HLtC_CombatSystem Documentation

Note that:

- Only the content that is essential to the project will be documented here, as parts of the project are either easily discernible, irrelevant to the project (packed in with the template/Debug), or repetitive (will be compressed).
- This project was made using the Third Person template project provided by Unreal Engine 5.4.4

Scripts

HTtC_CombatSystemCharacter.cpp/h

This script was not made from scratch, it is a heavily altered version of the default character script that comes with the Third Person template project.

Inherits from ACharacter.

Variables (Default)

There is a list of UInputAction* variables for each input action, which are used for binding these actions to their associated functions when they are triggered. There is also a UInputMappingContext* variable for the input mapping context.

Name	Type	Default Value	Note
CameraBoom	USpringArmComponent*	null	Container for the spring arm component
FollowCamera	UCameraComponent*	null	Container for the camera component

Variables (Public)

Name	Type	Default Value	Note
PlayerControlState	FString	null	The players current control/movement state
PlayerAction	FString	null	The players current action
StaticAction	bool	false	If the player can currently move (static actions freeze player movement)
StaticActionDurationTimer	double	0	Countdown till a static action concludes
CameraState	FString	null	The cameras current control/movement state
ControlSchemeIndex	int	1	The currently selected control scheme as an index
MoveSpeed_Slow	float	200	Move speed of player during "Slow" control state
MoveSpeed_Action	float	400	Move speed of player during "Active" control state
isSprinting	bool	null	Signals if the user is holding the sprint input
SprintSpeedAddition	float	200	Added to player speed when sprinting
CurrentAttackType	FString	null	Type of attack currently being used (Light/Heavy)
AttackMechanicsTrigger	bool	false	Pulse to activate associated attack mechanics (currently hitscan) for a single instance
LightAttackIndex	int	null	Index of the light attack in the chain currently being used
HeavyAttackIndex	int	null	Index of the heavy attack in the chain currently being used
LightAttackChainLength	int	5	Length of the light attack chain
HeavyAttackChainLength	int	3	Length of the heavy attack chain

LightAttackTimings	array(5) double	1, 0.7, 0.7, 0.7, 0.7	Duration of each attack in the light attack chain
HeavyAttackTimings	array(3) double	1.5, 1, 1	Duration of each attack in the heavy attack chain
AdditionalAttackBuffer	bool	null	Signals if the next attack in the chain should trigger as soon as possible. Set to true if the user tries attacking too soon after a prior attack
AdditionalAttackBufferTiming	double	null	The duration of time needed to pass after an attack is triggered for a followup attack to be triggered. Based on the complete duration of the prior attack
AttackBufferTimingMulti	double	0.5	The multiplier that determines the initial length of AdditionalAttackBufferTiming
Blocking	bool	null	Signals if the user is holding the block input
ArmLengths_Slow	array(3) float	175, 225, 275	Array of boom arm lengths, defining said lengths for when the control state is "Slow", and PlayerAction is "Idle", "Moving", or "Sprinting" respectively
BoomSocketOffset_Slow	FVector	(0, 50, 75)	The cameras offset from the player on the end of the camera boom when the control state is "Slow"
ArmLengths_Action	array(2) float	300, 350	Array of boom arm lengths, defining said lengths for when the control state is "Action", and CameraState isn't "Free", and is "Free" respectively
BoomSocketOffset_Action	array(2) FVector	(0, 0, 50), (0, 0, 100)	The cameras offset from the player on the end of the camera boom when the control state is "Action"
DesiredArmLength	float	0	The target boom arm length at any instance, needed for when

			the actual boom arm length is between values
DesiredBoomSocketOffset	FVector	null	The target camera offset at any instance, needed for when the actual camera offset is between values
CamShakeRising	bool	true	Signals if the camera is currently positively rising during the camera shake
CamShakeTiming	double	0	The current timing of the camera shake used in the interpolation to determine the cameras current offset
CamShakeTimingConstraint	double	0.1	Constraints CamShakeTiming. If CamShakeTiming is greater than the constraint, of less than the constraints inverse, then flip CamShakeRising so that camera shake moves in the opposite direction
CamShakeDeltaTimeDivision	array(2) float	2, 1.2	An array containing the values that divide the amount added to CamShakeTiming every frame, to vary the timing of the camera shake. They are for "Moving" and "Sprinting" respectively

Functions (Protected)

Name	Type	Arguments	Note
SetControlStateDefaults	void	float DeltaTime	Sets the default values associated with each control, action and camera state every frame
SprintingFlag	void	const FInputAction Value& Value	Executes when sprint input action is triggered. Sets isSprinting
LightAttack	void	const FInputAction Value& Value	Executes when light attack input action is triggered. Determines what light attack should be used and when
HeavyAttack	void	const FInputAction Value& Value	Executes when heavy attack input action is triggered. Determines what heavy attack should be used and when
Block	void	const FInputAction Value& Value	Executes when sprint input action is triggered. Sets Blocking

SetControlStateDefaults

```
void AHLtC_CombatSystemCharacter::SetControlStateDefaults(float DeltaTime) // Sets the
default values associated with each control, action and camera state every frame
{
   if (!StaticAction) // If the player isn't performing a static action (e.g. an attack)...
   {
      if (GetCharacterMovement()->Velocity == FVector(0.0f, 0.0f, 0.0f)) // If the player
is standing still...
      {
         PlayerAction = "Idle";
      }

      else // If the player isn't standing still...
      {
         if (isSprinting) // If the player is sprinting...
         {
            PlayerAction = "Sprinting";
         }

         else // If the player isn't sprinting...
         {
            PlayerAction = "Moving";
         }
      }

      if (PlayerControlState == "Slow") // If the control state is "Slow"...
      {
         GetCharacterMovement()->MaxWalkSpeed = MoveSpeed_Slow; // Set the players move
speed to the associated variable
         DesiredBoomSocketOffset = BoomSocketOffset_Slow; // Set the target boom offset to
the associated variable
         CamShakeTiming = 0; // Reset CamShakeTiming

         if (PlayerAction == "Idle") { // If the player is standing still...
            DesiredArmLength = ArmLengths_Slow[0]; // Set the target boom offset to the
associated variable
         }

         else if (PlayerAction == "Moving") { // If the player is moving...
            DesiredArmLength = ArmLengths_Slow[1]; // Set the target boom offset to the
associated variable
         }

         else if (PlayerAction == "Sprinting") { // If the player is sprinting...
            DesiredArmLength = ArmLengths_Slow[2]; // Set the target boom offset to the
associated variable
         }
      }

      else if (PlayerControlState == "Action") // If the control state is "Action"...
      {
         GetCharacterMovement()->MaxWalkSpeed = MoveSpeed_Action; // Set the players move
speed to the associated variable

         if (CameraState == "Focus") // If the player is locked onto an enemy...
         {
            DesiredArmLength = ArmLengths_Action[1]; // Set the target boom length and
offset to the associated variables
            DesiredBoomSocketOffset = BoomSocketOffset_Action[1];
         }
```

```
         else // If the player isn't locked onto an enemy...
         {
            DesiredArmLength = ArmLengths_Action[0]; // Set the target boom length and
offset to the associated variables
            DesiredBoomSocketOffset = BoomSocketOffset_Action[0];
         }
      }

      if (isSprinting) { GetCharacterMovement()->MaxWalkSpeed += SprintSpeedAddition; } //
If the player is sprinting, add the additional speed mod to the move speed

      CameraBoom->TargetArmLength = FMath::Lerp(CameraBoom->TargetArmLength,
DesiredArmLength, DeltaTime * 2.5f); // Set the boom length to a value interpolated between
its current and desired length

      if (PlayerControlState == "Action" && CameraState == "Free") // If the control state
is "Action" and the camera is open to player input...
      {
         float newCamShakeTiming = 0; // Define new value for later addition
         if (PlayerAction == "Moving") { newCamShakeTiming = DeltaTime /
CamShakeDeltaTimeDivision[0]; } // If player is moving, set newCamShakeTiming to the
DeltaTime divided by the associated index in CamShakeDeltaTimeDivision
         if (PlayerAction == "Sprinting") { newCamShakeTiming = DeltaTime /
CamShakeDeltaTimeDivision[1]; } // If player is sprinting, set newCamShakeTiming to the
DeltaTime divided by the associated index in CamShakeDeltaTimeDivision

         if (!CamShakeRising) { newCamShakeTiming *= -1;} // If the camera should be
dropping in the shake, invert newCamShakeTiming

         CamShakeTiming += newCamShakeTiming;

         if (CamShakeTiming >= CamShakeTimingConstraint) { CamShakeTiming =
CamShakeTimingConstraint; CamShakeRising = false; } // If CamShakeTiming is greater or
equal to CamShakeTimingConstraint, set CamShakeTiming to CamShakeTimingConstraint and
signal that the camera should drop
         if (CamShakeTiming <= -CamShakeTimingConstraint) { CamShakeTiming =
-CamShakeTimingConstraint; CamShakeRising = true; } // If CamShakeTiming is less or equal
to negative CamShakeTimingConstraint, set CamShakeTiming to negative
CamShakeTimingConstraint and signal that the camera should rise

         FVector ShakenBoomSocketOffset = DesiredBoomSocketOffset;

         if (PlayerAction == "Moving" || PlayerAction == "Sprinting") {
ShakenBoomSocketOffset += FVector(0,0,1); } // If the player is moving or sprinting, offset
the vector

         if (ShakenBoomSocketOffset != DesiredBoomSocketOffset) // If the shaken offset is
different from the desired offset...
         {
            DesiredBoomSocketOffset = FMath::Lerp(ShakenBoomSocketOffset,
-ShakenBoomSocketOffset, CamShakeTiming); // Set the desired offset to an interpolated
vector between the shaken offset and the negative shaken offset, based on CamShakeTiming
         }

         else // If the player isn't moving or sprinting...
         {
            CamShakeTiming = 0;
         }
      }

      CameraBoom->SocketOffset = FMath::Lerp(CameraBoom->SocketOffset,
DesiredBoomSocketOffset, DeltaTime * 10.0f); // Set the camera offset to an interpolated
vector between its current and desired location
```

```
    }

    else // If the player is performing a static action (e.g. an attack)
    {
        StaticActionDurationTimer -= DeltaTime; // Countdown of the action duration

        if (StaticActionDurationTimer <= 0.0f) // If the timer is less than or equal to 0...
        {
            // Set variables ready for the player to move freely again
            StaticAction = false;
            StaticActionDurationTimer = 0.0f;
            LightAttackIndex = 0;
            HeavyAttackIndex = 0;

            AdditionalAttackBufferTiming = 0.0f;
            AdditionalAttackBuffer = false;
        }

        else if (StaticActionDurationTimer <= AdditionalAttackBufferTiming &&
AdditionalAttackBuffer) // If the remaining action duration is less than the current
AdditionalAttackBufferTiming, and an additional attack has been buffered...
        {
            if (CurrentAttackType == "Light") // If the prior attack was "Light"...
            {
                LightAttack(false); // Trigger an additional light attack
            }

            else if (CurrentAttackType == "Heavy") // If the prior attack was "Heavy"...
            {
                HeavyAttack(false); // Trigger an additional heavy attack
            }
        }
    }
}
```

SprintingFlag

```
void AHLtC_CombatSystemCharacter::SprintingFlag(const FInputActionValue& Value)
{
```

```
    isSprinting = Value.Get<bool>(); // Set the value to if the input it being pressed or
released
}
```

LightAttack

```
void AHLtC_CombatSystemCharacter::LightAttack(const FInputActionValue& Value)
{
    if (HeavyAttackIndex == 0 && LightAttackIndex < LightAttackChainLength) // If not using
the heavy attack and the current light attack is not the last in the chain...
```

```
    {
        if (StaticActionDurationTimer <= AdditionalAttackBufferTiming) // If the remaining
duration on the current attack is less or equal to the AdditionalAttackBufferTiming...
        {
            // Set variables to trigger an attack
            StaticAction = true;
            Blocking = false;
            CurrentAttackType = "Light";
            AttackMechanicsTrigger = true;

            PlayerAction = "LightAttack_" + FString::FromInt(LightAttackIndex);
            StaticActionDurationTimer = LightAttackTimings[LightAttackIndex]; // Set the
attack duration based on what attack it is
            AdditionalAttackBufferTiming = StaticActionDurationTimer *
AttackBufferTimingMulti; // Set the attack buffer based on StaticActionDurationTimer
            AdditionalAttackBuffer = false;
            LightAttackIndex++;    // Update the index so it reflects the current attack being
used
        }

        else // If the user attempts to attack too soon after a prior attack...
        {
            CurrentAttackType = "Light";
            AdditionalAttackBuffer = true; // Buffer an attack to use as soon as it can be
        }
    }
}
```

HeavyAttack

```
void AHLtC_CombatSystemCharacter::HeavyAttack(const FInputActionValue& Value)
{
    if (LightAttackIndex == 0 && HeavyAttackIndex < HeavyAttackChainLength) // If not using
the light attack and the current heavy attack is not the last in the chain...
    {
```

```
      if (StaticActionDurationTimer <= AdditionalAttackBufferTiming) // If the remaining
duration on the current attack is less or equal to the AdditionalAttackBufferTiming...
      {
         // Set variables to trigger an attack
         StaticAction = true;
         Blocking = false;
         CurrentAttackType = "Heavy";
         AttackMechanicsTrigger = true;

         PlayerAction = "HeavyAttack_" + FString::FromInt(HeavyAttackIndex);
         StaticActionDurationTimer = HeavyAttackTimings[HeavyAttackIndex]; // Set the
attack duration based on what attack it is
         AdditionalAttackBufferTiming = StaticActionDurationTimer *
AttackBufferTimingMulti; // Set the attack buffer based on StaticActionDurationTimer
         AdditionalAttackBuffer = false;
         HeavyAttackIndex++;    // Update the index so it reflects the current attack being
used
      }

      else // If the user attempts to attack too soon after a prior attack...
      {
         CurrentAttackType = "Heavy";
         AdditionalAttackBuffer = true; // Buffer an attack to use as soon as it can be
      }
   }
}
```

Block

```
void AHLtC_CombatSystemCharacter::Block(const FInputActionValue& Value)
{
   if (!StaticAction) // If the player isn't doing an action...
   {
      if (Value.Get<bool>()) // If the blocking input is being pressed or held...
      {
```

```
            Blocking = true;
        }

        else // If the blocking input is being released...
        {
            Blocking = false;
        }
    }
}
```

Blueprints

BP_ThirdPersonCharacter

Variables

Name	Type	Default Value	Note

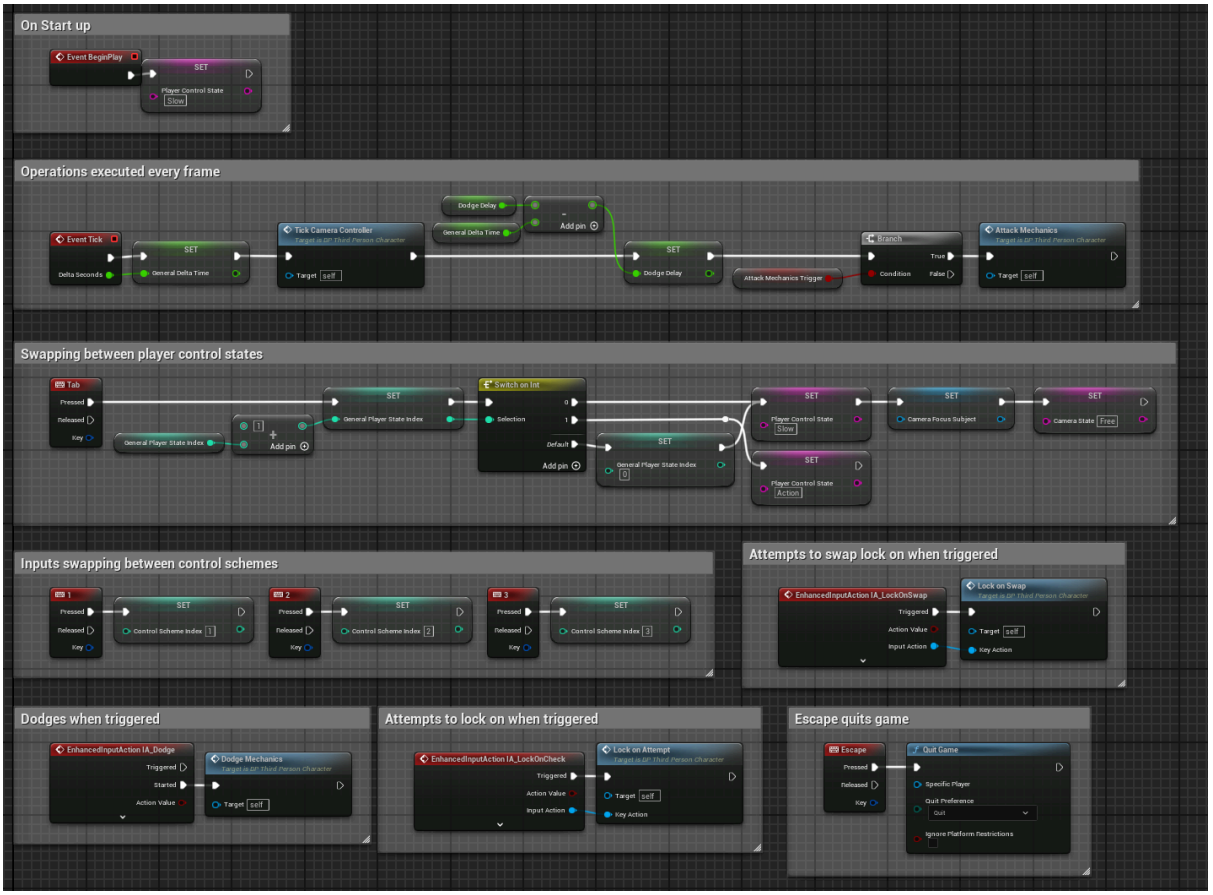
General_DeltaTime	float	null	Delta time every frame
Camera_FocusSubject	Actor	null	The actor being locked on to
Camera_FocusLerpTime	float	null	The lerp value used for camera focusing
Camera_FocusLerpRate	float	3	The multiplier applied to the lerp value for focusing
Camera_RepositionLerpTime	float	0	The lerp value used for camera repositioning
Camera_RepositionLerpRate	float	0.75	The multiplier applied to the lerp value for repositioning
Camera_RepositionRotation	Rotator	null	The rotation to reposition to
General_PlayerStateIndex	int	null	The current state index
Sound_HitDetectSFX	array(5) Sound Wave	Hit_1, hit_2, hit_3, hit_4, hit_5	Array of the sound effects that can be used when you hit an enemy
Dodge_Delay	float	null	The remaining duration of the dodge delay

Functions

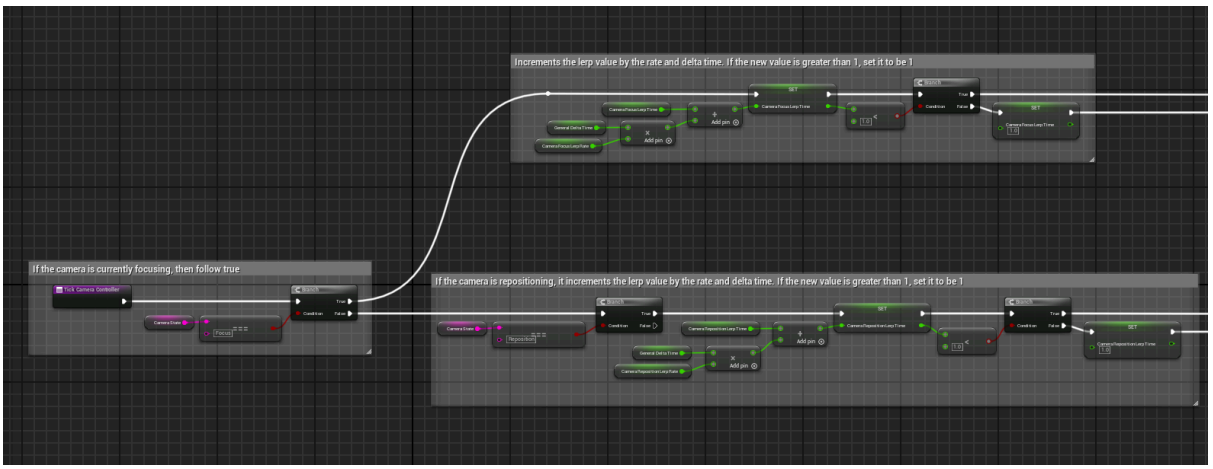
Name	Input	Output	Note
EventGraph	null	null	Root
TickCameraController	null	null	Updates camera position and rotation every frame
LockOnAttempt	KeyAction	null	Checks to see if there

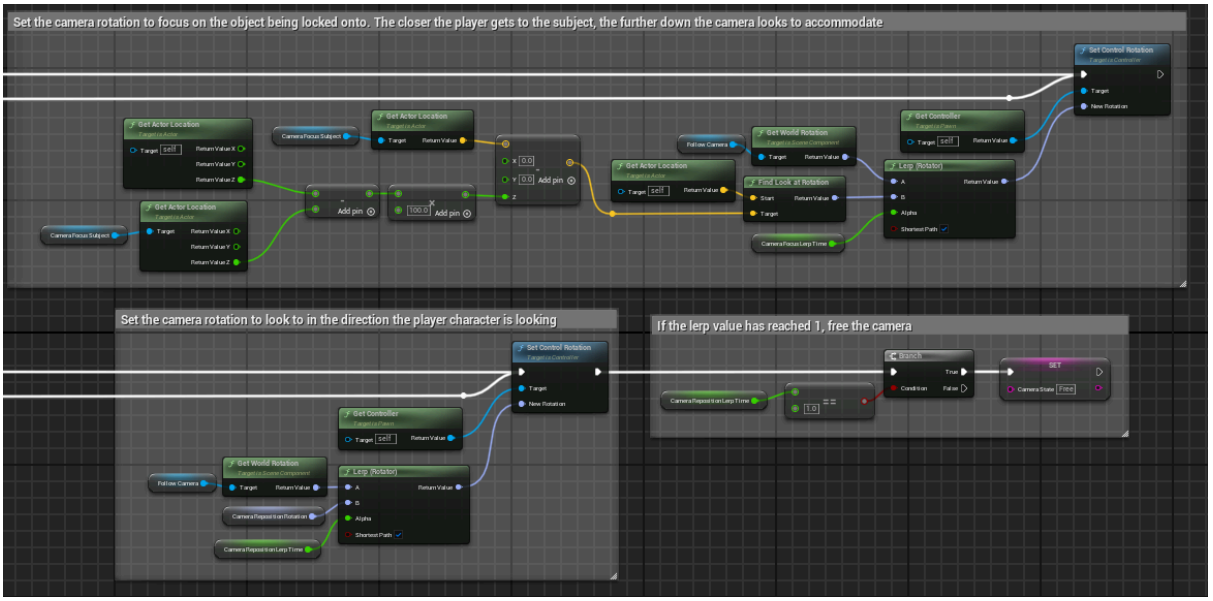
			is a pawn to lock on to
LockOnSwap	KeyAction	null	Checks to see if there is a pawn to swap lock on to
GetInputKey	KeyAction	ValidKeyInput, KeyNameString	Outputs the name of the input triggered as a string
AttackMechanics	null	null	Checks to see if there is a pawn to attack
DodgeMechanics	null	null	Adds force to player to dodge

EventGraph

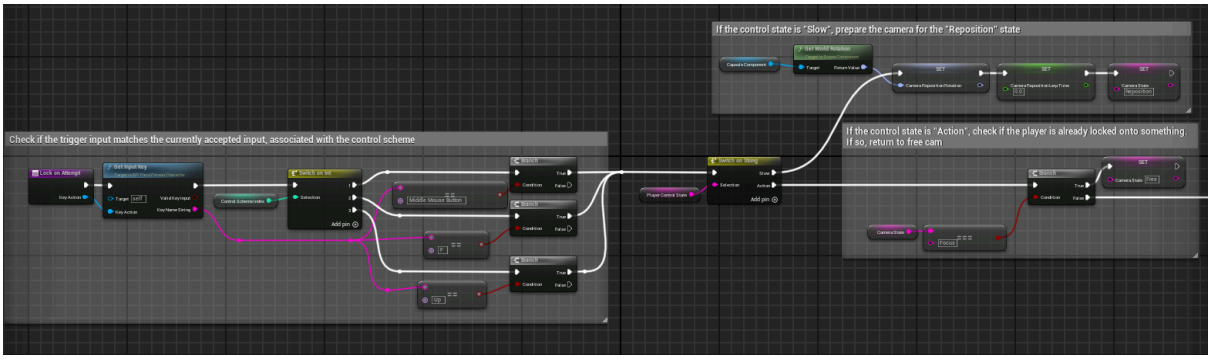


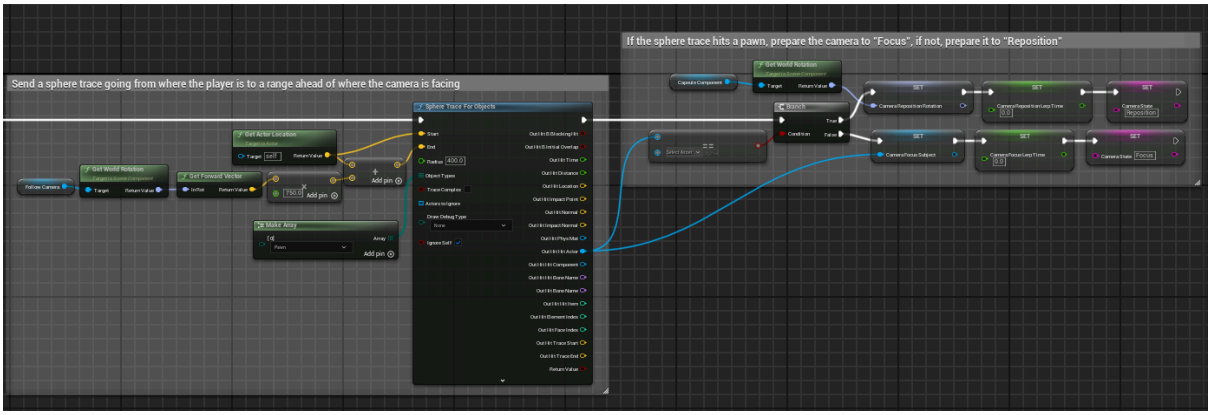
TickCameraController



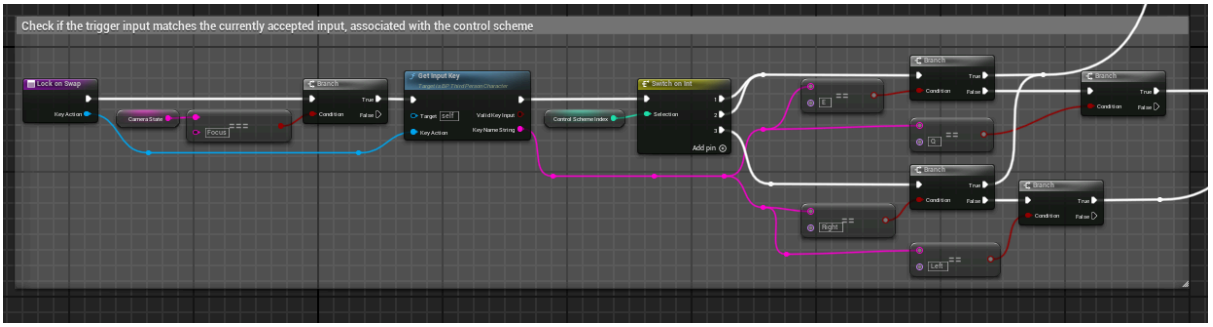


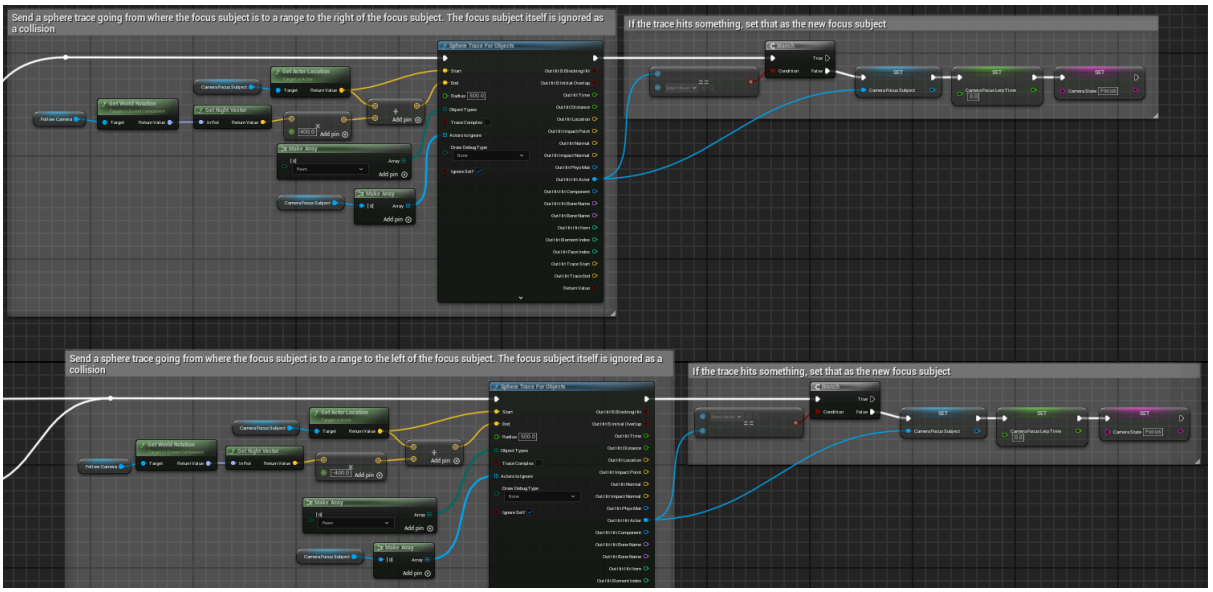
LockOnAttempt


Check if the trigger input matches the currently accepted input, associated with the control scheme
If the control state is "Slow", prepare the camera for the "Reposition" state
If the control state is "Action", check if the player is already locked onto something. If so, return to free cam

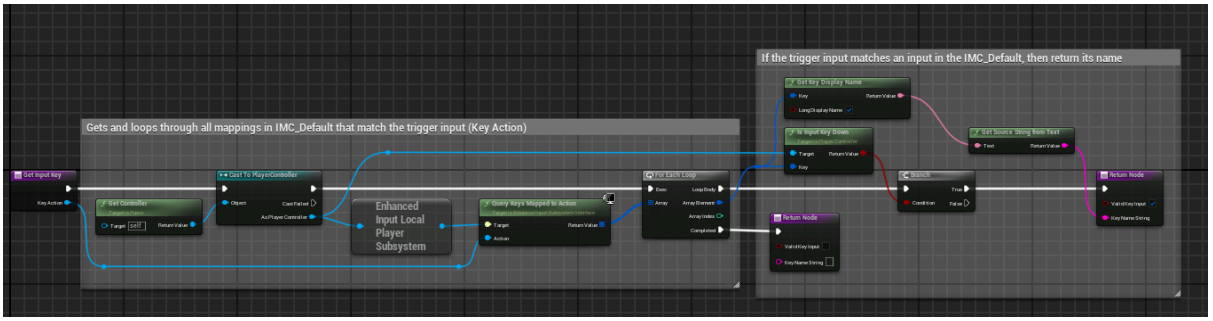

Send a sphere trace going from where the player is to a range ahead of where the camera is facing
If the sphere trace hits a pawn, prepare the camera to "Focus", if not, prepare it to "Reposition"

[LockOnSwap]

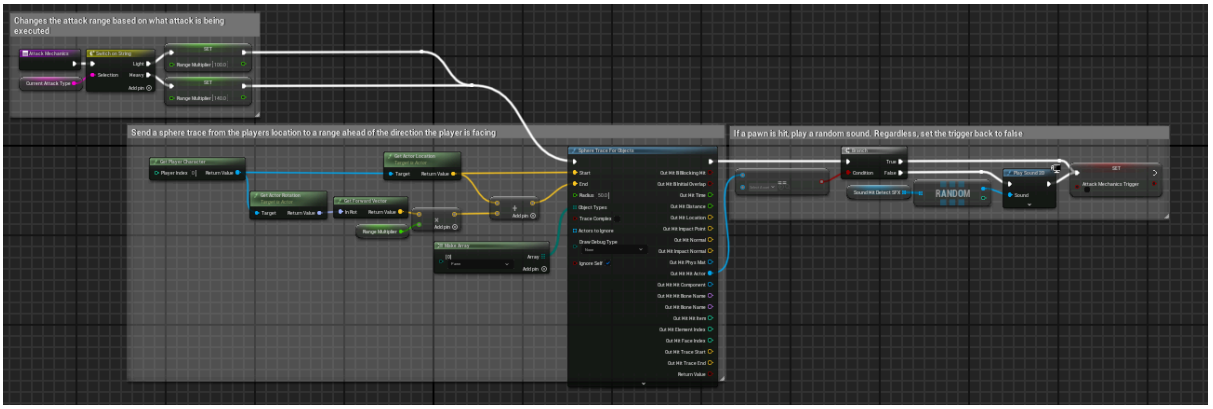




GetInputKey



AttackMechanics



DodgeMechanics

Additional Content

Blueprints

The main additional blueprint is a UI widget that displays debug and control information, the data provided by which should allow for the user to try and test different mechanics and control schemes within the project.

The other blueprint created in the project is **BP_Enemy**. This blueprint contains nothing of note, and exists in the level to demonstrate the lock on function and mechanics.

Input Data Assets

The default mapping contexts, or **IMC_Default** has been heavily altered, the changes include:

- Removal of **IA_Jump** mapping, as jumping was removed from the project.
- Addition of several new mappings, including:
    - **IA_SprintToggle**
    - **IA_LockOnCheck**
    - **IA_LockOnSwap**
    - **IA_Block**
    - **IA_LightAttack**
    - **IA_HeavyAttack**
    - **IA_Dodge**
- Addition of associated input keys and triggers to new mappings.

The new mappings required associated input action data assets, all of which were created and stored with the pre existing input assets.

Audio Assets

For use with the attack mechanics, audio assets were created. There are 5 of them, named **hit1~5**, and they all use default values.