Player Character Blueprint

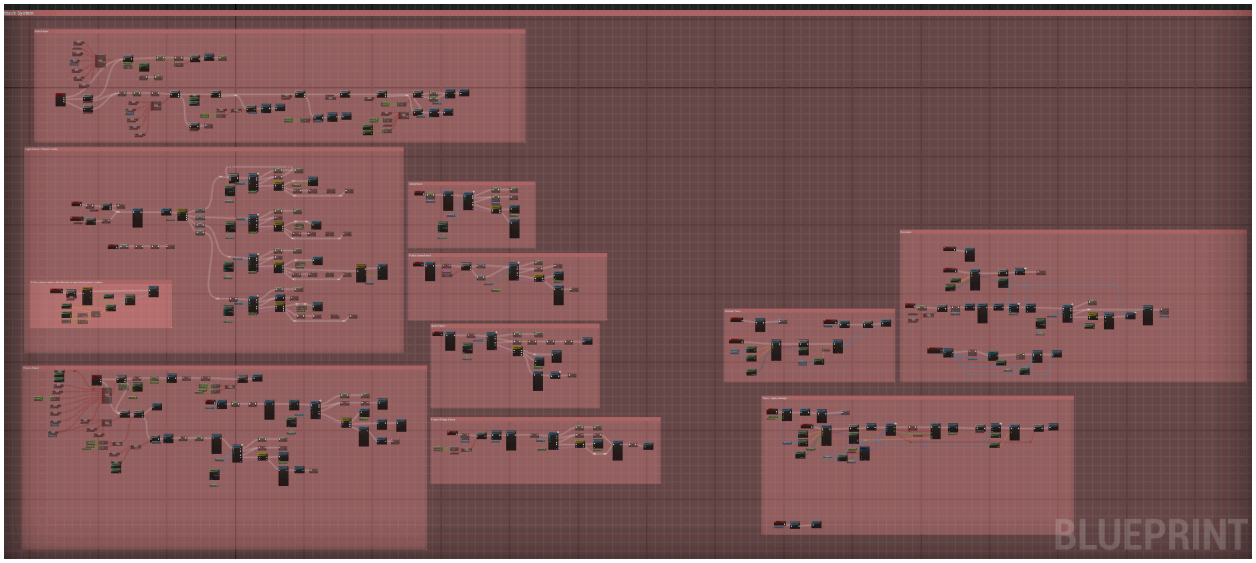

Figure 1: Overview for the player character's offense (attack system).

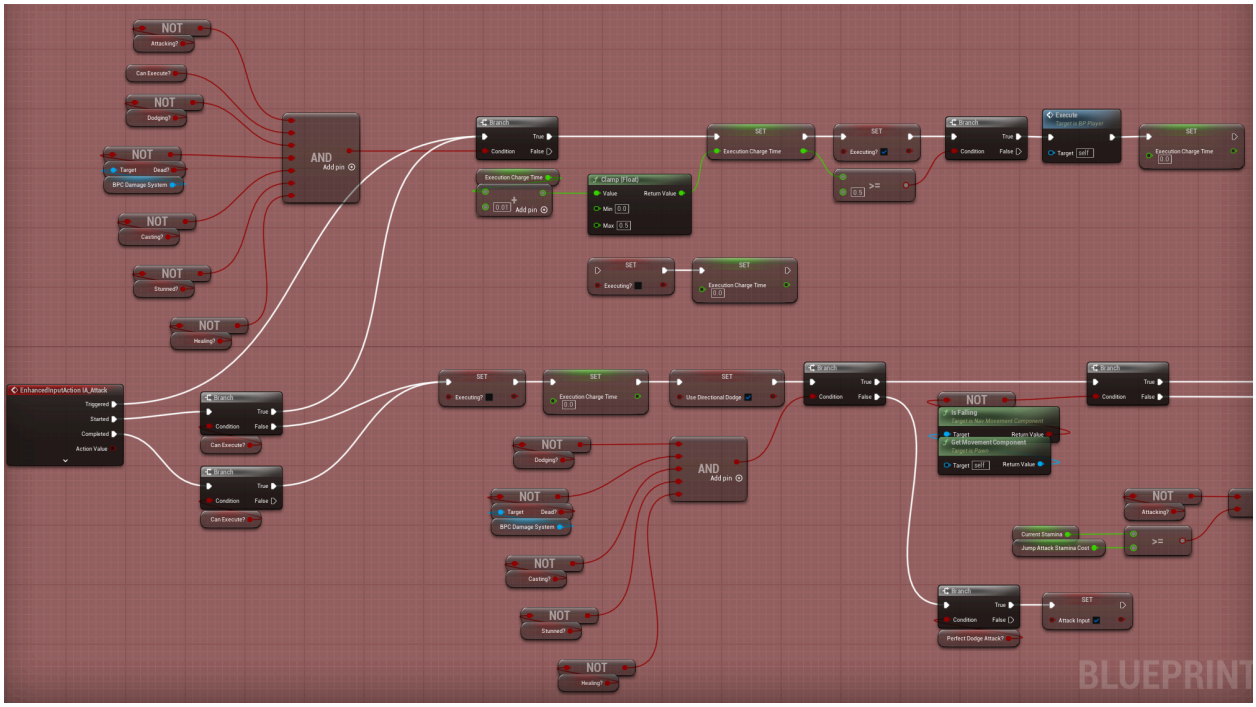

Figure 1.2: Left-Mouse-Click: (Hold or Tap) When input is given, check if the player is able to attack based on various conditions such as stunned state, already attacking, performing a dodge, etc. If the input is held for 0.5 sec, checks if there is an enemy within range that can be executed, if true, performs the execution, if false, doesn't do anything. If the input is a tap and conditions are met, will check additional conditions for special attacks (sprint attack / jump attack). Finally, if neither are true, will initiate a light attack combo-sequence (starting at 1).

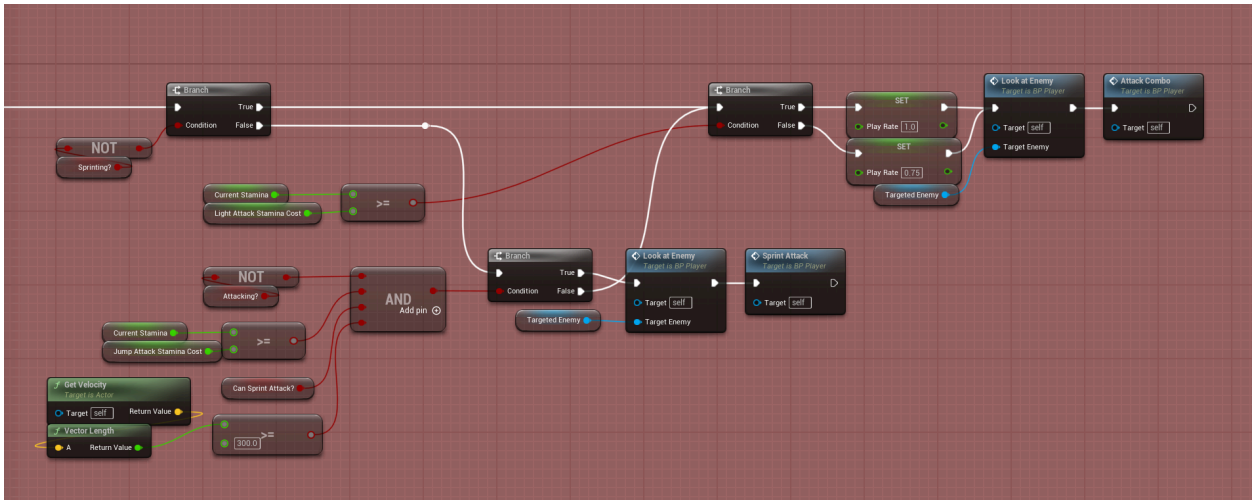

Figure 1.3: Continued logic from figure 1.2, if the player is sprinting, and has enough stamina, they do a sprint-attack, else they do a light attack combo. Additionally, if stamina is fully depleted, the player can only do light attack combos, but at a decreased rate of 25%.

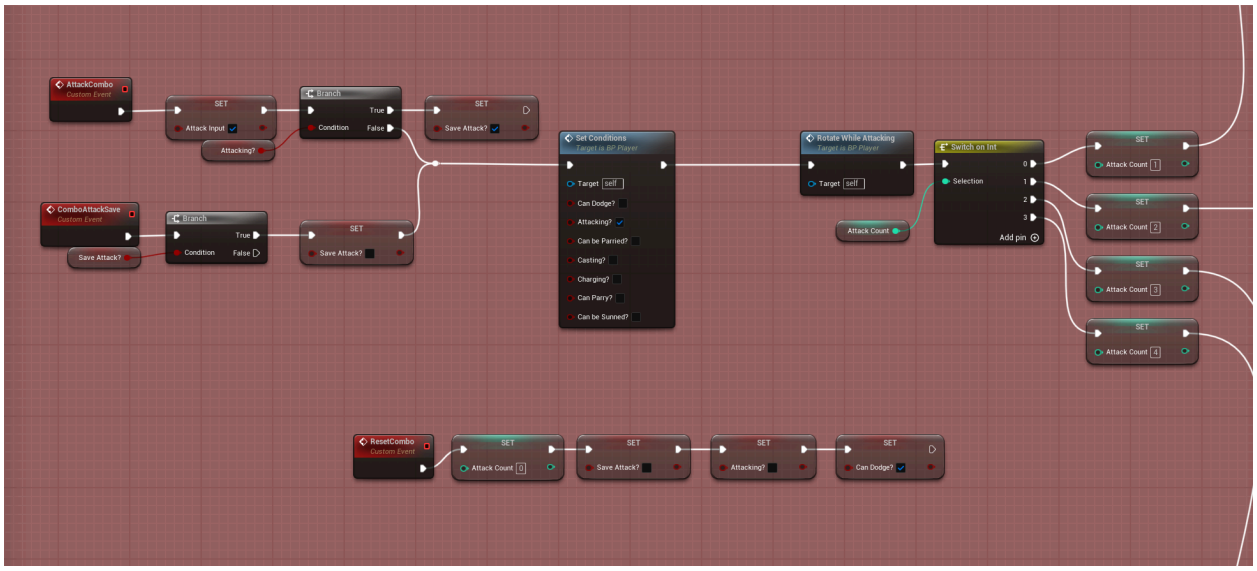

Figure 1.4: Light attack combo checks if an attack input is given during a sequence, if true saves the attack (to move on to the next combo in the sequence).

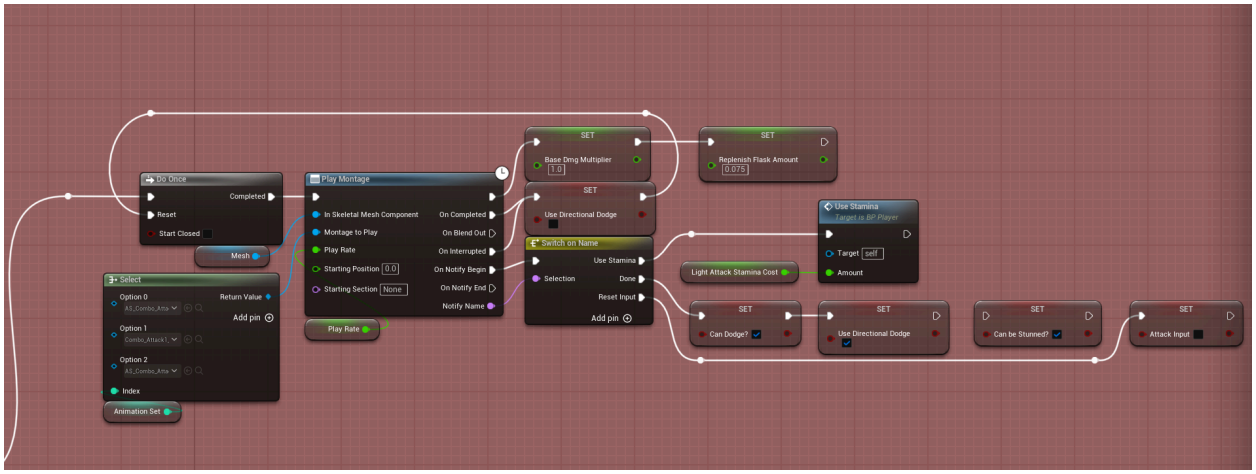

Figure 1.5: Logic for actually doing the light attack logic and animation. Each sequence in the combo has different values for stamina consumption, damage multiplier (scales with different weapons), replenishing flask (if fully depleted), etc. The select node on the bottom-left, chooses between 3 different animation sets based on the type of weapon currently equipped (heavy sword, regular sword, twin daggers). Sprint and jump attacks use a similar set up to this.

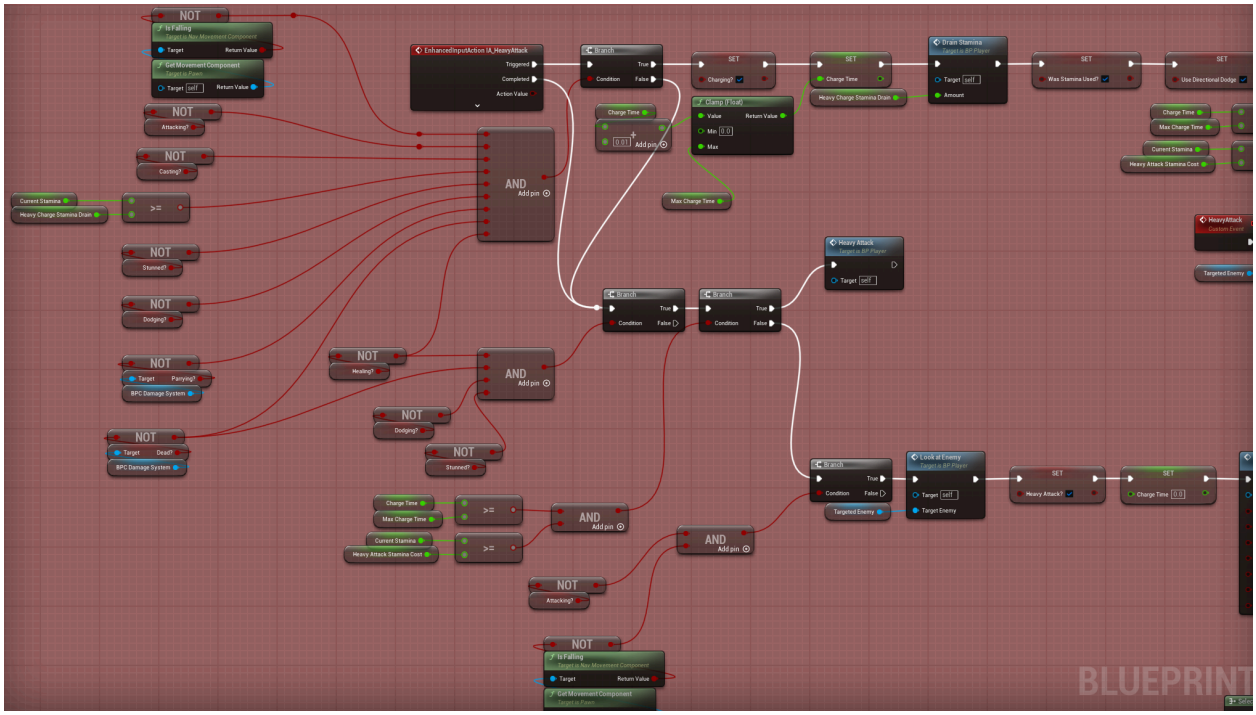

Figure 1.6: Right-Mouse-Clock (heavy attack). Uses similar logic to regular attacks by checking various player conditions that prevent the player from attacking. Can be tapped once for a heavy attack, or held for a charged heavy attack.

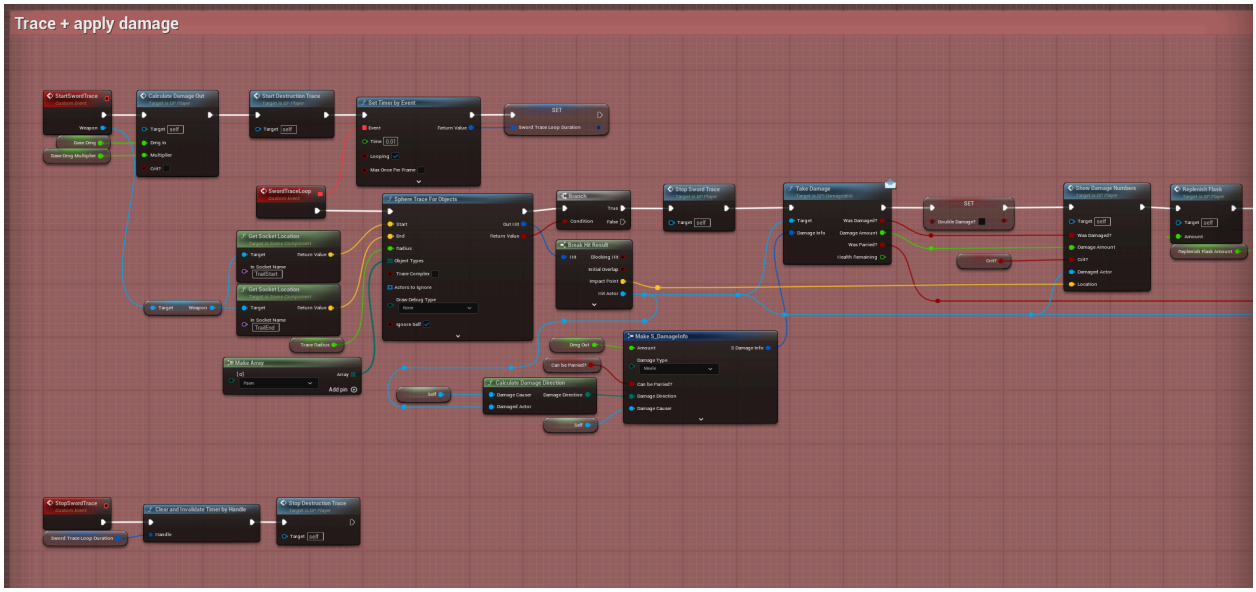

Figure 1.7: Logic for tracing sword location to apply damage. Once triggered, gets socket locations from the sword (start and end points) and does a sphere trace for a duration specified within individual attack montages. If collides with a pawn, stops the trace and applies damage to the pawn. *ShowDamageNumbers* is a custom function to display the amount of damage dealt to the enemy.

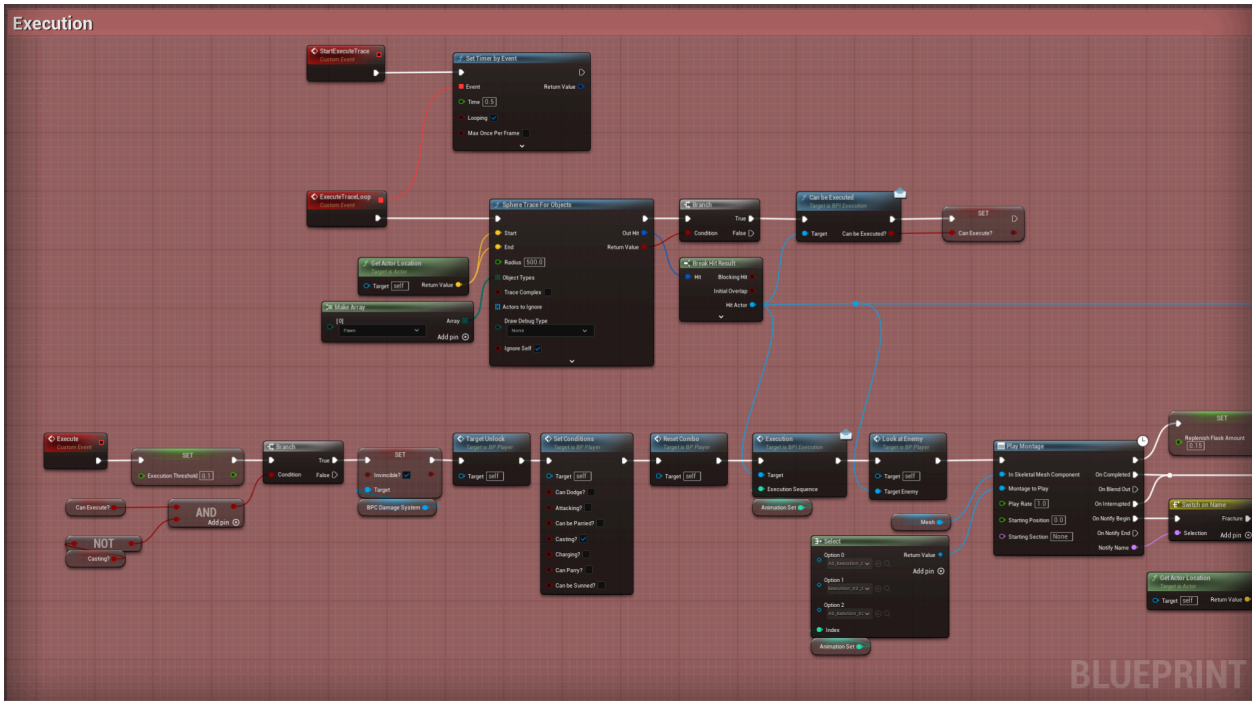

Figure 1.8: Execution. Starts a sphere trace around the player to check for enemy pawns. If collision is detected, calls the *CanBeExecuted* function for the enemy to check if they meet execution requirements (health threshold). If true and execution input is given, performs an execution on the enemy.

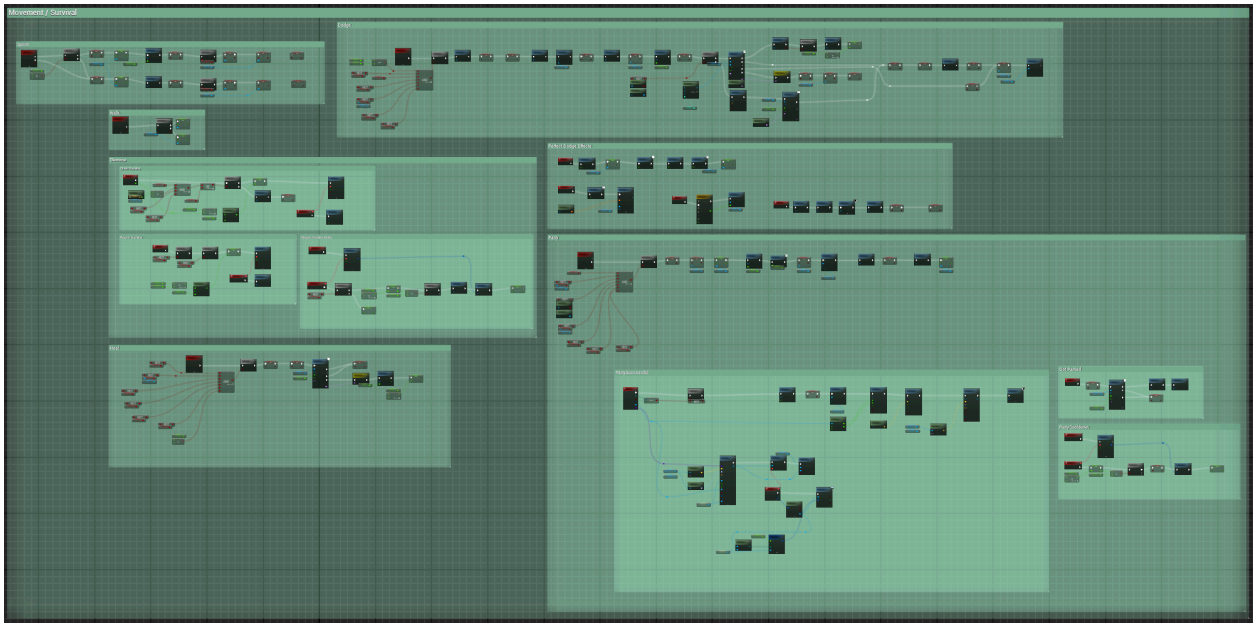

Figure 2: Overview for the player character's defense/movement.

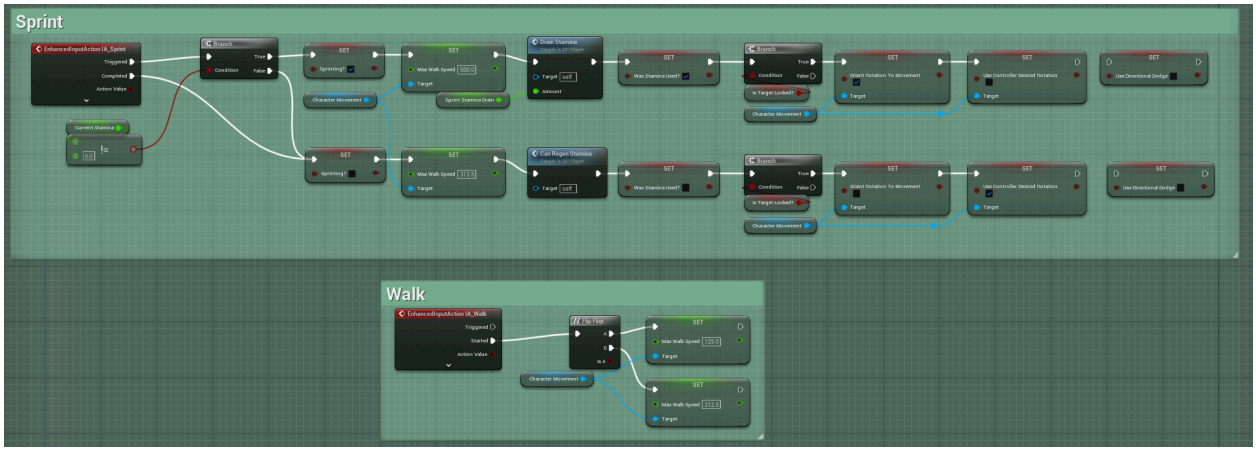

Figure 2.1: Sprinting and Walking. Holding **Shift** allows the player to sprint at the cost of stamina, if stamina fully depleted, stops sprinting. Tapping **Alt** toggles between walking and jogging, does not consume stamina.

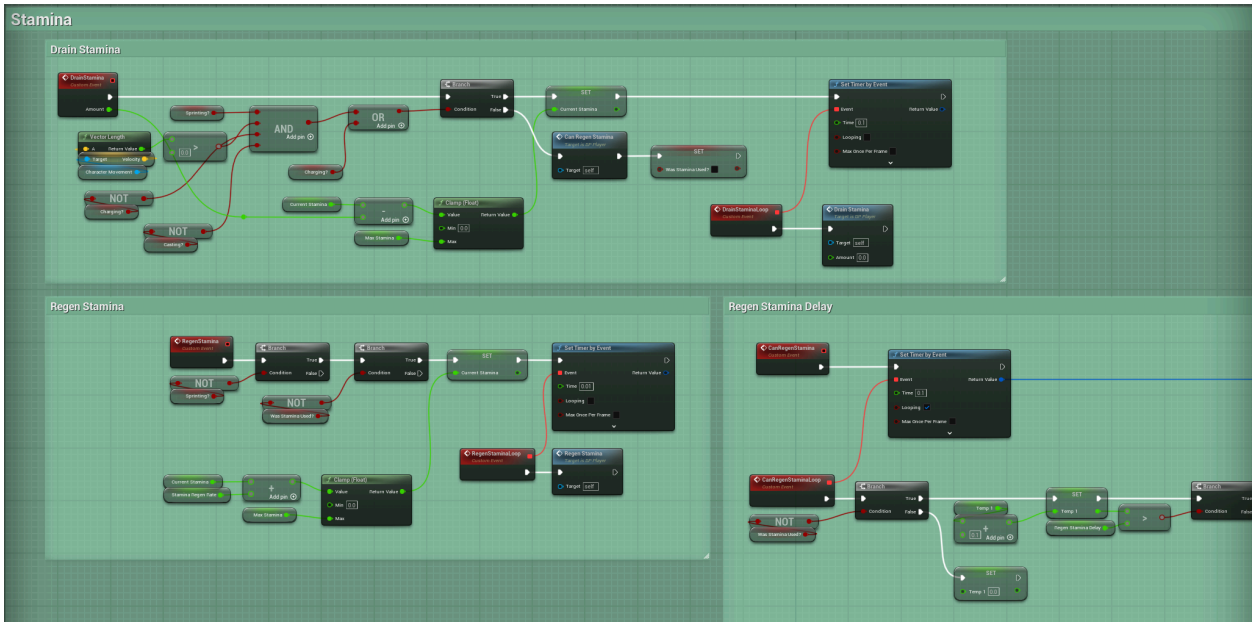

Figure 2.2: Stamina system. *DrainStamina* is used for tasks that slowly drain stamina like sprinting or charging a heavy attack. *RegenStamina* regenerates stamina at a given rate which can be increased by equipping better armor. *CanRegenStamina* delays stamina regeneration after it's been used.

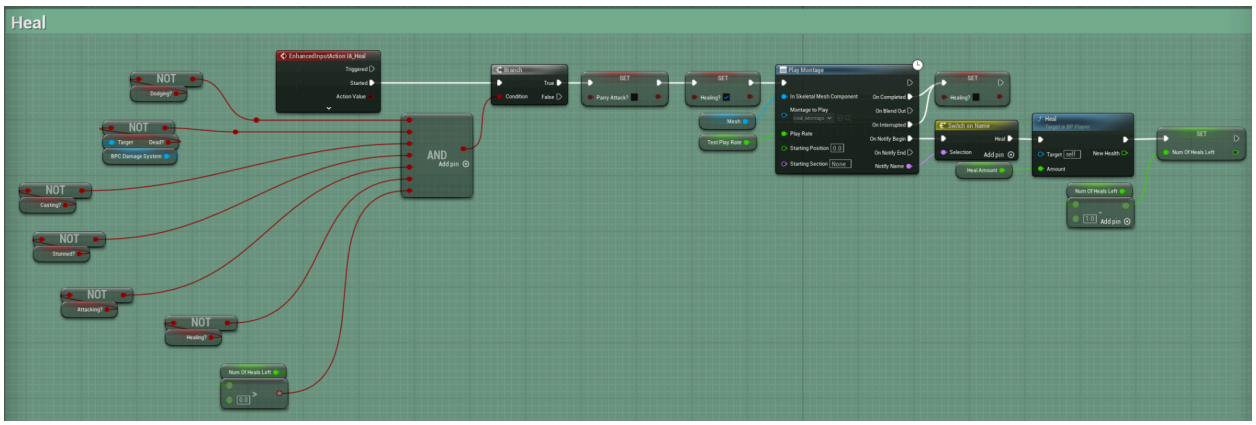

Figure 2.3: Healing. Checks for conditions that prevent healing like stunned state, attacking, etc. If passed all checks, plays healing animation and regains some health. The player starts with a flask size of 3 which can be increased by equipping better gear or regaining an extra flask during combat.

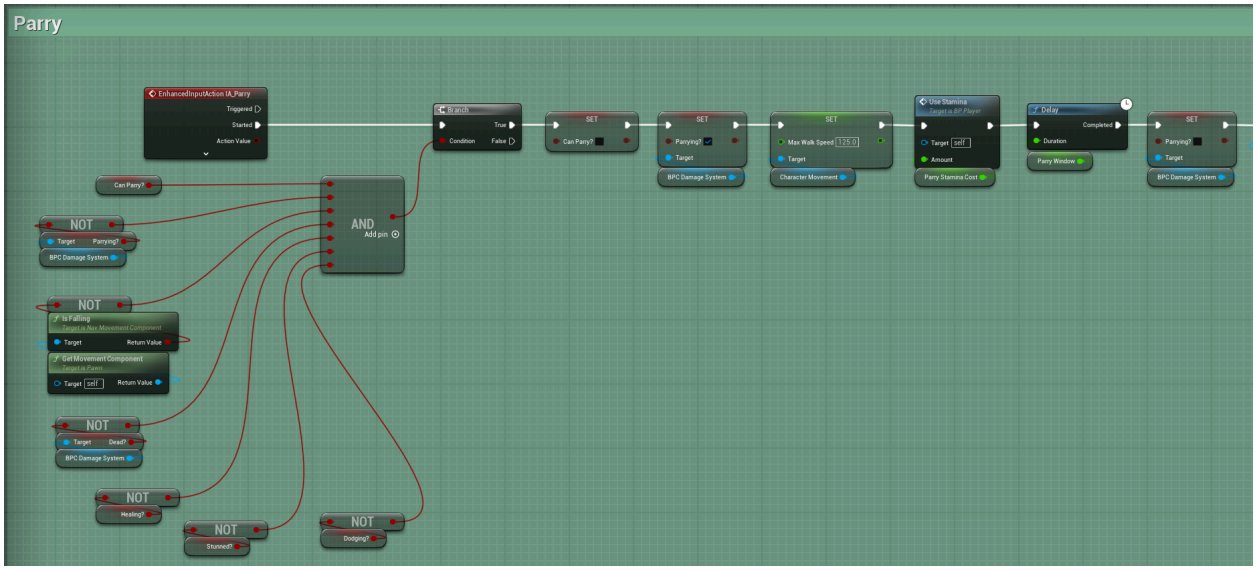

Figure 2.4: Parrying. If passes conditions, sets player state to parrying for a short duration assigned by the ‘ParryWindow’ variable. If an attack lands during parrying state, successful parry will be called.

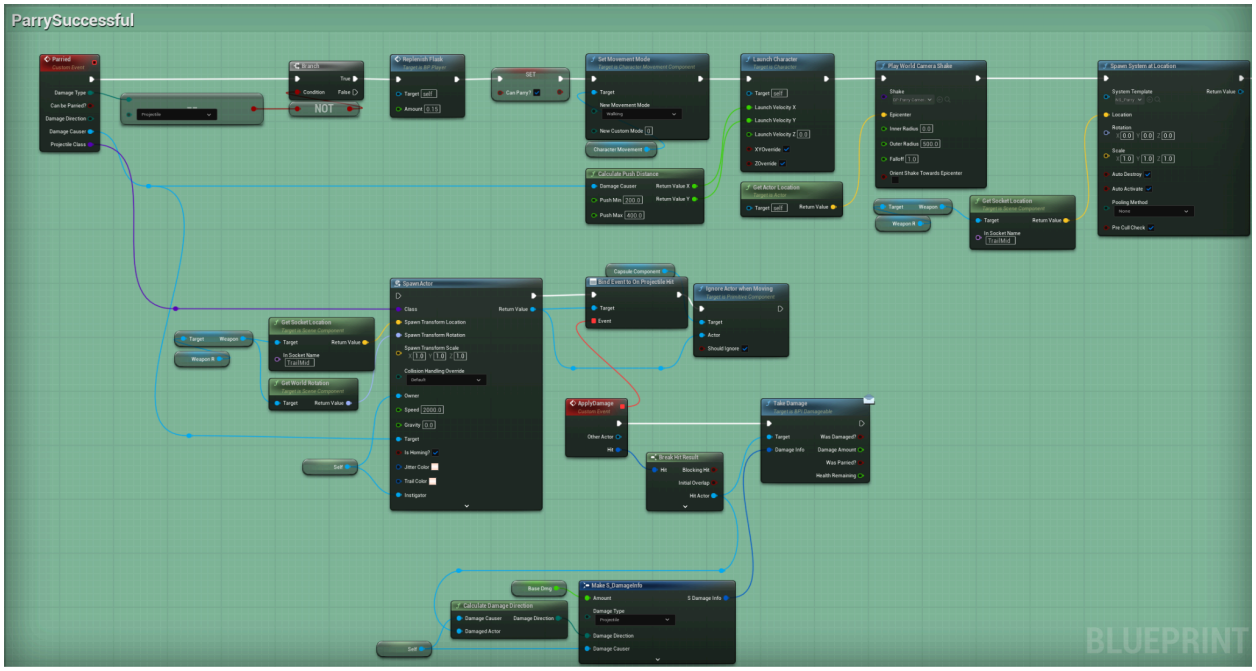

Figure 2.5: Successful Parry. When called, checks for damage type, if projectile, deflects it back at the enemy. Else deflects all damage and plays visual and sound effects to indicate the player did something cool.

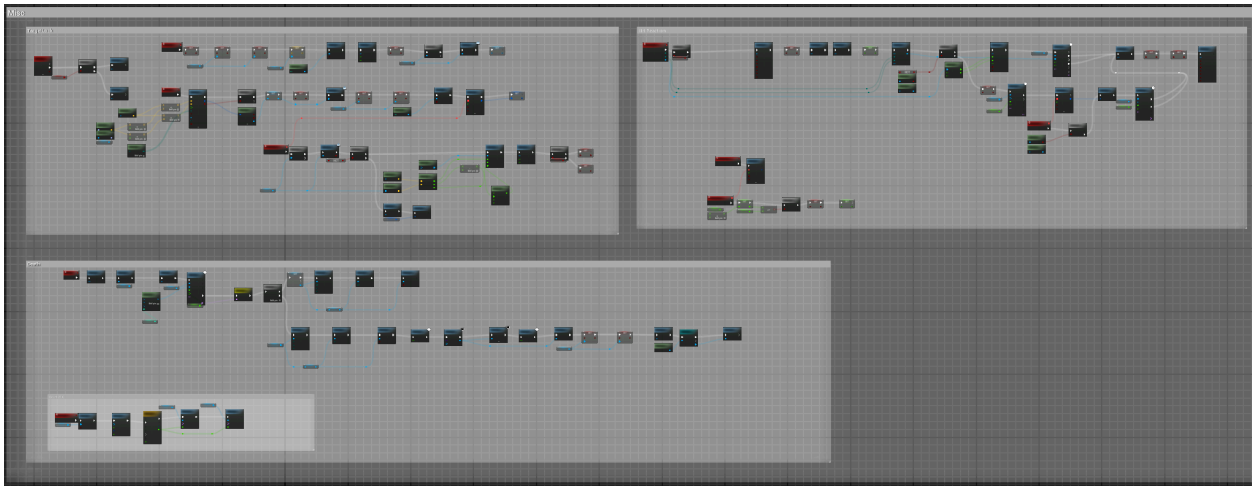

Figure 3: Overview for miscellaneous player logic.

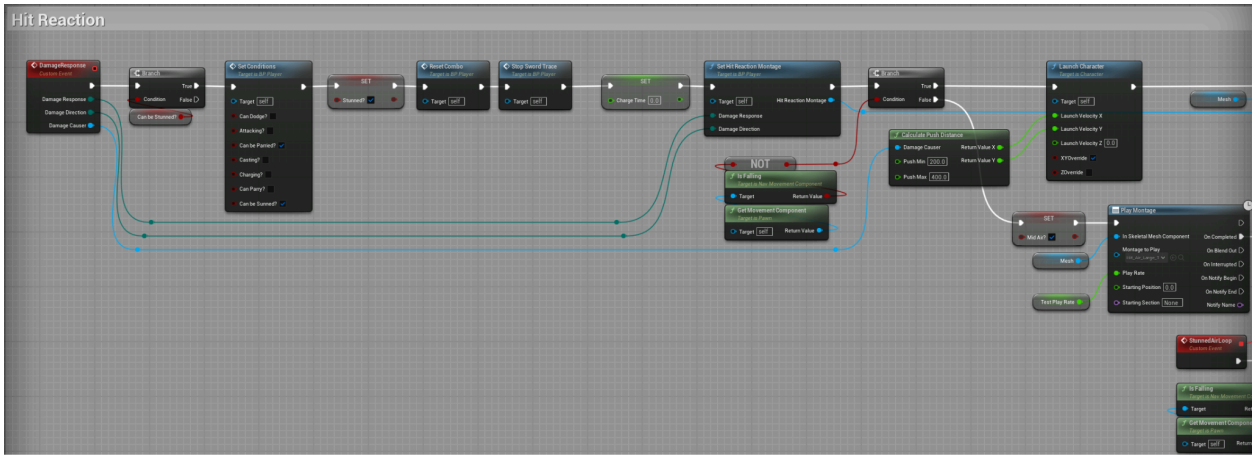

Figure 3.1: When the player is hit, *DamageResponse* is called that checks if the player can be stunned (i.e: not already stunned). If true, sets the player state to stunned, calls *SetHitReactionMontage* function, pushes the player backward, and plays appropriate animation based on damage direction and damage response. Also checks if the player is in air, which if true, makes the player fall on the ground.

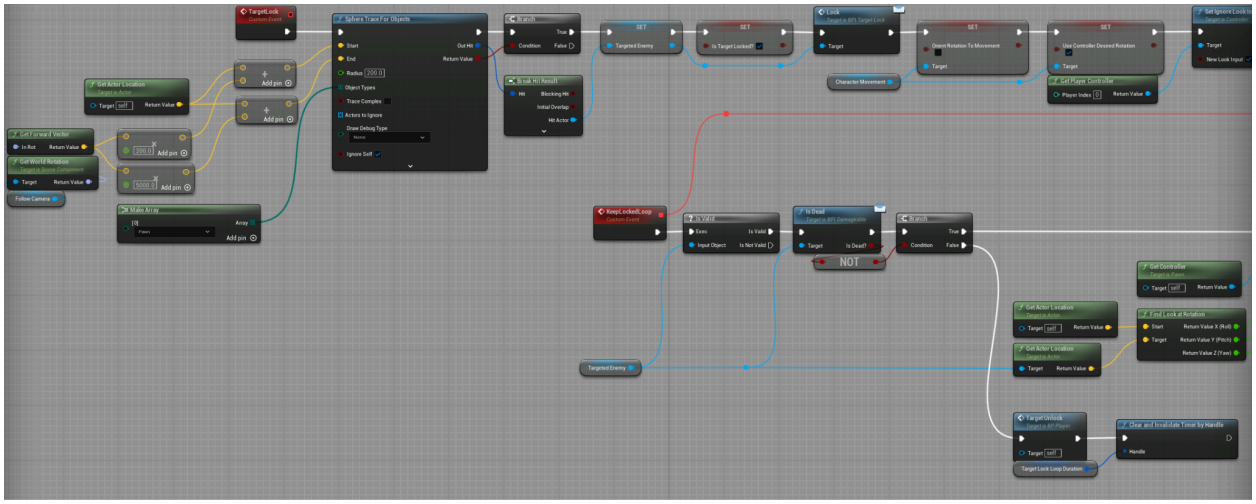

Figure 3.2: Target Lock. When called, starts a sphere trace to where the player is facing and checks for collision with enemy pawns. If collided, saves the hit actor as ‘TargetedEnemy’ and locks the camera to it. KeepLockedLoop is used to check if the target is still alive, if not, calls *TargetUnlock*.

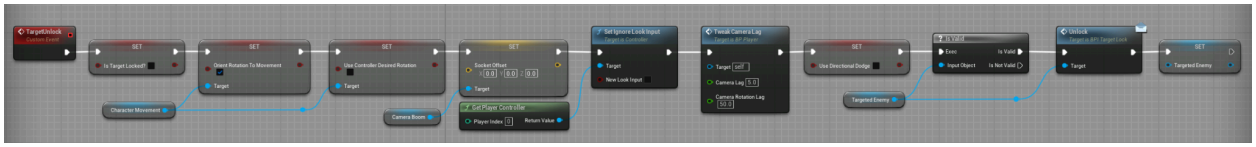

Figure 3.3: Target Unlock. When called, clears all logic set by *TargetLock*, and allows free camera movement.

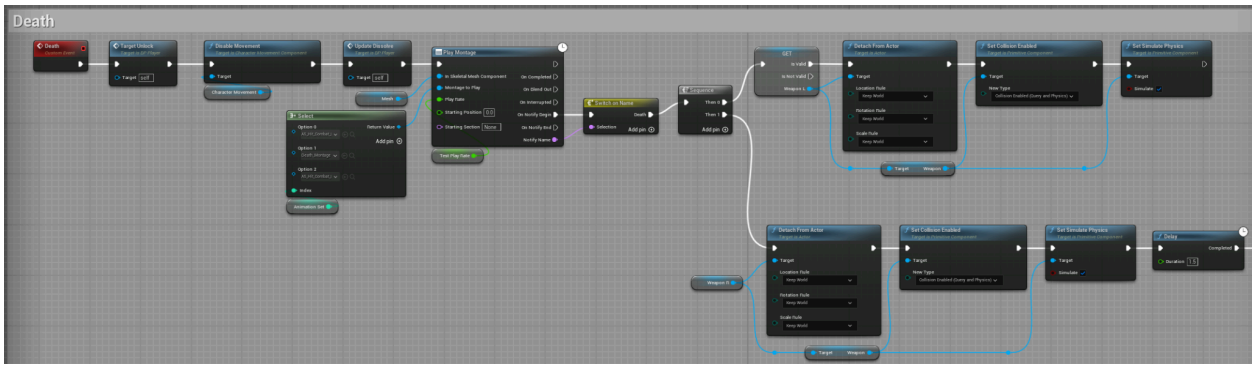

Figure 3.4: Death. When the player’s health falls below 0, *Death* will initiate. It clears all locked targets, disables movement, plays animation based on equipped weapon type, and detaches the weapon with physics enabled. Also, it plays death VFX and displays a death widget on the screen. Finally, it destroys the player character and creates a new instance of it at the last checkpoint.

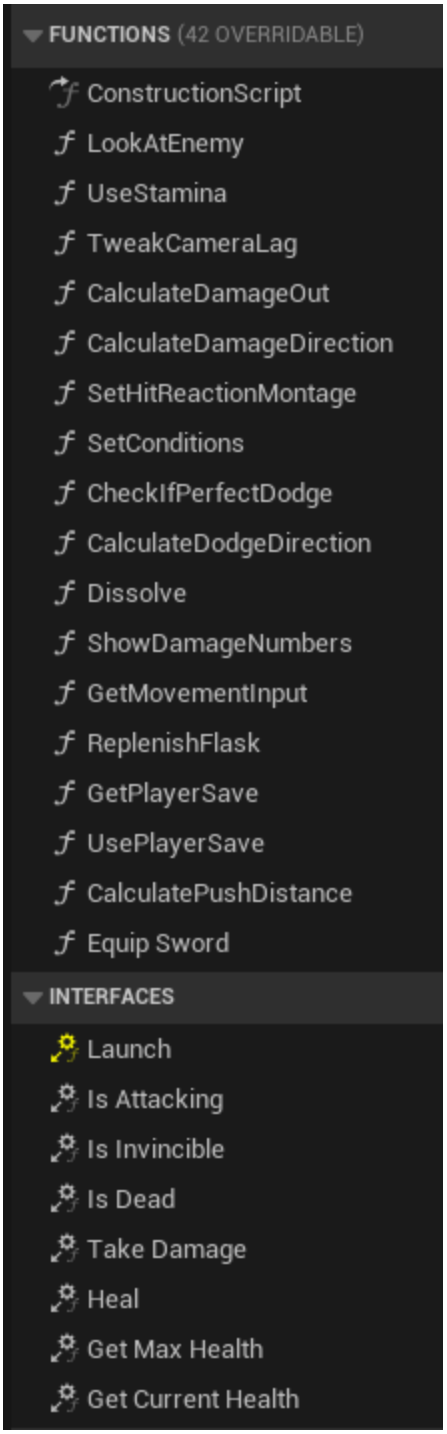

Figure 3.5: All the custom functions used for the player logic. (too many to give in-depth details of each)

Damage System

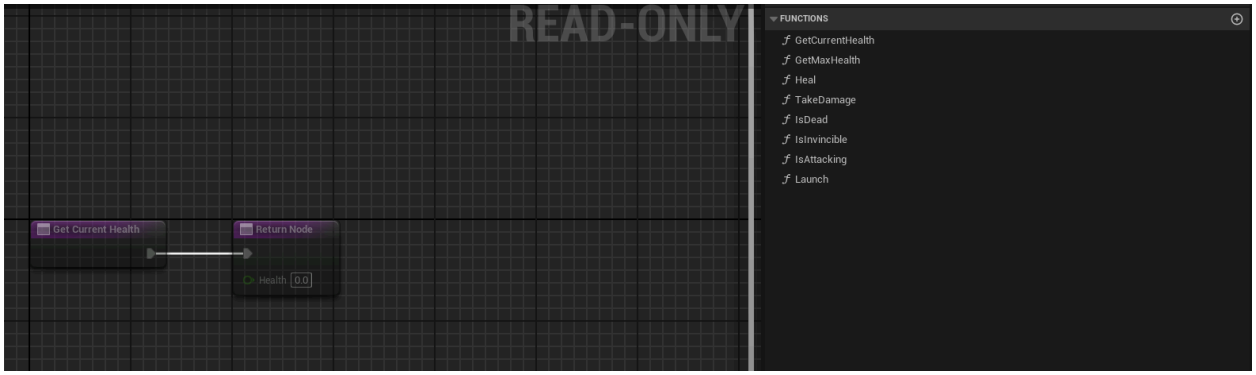

Figure 4.1: ‘Damageable’ interface used by all actors that use the damage system.

Figure 4.2: ‘DamageInfo’ struct used for applying damage.

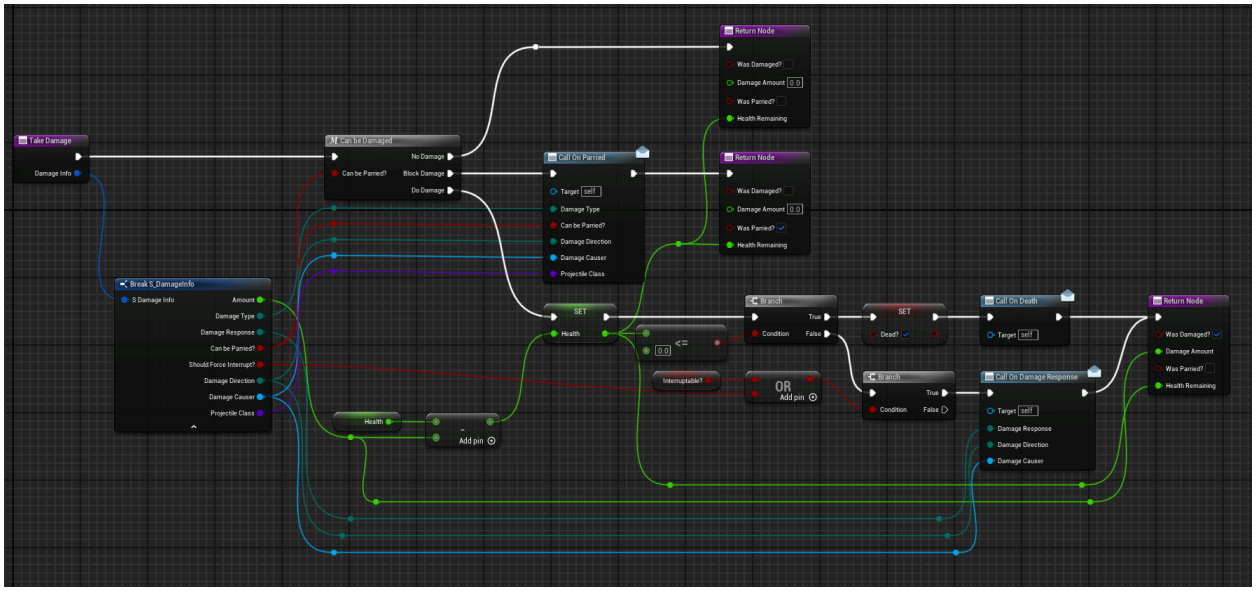

Figure 4.3: *TakeDamage* function shared across all actors that use the ‘Damageable’ interface. It’s responsible for checking for various conditions that occur when taking damage such as parrying, doing no damage, doing damage, and death.

Player Animation Logic

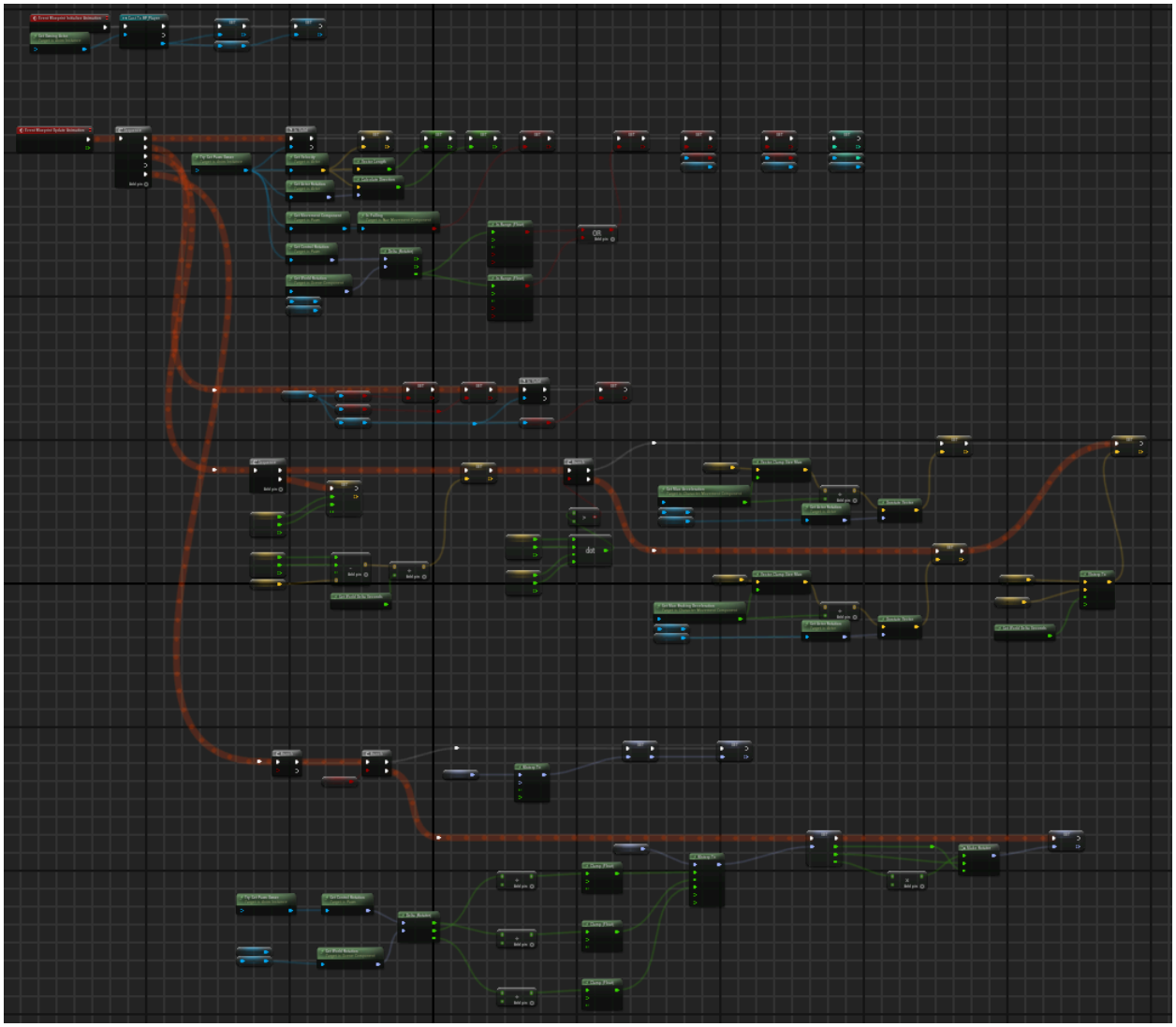

Figure 5: Overview of player animation blueprint for smooth transition between different player states.

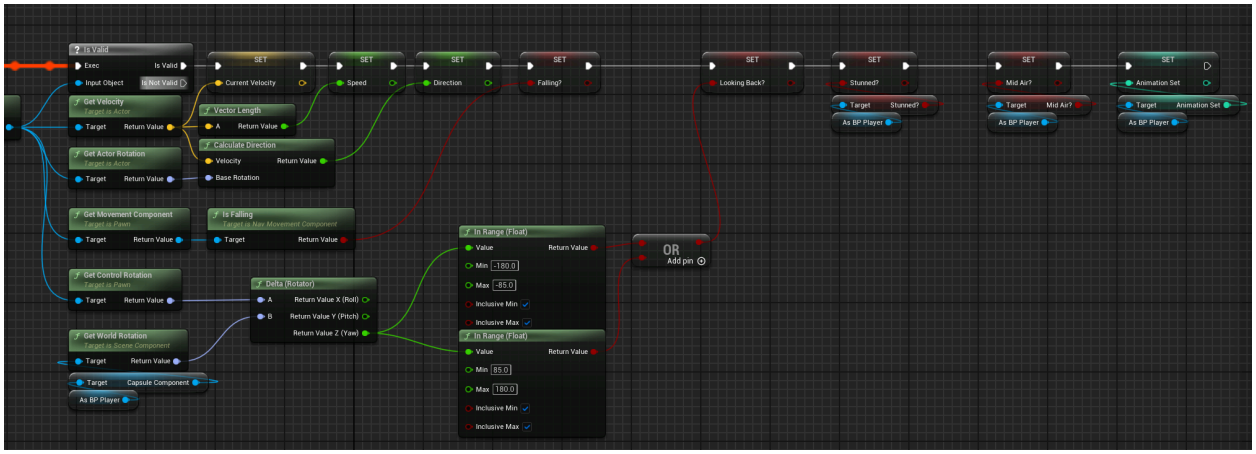

Figure 5.1: Setting various variables necessary for smooth transition between animations.

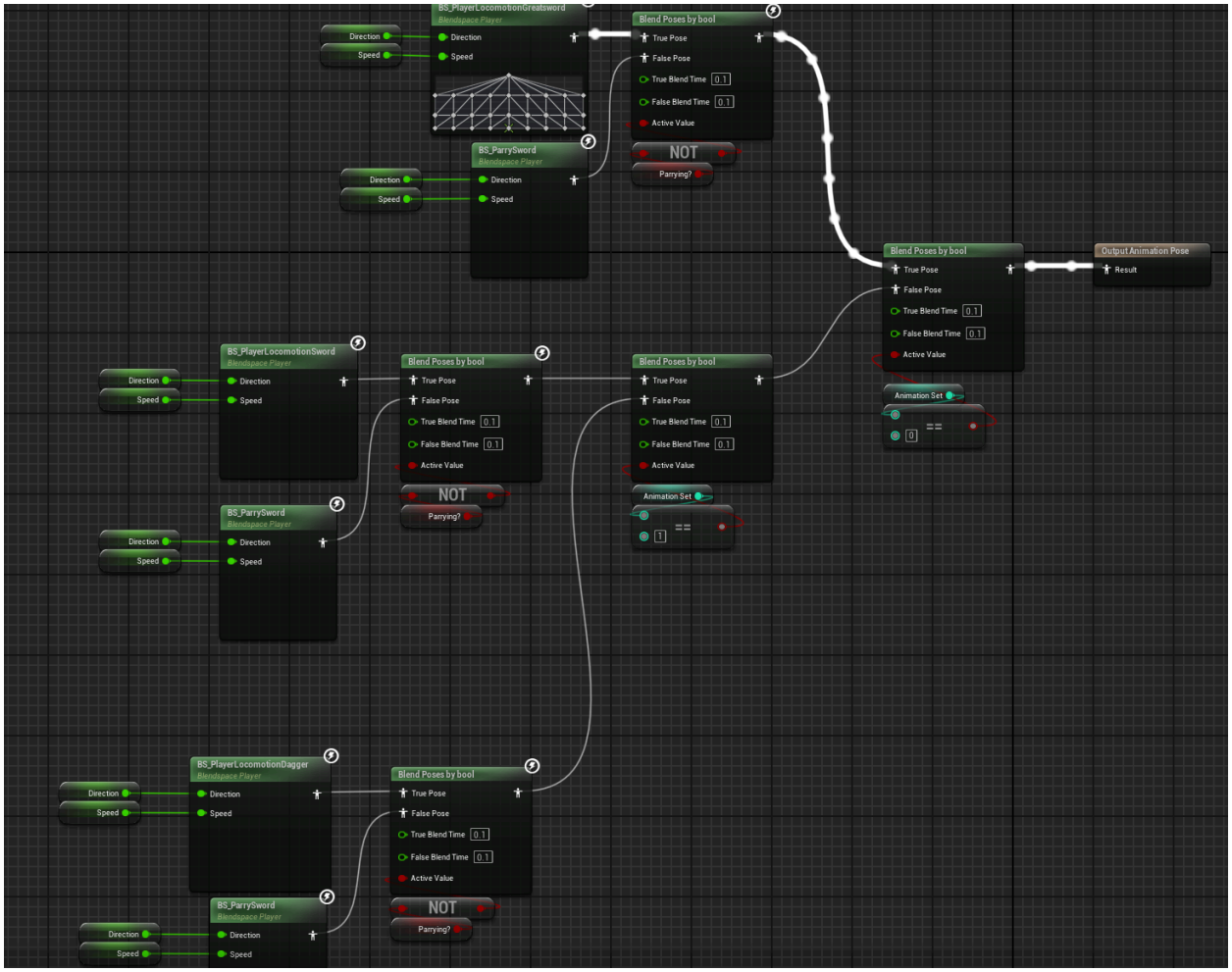

Figure 5.2: Logic that allows blending between different weapon stances.

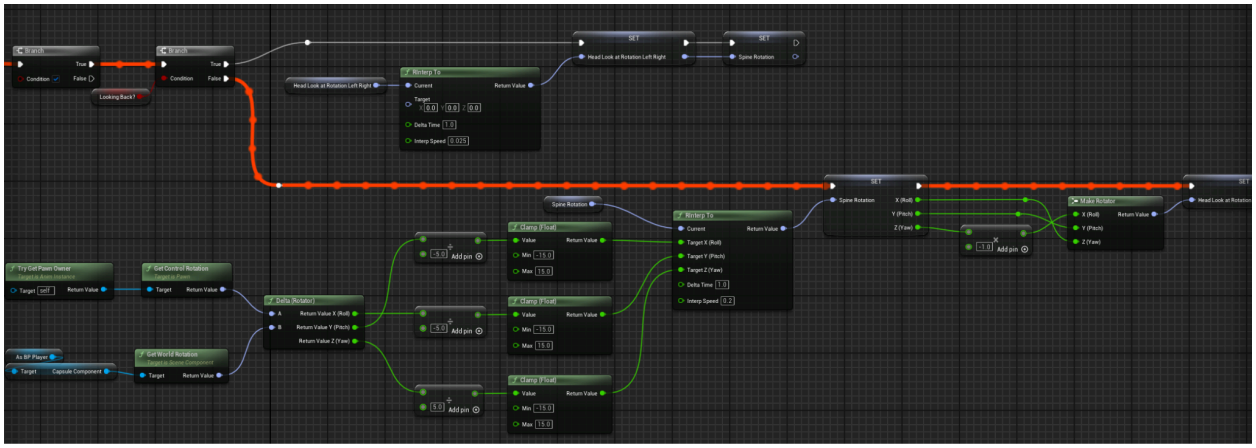

Figure 5.3: Logic used to rotate the player character's head in the direction of the camera.

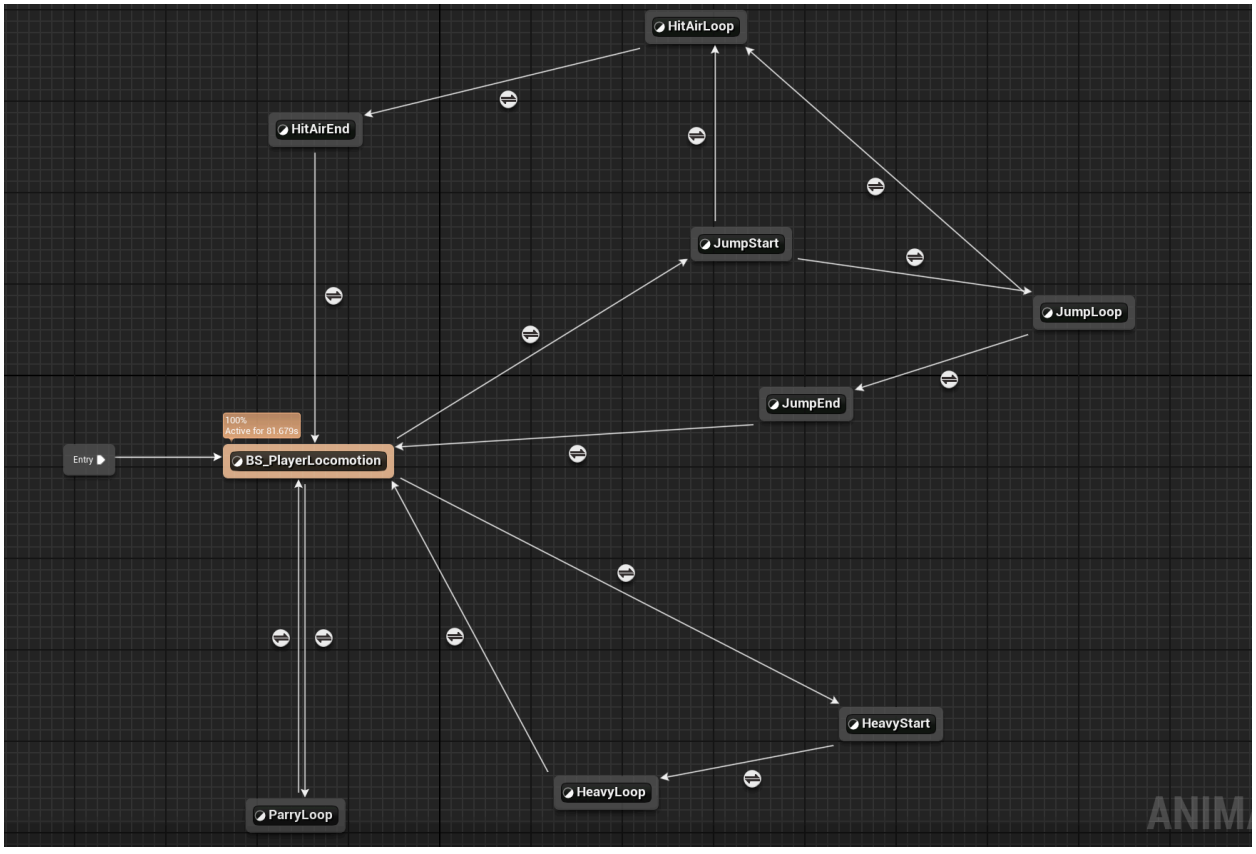

Figure 5.4: Main animation graph for the player allowing for smooth transition between various states such as jumping, charging a heavy attack, parrying, and getting hit mid-air.

Other Supporting Blueprints

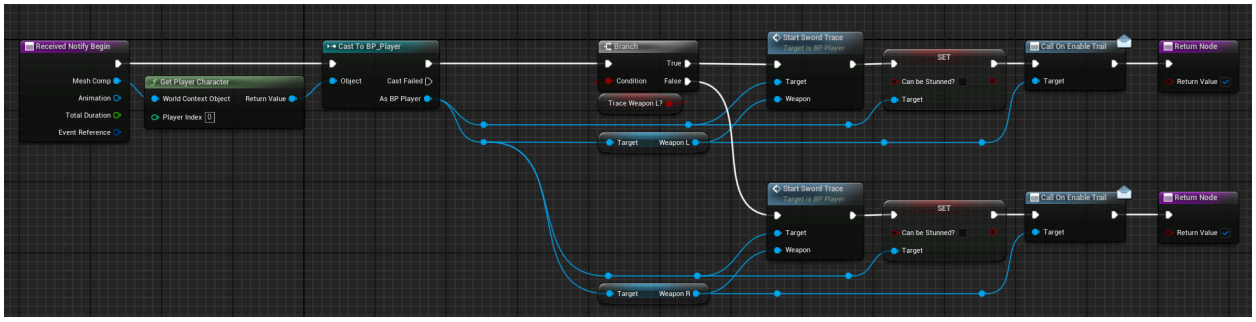

Figure 6.1: ‘SwordTraceNotify’ is a notify state used in player attack animations to indicate which part of the animation the sword trace logic should be used.

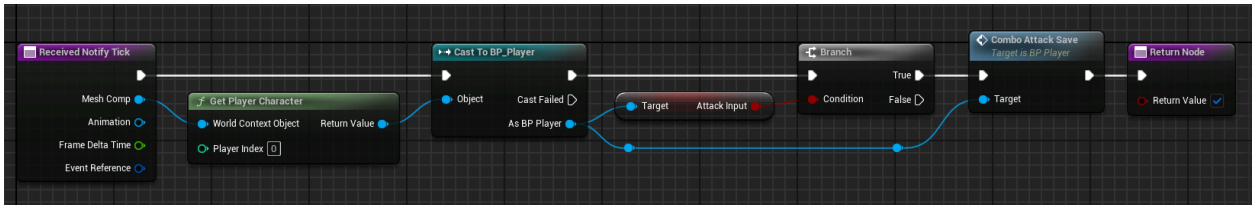

Figure 6.2: ‘SaveAttackNofiy’ is a notify state used in animations to indicate the window where the player can transition into the next combo sequence.

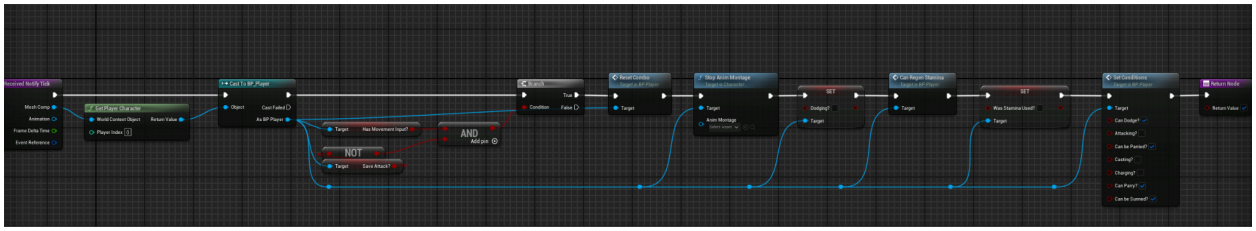

Figure 6.3: ‘CancelAnimNotify’ is a notify state used in animations to indicate the point which the player can cancel out of certain animations. (ie: as soon as the player is finished the attack they can cancel out of it and move without having to wait for the full animation to finish)