# Camera System

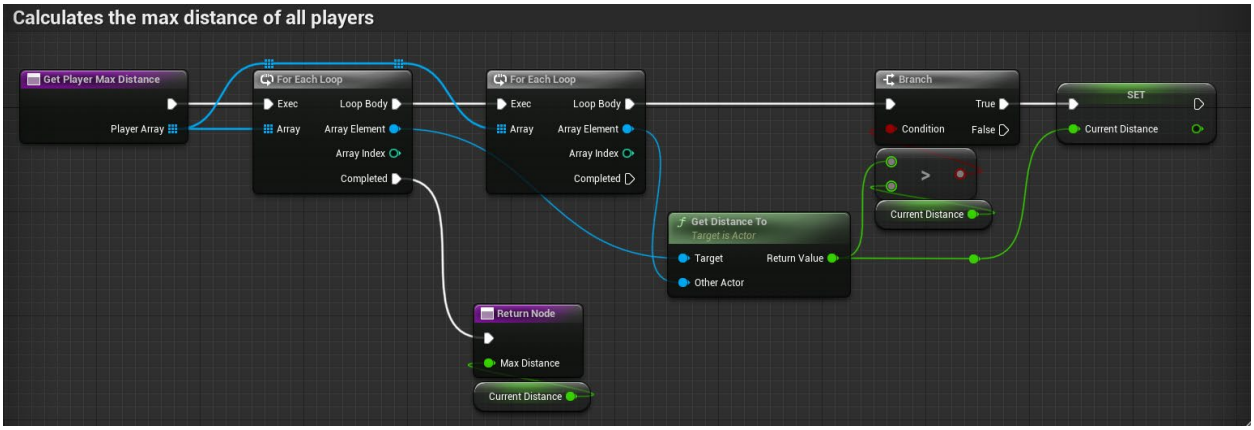
The topic I focused on for Modern Tech this term was the game's dynamic camera system. This system had two important features it needed to cover:

1. Fit all characters on screen at once.
2. Position itself between all players.

In this document, I will summarize my work by going over how the system works and was developed.

## Camera Zoom

To fit all characters on screen, the camera needs to zoom in when characters are close together or zoom out when they are far apart. So first, we need to figure out what the furthest distance between any two players is. To do this, the camera loops through an array of all the players and checks the distance between each player. If this value exceeds the current value of the "*CurrentDistance*" local variable, it is assigned as the new value of *CurrentDistance*. Once the loops complete, the value of *CurrentDistance* is returned.

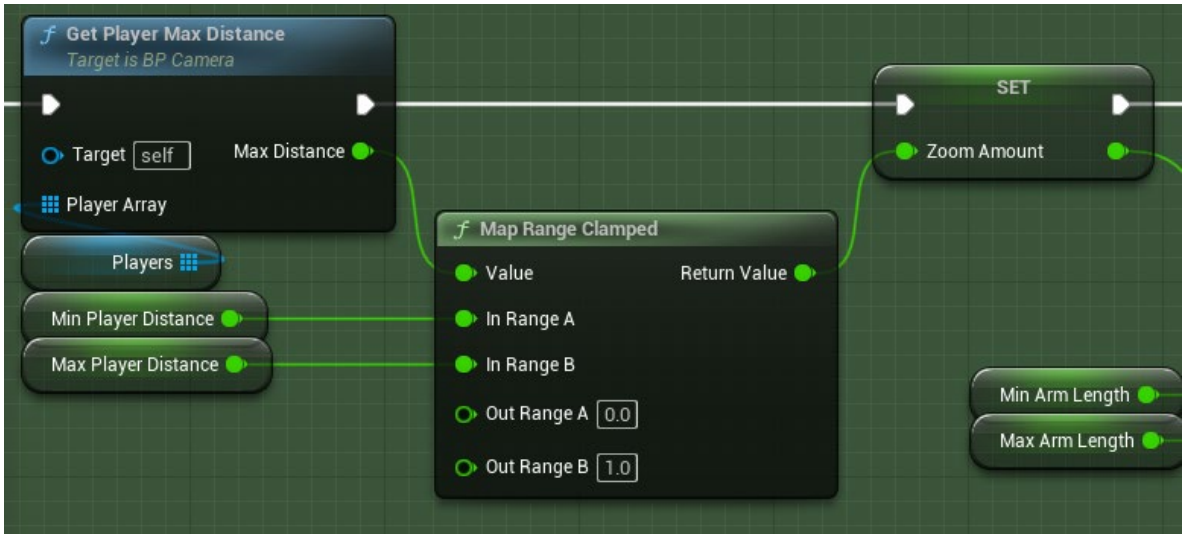



*1 Function for determining the furthest distance between any two players.*

Now that we know how far the two furthest players are, we need to figure out how far the camera needs to zoom out to fit all players on screen. To do this, we first need to know how close the characters can be before the camera stops zooming in, and how far they can be before it stops zooming out. Since the two stages in our game are of vastly different size, I set up two instance variables so that the level designers can set these values manually.

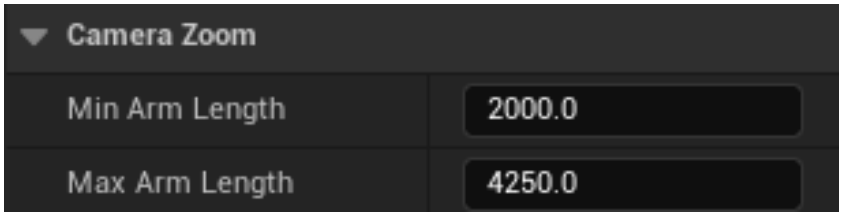

*2 Instance variables for the Minimum and Maximum player distance.*

Using these two variables, we can now map the distance the players are to a value between 0 and 1, with 0 being the Minimum player distance, and 1 being the maximum player distance. We clamp this value to 1, as we don't want the camera zooming out further than the level designer intended. This value is then stored as a variable called *ZoomAmount*.



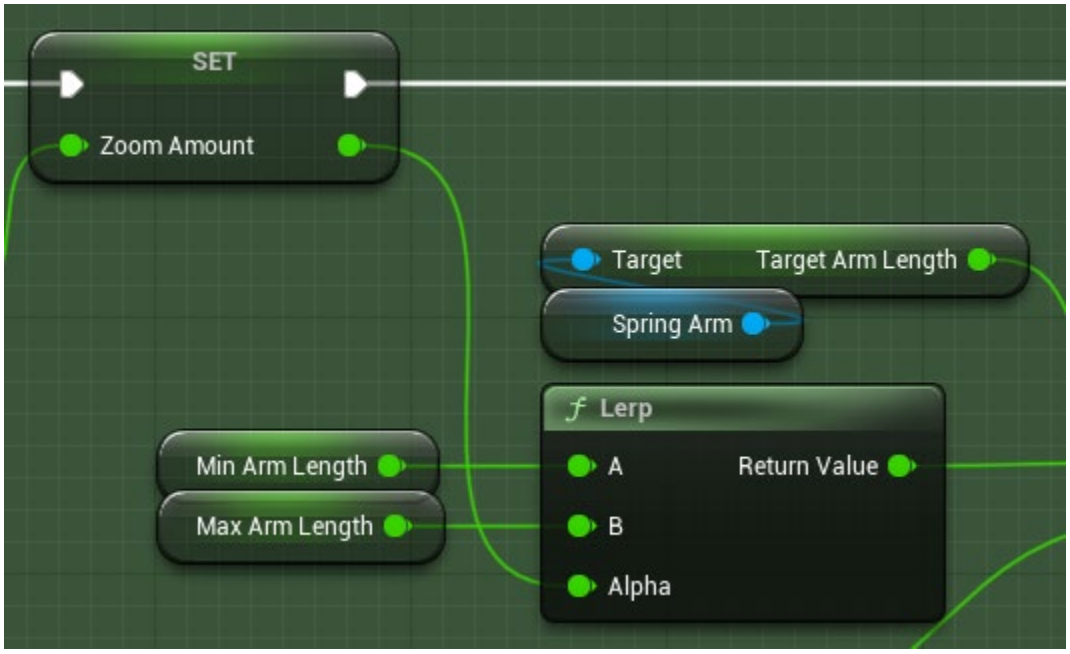

*3 Blueprint code for mapping the max player distance to a value between 0 and 1.*

Because Unreal cameras don't use a lens to zoom in or out, a spring arm is attached which pulls the camera in or out to create the "zoom" effect. So, to figure out how long the spring arm needs to be for the player's current distance, we need to know the values for the camera's spring arm when it is fully zoomed in (minimum length) and fully zoomed out (maximum length). Again, because the two levels are different sizes, I set these to be instance variables for the level designer to determine.



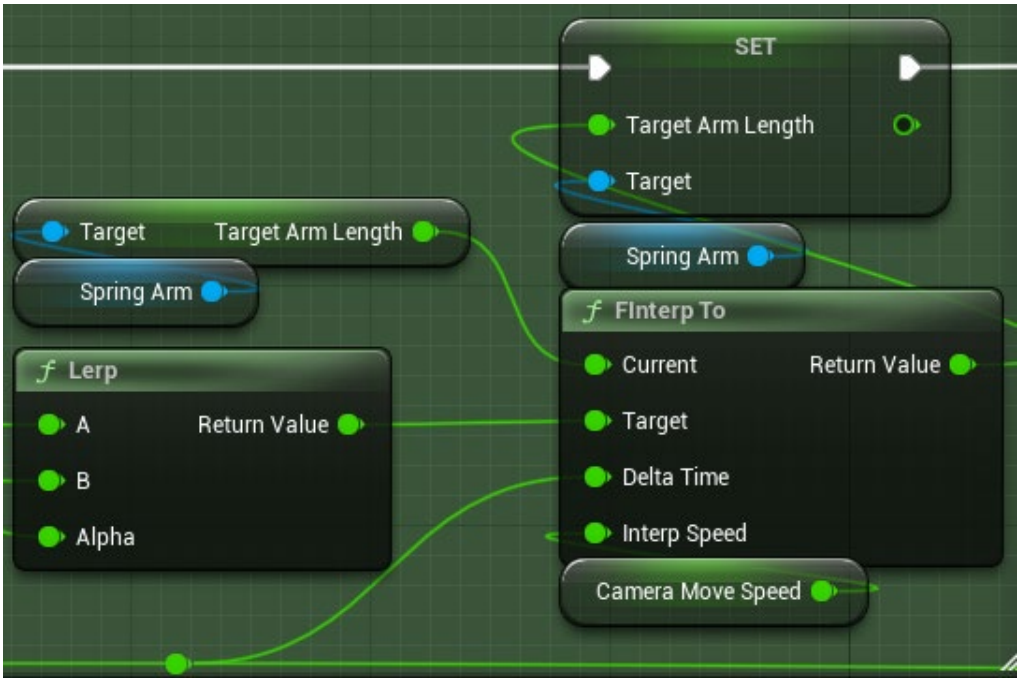

*4 Instance variables for the Minimum and Maximum length of the spring arm for the given level.*

Using these values to determine the range, and the *ZoomAmount* value we mapped earlier, we use a Lerp node to find what the corresponding value on this range is.




*5 Blueprint code for determining what length the spring arm should be.*

Finally, an *FInterpTo* node is used to set the Spring Arm Length. This node uses Delta Time with a set Camera Move speed to smoothly change the value over time,  rather than jumping to the desired value immediately.

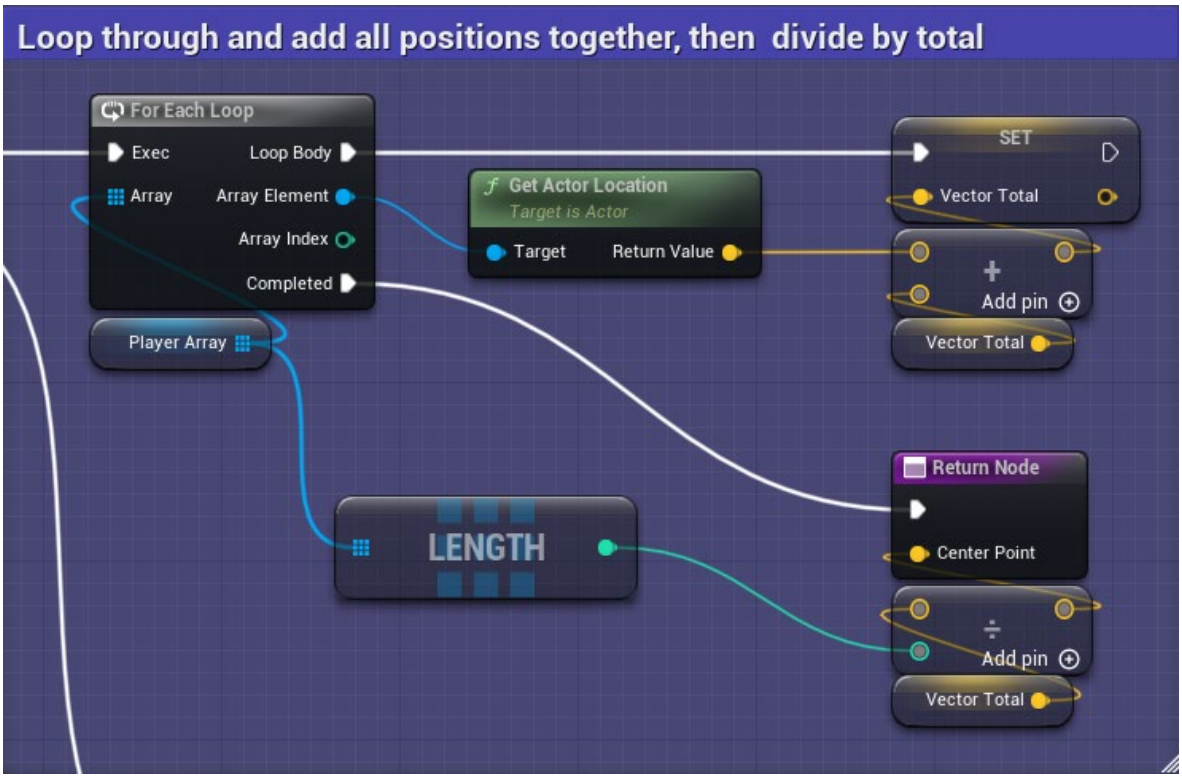



*6 Blueprint code of the arm length being set.*

# Tracking Position

Now that the camera is zoomed to it's desired value, it also needs to track the position of the players. To do this, we need to find the center point of all players in the match, and smoothly move towards it.

Finding the centre point is an easy task, as all that needs to be done is adding all player positions together, and then dividing by the player count.

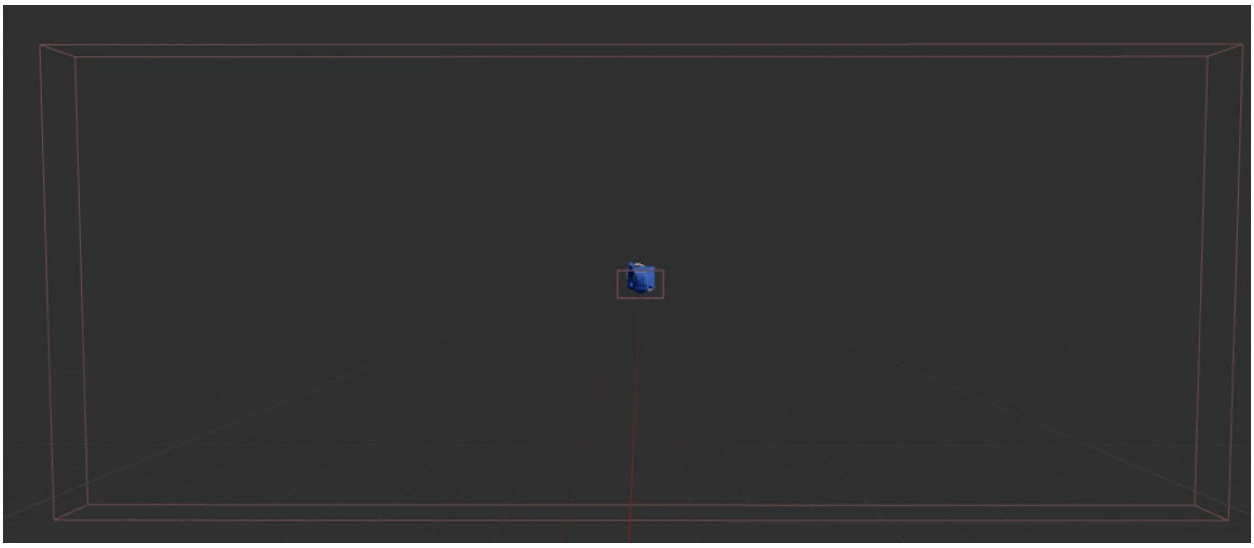



*7 Blueprint code of the center point being calculated.*

This, however, presents a problem: how do we prevent the camera from showing areas of the map that shouldn't be seen? We can set a boundary at the edge of the screen, but as the camera zooms in and out, it's hard to find a boundary line that works for all zoom values.

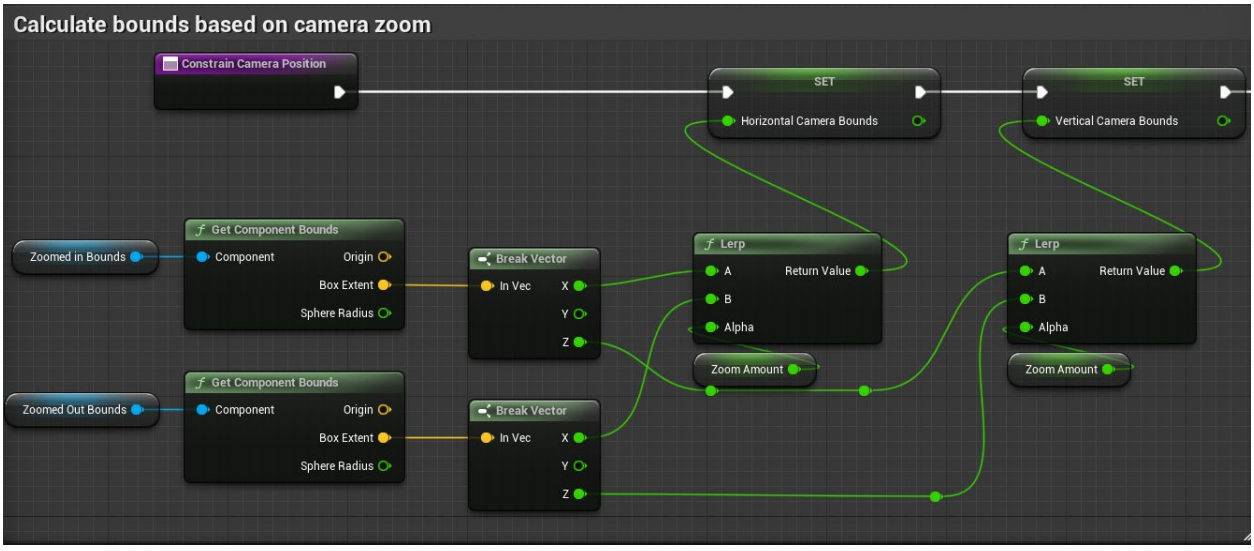
Luckily, we still have that 0 to 1 value we mapped in the previous section. Using this value, we can determine where the bounds of the camera should be. All we need now is the bounds of a fully zoomed in camera, and the bounds for a fully zoomed out camera  to set our range, similar to how we needed the minimum and maximum zoom values.

To find these, I set up two collision box components in the Camera actor to represent the bounds for the camera at it's fully zoomed in and fully zoomed out state.



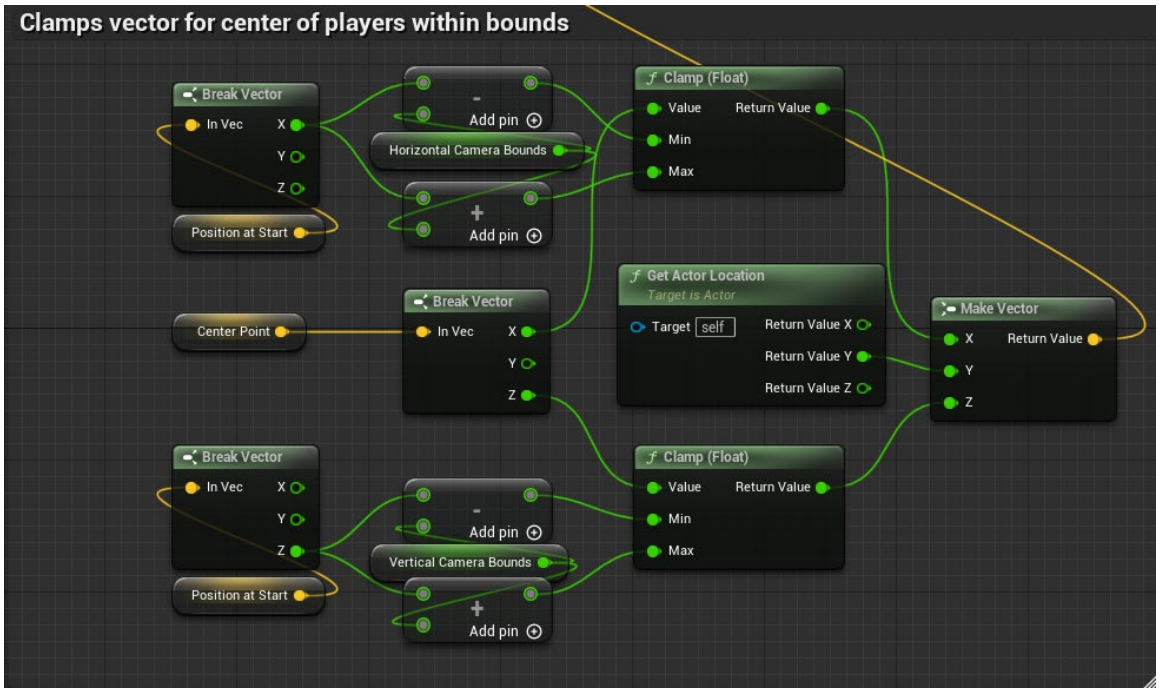
*8 Screenshot of the two collision boxes which represent the minimum and maximum bounds of the camera.*

Once these are set by the Level designer, we can use the X an Z bounds of each box to find the 0 and 1 values of Lerp nodes and use the Zoom amount to find where the X and Y bounds of our camera bounds need to be.



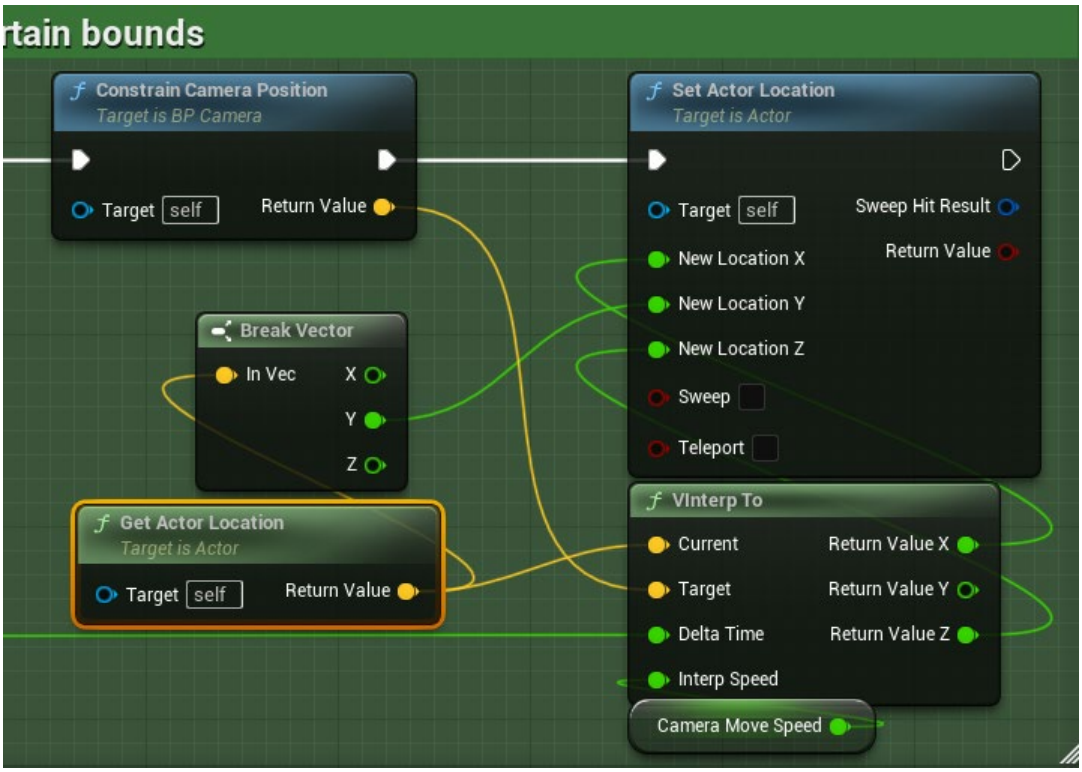

*9 Blueprint code of the new bounds being calculated.*

Now that we have our bounds, we just need to clamp the value of the centerpoint we calculated earlier, to keep it within our bounds before we move the camera to it.




*10 Blueprint code of the camera's new position being clamped.*

Finally, we use a *VInterpTo* node to smoothly move the camera to its new position, like how an *FInterpTo* node was used for the spring arm length.




*11 Blueprint code of the new camera position being set.*