Project Sunrise

Introduction

Project Sunrise is a blueprint only project that comes with various features to make it easier to setup various gameplay aspects. The features present are:

- Interaction System
- Interaction Camera System
- Signal Emitters & Receivers
- Dialogue System *(Saving data built in)*
- Quest System *(Saving data built in)*
- Actor Manager
- Generic States Save Component
- UI Stack Container System *(uses CommonUI)*

Due to being blueprint only, it makes it easily accessible to see how everything is setup and if necessary, make changes to the base classes. However, because the project has been setup with scalability in mind, normally, the only thing you'll need to do is create a child of the base classes and override some of the core functions.

This documentation will guide you through how to setup and use the systems highlighted above. If you haven't already, be sure to check out the 'Example Map' to see the systems in action.

Setup

If you are using the project as a base for your project, then there isn't anything to do. *(Recommended)* However, if you have migrated to project into an existing project there are a few things you'll need to do to ensure some features work correctly.

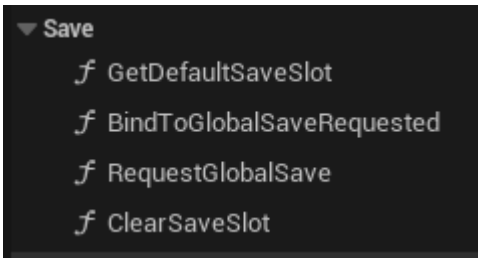
Please note: Whilst this is a comprehensive list of steps, there may still be minor settings/steps that have been missed.

Game Instance

Whilst there isn't much setup in the game instance, but this is used to allow the save options on the in-game menu to function. If you aren't already using a game instance, you can use the 'GI_ProjectSunrise' version and assign it in the project settings under the 'Maps & Modes' section.

If you already have a game instance setup, you will need to copy the functions located in the project sunrise version into your own. Once done, it would be a case of updating the save related functions inside the 'FL_ActorManager' function library to call the ones your own game instance.

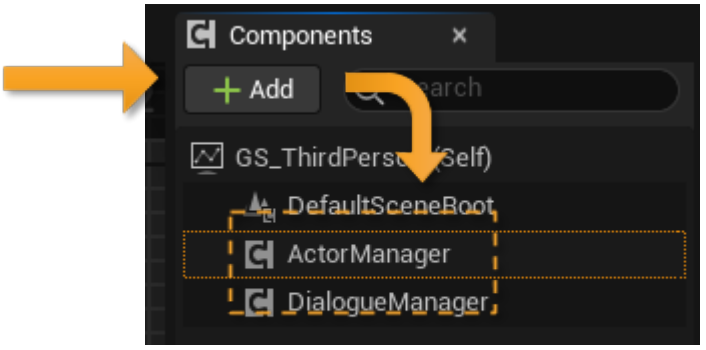
All the systems use the functions inside the function library when needing to call any of the save related functions on the game instance.



Game State

If you already have a game state setup and being used in the game mode, it's just a case of adding two components to it. These are:

- Actor Manager
- Dialogue Manager



If you aren't currently using a custom game state, you can select 'GS_ProjectSunrise' in your game mode.

Actor Manager

The actor manager is used for registering actors (using a unique name) at runtime so they can be easily retrieved. This is used heavily as part of the quest system to ensure quest listeners can easily get the references they need to specific actors.

*The registering of actors via the actor manager has been depreciated. See 'Actor Collection' for replacement system.

In addition to this, the Actor Manager is used as part of the generic states save component that allows you to easily save generic state data.

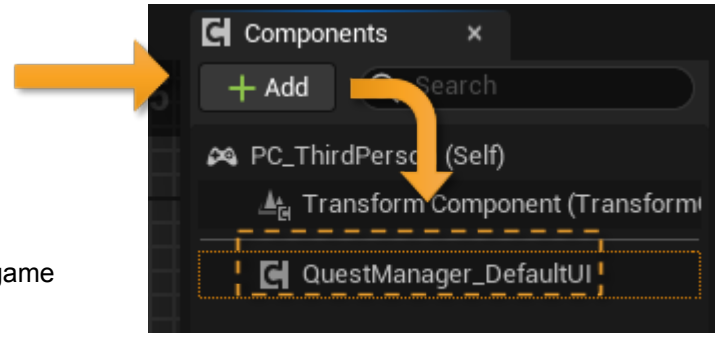
Dialogue Manager

The Dialogue manager is used by the dialogue components and like the actor manager, allows you to get a specific dialogue component based on its dialogue tag. Additionally, this manager handles saving and loading dialogue data. There are only two settings on these components that are configurable and that's related to saving. Feel free to tweak these if you need to.

Player Controller

Like the game state, if you already have a player controller setup, you will need to add a component to it. This is:

- Quest Manager Default UI



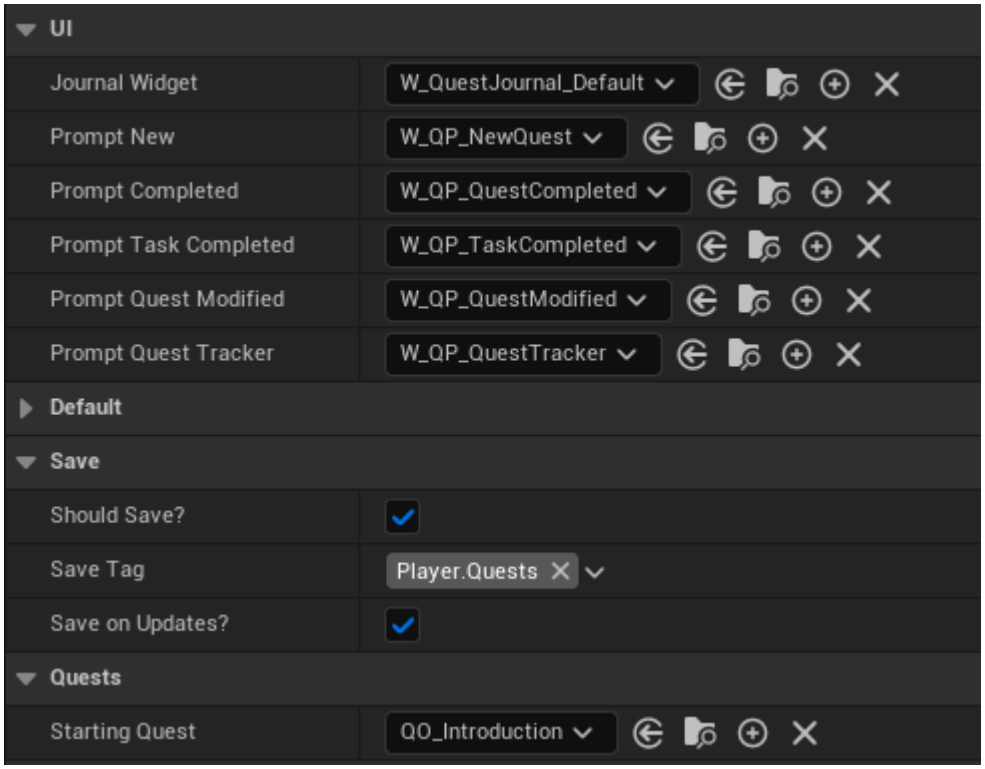
Alternatively, you can select 'PC_ProjectSunrise' in your game mode.

Quest Manager

Quest Manager Default UI is a child of Quest Manager where it has been setup to handle the various quest related UI prompts as well as the quest journal that shows active quests and completed quests.

If you wish to handle the UI side of the quest manager differently, you will have to create a child of 'Quest Manager' and instead place the child you have created on to the player controller.

If you are using the quest manager with the default UI, there are options relating to saving as well as the widgets it will use for the various prompts and journals. You can create your own children of the base widgets that are used but this will be explained in a different section. You will need to select the widgets to use as these aren't assigned by default. Below is an example of how it is setup in the example map should you wish to use the default prompts provided.

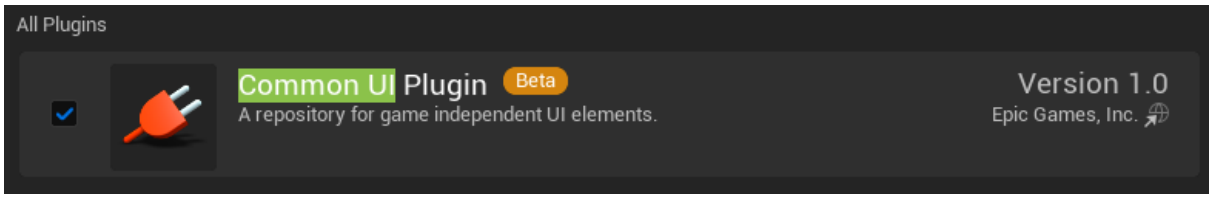


Regarding the 'Save Tag', this is a gameplay tag that is used to denote the folder structure and name of the save file that will be created when saving to disc. 'Player.Quests' will result in the save file being called 'Quests' and will be placed in a folder called 'Player'. If you wish to change the folder structure, you can add a new gameplay tag and select it here.

HUD

All the UI elements setup in the project utilize common UI and the widget stacks it provides. Because of this, you will need to make sure the 'Common UI Plugin' is enabled for your project if you have migrated the assets to an existing project.

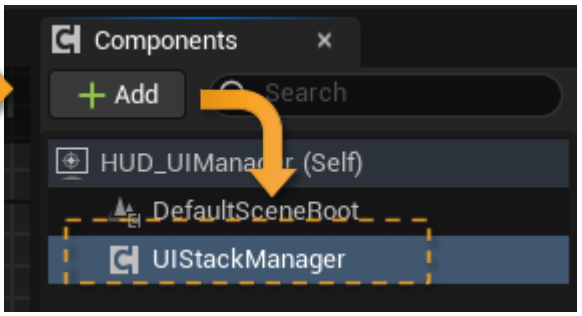
If you wish to handle all the UI differently than what is provided in the project, you can ignore this step but be aware that some of the classes utilizing the UI system setup for this project will have errors at runtime.



To ensure the UI functions correctly, you will need to place the below component onto the HUD class.

- UI Stack Manager

If you aren't currently using a HUD class, you can select 'HUD_UIManager' in your game mode.



UI Stack Manager

The UI Stack Manager handles the various stack containers we use to enable use to push widgets to a stack.

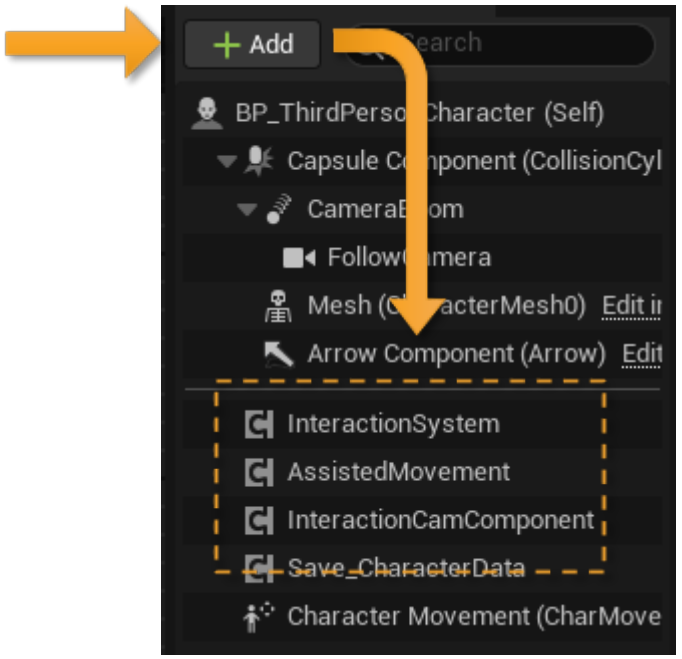
In addition to this, it also manages tagged slots which allows you to push an activatable widget to a slot without having to get a direct reference to the specific widget container.

The only variable on this component you can change is the default HUD widget. For now, you can just select 'W_MainHud'. This widget is already setup to handle the various UI elements used in the project. You can always change this later when you decide to make your own.

Player Character

If you are using a preexisting character, there are three components that will need to be added to enable the interaction system to work correctly. These are:

- Assisted Movement
- Interaction System Default UI
- Interaction Cam Component
- Save_CharacterData



Assisted Movement

The assisted movement component is used by the interaction system to help move the player character during an interaction.

There's nothing stopping you from utilizing this in other places if you need to move the player character.

The only thing you can change is the 'Find Nav Mesh EQ'. This is used to help getting the player character onto the nav mesh before attempting to use the nav mesh for any movement. If you wish to change how the location is determined, you can create your own EQS and select it here.

Depending on your character setup, you might benefit from enabling 'Use Acceleration for Paths' on the character movement component. This is because the assisted movement simulates movement inputs when performing short movements off the nav mesh. Using acceleration for paths helps keep the movement speeds more consistent. This is something to play around with to see what your happy with.

Interaction System

The Interaction system component is the main component for allowing the player to interact with interactables in the level. The default UI version is a child of the base interaction system that handles showing the interact prompts. If you wish to handle this differently, you can create a child interaction system and override the 'UI_ShowPrompt' and 'UI_RemovePrompt' functions.

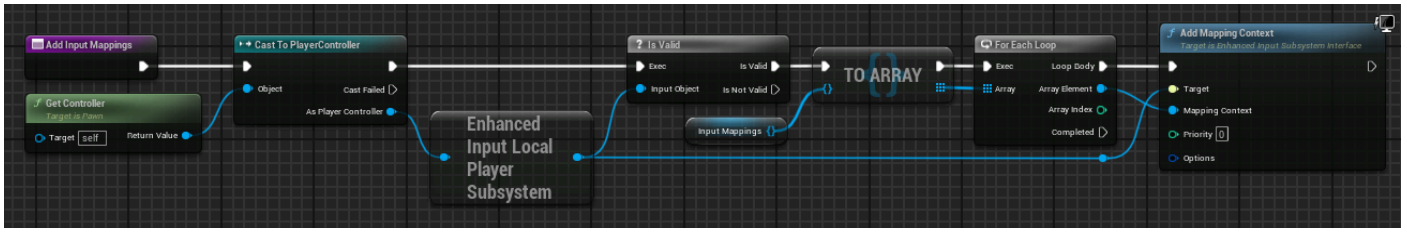
There are some settings under the 'Trace' category that you can tweak to your liking. By default, the system will trace from the camera into the world looking for interactables. You could change this to trace from the character (owner) if you required. Beyond this, there are some settings for change the trace distances and radii. If you wish to see the traces, you can enable this using the 'Trace Debug Type'.

With the interaction component added, we now need to setup the inputs. The character used in the project uses 4 input context mappings. We separate various inputs into their own input context mappings so that we can easily add and remove various inputs using the interaction system. For example, we can remove the movement input context mapping during an interaction to prevent the player from moving but leave the look input context mapping so the player can still look around. You can have a play around with how you want these to be handled depending on how you wish your interactions to be setup.

The 'IMC_Interaction' contains the input actions just for interacting. As the interaction system supports different keys triggering different events in an interactable, we need to have these defined in the character. The one provided in the project are as follows:

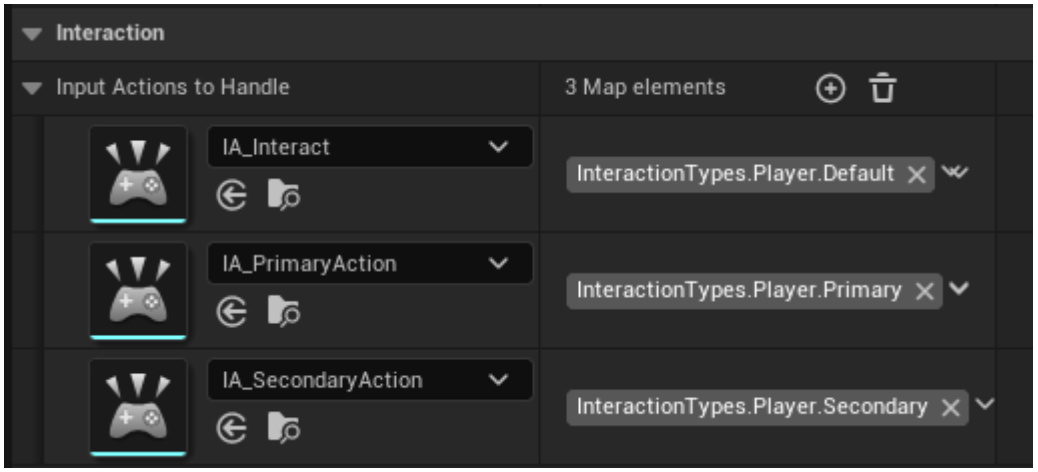
- Interact: E Key
- Primary Action: Left Mouse Button
- Secondary Action: Right Mouse Button

After setting up the input context mappings, we need to make sure they are being added to the Enhanced Input Local Player Subsystem on begin play. If you've seen the character setups in the templates provided by Epic, we can do something similar. Alternatively, you can use a function as shown below. This function allows you to specify multiple input context mappings (set variable) and add them all to the subsystem.



Once you've decided on your inputs (you can always add more later), we need to set them up to work with the interaction system.

This is as simple as adding the input action event to the event graph and then calling the 'Initiate Interact' function on the interaction system. On this function, we just need to connect to input action that is returned on the input action event. With that done, we then need to denote on the interaction system component what input actions it should handle and what the associated interaction type tag it should have. This is done in the 'Input Actions to Handle' variable on the interaction system component.



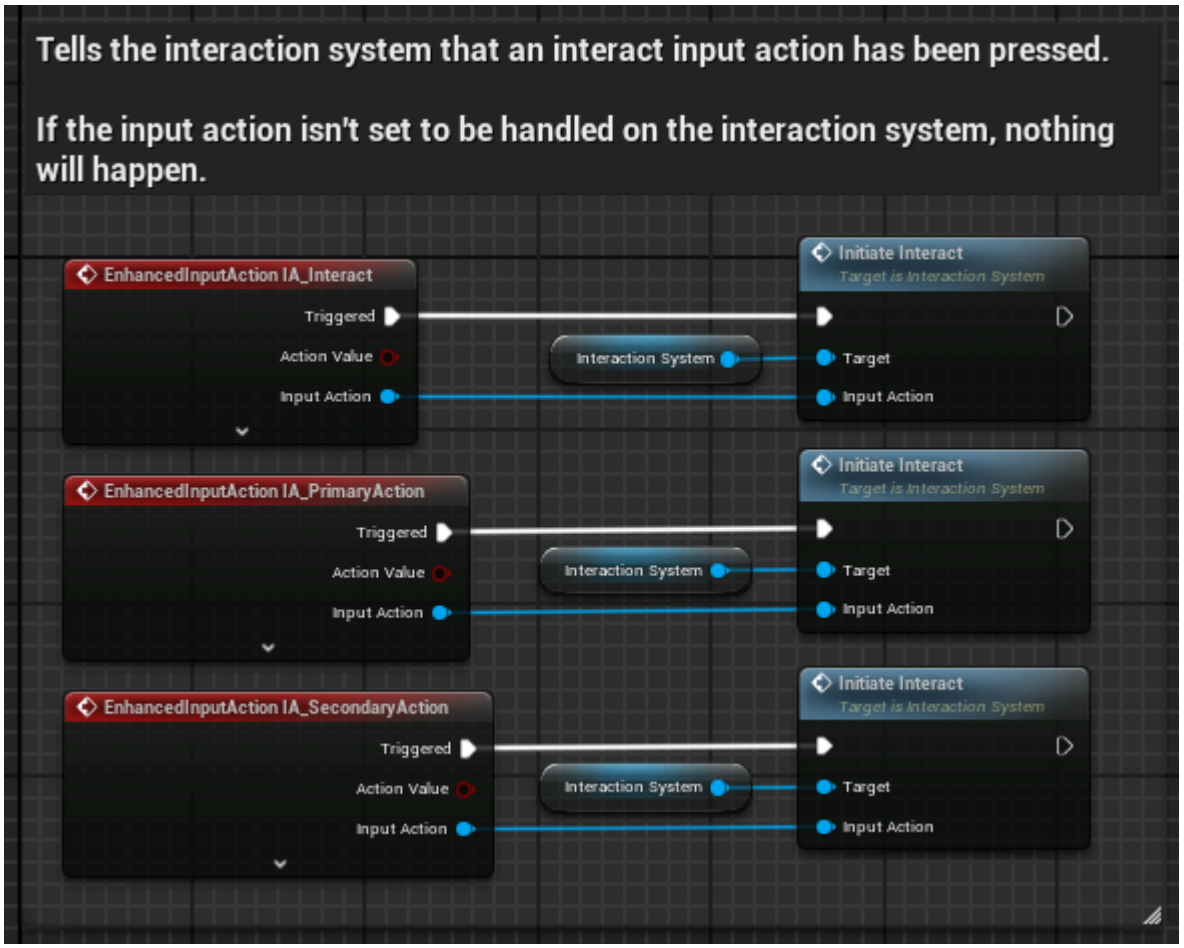
These can be updated/changed at runtime which can affect what interaction events the player can handle at specific times.

If you wish to have more than 3, create an input action for it and add it to the interaction input context mapping before then setting up the input action in your character. Below is an example of the 3 input actions already setup in the project.

As a reminder, you can assign whatever keys you desire to the specific input actions.

If you are unfamiliar with the Enhanced Input system, you can read through the documentation provided by Epic for more information.

https://dev.epicgames.com/documentation/en-us/unreal-engine/enhanced-input-in-unreal-engine



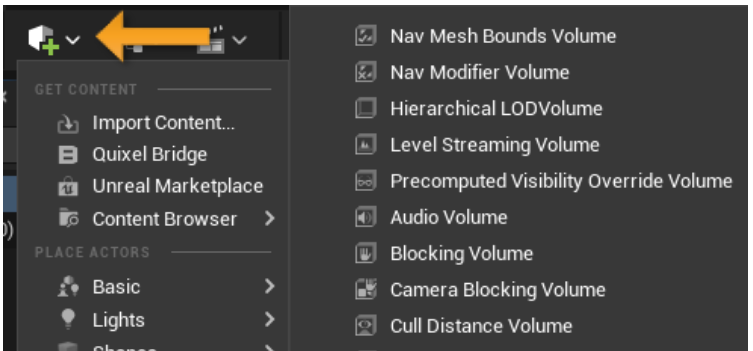
Interaction Camera System

Other than adding this component to the character, there's nothing else you need to do or configure for this to function. It just needs to be present on the character.

Post Processing Volume

As part of the interaction system, we use an outline effect to highlight the interactables. If you don't wish to use this, you can ignore the following steps.

For this to function, the level must be setup to use the relevant post process material. As well as having stencil custom depth enabled in the project settings.

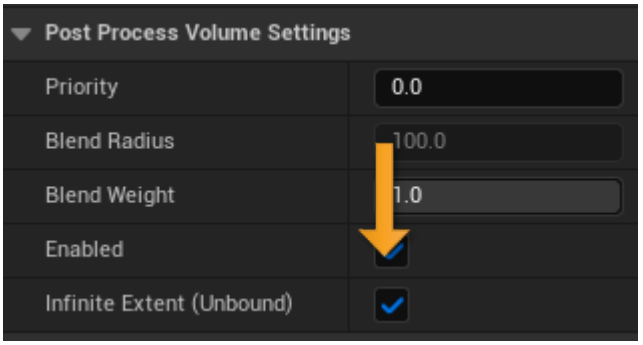


If you don't already have a post process volume in your level, first start by adding one.

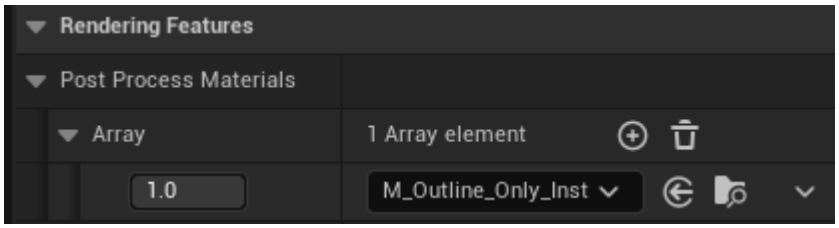
Once this has been added to the level, with the volume selected, go to the detail panel and locate the 'Post Process Volume Settings'. Under this

section, make sure that ‘Infinite Extent (Unbound)’ is ticked. (true)

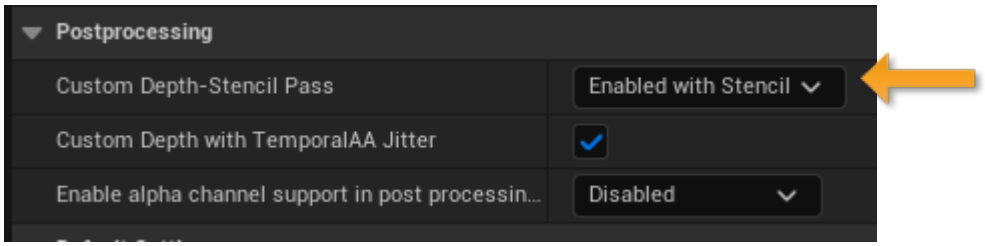
This is used to make sure the outline effect will happen regardless of where in the level the character is.



With the volume in the level, we can now add the post process material that will handle the outline effect. To do this, with the volume still selected, in the details panel, locate the ‘Rendering Features’ section. Under here, we want to add a new element to the ‘Post Process Materials’ and select ‘M_Outline_Only_Inst’ from the drop-down menu. This is the material that will apply the outline effect using stencil custom depth.



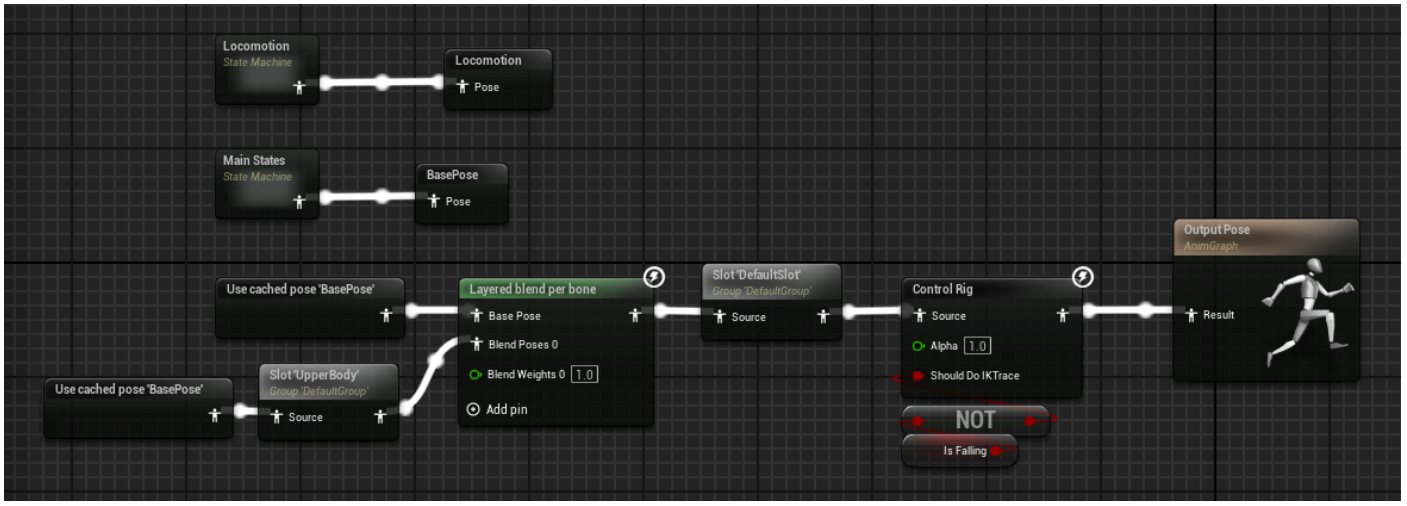
For the outline to be correctly applied, we need to make sure that custom depth stencil pass is enabled in the project settings. This can be located under the rendering section.



Animation Blueprint

Whilst project sunrise doesn’t do much in the way of animations, it does however require the animation blueprint to be setup to allow for montages to play. For this we must make sure that slots have been added to the anim graph.

The below is an example of the minor modifications made to the Manny animation blueprint used for Manny and Quinn. These changes will allow the character to play full body montages (default slot) and upper body montages (upper body slot). The layer blend by bone, blends from the ‘Pelvis’ of the skeleton.



As your project grows, chances are further changes will be made to the animation blueprint to facilitate your animation needs. As this happens just be aware of the slots that are required to allow montages to play as part of the interaction and dialogue systems.

It is recommended to try use the same skeleton where possible for humanoid characters so that animations (such as montages) can more easily be used and shared between characters.

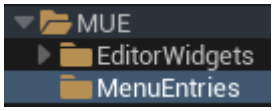
Save Character Data

The ‘Save_CharacterData’ component is used to save the transform of the player character, the control rotation, and the handled input actions on the interaction component. There’s nothing to configure with this component so it just needs to be on the character. If you wish to handle saving the character data differently, you can omit it.

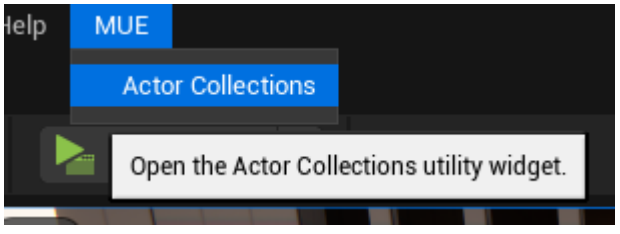
Actor Collection System

The Actor Collection System replaces the ability to register actors using a name or gameplay tag that can then be easily retrieved at runtime.

This system is self-contained and doesn’t require any setup to use. However, it does come with editor utility widgets that makes using it easier. Locate the ‘MUE’ folder inside of project sunrise and run all the files inside the ‘Menu Entries’ folder.



Once complete, this will give you a menu at the top that can be used to open the actor collection utility widget when you need to use it.



Startup

If you don’t wish to run the above each time you open the project, you can add some lines to the ‘DefaultEditorPerProjectUserSettings.ini’ config file that will automatically run the files when opening the project.

```
[/Script/Blutility.EditorUtilitySubsystem]
+StartupObjects=/Game/ProjectSunrise/MUE/MenuEntries/MenuEntry_MUE.MenuEntry_MUE
```

+StartupObjects=/Game/ProjectSunrise/MUE/MenuEntries/MenuEntry_ActorCollections.MenuEntry_ActorCollections

The above is already included in the Project Sunrise template.

Setup Conclusion

With the setup done, the systems provided in project sunrise should be functional within your existing project. For best results however, it is recommended that you use the project as base when starting a new project.

Creating a Simple Interaction

With the systems setup, the first thing we want to do is setup an interaction. Interactions are an important aspect of any game as this is what allows the player to (as the name suggests) interact with the world you create.

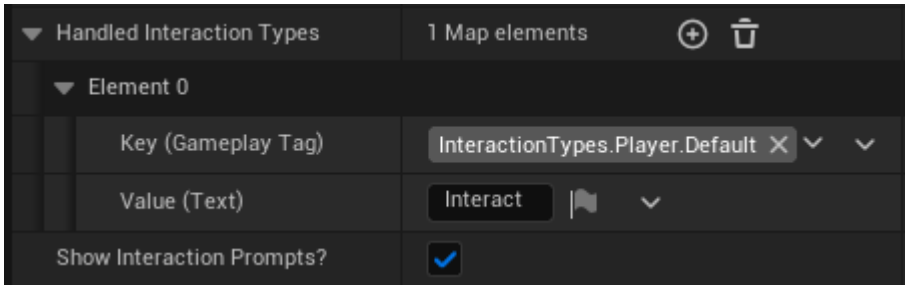
The first thing to do is to add the ‘Interactable’ component onto an actor you wish to be interactable. With the component selected, you will have access to many configurable settings specific to the actor, some of which you can event tweak after it’s been placed in the level on a per instance level.

For something to be interactable, the actor must contain a component that has collision such as a static mesh.

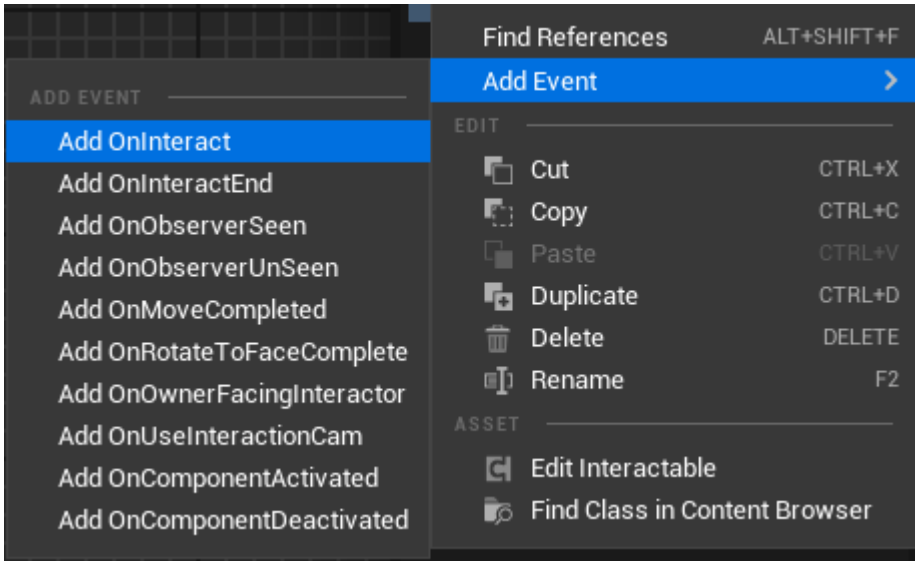
With the default settings, when you look at the actor in the level, it should show the outline effect and interaction prompt (without any text).

Next, we want to specify the interaction types this actor will handle. For this, with the interactable component selected, in the details panel locate the ‘Handled Interaction Types’ var and add a new element. For this you will need to provide the gameplay tag associated with the input you’re wanting to use. In the example below, we use ‘Default’ of which is for the E key. (if you’ve changed the input keys this might be different) You can add multiple interaction types if you want different things to happen depending on the key pressed.

In addition to this, you can specify the text you would like to be displayed in the interaction prompt. If you don’t want the prompt, you can leave this blank and untick the ‘Show Interaction Prompts?’.

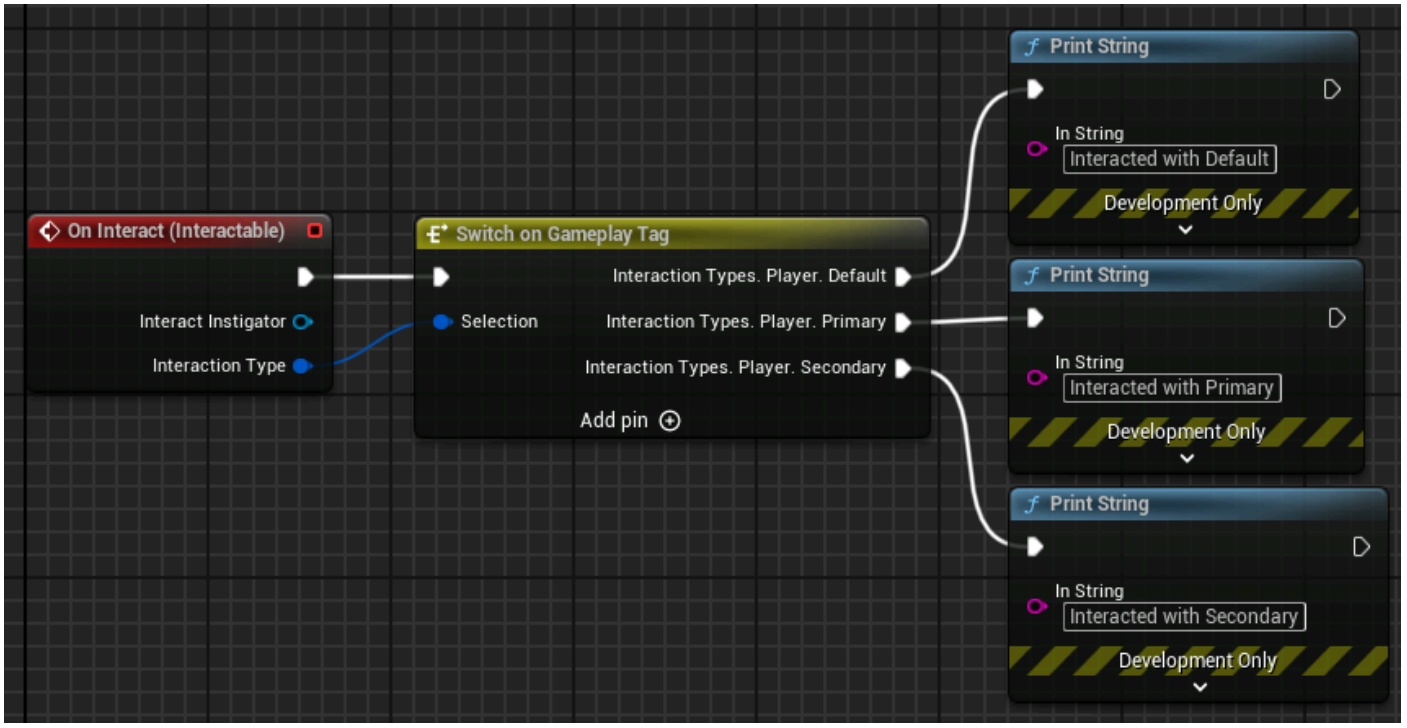


With the interaction types setup, we now need to actually handle them. For this, we need to right click on the Interactable component and go to ‘Add Event’ and then select ‘Add OnInteract’.



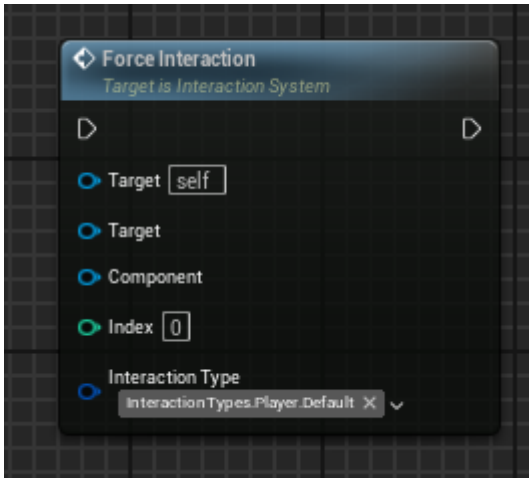
This will then add an event to the event graph like below. This event provides a reference to the actor that has performed the interact (normally the player character) as well as the interaction type that triggered the interaction. From here it’s a case of setting up the logic for what you want to happen when the interaction is triggered.

If you are handling multiple interaction types in the actor, you can pull from the ‘Interact Type’ output on the event and add a switch on gameplay tag. It would then be a case of adding the interaction types to the switch. This will allow you to control what happens depending on the interaction type.



Force Interaction

On the interaction system component, you can call the ‘Force Interaction’ function if you wish to force the player character to interact with a specific actor. An example of how this can be used can be located in the ‘QT_Intro_03_SpeakToSunrise’ quest task. When this task is completed, it forces the player character to interact with a door. This method allows you to run it through the interaction system and receive some of the benefits of doing so, such as movement and montages.

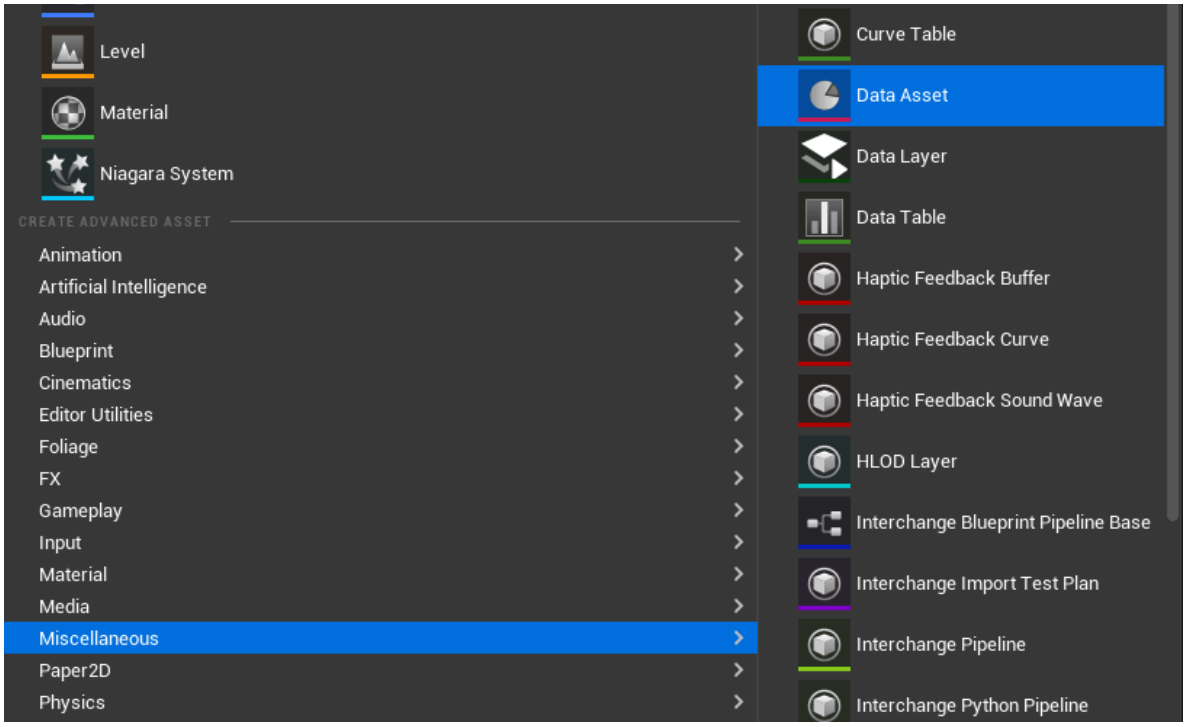


You can specify the component/index that might be required to perform the interaction if the desired actor has been setup to use it. (based on what the player is looking at)

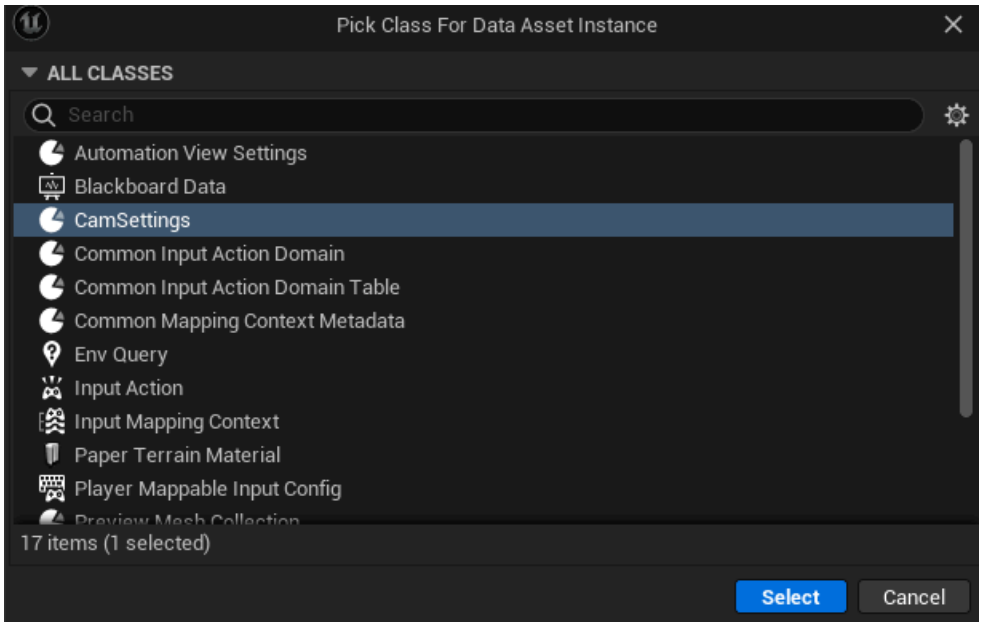
Creating an Interaction Cam

When interacting (Or during a conversation), you might want to use an interaction cam. This allows you to create a simple camera change just by specifying a few settings. The interaction camera system works using a spline. It’s start and end points are set based on the location of the player and the thing you interact with. It then uses a ‘Cam Settings’ data asset to change where along the spline the camera should be placed and the length of the spring arm it is on. There are other settings as well, but you can make as many of these as you like for the types of camera settings you might want.

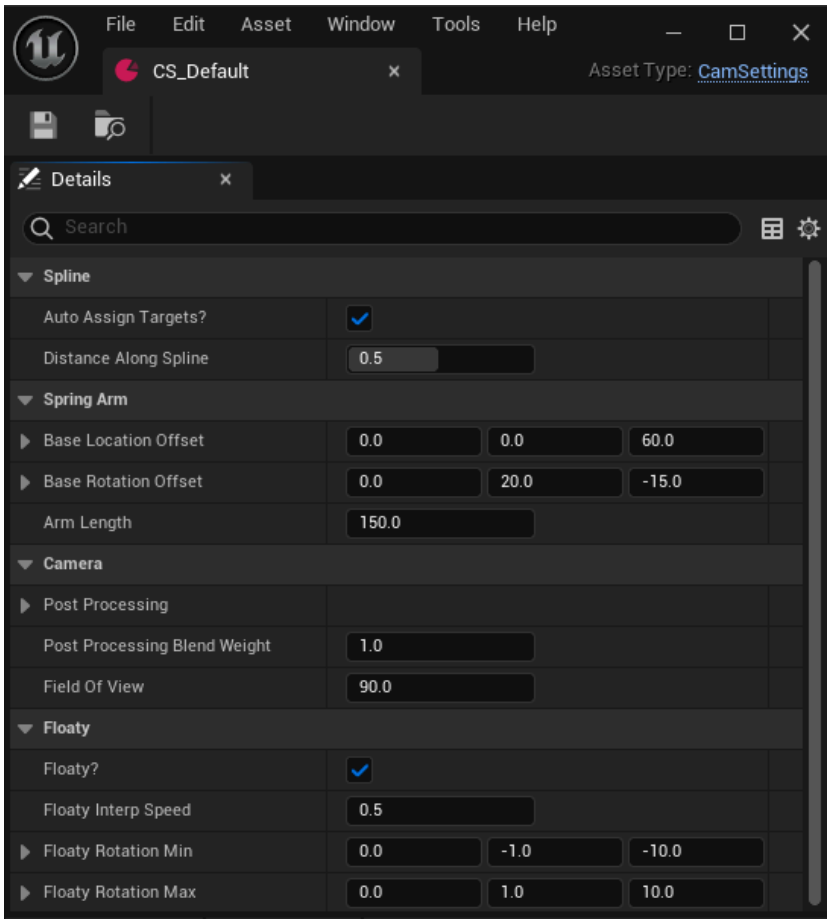
To create a Cam Setting, go to the content browser and go to create a new BP, in the dropdown menu, go to ‘Miscellaneous’ and select ‘Data Asset’ from the following menu.



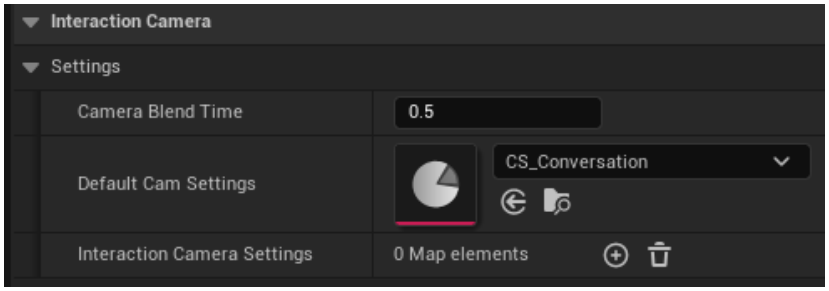
This will then give you a prompt to select the base for the data asset. For this, select ‘CamSettings’ from the list and click select.



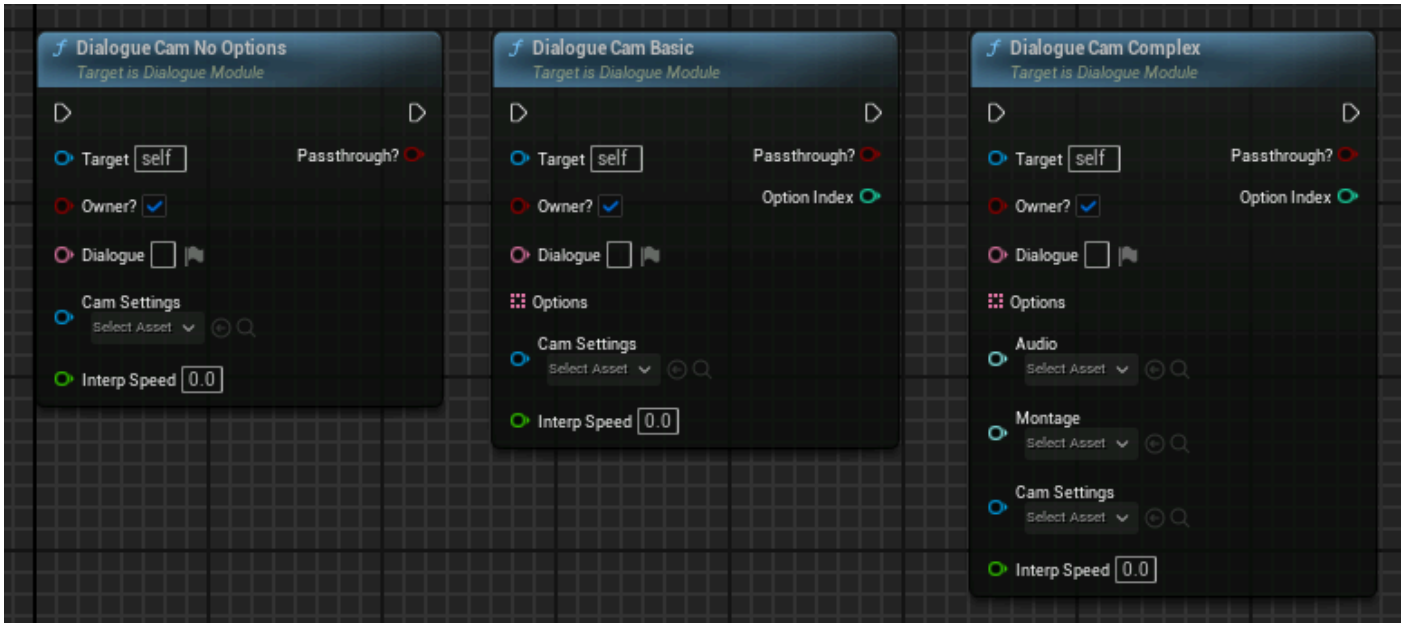
If you double click the data asset, it’ll open to show the settings you can change to configure the camera system.



When setting up an interaction (interactable component), you can specify cam settings to use. If provided, it will automatically create the relevant actor to facilitate the camera change.



Additionally, when setting up dialogue using the dialogue modules, there are dialogue nodes that allow you to specify what camera settings should be used, along with how quickly it should transition from the old to new settings. (if an interaction cam is already being used) If the interp speed is 0, the change will happen instantly.



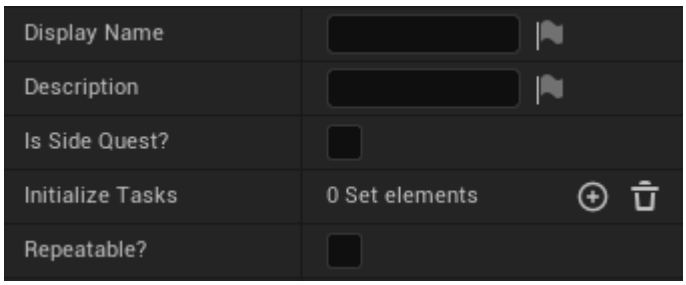
Creating a Quest

Creating a quest comes in three main parts, each of them serves a specific purpose when building a quest. These are:

- Quest Object
- Quest Task
- Task Listener

Quest Object

A quest object helps manage the specific quest. This would contain the quest name and description that will be shown to the player.

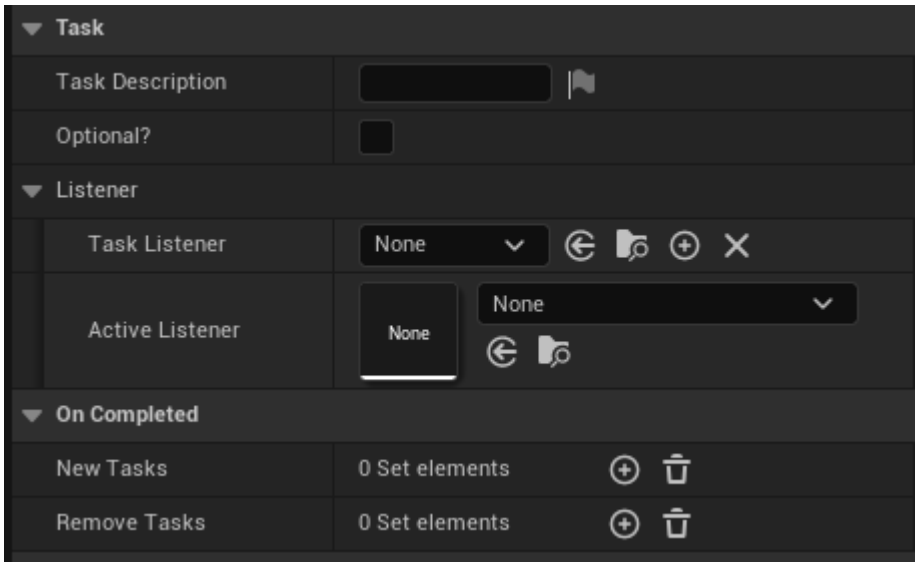


As the quest object does not keep track of all the individual tasks, we only need to specify the starting tasks that will

be provided when the quest is first created. You can have multiple if desired. New tasks that are added are defined by the task that is completed.

QUEST TASK

A quest task is an individual objective intended for the player to complete. It includes a task description that will be presented to the player. Additionally, you can define which new tasks should be added and which tasks (if active within the quest object) should be removed upon completion of the task.



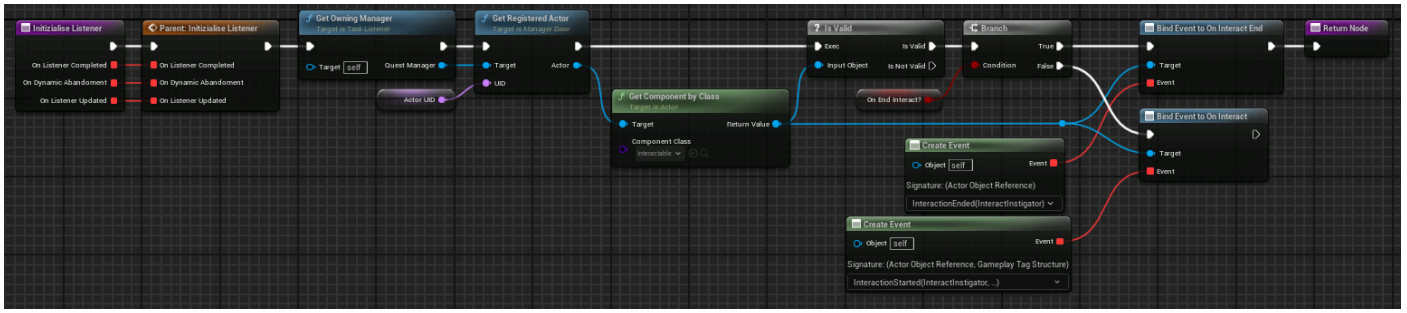
As a part of the quest task, you must designate a task listener. This object monitors events in the game world to determine whether the task has been fulfilled.

TASK LISTENER

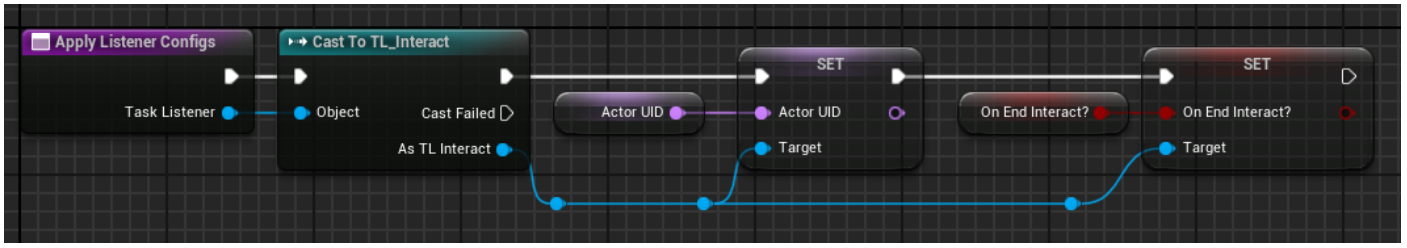
A task listener is used by a quest task to keep track of the actions/events that need to happen for the task to be considered complete. You can use the same task listener for different tasks if you desire. The quest listener has no variables for you to configure and instead is an object that you override functions inside.

When the quest listener is first created (at runtime), ‘Initialise Listener’ will be called and is intended for you to bind to the various event dispatchers so it can keep track of what is required for the quest to be completed.

Below is an example of the initialise event for the interact task listener. The role of this listener is to listen out for when someone has interacted with an actor that has the ‘Interactable’ component on it. When initialised, it binds to the relevant events it needs to know if it’s happened.



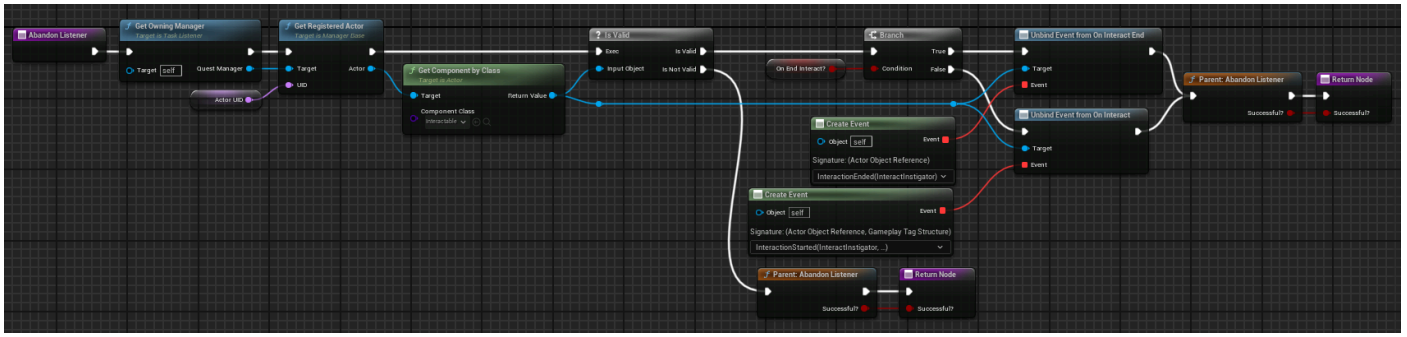
As the interact listener is design to be generic, there are variables that need to be set before the initialise event happens. For this you have two opens, either create a child of the task listener or create a new quest task base (child) that uses the listener. If you choose the second option, you can override the ‘Apply Listener Configs’ function where you can cast to your specific listener class and set the relevant variables before its initialised.



Doing it this way allows you to create common quest tasks where you can define variables for the generic task listener that will save you having to create a new listener for each quest task. Be sure to check out the base tasks that have been made for this project to see how they are setup.

Be sure to check out the base tasks and the associated listeners to see how to setup new base tasks.

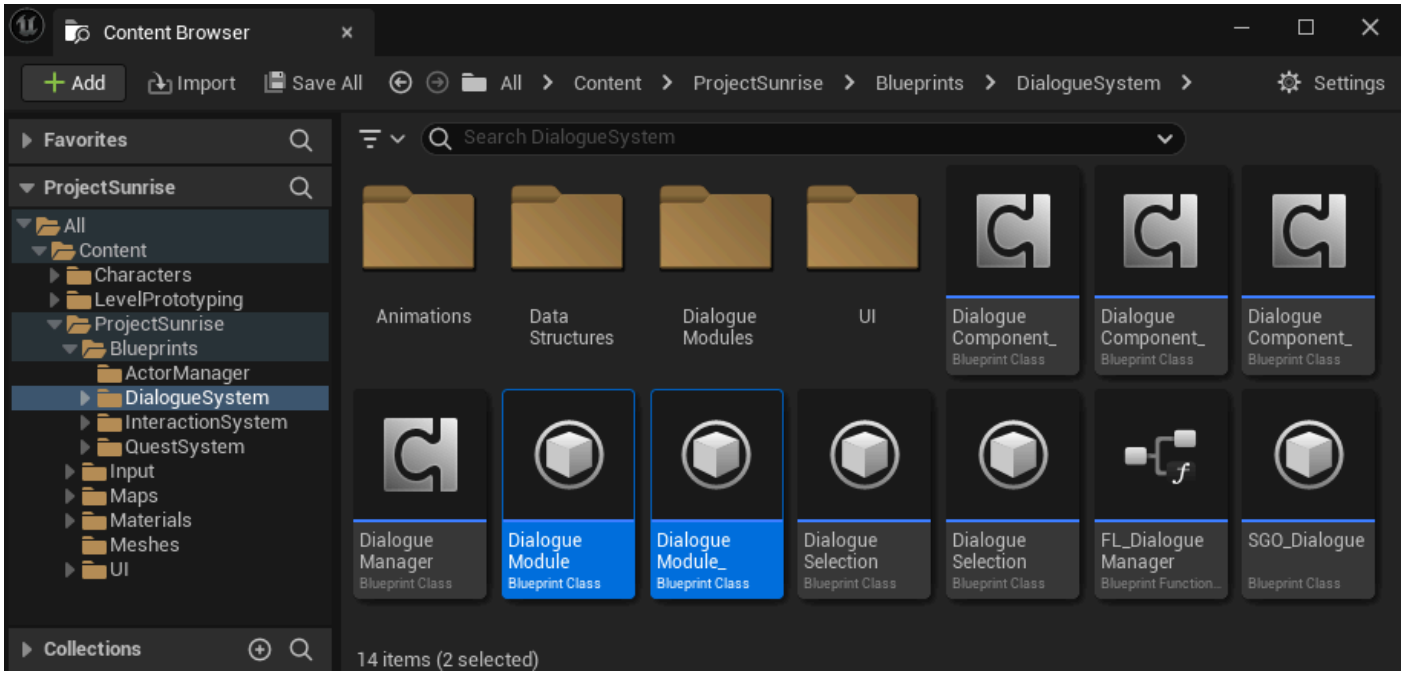
In addition to the initialise function, you will also need to override the 'AbandonListener' function. This is called whenever the listener is abandoned (for whatever reason). The purpose of this function is to unbind any event dispatchers that were setup during initialisation. This is to prevent unnecessary code from running when it's no longer needed.



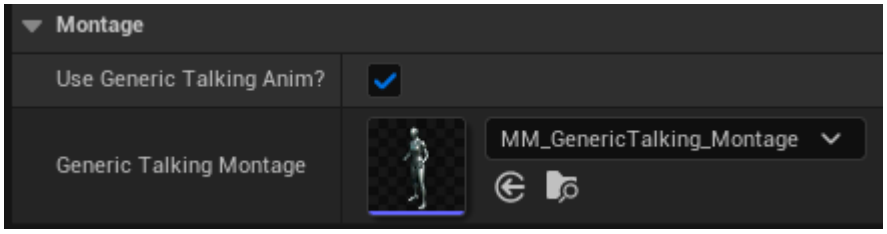
Creating a Dialogue Module

To create a Dialogue Module, create a child of either 'DialogueModule' or 'DialogueModule_Extended'. These can be located inside the 'DialogueSystem' folder. The extended version has additional helper functions for getting the quest manager and assigning quests. As your project grows, you might want to create a child of the relevant base dialogue module so you can add your own additional functions that would then serve as the new base for your dialogue modules.

If needed, you can always reparent dialogue modules you have created if needed, however it can be good to make sure everything is saved before proceeding.

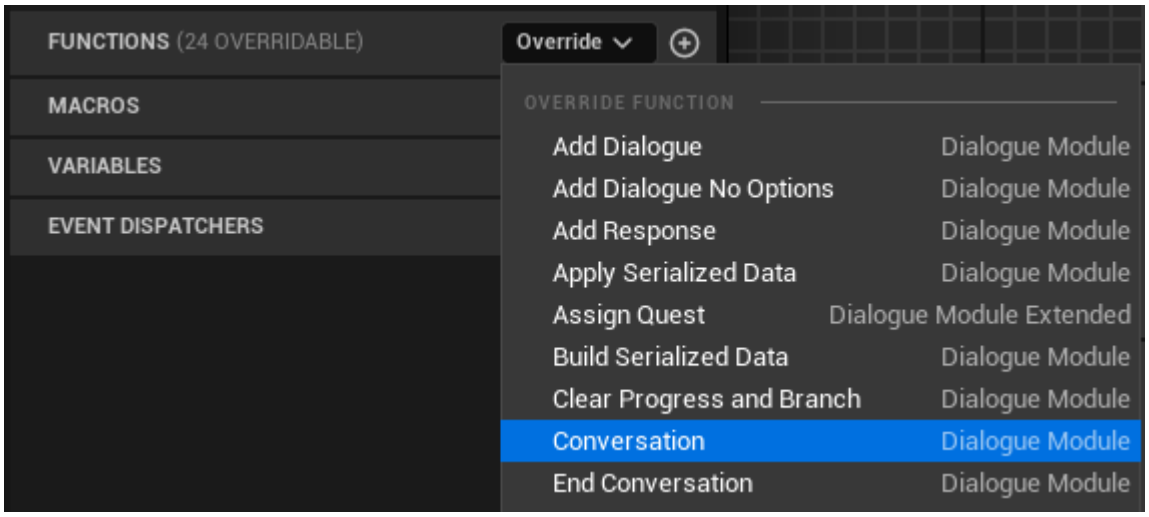


Once you've created a child of one of the dialogue modules, open it up. In terms of the default settings, the only thing to configure is if you want this specific module to use a generic talking montage. You can disable/change this if required.

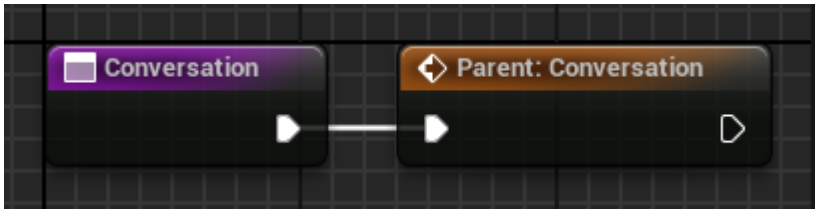


Building your Conversation

With that done, it's time to start setting up the conversation within the dialogue module. In the functions section, click on 'Override' and select 'Conversation' from the list.



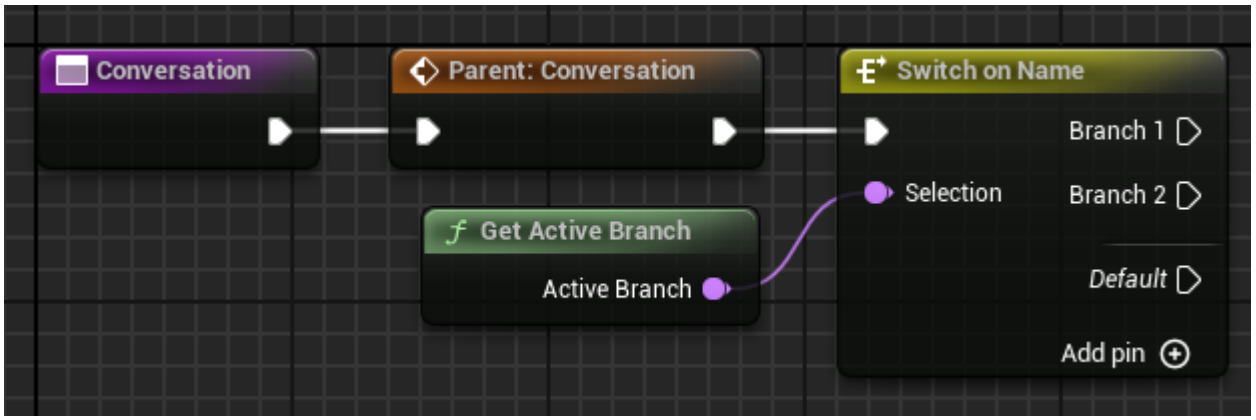
This will override the 'Conversation' function. The purpose of this function is for you to setup the conversation inside. When creating a new dialogue module, you will always have to override this function. As there is nothing in the parent function, you can choose to delete the orange call to parent node.



Conversation Branching

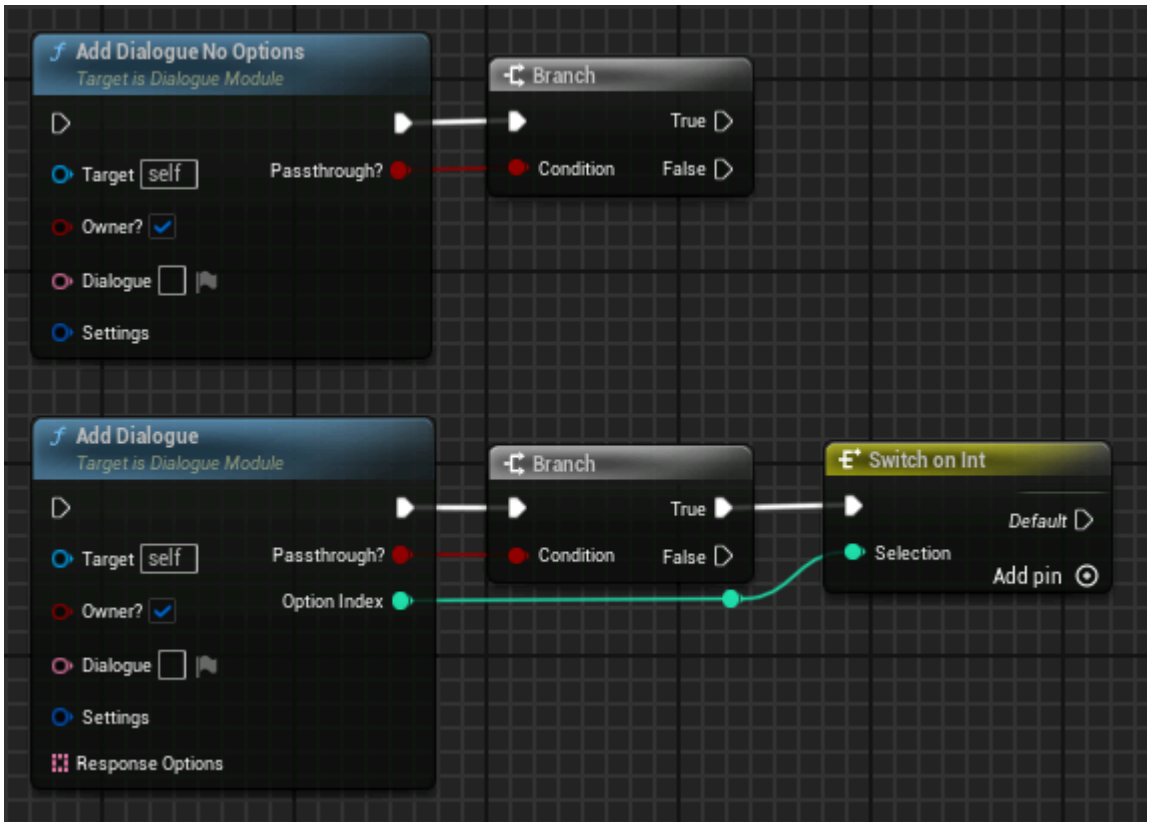
If you wish to use dialogue branches, you can use a switch on name node and connect the ‘Get Active Branch’ node. There are functions in place to change the branch name. This would then be connected to the conversation execution chain. Branch names will be set by through the dialogue tree when required, all you must do is add an entry to the switch on name to accommodate it. You can add as many branches as you would like.

Be aware that the starting branch will always be ‘Default’. We will go over branching more later in this section.



Adding Conversation Dialogue

Now it’s time to add some dialogue, for this we use the two ‘Add Dialogue’ nodes. The ‘Add Dialogue No Options’ is the simplest of the two nodes and is for when you don’t wish to present any options to the player.

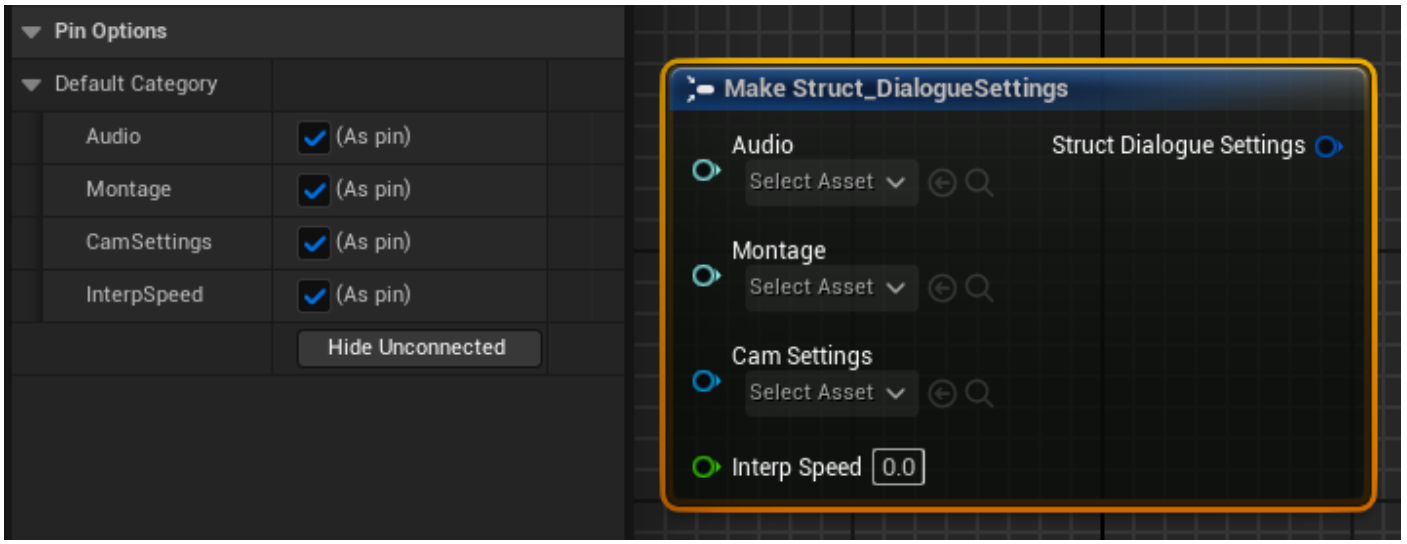


Common inputs and outputs on the node are as follows:

bOwner? – This relates to if this piece of dialogue is to be spoken by the owner of the dialogue component using this module. If false, it will refer to the instigator. (The player that triggered the conversation to start) In addition to this, it will affect the name that is displayed for who is saying the dialogue. If using a montage or audio, this will affect what it is played on.

Dialogue – This is (as the name suggests) the dialogue you want to display.

Settings – This is a structure of additional settings such as audio, montage and camera settings you can use for this specific dialogue. You can either split the pin or make Struct_dialogueSettings. If you choose to make structure, you can hide unused pins in the details panel.

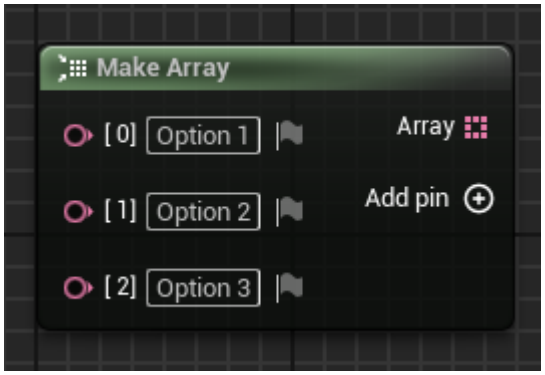


bPassthrough? – This refers to if the dialogue has stopped at this node. If this is the case, the returned value will be false. If the dialogue is (as the name suggests) passing through, it will be true. When setting up the dialogue you will be required to add a branch from this output. When chaining 'Add Dialogue' nodes, these would be connected to the true on the branch.

If you wish something to happen when the dialogue is first displayed, you can connect the logic to the false.

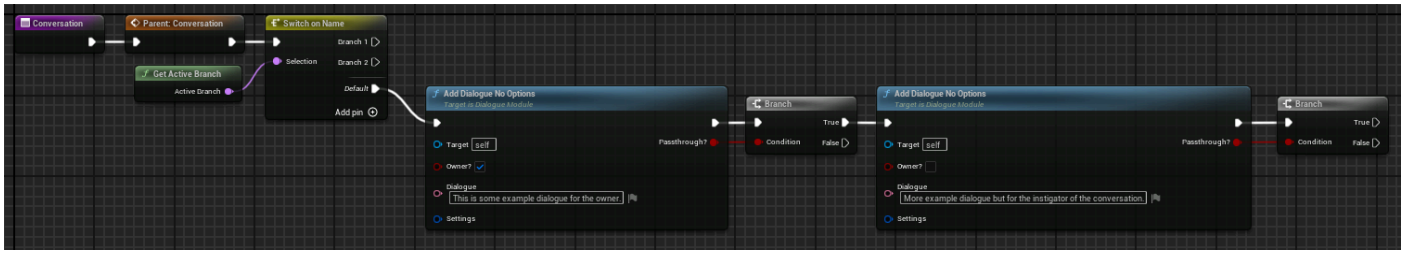
Option specific input and outputs are as follows:

Response Options – This is an array of the options you would like to show to the player. For this, make a text array and add as many options as required.



Option Index – This is in relation to the option that has been picked by the player. If for example, you gave the player 3 options and they picked the third one, the index would be 2. The index will match the index on the supplied options array. When setting up branching dialogue, you will use a switch on int node that is controlled by the options index.

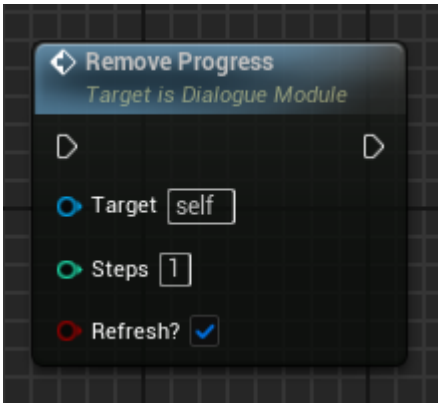
Below, I have added two add dialogue nodes with some sample dialogue inside. The first will be spoken by the owner while the second will be for the instigate. Take note of how these are chained together on the branch node. When the dialogue is progressed, it will traverse through each node until passthrough is false where it will then use the data supplied into the node to update the conversation widget.



Dialogue Progress

As it traverses through these nodes, it will keep track of the progress that has been made and is stored in the ‘Dialogue Progress’ variable.

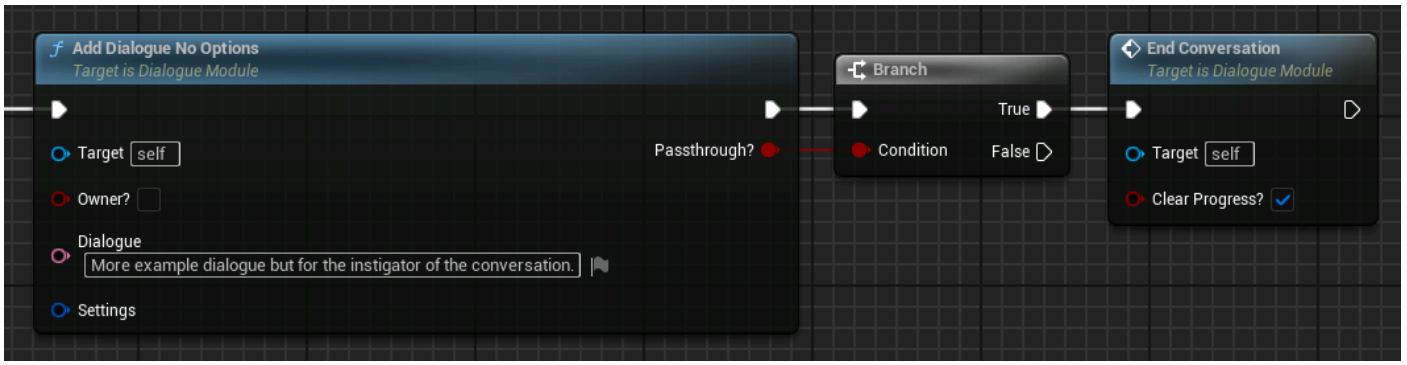
If you’re making complex dialogue flows, you can choose to manipulate this so that the dialogue starts and progresses from where you want. You can use the ‘Remove Progress’ node to go backwards in the dialogue tree.



Ending a Conversation

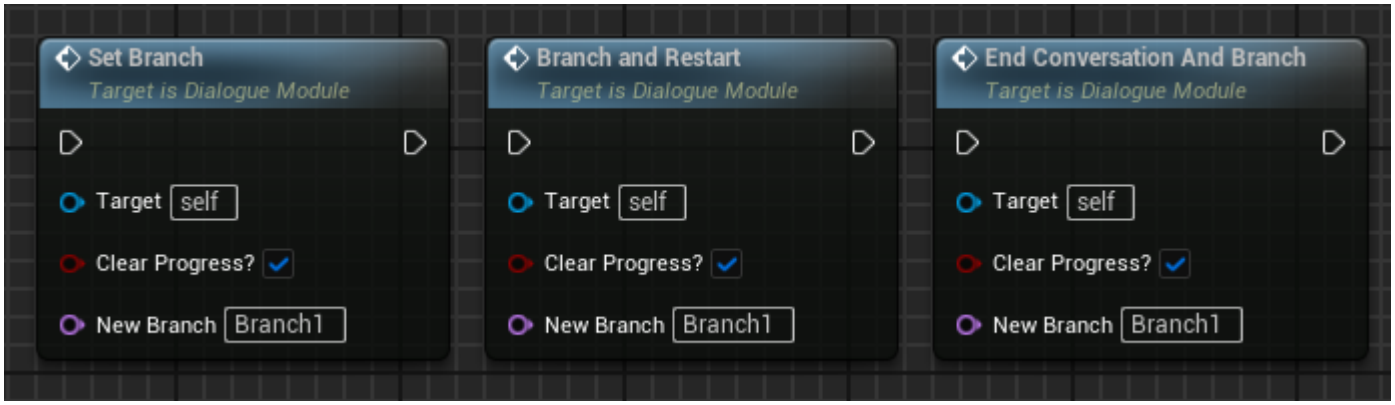
Should you wish to end the conversation, you can use the ‘End Conversation’ node. This will close the conversation widget and call the ‘OnConversationClosed’ event dispatcher on the dialogue component. You can handle what additional things might happen by binding to this event in the actor that has the dialogue component.

Be aware that in most cases, you will need to clear progress when closing a conversation otherwise this will result in the conversation been closed again when the conversation is reopened. As mentioned above, you can manipulate the ‘Dialogue Progress’ variable if necessary.



Conversation Branching Continued

If you wish to use a different branch there are a few nodes you can use the switch the active branch.



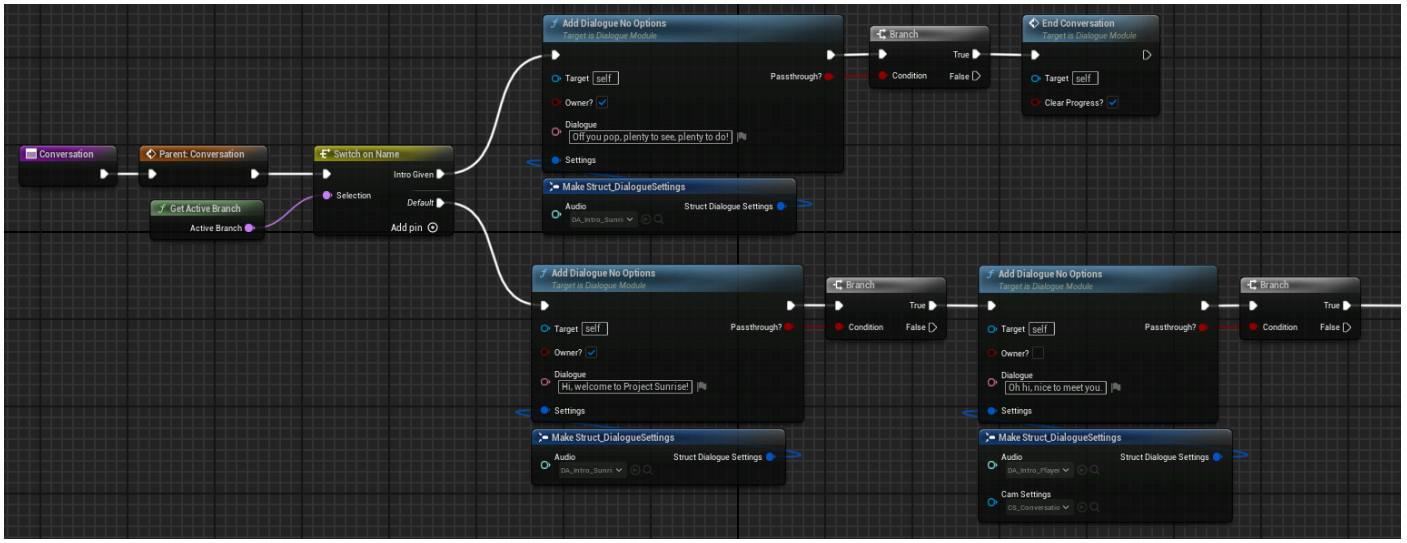
The first one is just ‘Set Branch’. This will switch the active branch to the specified name. There is an option to clear the progress for the current branch before it is switched. In most cases, this would be true, especially if you don’t plan to return to the branch as this will reduce the data that needs to be saved for the dialogue module.

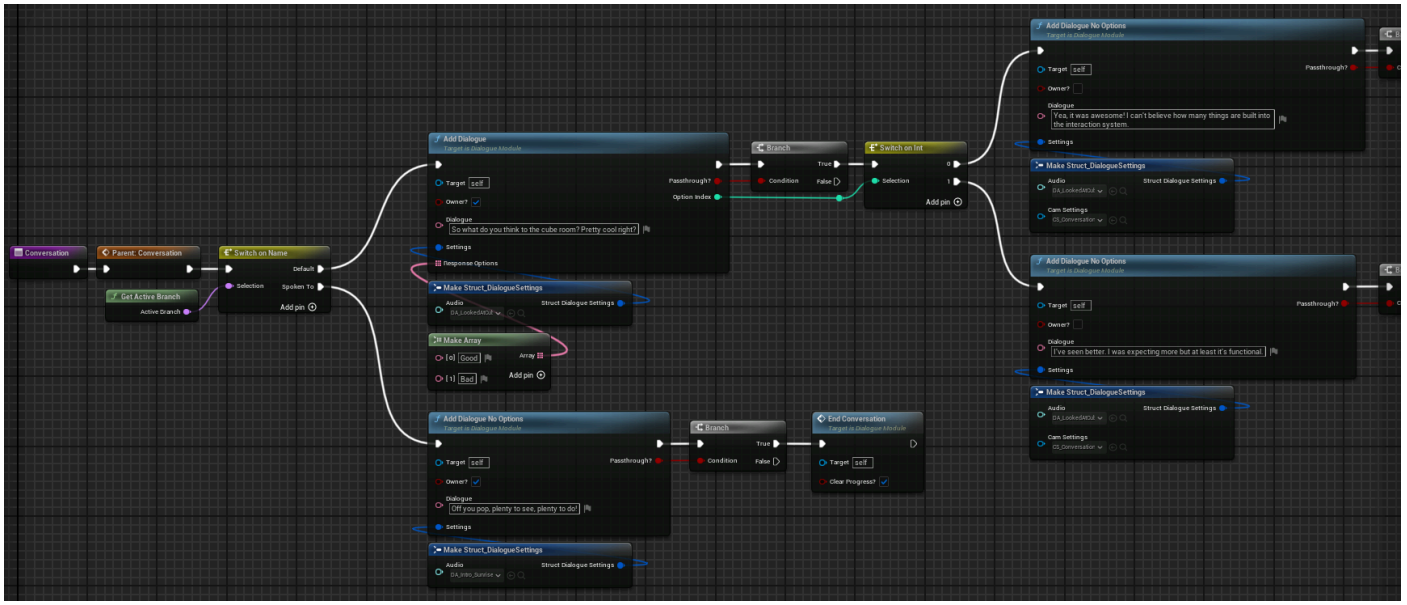
The next one is ‘Branch and Restart’. This will switch the branch and then restart (refresh) the dialogue tree from the new branch.

The final one is ‘End Conversation and Branch’. This will end the conversation and then update the active branch. This will mean when the conversation is reopened, it will start from the branch it was set to.

Examples

Below is an example of the how two of the modules used for Sunrise are setup. You can open ‘DM_Sunrise_01_Intro’ and ‘DM_Sunrise_02_LookedAtCubes’ to see the full dialogue flow. Be sure to look through the other dialogue modules to see how you can set them up to get the dialogue flow you want.

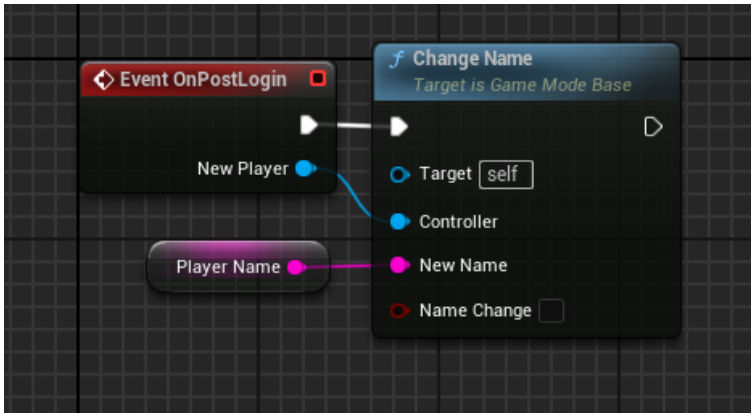




Player Name

The name specified for the player character that is used in the dialogue is based on the name specified in the player state. This name can be updated inside the game mode. A simple setup for this has been included. You can update the ‘Player Name’ variable to the desired name or handle this differently if required.

Please note that the ‘Change Name’ function is an engine function located on the game mode base.

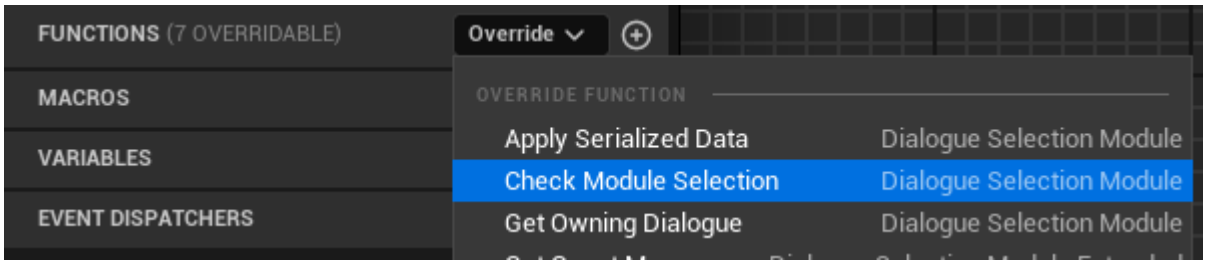


Creating a Dialogue Selection Module

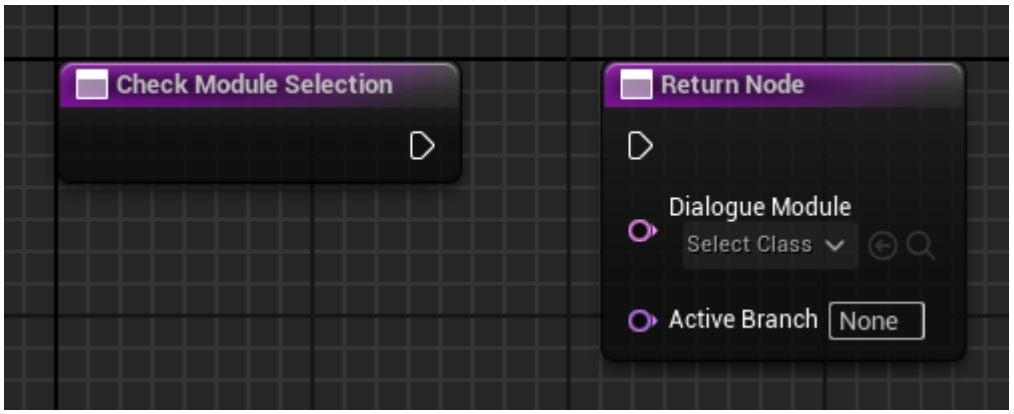
To create a Dialogue Selection Module, create a child of either ‘DialogueSelectionModule’ or ‘DialogueSelectionModule_Extended’. These can be located inside the ‘Dialogue System’ folder. The extended

version contains additional logic for getting the quest manager if needed for checking if any quests are active or been completed.

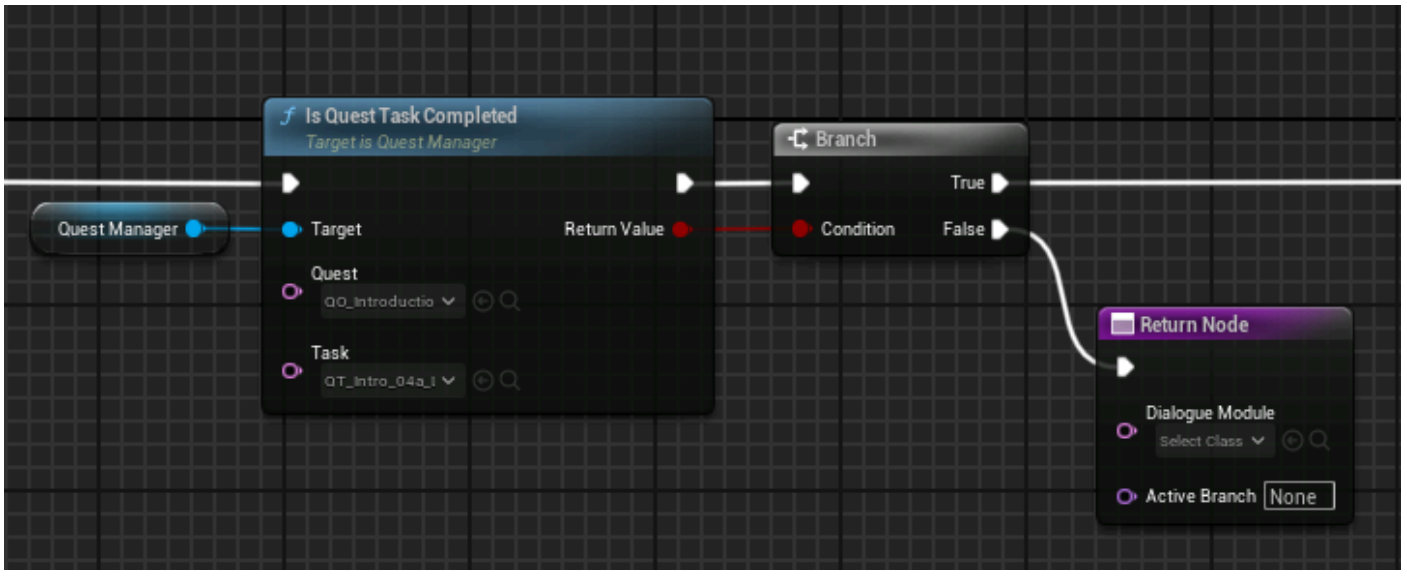
With the selection module created, open it up and override the ‘CheckModuleSelection’ function. This is where you would put the logic and condition-based checks to determine what dialogue module the owner of the selection module should use.



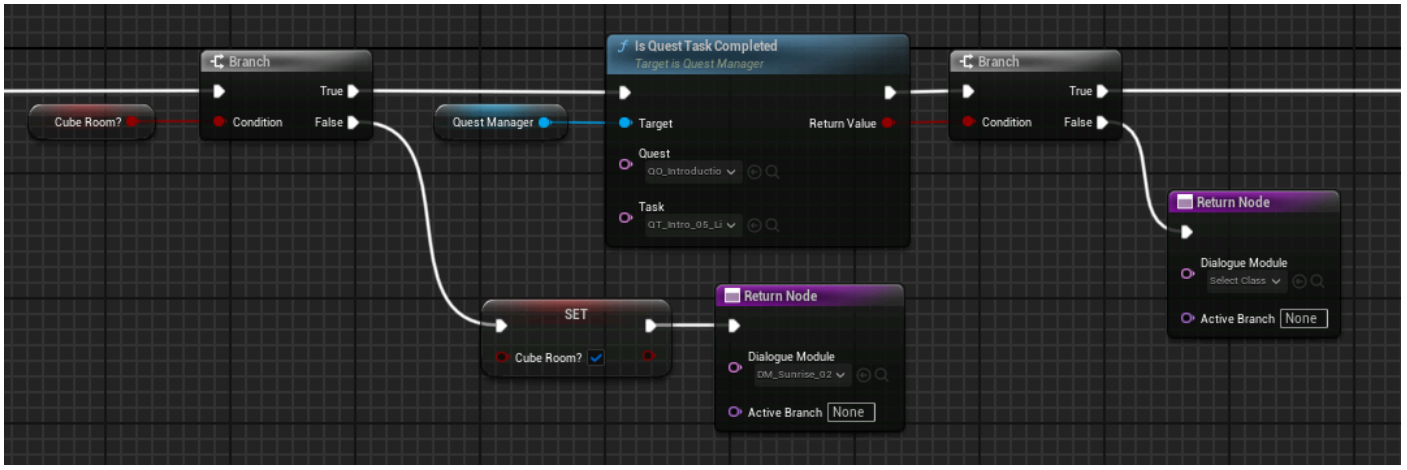
In this function, you can remove the orange parent call node as there’s nothing in the parent to call for this function. This function has a return node that requires a Dialogue Module the Active branch name. If left at Null and None, the dialogue component won’t do anything regarding changing the dialogue module or branch.



At this point it’s just a case of adding the checks and returning the relevant module. If we look at the Dialogue Selection Module for Sunrise, we first check if the player has completed one of the introduction tasks. If not, nothing changes, if they have, we proceed to additional checks.



The previous quest task we check is if the player has looked at all the cubes in the starting room. If we make it to the image below, it means that the task has been completed so here we check if they have had following dialogue related to the cube room. If not, we return the new dialogue module. The following check won’t return anything if the next task isn’t completed. This means that the dialogue component will continue to use the previous dialogue module (assuming it hasn’t been changed somewhere else).



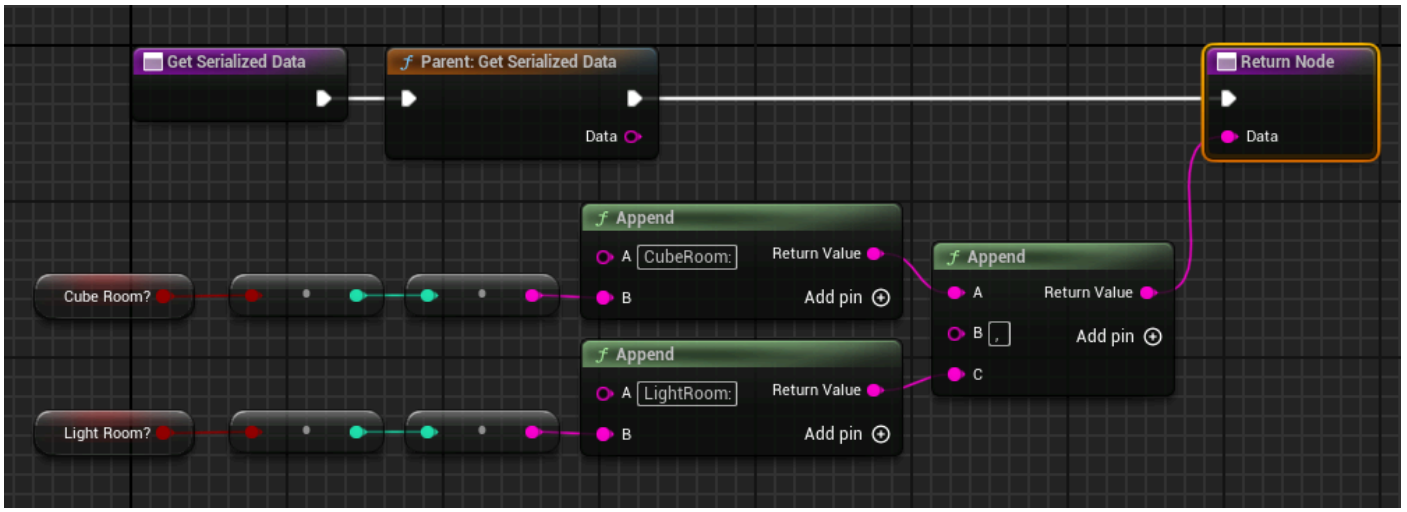
Feel free to check out the other examples for the Dialogue Selection Module within the example map.

Saving Additional Data

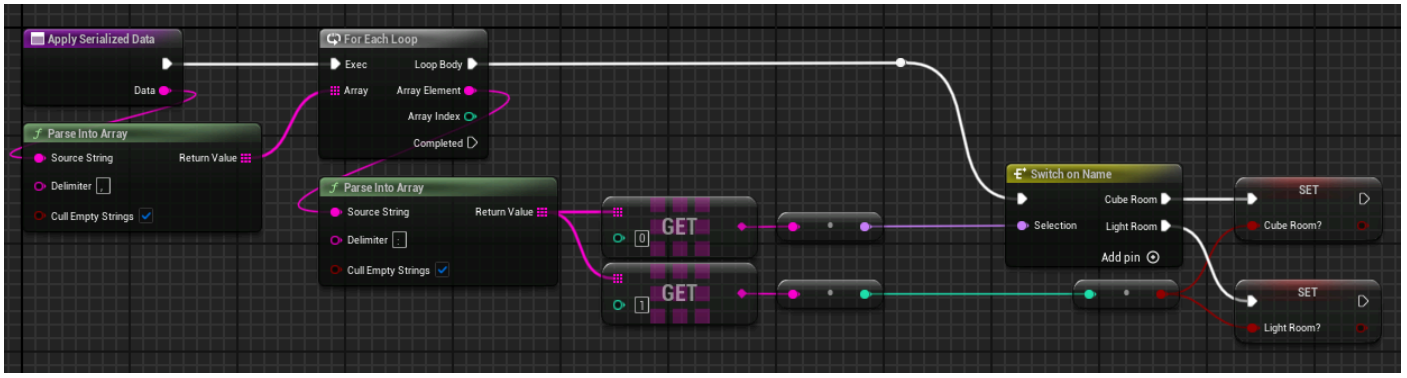
As you may have noticed from the last screenshot in the previous section, we use a bool to control the flow of the checks. This means that this data needs to be saved to ensure that when we load the dialogue selection module, it has the same data to perform the relevant checks.

This is simple enough by overriding the ‘GetSerializedData’ and ‘ApplySerializedData’ functions when needed.

The ‘GetSerializedData’ function is used as part of the save system built into the dialogue system. It is used to pull the additional data that needs to be saved for this selection module. Below is as example of the how we convert the values of two bools in the selection module for Sunrise into a serialized string. How you wish to do this it up to you, but you will need to format you’re string in a way that will allow you to get the data back out.



With the data saved, we need to then extra the data from the saved string and reapply it to the relevant variables. When the selection module is initialized, it will call the ‘ApplySerializedData’ function and pass the saved string to it. Below is an example of how we extract the data from it and reapply the data to the relevant variables.

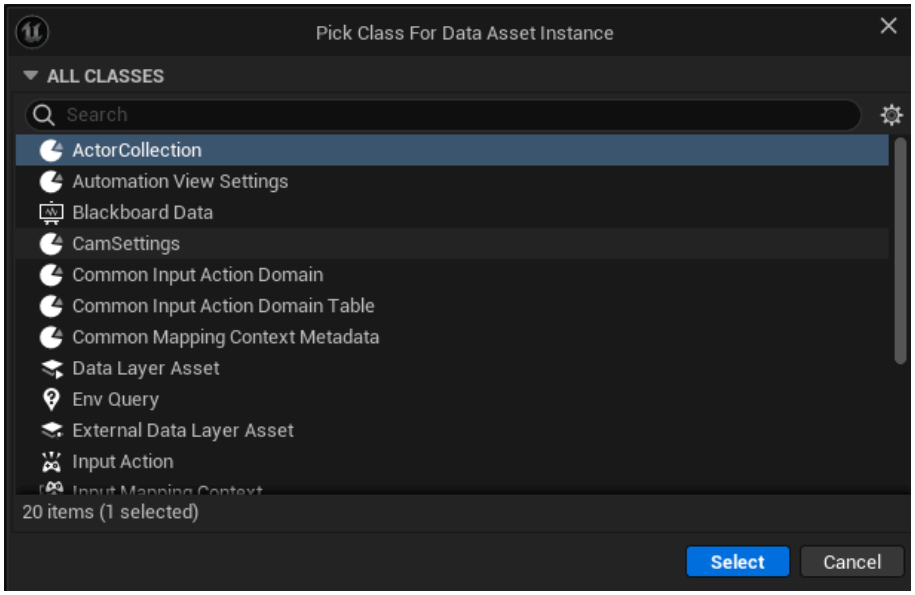


When setting up additional variables inside your selection modules, just bear in mind that if you need the data to save, it must use variable types that can be converted to a string (and back).

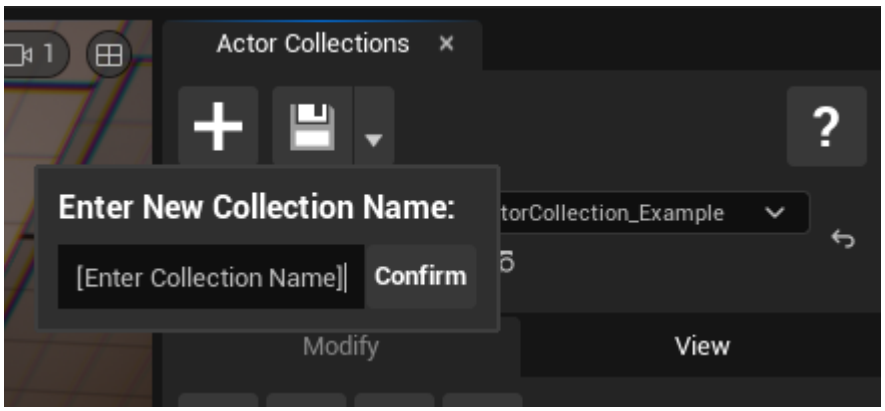
Actor Collection

Creating a Collection

Creating a new actor collection is as simple as creating a new data asset using the ‘Actor Collection’ asset type.



Alternatively, if you’re using the editor widget, you can click the + icon at the top left. This will give you a prompt that you can use to enter the name of the collection. Once created, it will automatically be placed in the collections folder and made the active collection used by the widget.

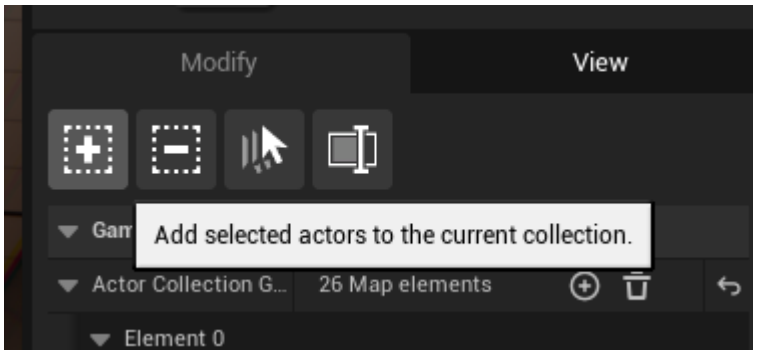


Registering an Actor

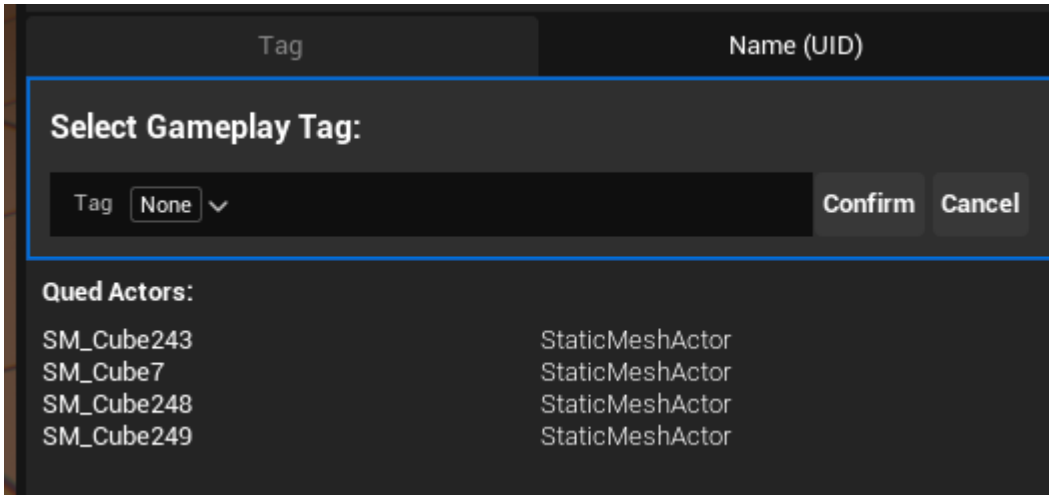
There are multiple ways to register an actor but these will fall into either registering via the editor or at runtime.

Editor

To register via the editor, first you need to make sure you have a collection selected in the widget you wish the actor to be registered inside. Once done, select the actors you want to register in the level and click the + button under the modify section.



This will then cycle through each selected actor and ask you to provide either a gameplay tag (recommend) or name. The camera will jump to actor that is currently being registered. A list of the actors that are pending to be registered will be shown below. When a tag or name has been confirmed, it will jump to the next one in the list. All tags and names used must be unique to that collection. You can use the same tag or name in different collections if required.

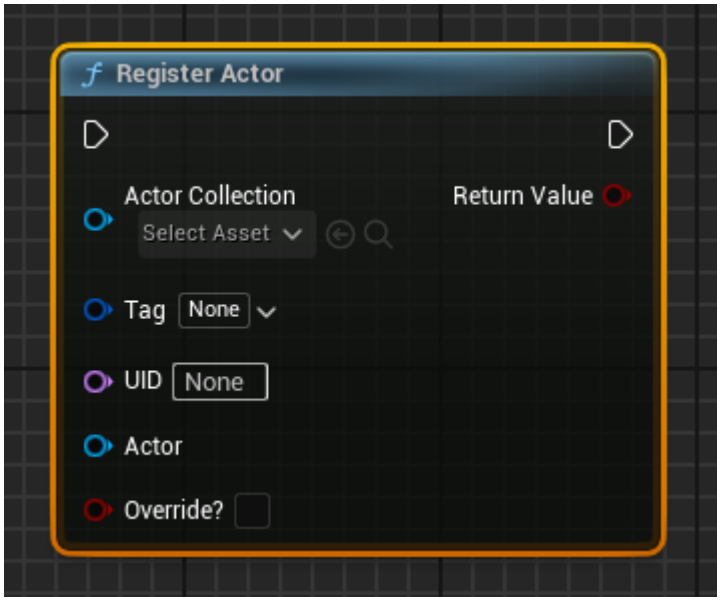


Actors already in collection will be ignored when attempting to add using the widget. Whilst you can manually register the same actor with multiple tags or names, this isn't recommended as it will cause issues if you want to get the tag for a registered actor.

Runtime

Registering at runtime is great for more dynamic items where they might be spawned after specific events have happened. To do this, it's as simple as calling the 'Register Actor' function. This is provided by the actor collection function library.

Select the collection you want the actor to be registered too and provide either the gameplay tag or a name. Please note that the gameplay tag will be priorities so if a gameplay provided is valid it will attempt to use it over the name. Lastly supply the actor you want to register.

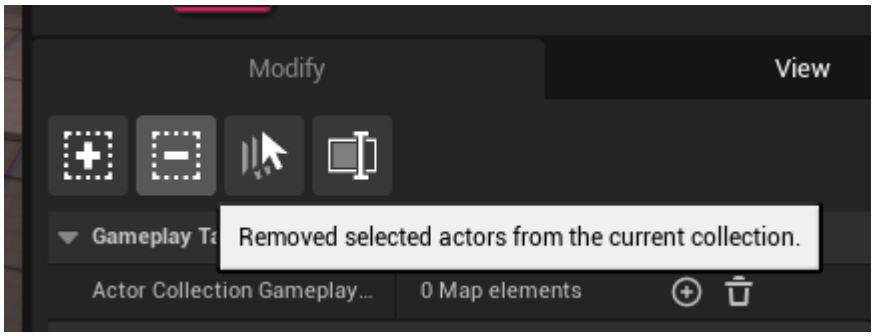


The override option allows you to override what ever actor is already registered with the given tag or name. If the tag or name is already being used and override is unticked, the registration will fail. You can check this by using the return value on the node.

Unregistering an Actor

Actors that are registered at runtime will automatically be unregistered when destroyed/unloaded. This contrasts with actors registered via the editor.

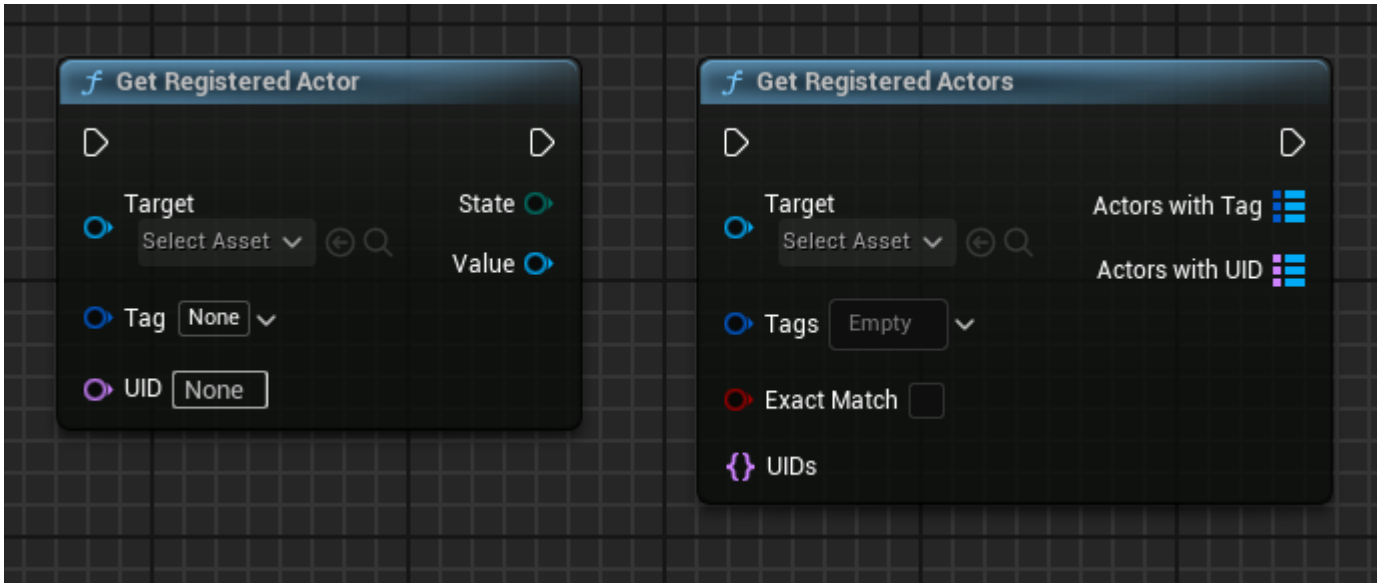
Actors registered via the editor are persistently registered, this is because they are a part of the level. If however you want to remove actors you have registered via the editor, you can do so by selecting the relevant ones and using the – button under the modify section on the widget.



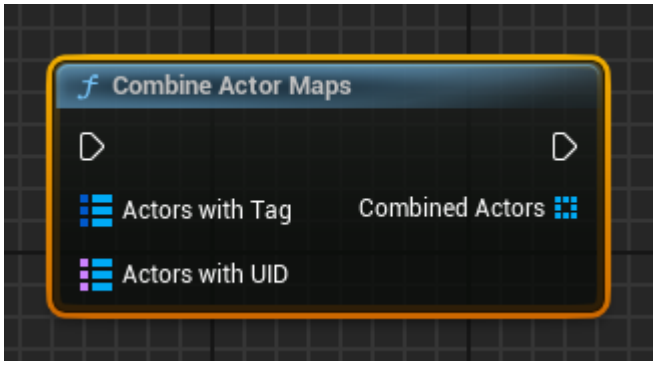
Getting a Registered Actor

When it comes to getting a registered actor there are two nodes. These allow you to get a single actor or multiple. Simply specify the collection you want to get the actor from and specify either the tag or name.

If getting multiple registered actors, you might need to loop through them. Be aware, you can get multiple actors using a mix of tags and names if desired.



There is a helper function in the function library that will combine the actors into a single array if required.



Change Log

1.1

Bug Fixes:

- Finding the closest interactable now correctly gets the closest interactable.
- Fixed an issue where using hold times of 0 wouldn't allow other input types to function once triggered.
- Fixed an issue where a 2x2 pixel area in the top of the screen is visible when there's no outline being used.

Changes:

- Setup implementation on the game mode for setting the player's name that will be used on the dialogue system.
- Tweaked the EQS used to find move to markers to consider the nav mesh where possible.
- Assisted move will now check to see if the nav move can be skipped if the character is close enough to the move to location and there is a clear line of sight.
- Included a 'Force Interaction' function on the interaction system which will allow you to force an interaction as if the player had triggered it themselves.

New Features:

- Actor Collections – The actor collection system replaces the actor manager system of registering and unregistering actors with names. You can define 'Collections' that you can register actors to with either a gameplay tag or name. Actors can be registered via the editor without the need for addition components to be added to actors.