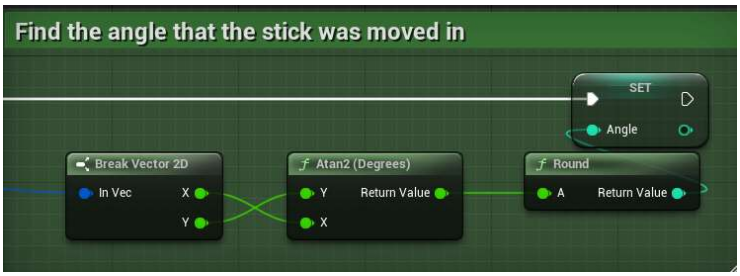
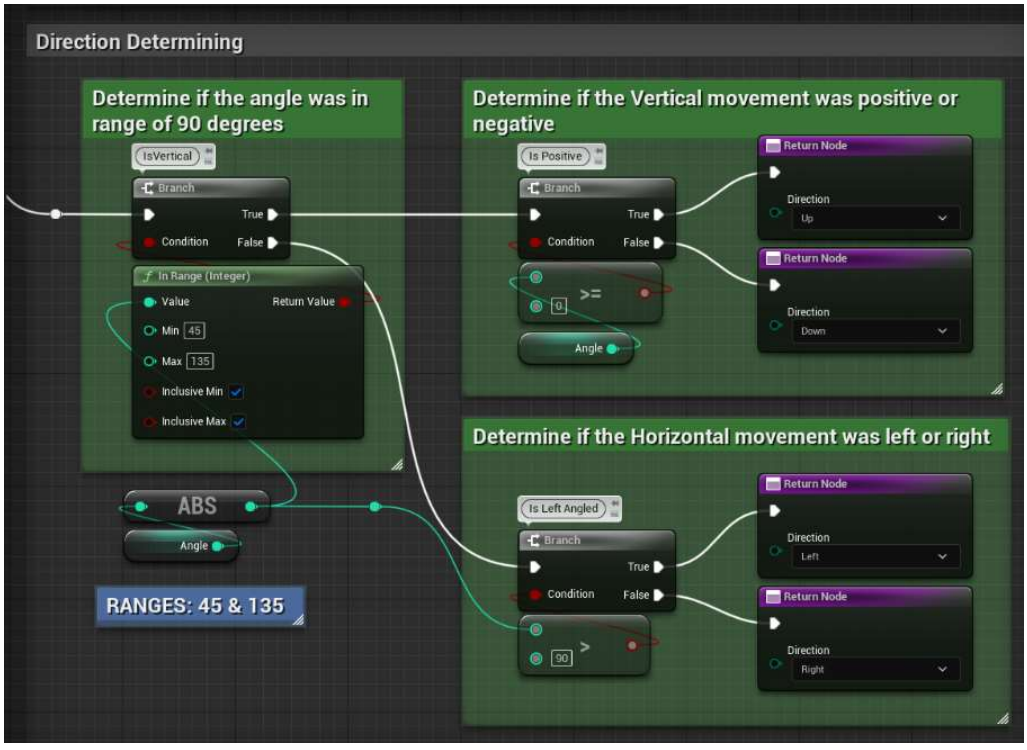
Flex Marks - 1

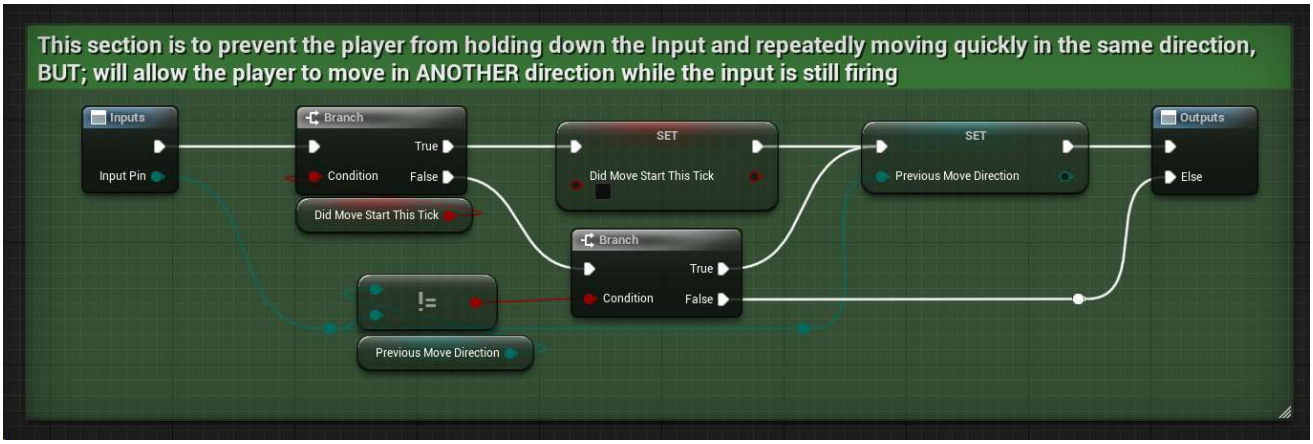
With the way that Epic Games set up Unreal Engine's widget system, it is very hard to make a gamepad (a controller) interact with UI in a consistent, satisfying manner, let alone four of them at once. I spent a considerable amount of time in this portion of this project working on the UI system, allowing gamepads to interact with it and for multiple players to use the UI at once. I named this the MUGWUI system, standing for *Multi-User Gamepad Widget User-Interface*.

MUGWUI is made up of two core components, the base, and the user interactable base.
The base serves as the container and directory for any objects within and tells any interactable base objects inside how to react to a user's inputs.
The interactable base serves to react to the inputs given. Each interactable base has a structure of surrounding objects, which contain what interactable base object will be focused on if the player was to move in that direction.

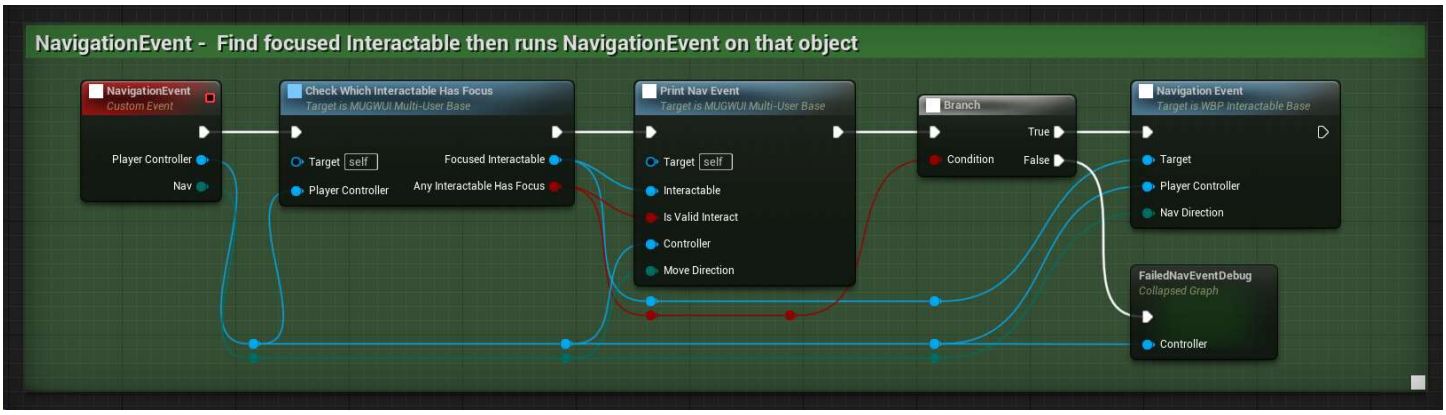
I took advantage of Unreal's new Enhanced Input System to feed input actions into the MUGWUI system. While with a keyboard or d-pad, movement is quite simple, with a joystick however, it is different as you must consider all the entire 360 degrees it can move in.
To determine this, after the user gives a movement input; I use a rounded Atan2 node to calculate the angle. With this angle, it is then determined what direction the joystick is moved in.
It also must be considered if the player has already navigated once, as the Enhanced Input System sends an event every tick the button is triggered.

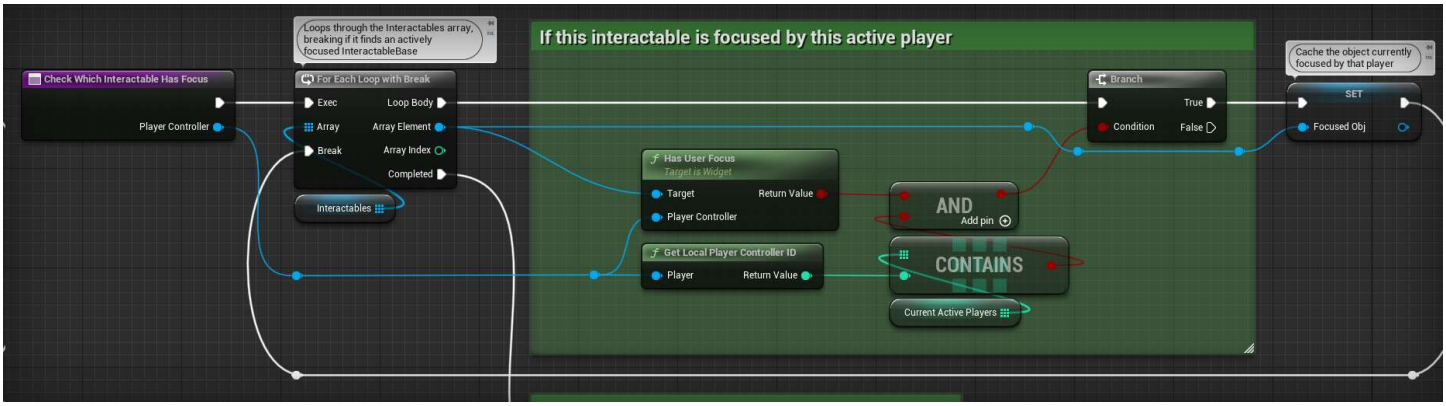




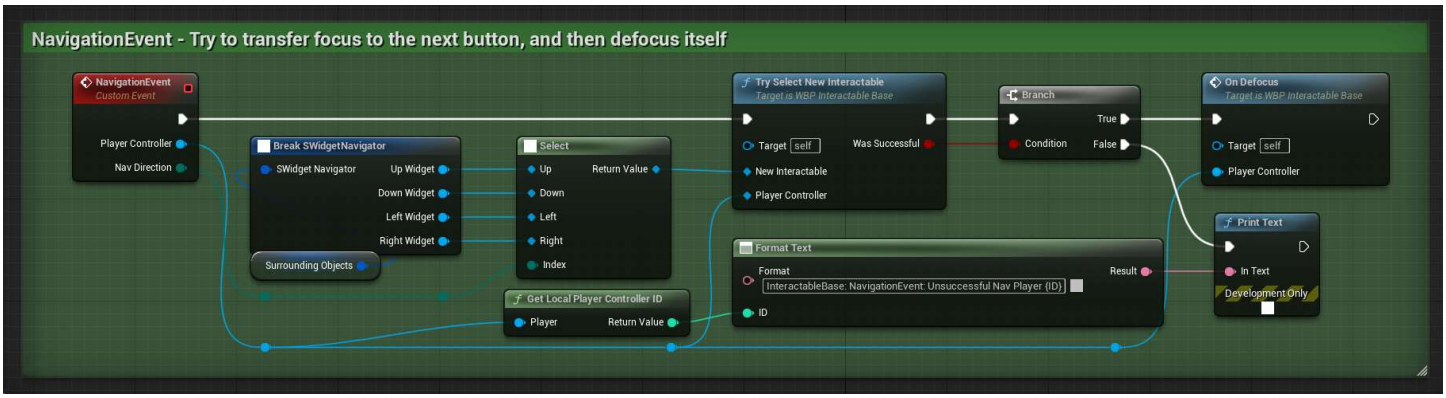


After calculating the direction, it then gets sent to the MUGWUI base, which will iterate through all of its interactable base objects to see which interactable base the player is actively focused on. Then, using the found object, it then calls the navigation event on that interactable base





Inside the interactable base, the navigation event uses the direction calculated in the player controller to determine what of the surrounding objects it should switch its focus to, if the interactable base corralating to the direction is not null and the interactable base is enabled.



With the swapping of widgets, the visuals need to update as well, but the interactable base is just that, an abstract base. This is why I made children inheriting from the class, versatile buttons. These give the interactable base a visual component.

Flex Marks - 2

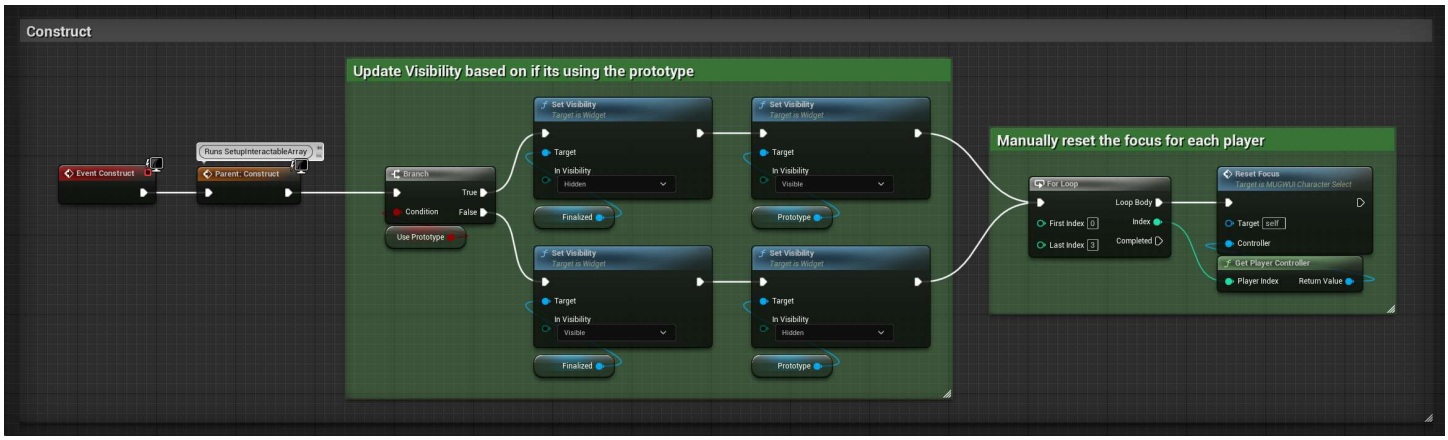
Summary

These past two weeks I mainly focused on the improvement of the UI system. I mainly focused on bringing the CharacterSelect into a 3D widget, fixing the mouse breaking the focus, and adding D-pad and Arrow keys to the input.

3D Space

When we moved the CharacterSelect widget into a 3D space, it came to our attention that the input was no longer reaching the widget. This was because we needed to turn on the ReceiveHardwareInput flag of the Widget component, and fix the issue that came with it; the events CharacterSelect::ResetFocus and CharacterSelect::Construct were being called out of order, which meant that the Interactables array and those like it were empty when the user was initially meant to be sent to the button at the top of their character selection strip.

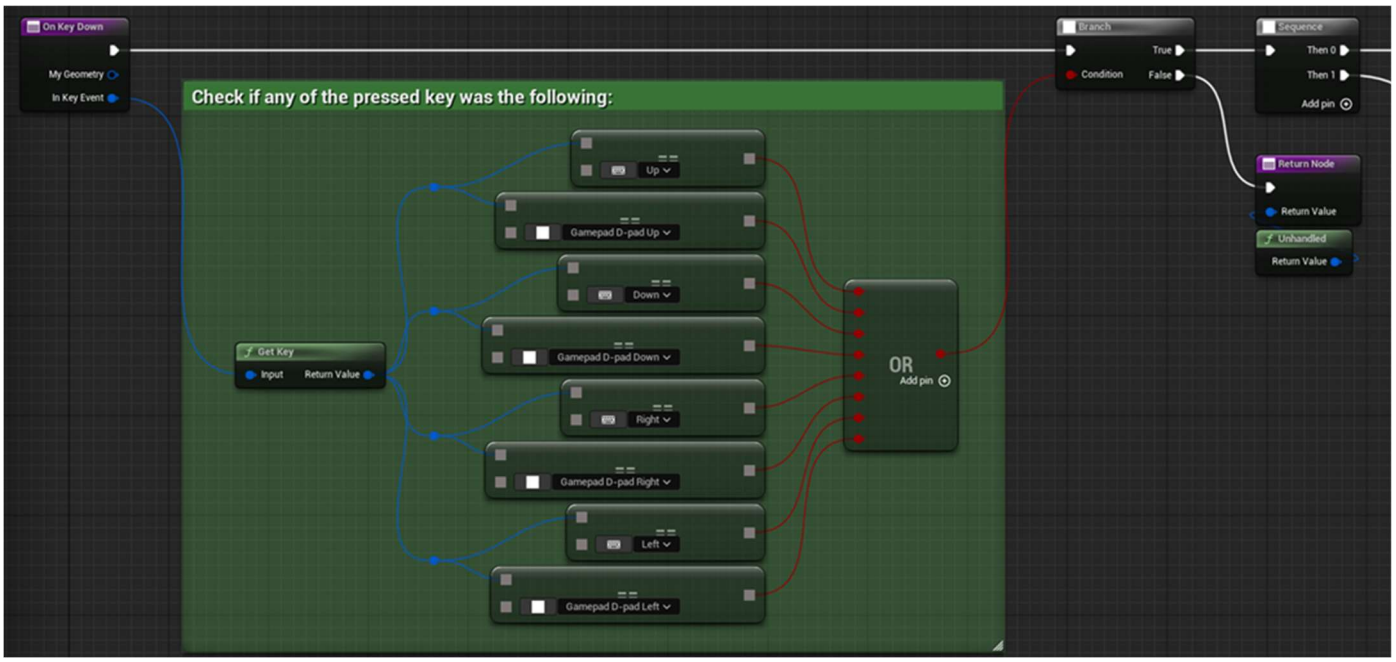
I fixed this by adding an extra call to the CharacterSelect::ResetFocus event at the end of CharacterSelect::Construct, that ensures the arrays necessary for the system – to correctly find a button for the player to focus on – is full.



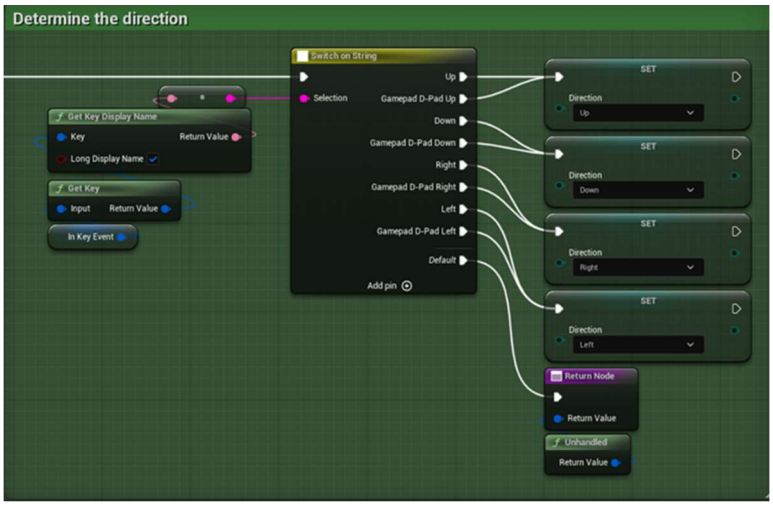
D-Pad and Arrow Keys

When trying to add these inputs to the UI input mapping, we discovered that UI takes input priority over the Enhanced Input.

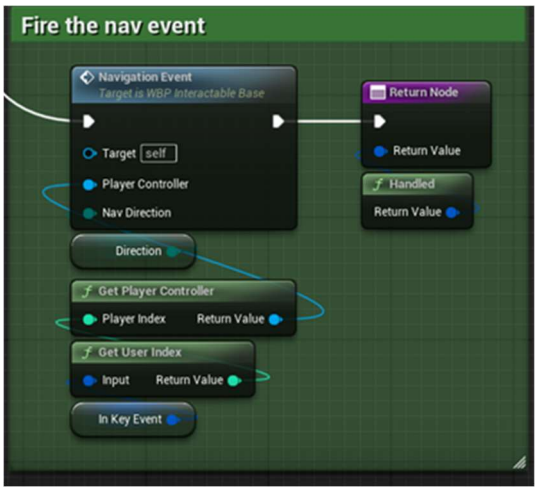
To fix this, we overrode the OnKeyDown event in InteractableBase and check if the key event it received was any of the inputs from the D-Pad or Arrow keys. If it is, it will continue into a sequence that will check which direction the input was (in literal terms of the key's name), and feed that direction into the InteractableBase::NavigationEvent like in PC_Menu. If the received key was not one of the D-Pad or Arrow keys, the event is returned as unhandled, which allows the input to continue to flow into the Enhanced Input System and trigger the Input Actions it has.
Not doing this final step blocks all input from the Enhanced Input System.



Determine if any of the D-Pad or arrow keys were pressed. If true, continue to a sequence of two. If false, let the input pass onto the Enhanced Input System.



Sequence 1: Determine the direction of the movement

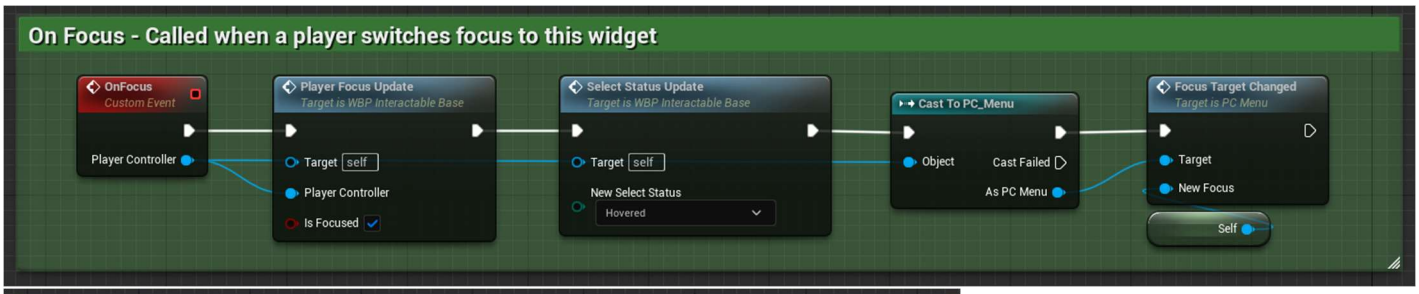


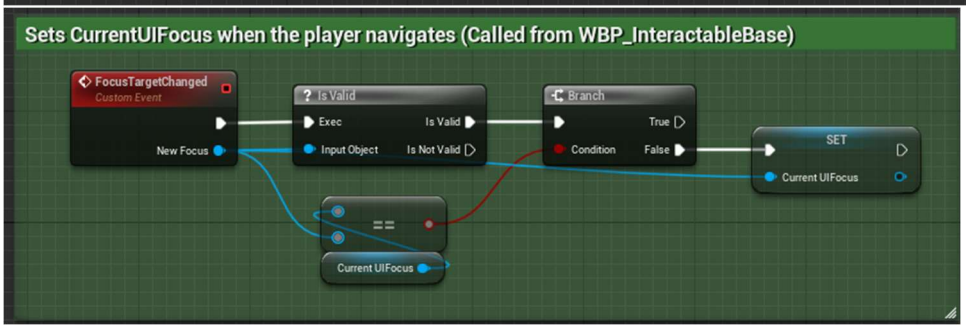
Sequence 2 : Output the direction of the movement into a NavigationEvent call

Focus Bug

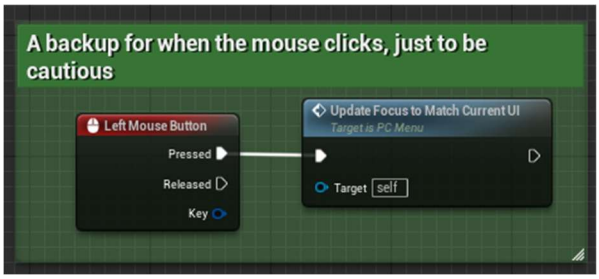
The last thing I fixed was the long existing bug that removed focus from InteractableBase buttons whenever the user clicked on the game screen.

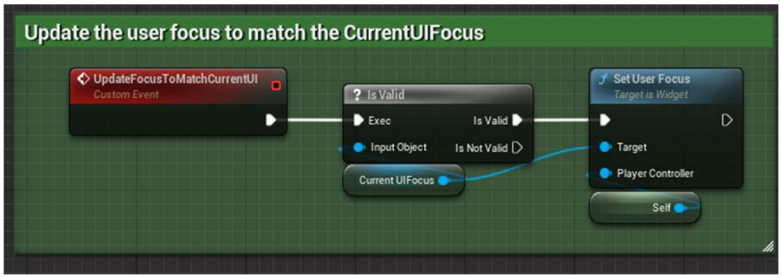
The solution came as adding an extra InteractableBase variable in PC_Menu named CurrentUIFocus, which whenever the player successfully transferred focus to another button, would call from InteractableBase::OnFocus to PC_Menu::FocusTargetChanged with the new Interactable to focus on.



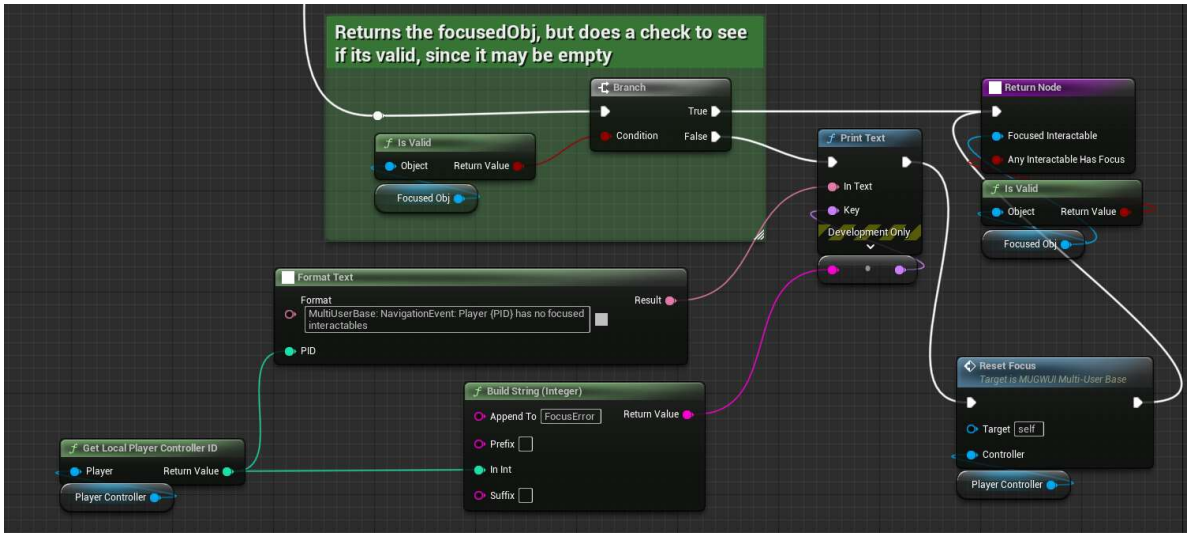


With the CurrentUIFocus updated, every 0.5 seconds OR when the player clicks their mouse, the UserFocus will be reset to whatever CurrentUIFocus is.





We also made changes that in MultiUserBase::CheckWhichInteractableHasFocus, if the player is found to have no InteractableBase focused within the MultiUserBase object instance, it will use MultiUserBase::ResetFocus to ensure it will be focused in the widget it should be.



After Iterating through all the items in the Interactables array, if no object inside the array is found to be focused by the user, it will alert the user and reset the focus.

Flex Marks - 3

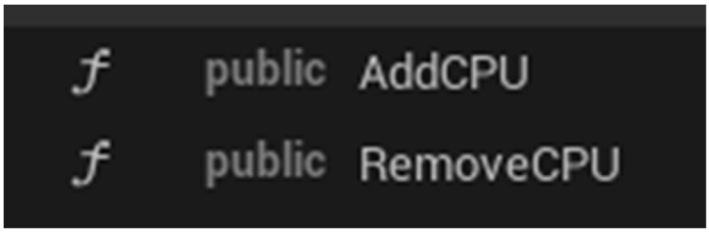
Added UI support for adding and removing CPU's when selecting a character

Why: Users need to be aware of how many CPU's have been added to the game. They also need to be able to add or remove CPU's at will to tailor the match to their preferences.

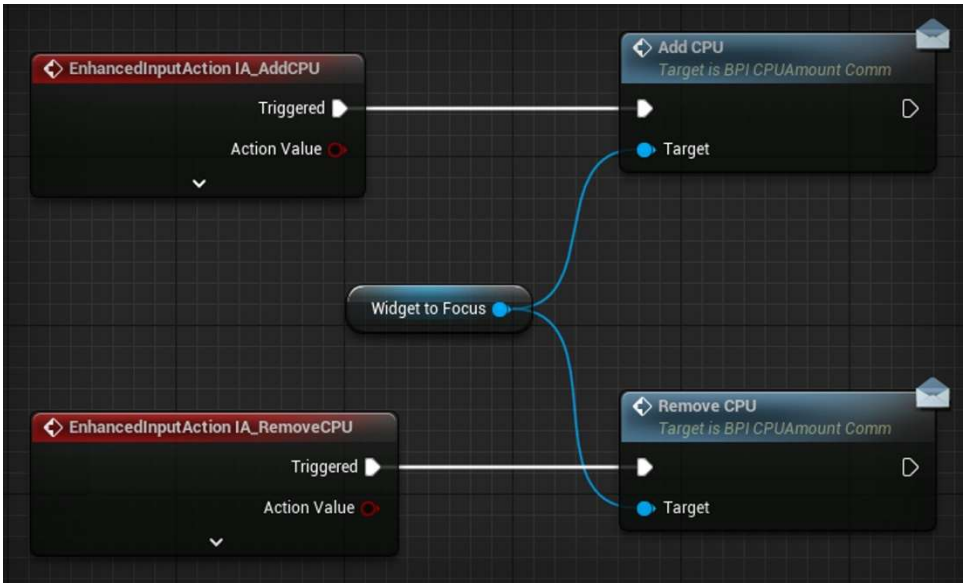
Adding and removing CPU's

Our team decision on how users could add/remove CPU's was to utilize the left and right triggers on the back of the controller. When the player is on the character select screen, by hitting the triggers you can toggle the amount of CPU's in your game.

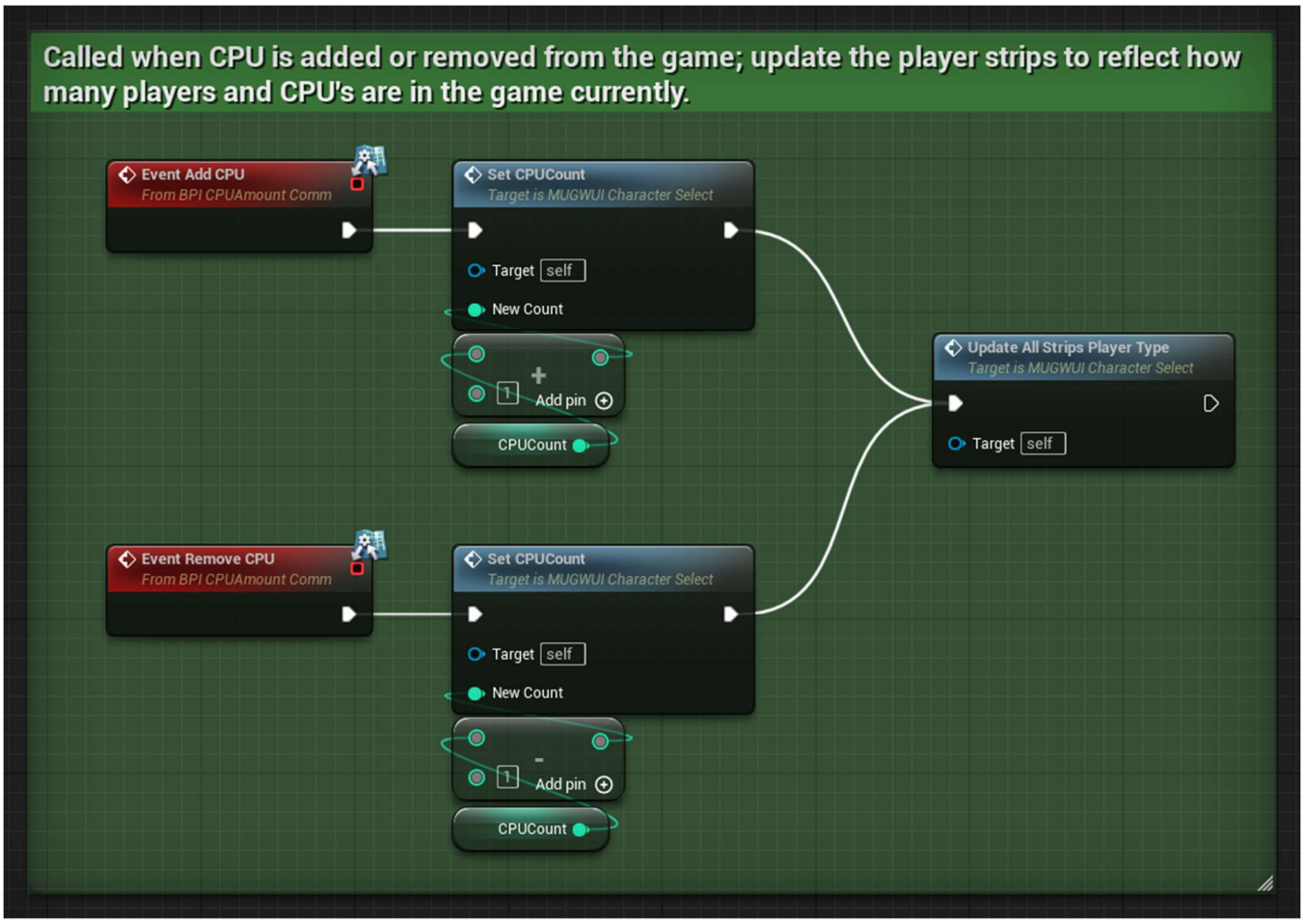
To communicate to the UI, I created a new Blueprint Interface integrated into `WBP_CharacterSelect` called `BPI_CPUAmountComm`. The `PC_Menu` controller then sends the `AddCPU`/`RemoveCPU` events to the `WBP_CharacterSelect` through the currently cached widget.



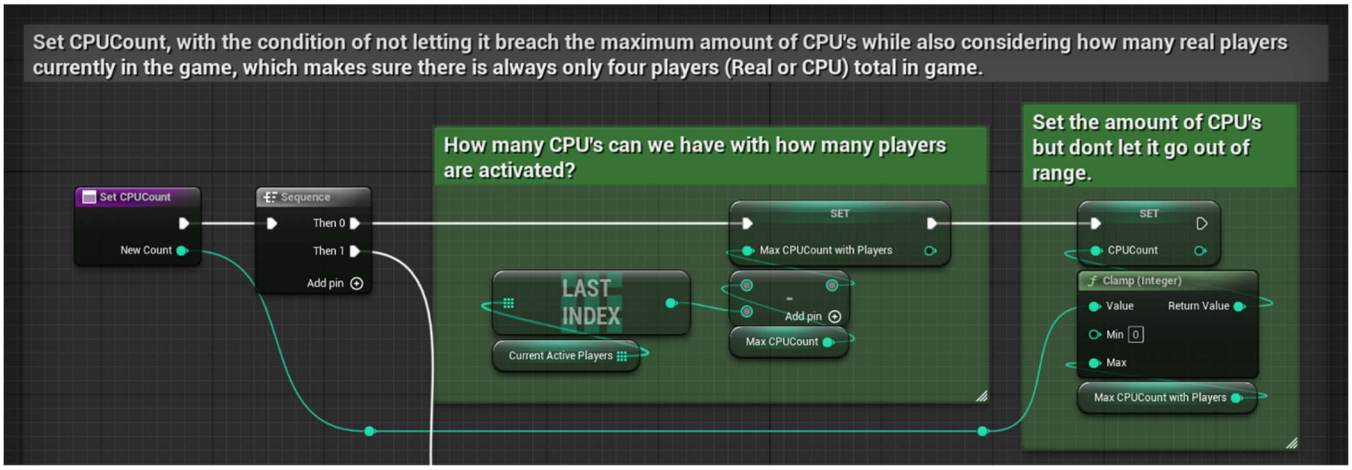
Functions available from `BPI_CPUAmountComm`



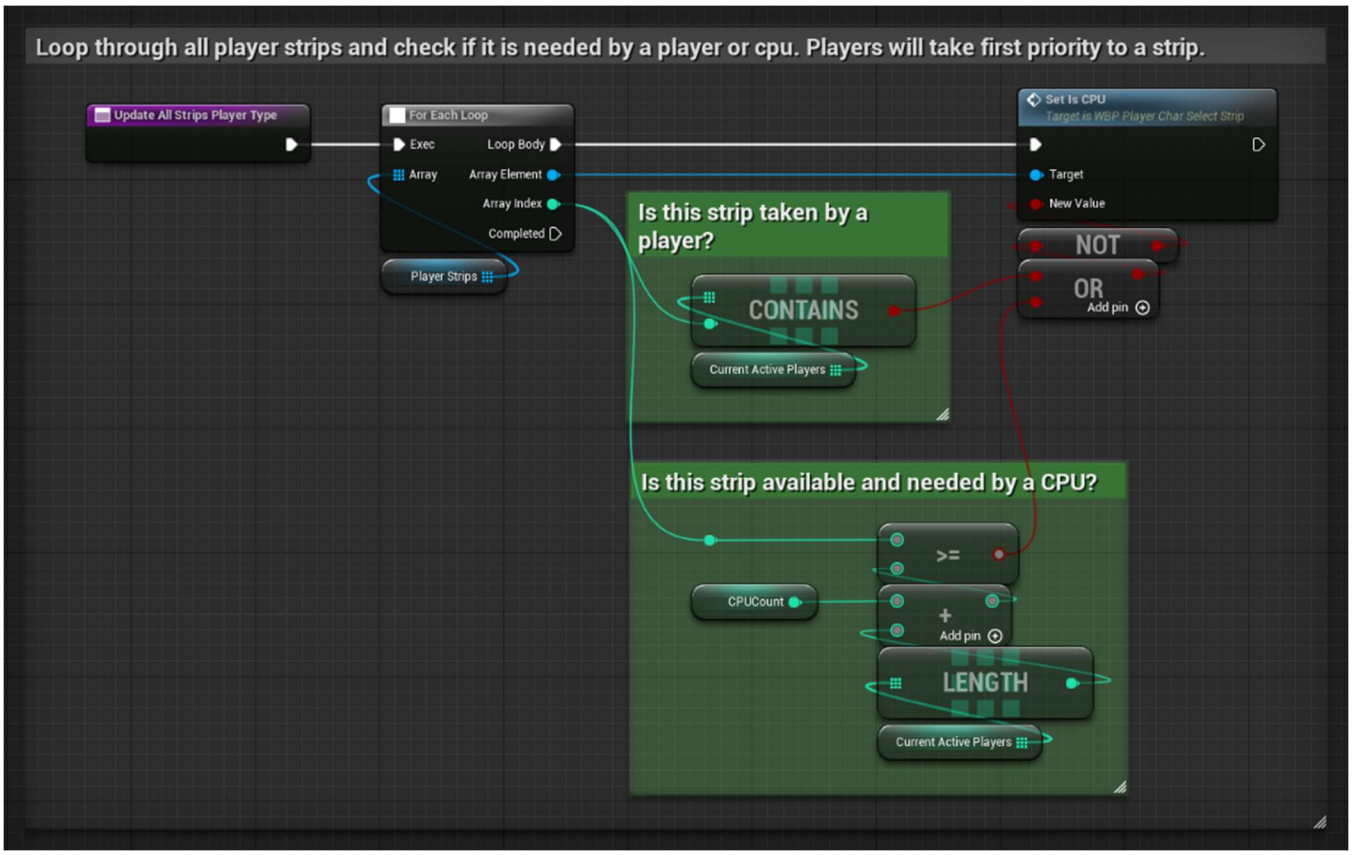
`PC_Menu`, calling `AddCPU` and `RemoveCPU` on the `WidgetToFocus` (which will only work on Character Select, since its the only one with the interface implemented.)



`WBP_CharacterSelect::AddCPU` & `WBP_CharacterSelect::RemoveCPU`, which will increment or decrement the `CPUCount` with `SetCPUCount`



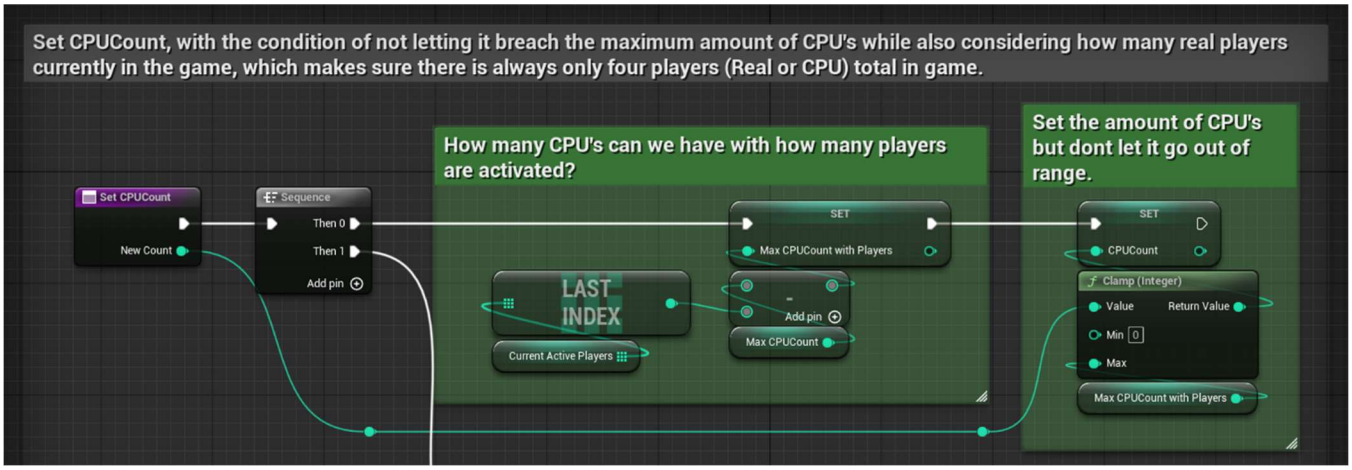
`WBP_CharacterSelect::SetCPUCount`, which will update the `CPUCount` while keeping it within the bounds of 4 players total. (The cut off part of this image is a debug print to notify how many CPU's there are, how many CPU's there can be, and if there are more CPU's than that amount).



`WBP_CharacterSelect::UpdateAllStripsPlayerType`, which will tell the Character Select Strips if they are for a CPU or not.

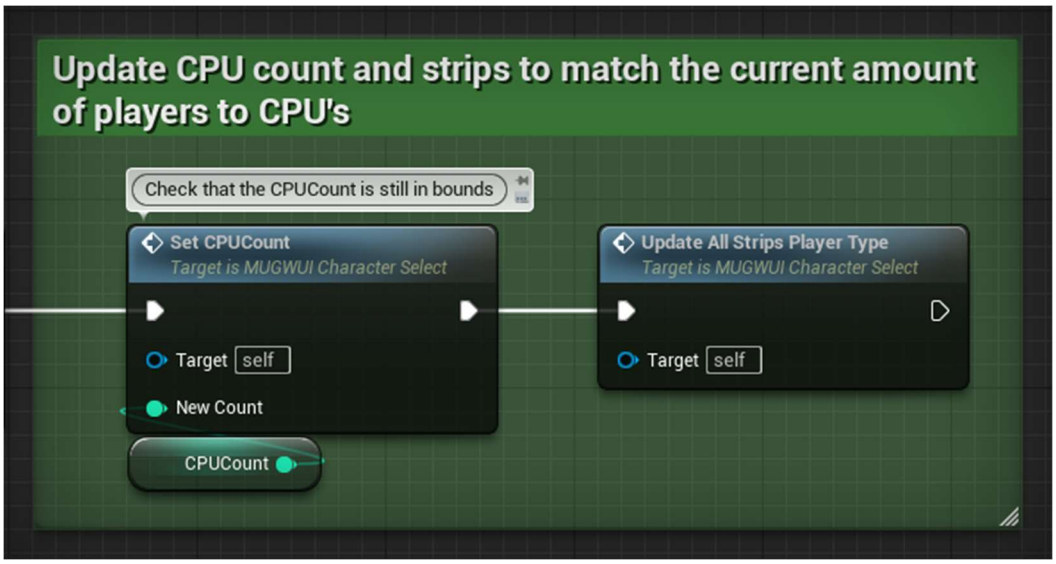
Player activation while CPU's are present

Because of the player limit (4 players max), there can only be as many CPU's as there are **inactivated** players. Every time one of the `BPI_CPUAmountComm` events are called in Character select, and since `CurrentActivePlayers` will have the correct amount of activated players, that can be used to calculate how many open slots are available for a CPU to fill.

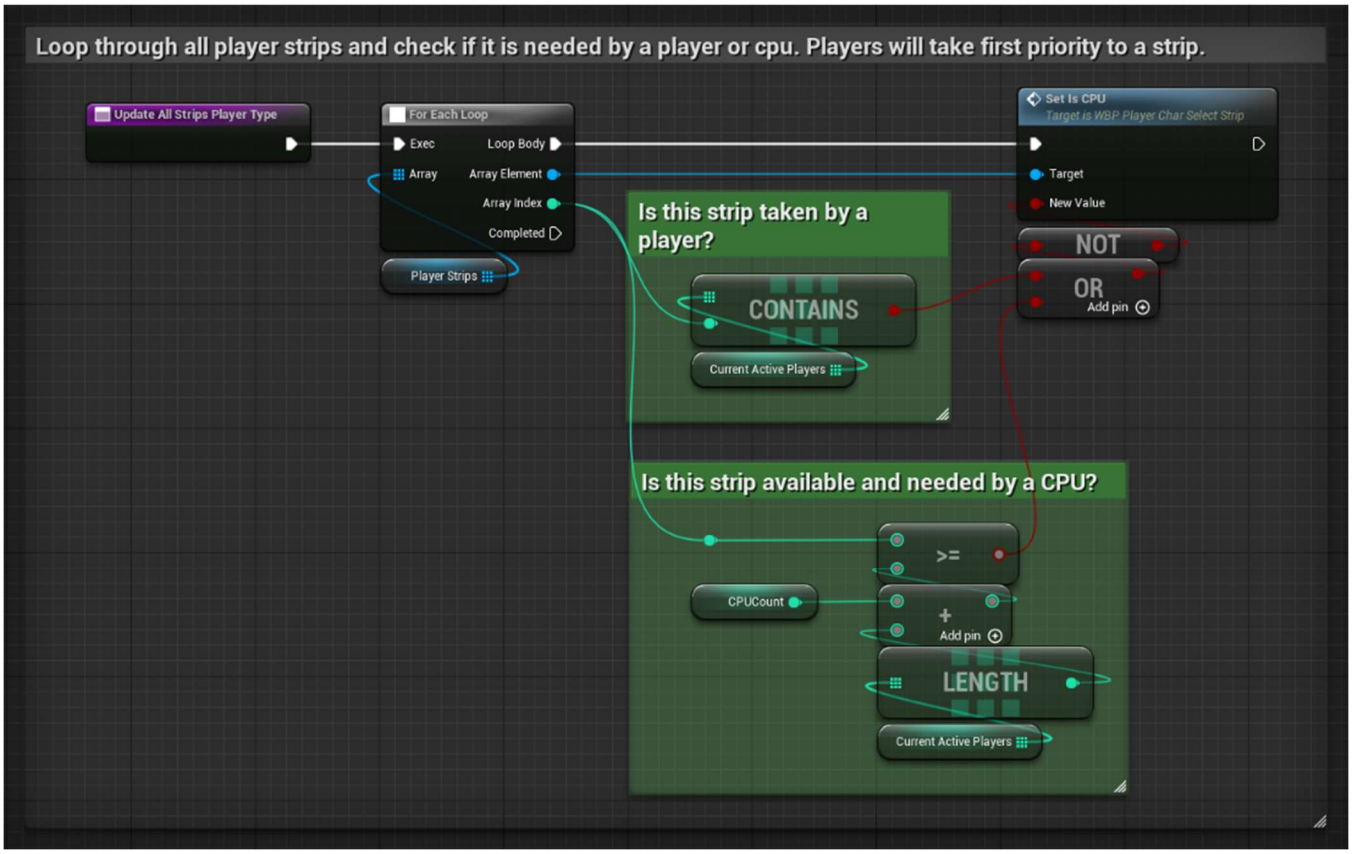


`WBP_CharacterSelect::SetCPUCount`, which will update the `CPUCount` while keeping it within the bounds of 4 players total. (The cut off part of this image is a debug print to notify how many CPU's there are, how many CPU's there can be, and if there are more CPU's than that amount).

And then, when `WBP_CharacterSelect::PlayerActivated` is called, it will run `SetCPUCount` and `UpdateAllStripsPlayerType` to make sure that there isn't too many CPU's left and "insert" the new player between the CPUs and existing players.



`SetCPUCount` and `UpdateAllStripsPlayerType` at the tail end of `WBP_CharacterSelect::PlayerActivated`



`WBP_CharacterSelect::UpdateAllStripsPlayerType`, which will tell the Character Select Strips if they are for a CPU or not.