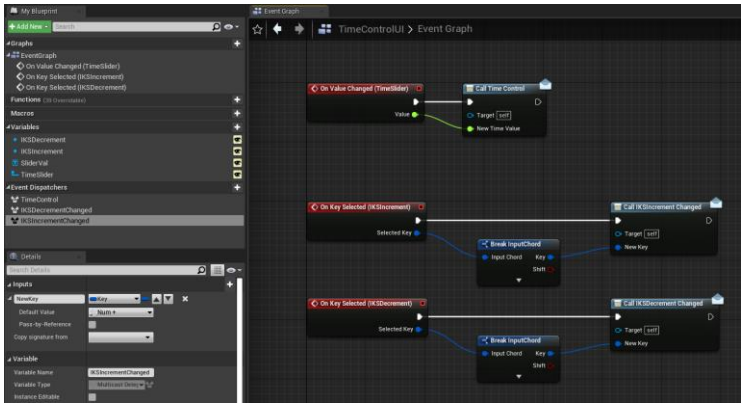
Again, remember to set your IKS' Is Variable checkboxes to true (ticked).

Compile and save, and now, in your graph, click on IKSIncrement and IKSDecrement or whatever you called them, and click the green plus signs next to "On Key Selected". Now you should have two more events in your graph (as On Value Changed should of course still be there).

Now, to take these selected keys and pass them into your mod, you need to create two new event dispatchers, called something like "IKSDecrementChanged" and "IKSIncrementChanged". Inside them, create an input variable called something like "NewKey" and set its type to Key. It should have a default value of whatever your default keybindings you want to be, so in my case increment one should be Num + and decrement one should be Num -.

You will notice that you can't just take selected key output and drag it right into your new key variable. What you need to do, is break the selected key input chord up, and get the key output from that.

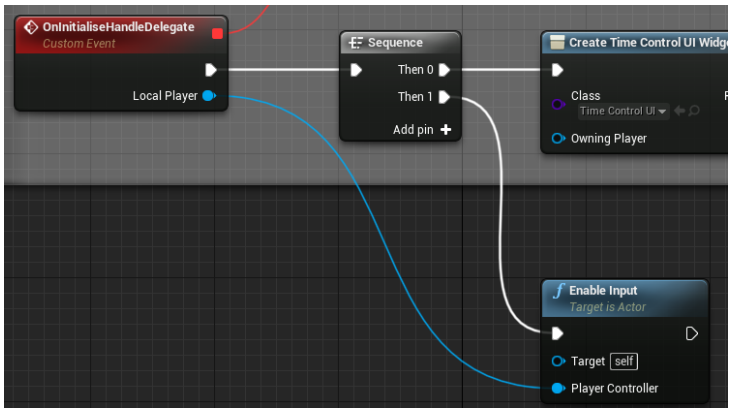
So your TimeControlUI graph should look something like this:



Now that we have done this, we need to work on the hardest part – creating the input action mapping in your mod.

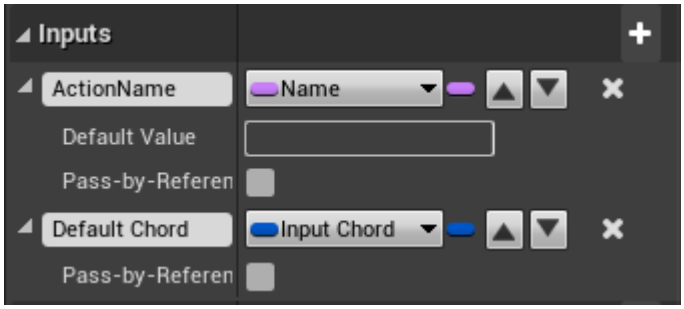
Creating input action maps from keybinding in the widget

When your mod starts up in the game, you need to create two new action maps – one for increasing and one for decreasing. We will initialise all of this before we start the widgets and everything. So, using a sequence node (very useful) between OnInitialiseHandleDelegate and Create Time Control UI Widget nodes, connect the “Then 0” node to a new node called “Enable Input”. This simply turns on the ability to take inputs. The important thing is that you need to plug in local player from oninitialise to enable input. My explanation is probably terrible so here is what you should do:

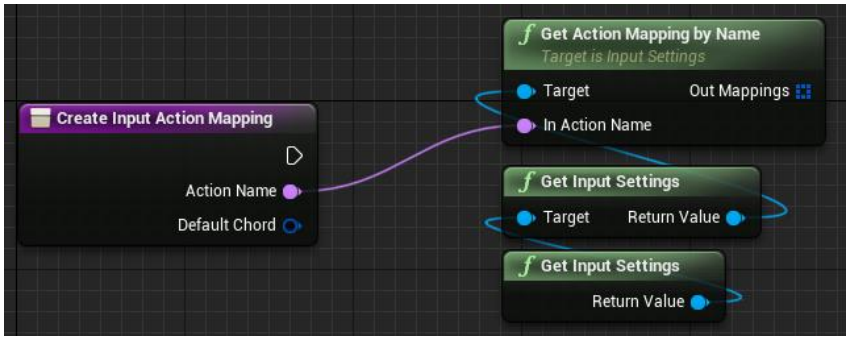


Okay, now we get super complicated. We need to create a function called something like “Create Input Action Mapping”. What this will do, is take a custom named action mapping and a default key and save the action mapping to the game’s key mappings. Luckily, there is nothing custom to DRG required here, so we can just use everything from UE.

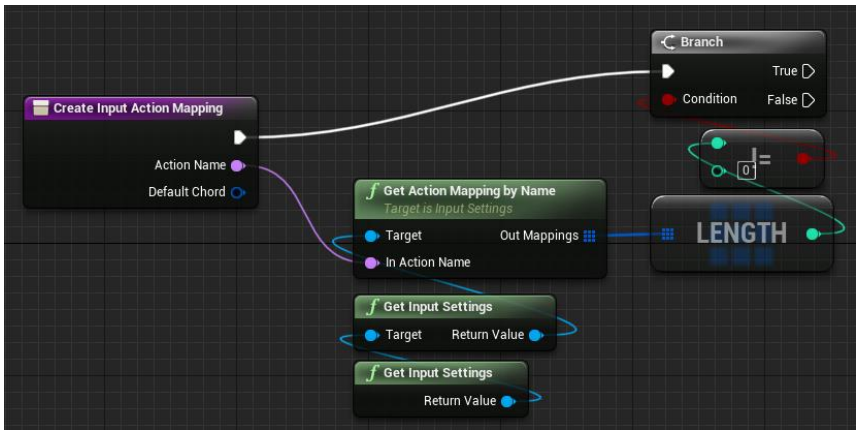
Still, the function is quite complicated, so I will go through step-by-step how to create it. Click on your newly created function and add two new inputs. One called “ActionName” of type Name. Another called “DefaultChord” of type Input Chord.



Inside this function, we first need to get your project input settings, and from that, getting the action mapping by name, which in this case would be by ActionName. To do this, we call a node “Get Input Settings”, then from the return value, another node of the same name (I have no clue why, but this is the only way I got this to work haha), then from that return value, into a node “Get Action Mapping by Name”. It asks for action name input, so use your function’s input for that. If you hover over the nodes, some tooltip will show explaining generally what these nodes do. So now your function should look like this:

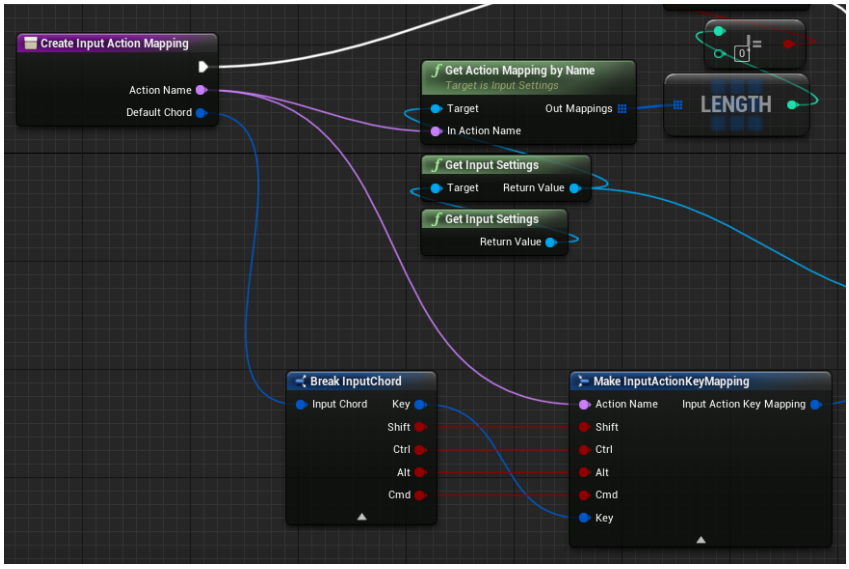


Now, we need to check that the out mappings well, has anything inside of it. This is basically just a sanity check that you haven't skipped making your input mappings inside your project settings. So quickly, we get the length of the array, check that it isn't equal to 0, and then go on from there:



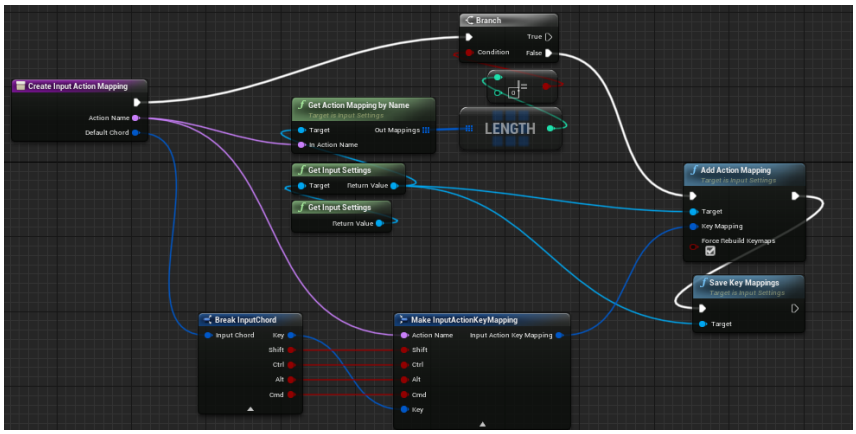
Now, drag off the return value from the second (highest here) Get Input Settings node and make an "Add Action Mapping" node. So the target will be your input settings. Connect the false branch execution path to this new node. If you try to compile your function now, it will throw an error, saying that you need to input a key mapping.

To get this, drag off default chord input and type break and create a "Break InputChord" node. Then drag off key and create a node "Make InputActionKeyMapping". Click the dropdown arrows on these and connect Shift to Shift, Ctrl to Ctrl, etc. Also take another line from action name input to the new make mapping node input. Like this:

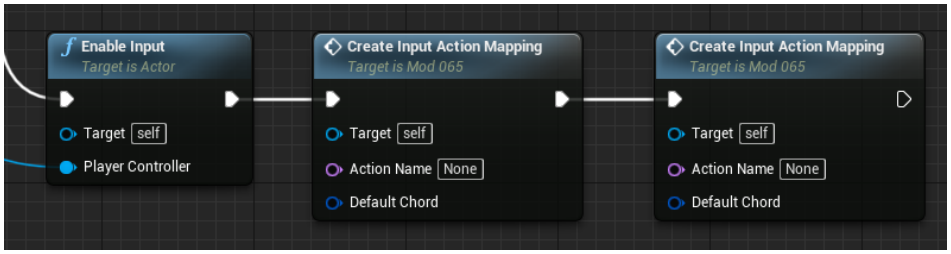


You are probably wondering, “but why can’t I just input a key into the function and avoid all this breaking and making rubbish?”. Well, the problem is that some action mappings may use multiple keys for their key bindings, so you need to take a chord, break it into its member fields, then take those and build up a new action key mapping, using your action name.

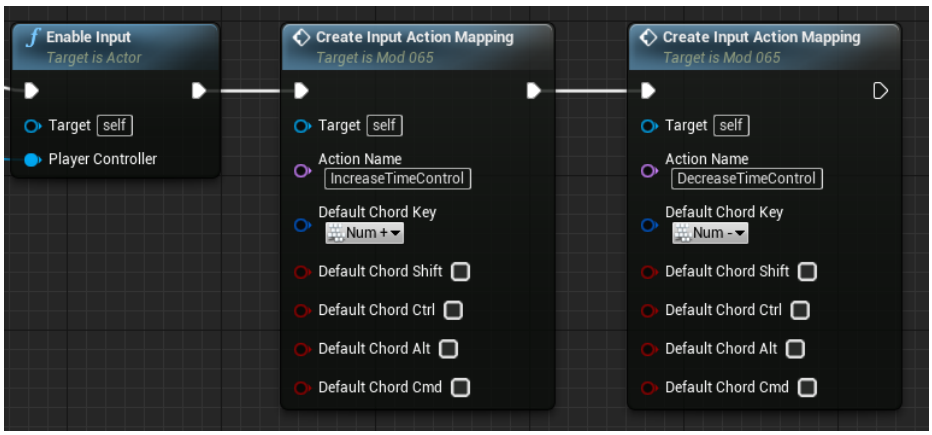
Now connect the output of the make node to the key mapping input on Add Action Mapping. To finish off the function, make a node along the execution line, called “Save Key Mappings”. The target for this, again, is from the second Get Input Settings node. So now, your entire function, should look like this:



Compile and go back into event graph. Drag off Enable Input node and call your Create Input Action Mapping function by typing its name. Then drag off that, and call it again.



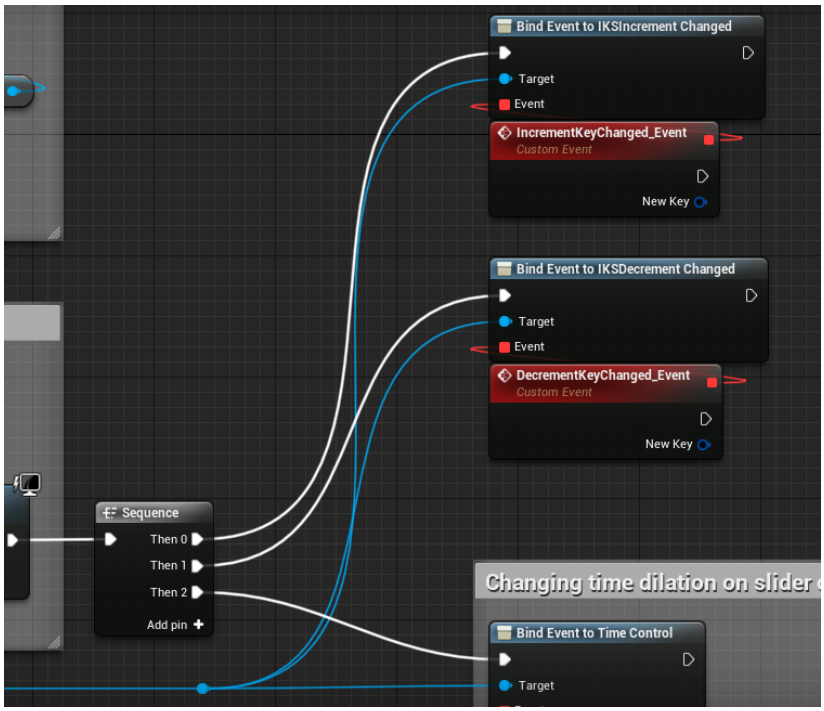
Now, right click Default Chord inputs on both and hit “split struct pin”. This will make the nodes deeper. But now, you can set your default chord key to what you want. So, the first one, set it to your increment chord key. For the second, set it to your decrement chord key. Also, your action names need to be set. You need to set them to the same names as those in your project settings. So, they should now look like this:



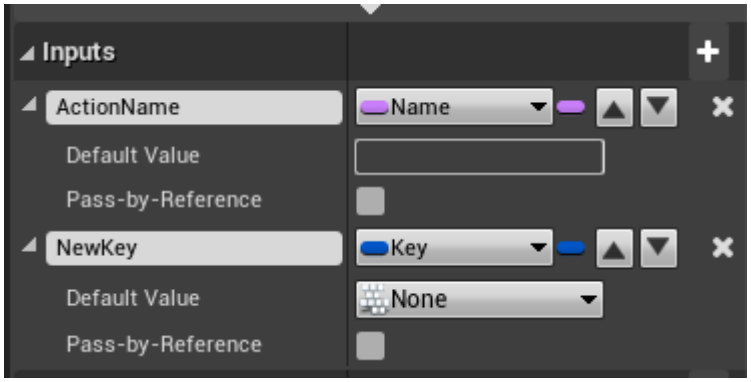
Editing input action maps

The complicated parts aren't over yet. You need to create *another* function for editing your key binds. First though, let's call those IKSIncrement and IKSDecrement Changed events that we created from the UI. We'll sequence execution so that these are called first, then the changing time dilation on slider change, but really it doesn't matter. The sequence nodes mostly keep your project well organised.

Do what we've done before with binding custom events – remember to drag off the time control UI widget return value line (remember you can double click on it to create a branch), then type in you event name. So my BP looks like this:



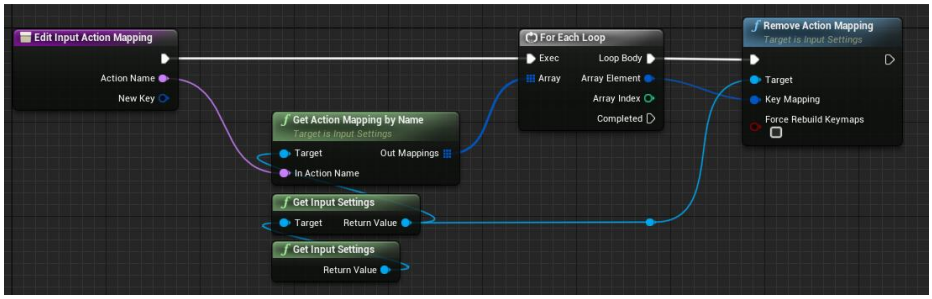
Now comes the second hard part – creating that edit input action mapping we discussed earlier. Make a function called "Edit Input Action Mapping" with two inputs. One called "ActionName" of type Name, the other "NewKey" of type Key. Default value can just be None.



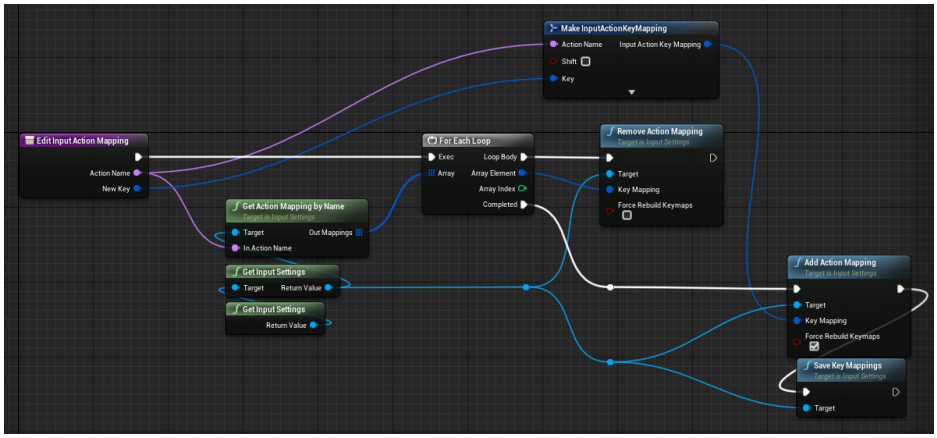
First off, we need to make that group of nodes that leads to Get Action Mapping by Name node, so that we can get those Out Mappings array. What we need to do with this array, is loop through the elements (mappings in the specified action name), and then remove all the action mappings. This then sets us up to add a new action mapping with our key of choice. There are no edit action mapping nodes so we have to do this, and of course we don't want to keep adding new action

mappings on top of each other otherwise we'd have a bunch of keys after many key changes that would all do the same things – this would be an absolute pain.

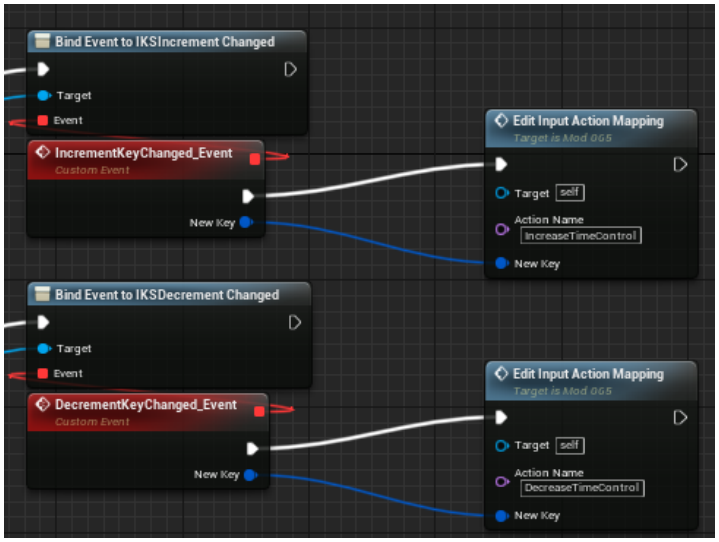
So, drag off Out Mappings and create a for each loop. Inside the loop body, run the Remove Action Mapping node, with target as the return value from the second Get Input Settings node and the Key Mapping the Array Element from the loop. So, your function should currently look like this:



When the for loop has completed, we just add the Add Action Mapping and Save Key Mappings nodes like last time. And since our input key into the function is just a key and not a chord, we can just Make the InputActionKeyMapping from the key and action name and plug that into our key mapping. And so, this is all your function should look like:

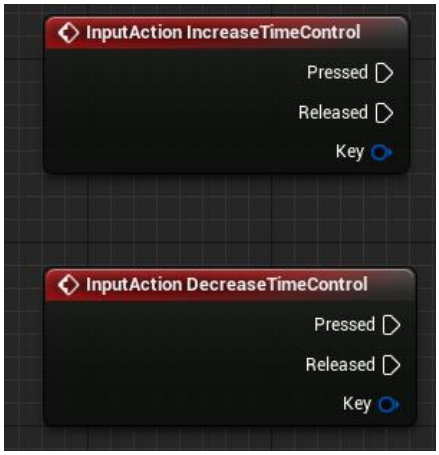


So, back inside your event graph, call this function once for your increment key changed event and another for your decrement key changed event, again using the same action names as your project settings.



Calling input actions and manipulating time dilation from those events

Now we can *finally* get onto the actual editing the time dilation based on when your input keys are pressed, regardless of what the key is mapped to. We can do this easily, as there are nodes categories called Input Actions. And inside there, should be your custom input actions. So right click somewhere and create these two nodes from this, which should look like this:

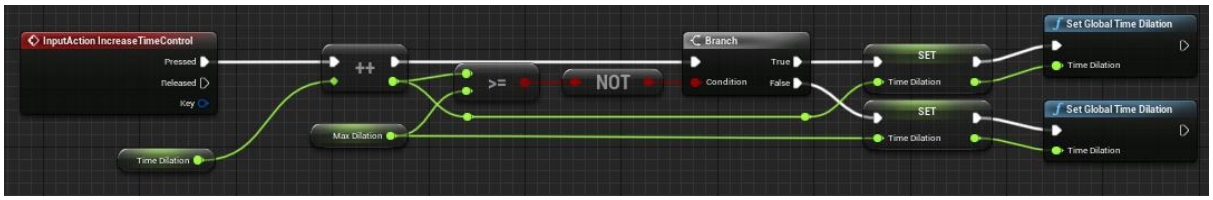


We want to execute our instructions when the key is pressed, so we will be dragging off those. Now, we will be doing some fun maths! (Totally not completely overcomplicated by the pain that is BPs). There are a couple of things to think about first:

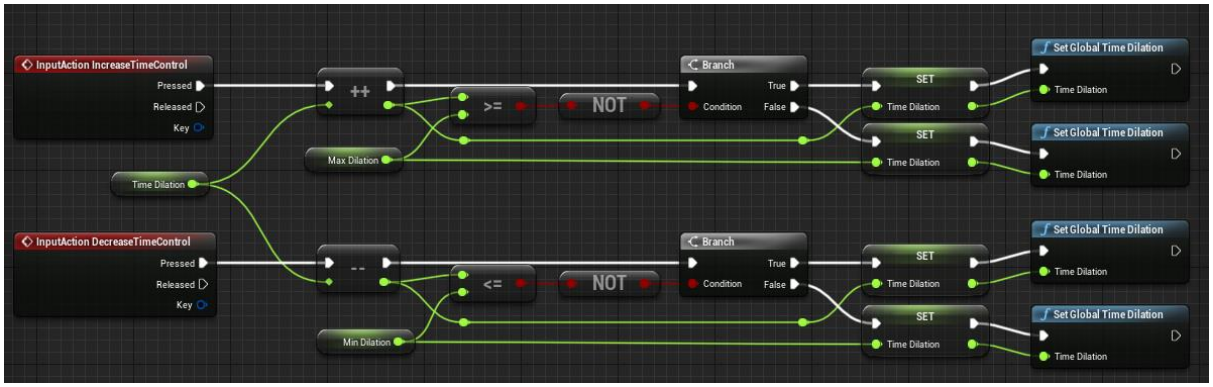
- We need to limit the dilation to a maximum (as any higher than like, 8, *really* lags out the game)
- We need to limit the dilation to a minimum (any lower than 0 is like… why??)
- We need to update the slider value, the text box under the slider and the text box on the HUD as we press the buttons, otherwise there will be number inconsistencies.

So taking in these considerations, let's make a float variables called MaxDilation and MinDilation. This will have our maximum/minimum dilation in a variable, which removes magic numbers and allows us to change this maximum/minimum with user input or whatever much more easily later. Although no idea why you would want to do that, this is just good programming practice.

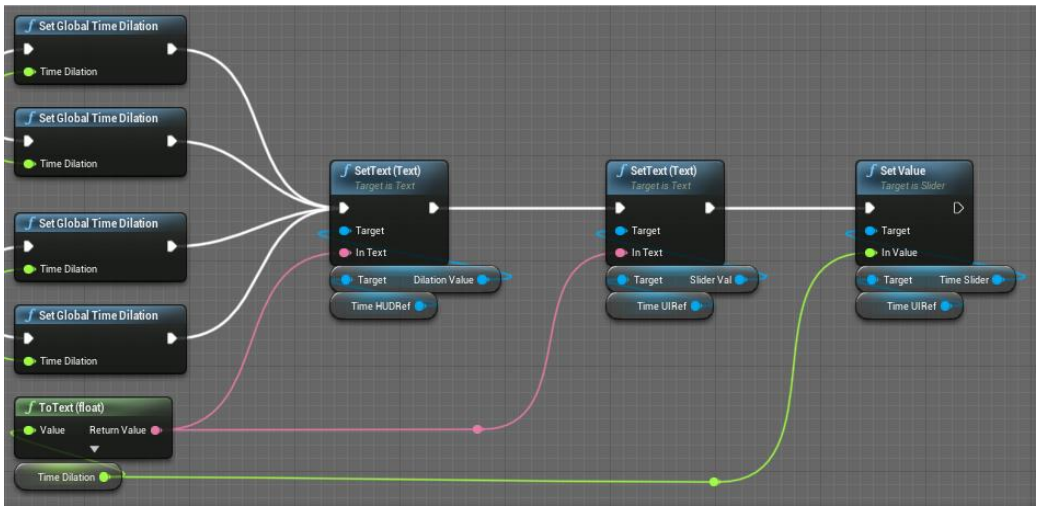
Let's just work on the increment part first, then we can work on decrement, as it will just be a copy and paste but the other way around. So, we start off by getting the value of Time Dilation and incrementing it, with the "++" node. Then, we want to check if the newly incremented value is larger than or equal to max dilation. If it is, we want to set time dilation to max dilation. If not, we set it to our newly incremented value. E.g., if we increment 3.5, we get 4.5, which is smaller than 8, so we keep it at 4.5. If we increment 7.8, we get 8.8, which is greater than 8, so we set it at 8. This way it never gets higher than 8. Then we just set the global time dilation from that same value. The logic for this is then simple:



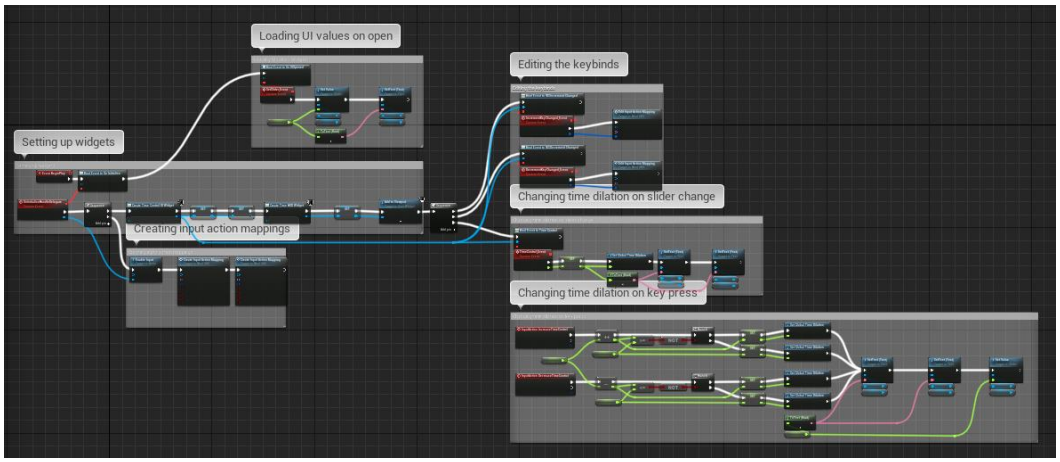
Now for decrement, we do the same thing, but we decrement, then check that it is smaller than or equal to min dilation:



Now, all we need to do is update the various slider/text box values to keep consistency across the mod. Just use the same normal SetText (Text) nodes and the Time Dilation float to string as their text. For setting a slider value, it is just a Set Value node. Then add a comment box over it and it should look like this:



And that should be it! The mod is done! Clean everything up so it is all commented up and readable and you should have something that looks a bit like this:



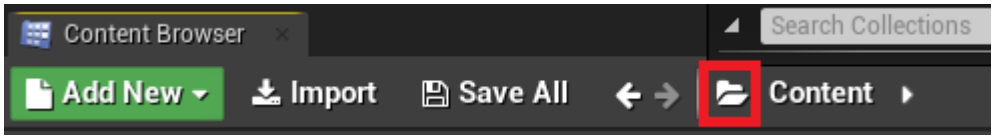
And that's your mod finished! Feel free to add as many new features as you want to this tutorial mod, although please do not publish your mod to mod.io or the modding repo.

Dummying the C++ class into the project

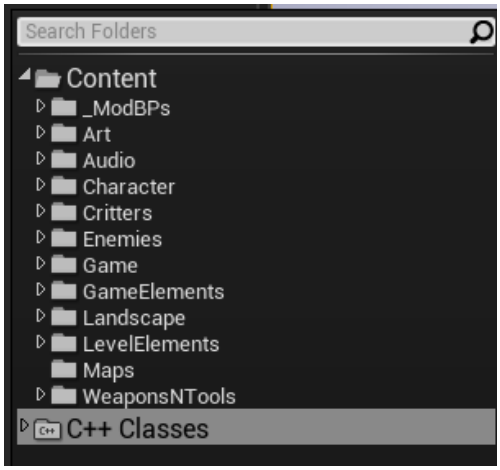
So, now go into your UE project. Remember that your project must be called FSD. Anywhere inside your content browser, right click and click “New C++ class”. Now, it will ask you to choose a parent class. Because our Kill function was in a class that is a child of UActorComponent, you need to select Actor Component as your C++ parent class. If you don’t do this, your dummy class won’t work.

You now must name your C++ class the same name as the struct in the header dumps, minus the ‘U’ on the front. You will have to do the same with the ‘A’ on the front of Actor classes – these letters UE append onto the end during runtime which is why they are in the dumps. So, in our case we name it “HealthComponentBase”. Don’t change the default path that it creates – it should be in /FSD/Source/FSD/.

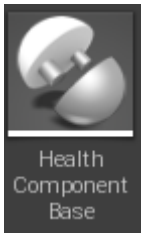
Now, to get to your file, click this folder highlighted in red:



Now you should see a folder called C++ Classes:

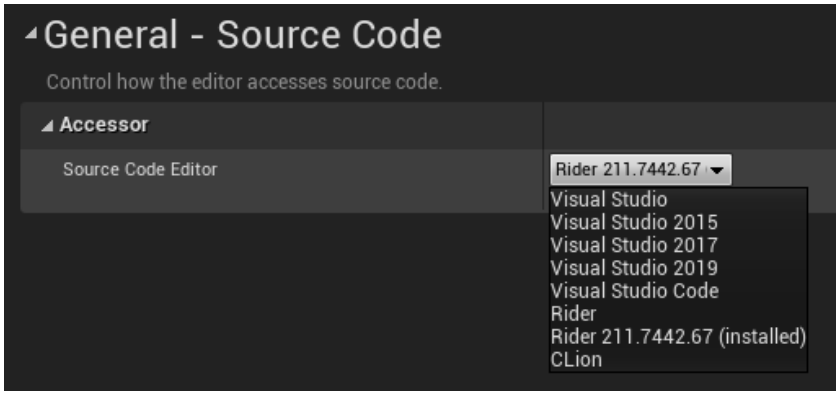


Click on this, navigate into FSD, and your class HealthComponentBase should be there. The icon should look like this:



Double-click this and it will open in your editor.

By default, UE will use Visual Studio as your default editor. To change the editor that it will open, open Editor Preferences, navigate to the Source Code category under General, and there should be an Accessor tab. In that is the dropdown to select your source code editor. At the time of writing, I am using “Rider 211.7442.67” which is Rider for Unreal Engine.



The code you open should be called “HealthComponentBase.h” and should look like this:

```
1   // Fill out your copyright notice in the Description page of Project Settings.
2
3   #pragma once
4
5   #include "CoreMinimal.h"
6   #include "Components/ActorComponent.h"
7   #include "HealthComponentBase.generated.h"
8
9
10  UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
    0 derived blueprint classes
11  class FSD_API UHealthComponentBase : public UActorComponent
12  {
13      GENERATED_BODY()
14
15  public:
16      // Sets default values for this component's properties
17      UHealthComponentBase();
18
19  protected:
20      // Called when the game starts
21      virtual void BeginPlay() override;
22
23  public:
24      // Called every frame
25      virtual void TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction) override;
```

Although of course the syntax highlighting will differ based on what IDE you are in. Some IDEs (Visual Studio for example, as far as I am aware), do not automatically generate this line:

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
```

If it doesn’t, write in this line yourself, in the same place as the screenshot above.

Now, to actually write your dummy C++ function – in our case the Kill() function – navigate to the line under line 25 (virtual void TickComponent…) and type “UFUNCTION(BlueprintCallable)”. Note the lack of semi-colon on this line. This is because we aren’t done with this line yet, however I want to explain what UFUNCTION and BlueprintCallable mean.

UFUNCTION is a C++ function that is recognised by the UE4 reflection system. This means that you can expose any function specified by UFUNCTION to blueprinting, which is what we will want. You can read more about UFUNCTIONs here. BlueprintCallable is a function specifier (which controls how the function will behave inside blueprints), that allows the specified function to be executed in a blueprint. There are other function specifiers, like BlueprintPure, which you may use in the future. You can read more about these function specifiers here.

Now that I have explained what they mean, finish the line by copying in the exact dummy function from the dumps from earlier. However, since we don’t want to provide any parameters into our

dummy function, we can ignore the parameter (struct AActor* DamageCauser). So now your line should look like this:

```
UFUNCTION(BlueprintCallable) void Kill();
```

Completely optionally, some people like to format their UFUNCTION specifiers so that the line above actually looks like this:

```
UFUNCTION(BlueprintCallable)
    void Kill();
```

This is completely up to personal choice how you lay out your code here, and it has no change on function.

Now, inside "HealthComponentBase.cpp", the file created alongside, all you have to do is generate the definition of the Kill() function inside of it. Visual Studio and Rider give the option to do this automatically, but this is how it should look:

```
void UHealthComponentBase::Kill()
{

}
```

Make sure you don't delete anything already in the file.

Now for the painful part, if you haven't setup your IDE properly or whatever. This tends to go wrong a lot for many people, I guess it's just sod's law. But you need to click "Run FSD", or whatever the run button is in your IDE. This will take a bit, as it is building UE from your C++. If any errors are thrown and you are certain you did exactly what I showed here, please do not hesitate to ask for help in #mod-chat as this can be painful without an extra set of eyes. When writing this tutorial, I got a bunch of dumb errors I didn't understand, which ended up being due to some weird memory issues because I had multiple UE projects called FSD open at once. Whoops.

When your build is successful, immediately click the stop button, if your UE is still open. It will then just hot reload UE with the new build. Alternatively, before you build, you can close UE and when you build, it will open UE, running off that build. If you're confused from reading this terrible explanation, you'll work it out eventually :)

Feedback

Hello modder! If you found this guide useful, I invite you to rate it in this form. Feedback is optional, but very welcome.