Catalogue

1. Cardinal Direction (Procedural 8 Direction)

Enumerations used in cardinal direction, will be demonstrated in V2

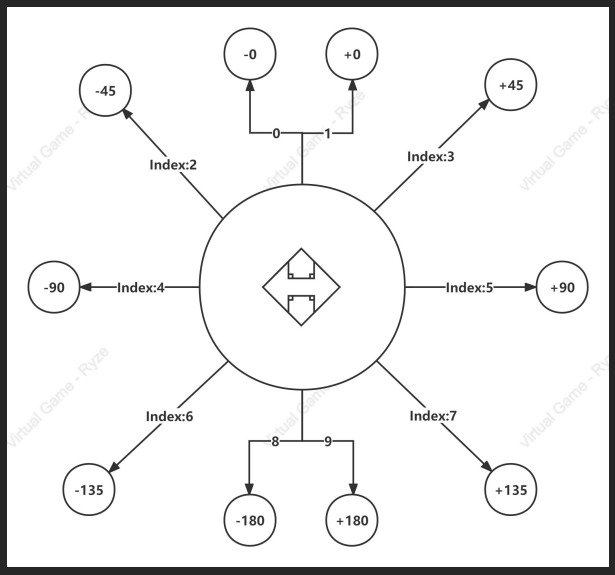
```
/** Types of virtual motion cardinal direction */
UENUM(BlueprintType)
namespace EMotionCardinalDirection
{
    enum Type
    {
        Forward,
        Backward,
        Left,
        Right,
        NONE,
    };
}
```

2. Angle Direction

(1) Enumeration used by angle direction

```
/** Types of virtual motion angle direction */
UENUM(BlueprintType)
namespace EMotionAngleDirection
{
    enum Type
    {
        ForwardLF,
        ForwardRF,
        ForwardLeft,
        ForwardRight,
        Left,
        Right,
        BackwardLeft,
        BackwardRight,
        BackwardLF,
        BackwardRF,
        NONE,
    };
}
```

(2) Concept map of Angle direction



(3) There are left and right foot(LF/RF) in both front and back directions, that is, 0 to -180 counterclockwise rotation direction and 0 to +180 clockwise rotation direction.

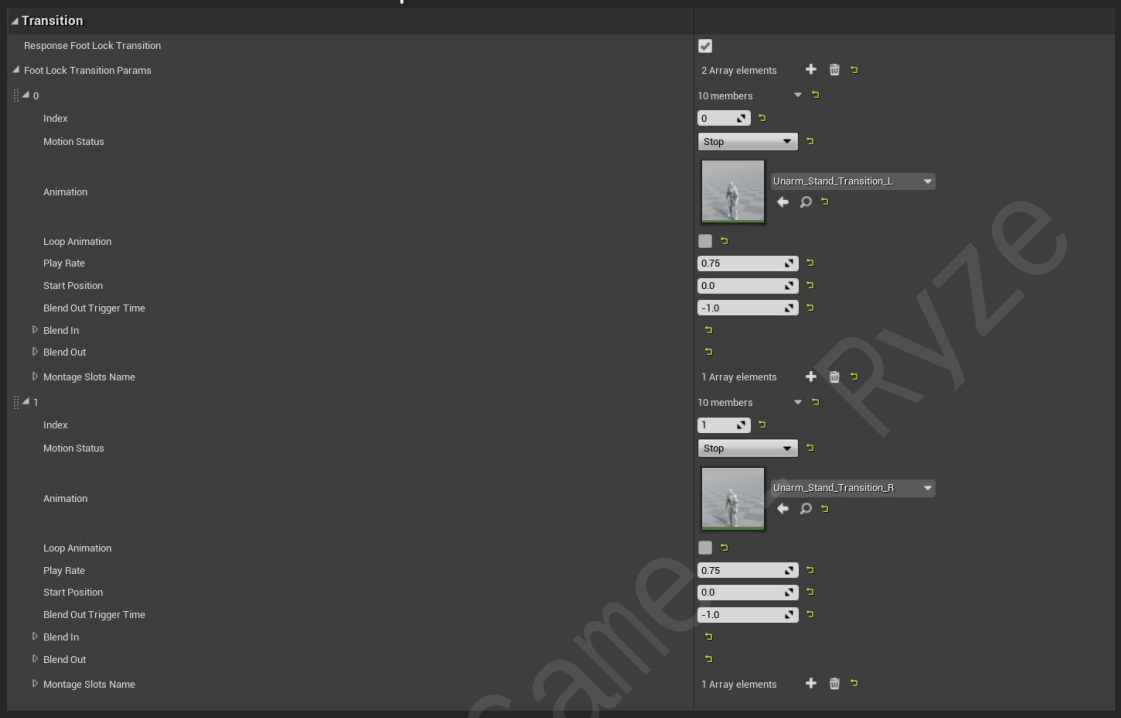
3. Locomotion Status

Enumeration used in animation state

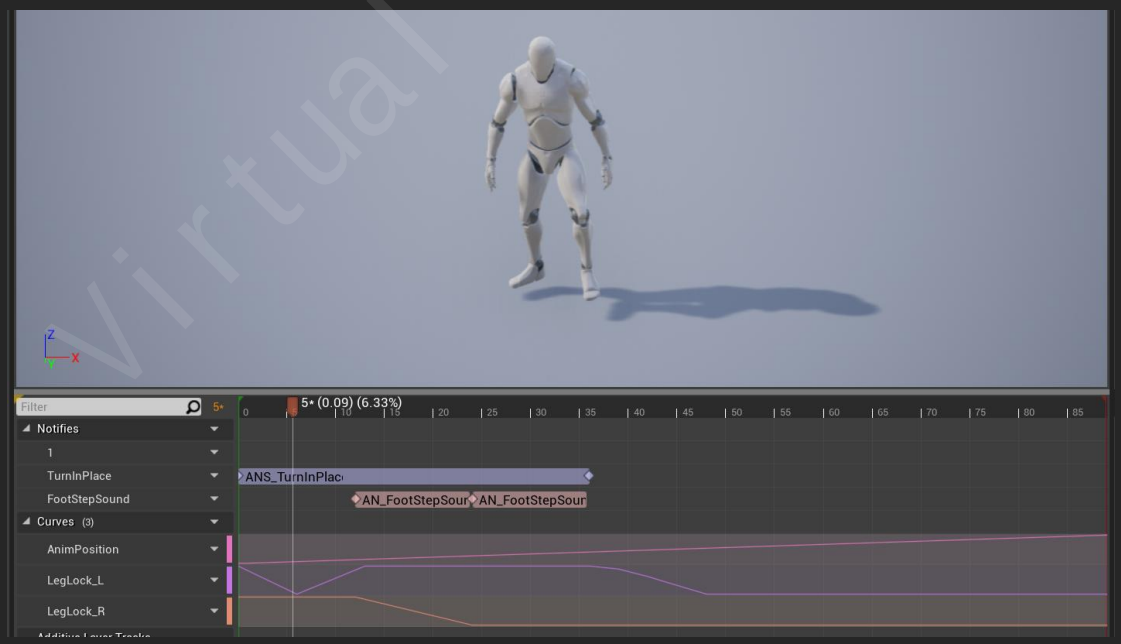
```
/** Types of virtual motion matching status */
UENUM(BlueprintType)
namespace EVirtualMotionStatus
{
    enum Type
    {
        Idle,
        Transition,
        TurnInPlace,
        Start,
        Moving,
        Pivot,
        Stop,
        JumpStart,
        JumpFalling,
        JumpLand,
        MAX UMETA(Hidden)
    };
}
```

4. Program Pose Transition

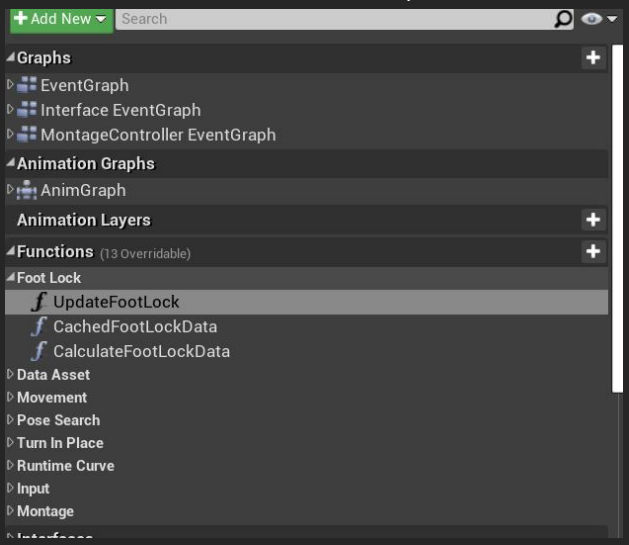
(1) Array index sequence corresponds to left-foot start[0] and right-foot start[1]
(2) This function is mainly used as short stop animation, and as the program transition of blend between poses

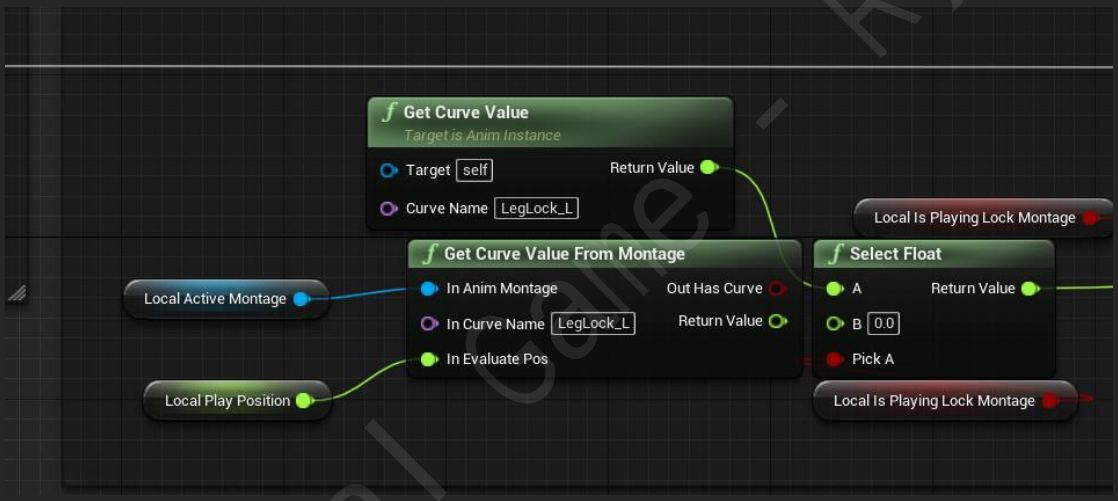


(3) You need to configure the corresponding Leg Lock_L(R) locking curve

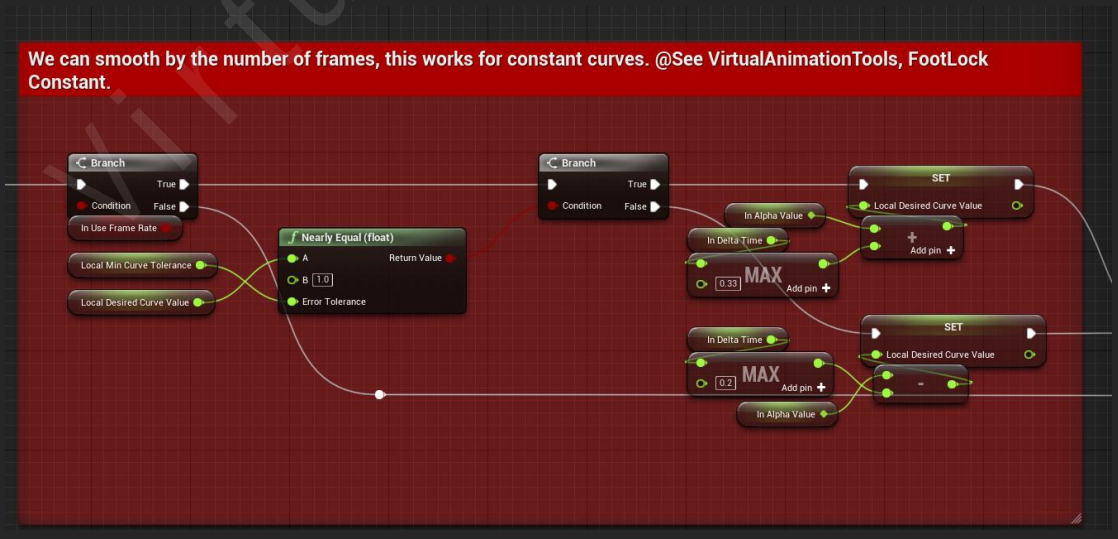


(4) We can choose the mode of acquiring curve data (static asset curve data, Runtime blended curve data)

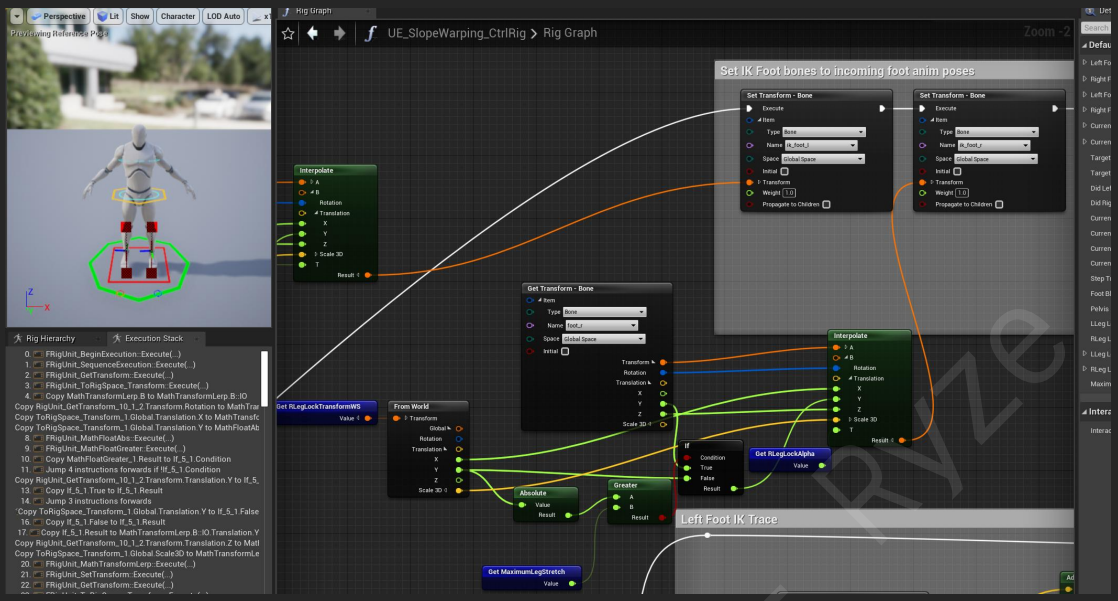




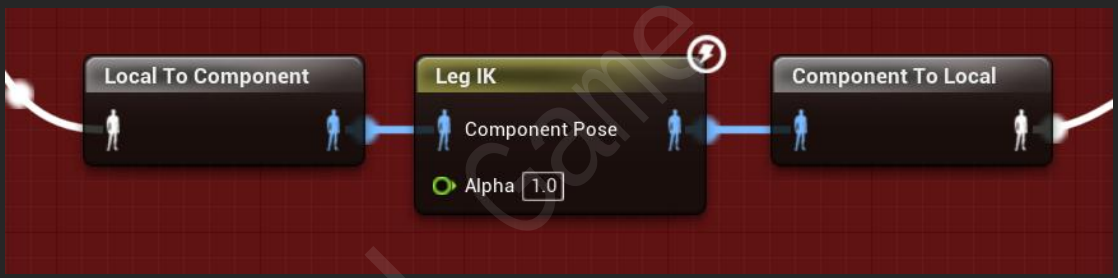
(5) Frame transition logic (to avoid the effect of pose change due to sudden drop/rise in value)



(6) Control Rig use cached component space transform data apply to foot bone data (a new leg IK animation node will be released later including lock feet, distance matching, etc.)

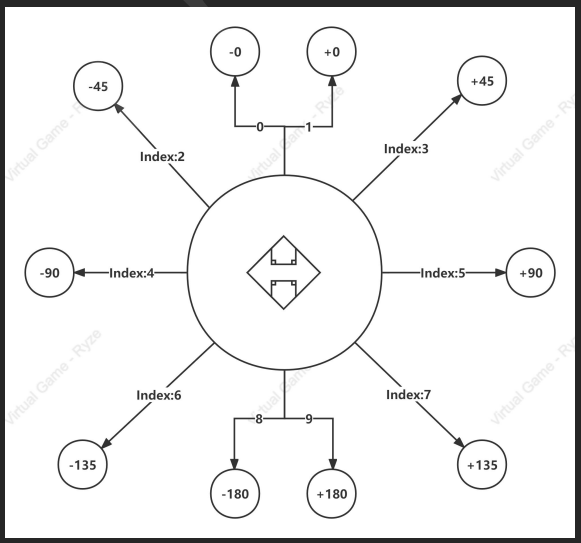


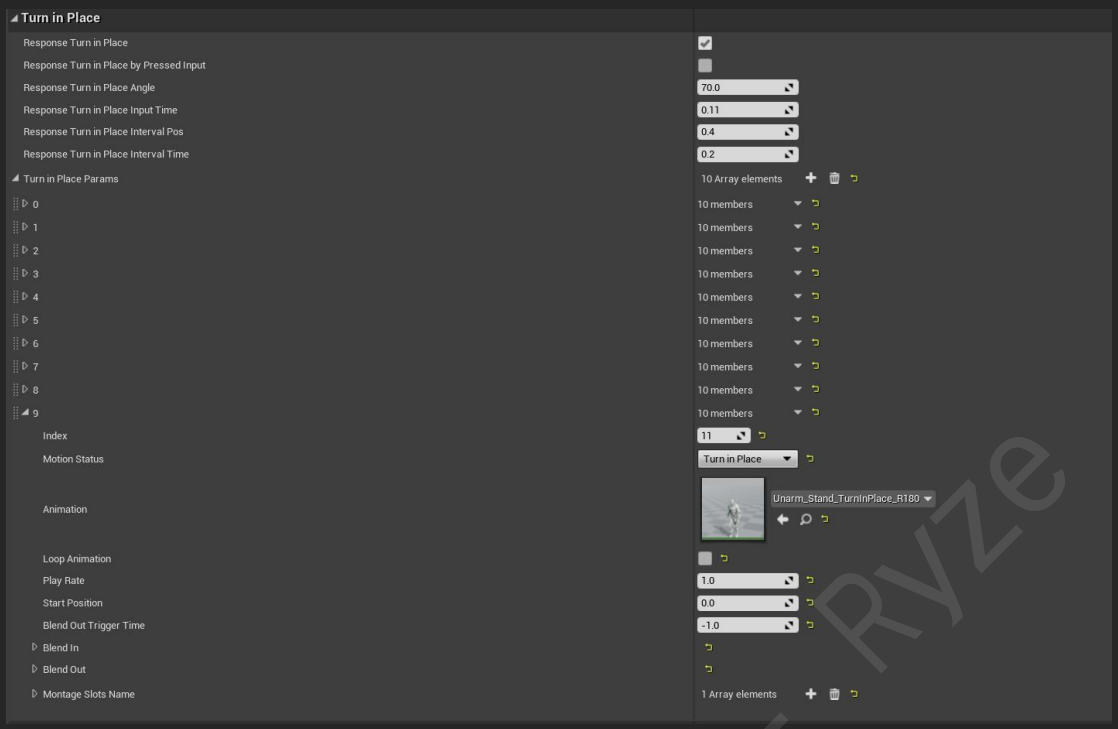
(7) Apply with leg IK animation nodes



5. Turn In Place

(1) The array index corresponds to Angle direction





(2) Parameter

bResponseTurnInPlace: whether to play turn in place animation

bResponseTurnInPlaceByPressedInput: to respond to the input mode of turn, press Input key to respond immediately, or release the Input key to respond

ResponseTurnInPlaceAngle: first person mode responds to turn in place angle range (this should be changed to left and right range)

ResponseTurnInPlaceInputTime: first person mode responds to the wait time for turn in place

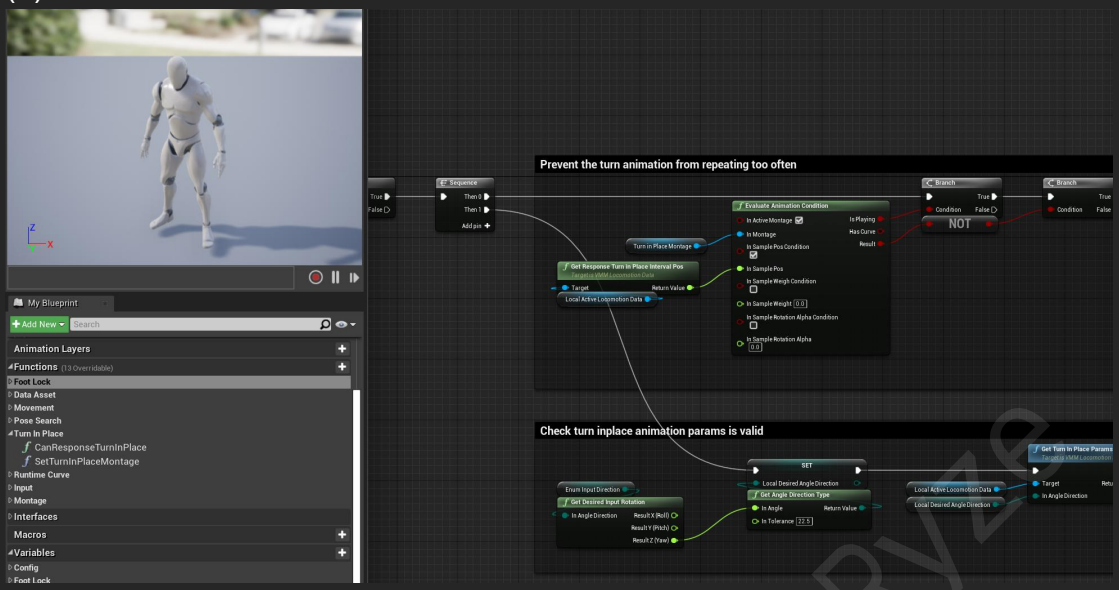
ResponseTurnInPlaceIntervalPos: if a turn animation is playing, the turn can only be updated if the play time is greater than this value

ResponseTurnInPlaceIntervalTime: if a turn in place animation is playing, the turn can only be updated if the interval between last plays is greater than this value

```
    /** Defines whether to perform a turn animation in response */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TurnInPlace)
    bool bResponseTurnInPlace;

    /** If true, it will execute turn when the button is pressed, otherwise it will execute the released input */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TurnInPlace)
    bool bResponseTurnInPlaceByPressedInput;

    /** Minimum angle for responsive turn animation */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TurnInPlace)
    float ResponseTurnInPlaceAngle;

    /** If the time interval between start input and stop input is less than this value, we will play the turn animation */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TurnInPlace)
    float ResponseTurnInPlaceInputTime;

    /** If the turn animation is being played, check whether the current animation playback position is greater than this value, otherwise the ne
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TurnInPlace)
    float ResponseTurnInPlaceIntervalPos;

    /** Minimum interval time for responding to turn animations, Only used in tick events, in order to get higher weight animation wait time */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TurnInPlace)
    float ResponseTurnInPlaceIntervalTime;
```

```
    /** Return turn in place params from desired angle direction. */
    UFUNCTION(BlueprintCallable, Category = TurnInPlace, meta = (DisplayName = "Get Turn In Place Params"))
    FMotionAnimParamsData K2_GetTurnInPlaceParams(const EMotionAngleDirection::Type& InAngleDirection);
    const FMotionAnimParamsData* GetTurnInPlaceParams(const EMotionAngleDirection::Type& InAngleDirection);

public:

    /** Return the response turn in place animation condition */
    UFUNCTION(BlueprintPure, Category = TurnInPlace)
    FORCEINLINE bool CanResponseTurnInPlace() const { return bResponseTurnInPlace; }

    /** If true, it will execute turn when the button is pressed, otherwise it will execute the released input */
    UFUNCTION(BlueprintPure, Category = TurnInPlace)
    FORCEINLINE bool GetResponseTurnInPlacePressedState() const { return bResponseTurnInPlaceByPressedInput; }

    /** Return the response turn in place animation minimum angle */
    UFUNCTION(BlueprintPure, Category = TurnInPlace)
    FORCEINLINE float GetResponseTurnInPlaceAngle() const { return ResponseTurnInPlaceAngle; }

    /** Return the response turn in place animation input time */
    UFUNCTION(BlueprintPure, Category = TurnInPlace)
    FORCEINLINE float GetResponseTurnInPlaceInputTime() const { return ResponseTurnInPlaceInputTime; }

    /** Return the response turn in place animation interval position */
    UFUNCTION(BlueprintPure, Category = TurnInPlace)
    FORCEINLINE float GetResponseTurnInPlaceIntervalPos() const { return ResponseTurnInPlaceIntervalPos; }

    /** Return the response turn in place animation minimum interval time */
    UFUNCTION(BlueprintPure, Category = TurnInPlace)
    FORCEINLINE float GetResponseTurnInPlaceIntervalTime() const { return ResponseTurnInPlaceIntervalTime; }
```

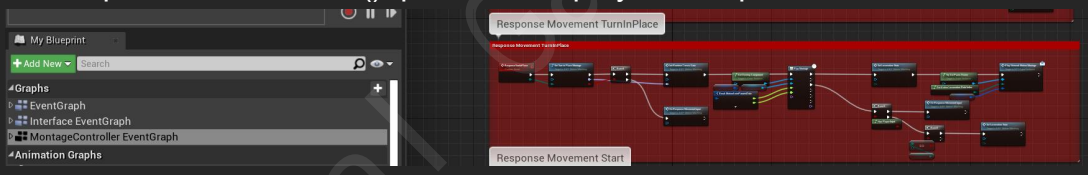
(3) Functions



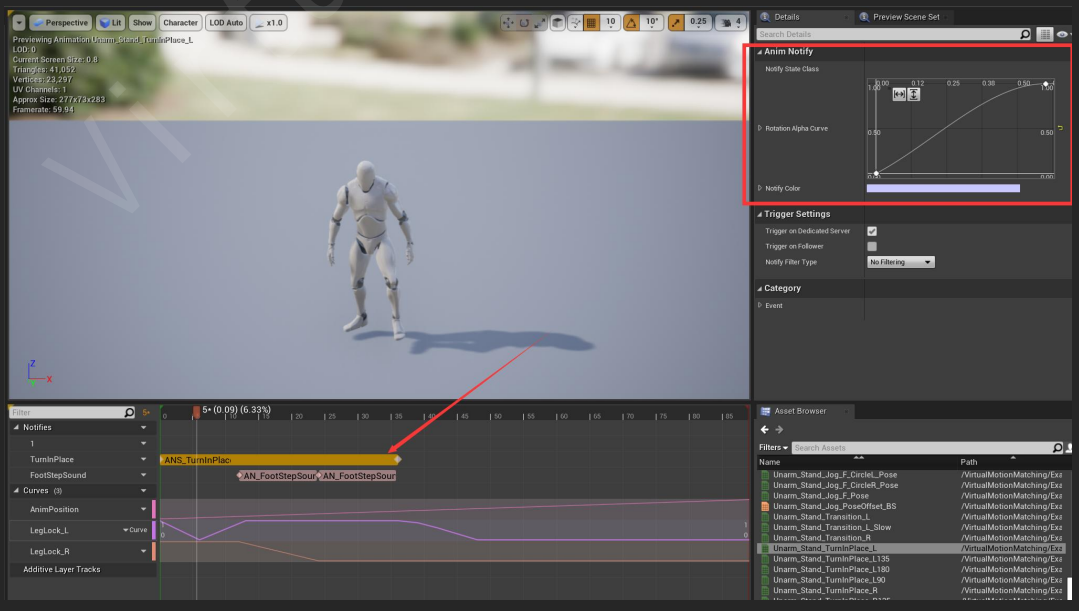
a. CanResponseTurnInPlace(): determine whether a turn in place can be responded to and whether the currently expected turn animation data is valid

b. SetTurnInPlaceMontage(): cache the current turn in place animations that should be played

c. ResponseTurnInPlace(): perform the play turn in place animation



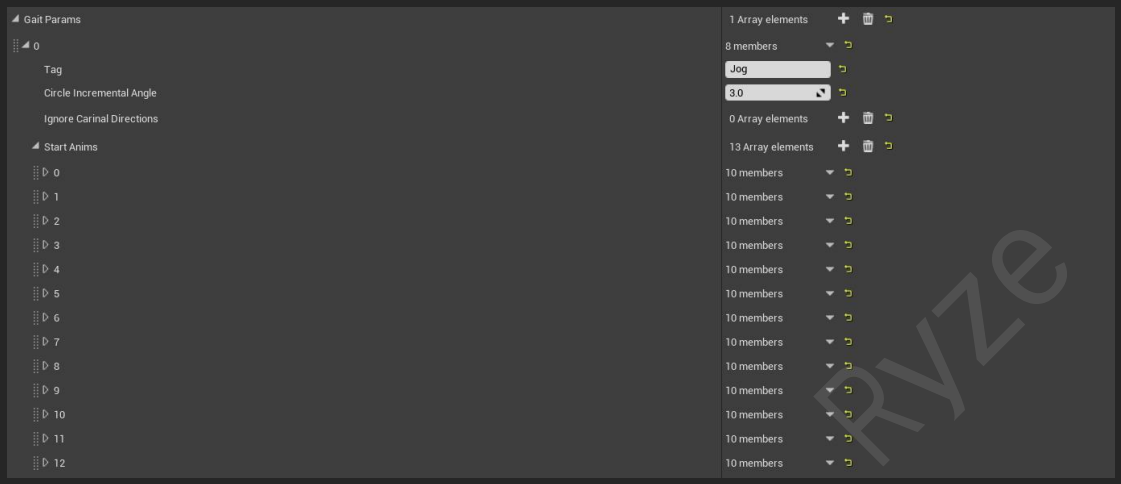
(4) Program animation curve rotation (simply apply 0-1 Alpha rotation curve)



6. Movement Start

(1) The array index corresponds to **Angle Direction + Cardinal Direction (Backwrd/Left/Right)**

0 - 9Angle Direction 10:Backward Start 11:Left Start 12:Right Start



(2) Parameter

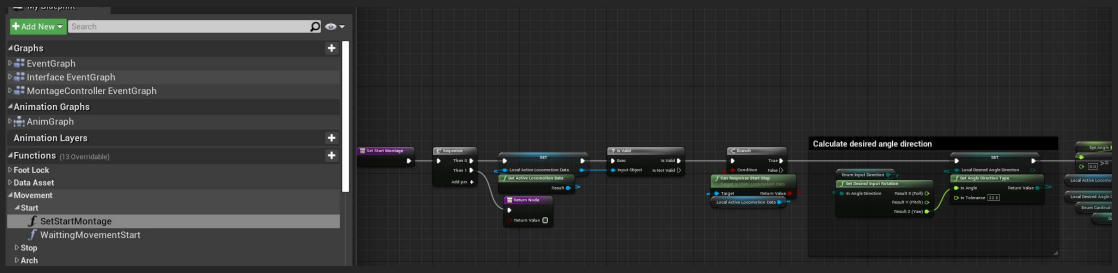
bResponseStartStep: whether to play the movement start animation

```
#pragma region Start

protected:

    /** Defines whether to perform a start animation in response */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Start)
    bool bResponseStartStep;

public:

    /** Return start animation params from desired data. */
    UFUNCTION(BlueprintCallable, Category = Start, meta = (DisplayName = "Get Start Params"))
    FMotionAnimParamsData K2_GetStartParams(bool InRightFoot, EMotionAngleDirection::Type InAngleDirection, EMoti
    const FMotionAnimParamsData* GetStartParams(bool InRightFoot, EMotionAngleDirection::Type InAngleDirection, E

public:

    /** Return the response start step animation condition */
    UFUNCTION(BlueprintPure, Category = Start)
    FORCEINLINE bool CanResponseStartStep() const { return bResponseStartStep; }

#pragma endregion
```

Data is stored in GaitParams

```
#pragma region Animation

protected:

    /** Idle animations. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    UAnimSequence* BasePose;

    /** Gait animation in place, we can customize the index, Start, Moving, Stop, Pivot, Jump etc.. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    TArray<FMotionGaitAnimsData> GaitParams;
```

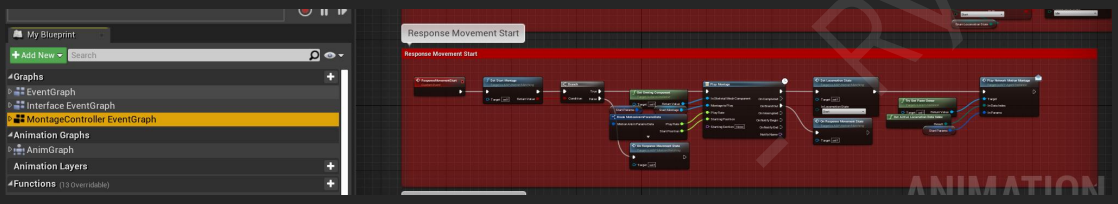
(3) Functions



a. SetStartMontage(): cache the currently expected movement start animation

b. WaitingMovementStart(): waiting for the response to play movement to start, usually to wait for other states to end

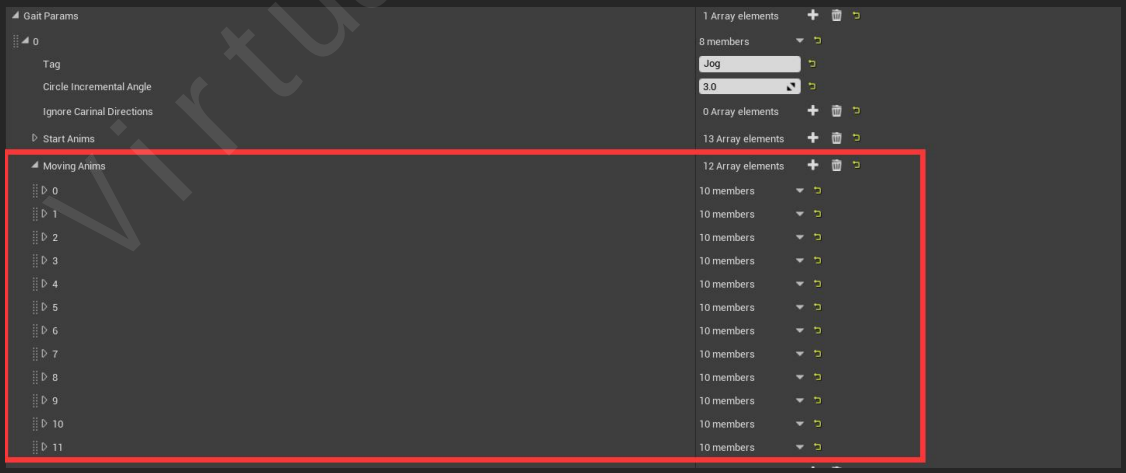
c. ResponseMovementStart(): perform the play movement start animation



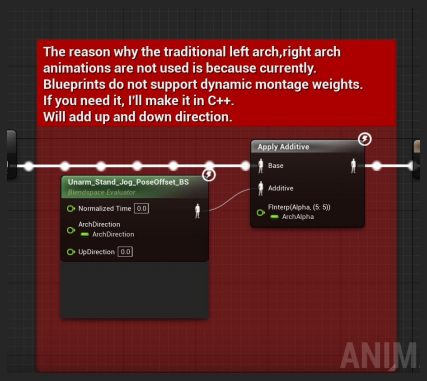
7. Movement Moving

(1) The array index corresponds to **Cardinal Direction+ Left/Right Circle**

0:Forward 1:Forward Circle L 2:Forward Circle R
3:Backward 4: Backward Circle L 5.Backward Circle R
6:Left 7:Left Circle L 8:Left Circle R
9:Right 10:Right Circle L 11:Right Circle R



(If there is no corresponding animation, we will use the program to offset the overlay asset, in this case using the first frame of Circle L/R as the tilt pose, and then using the VAT Bone Blend Layer Tool to match the hands, feet, and head to the forward pose)



(2) Parameter

CircleIncrementalAngle: rotation increment of each frame, which refers to rotation frame difference curve of arc walk animation (VAT generated)

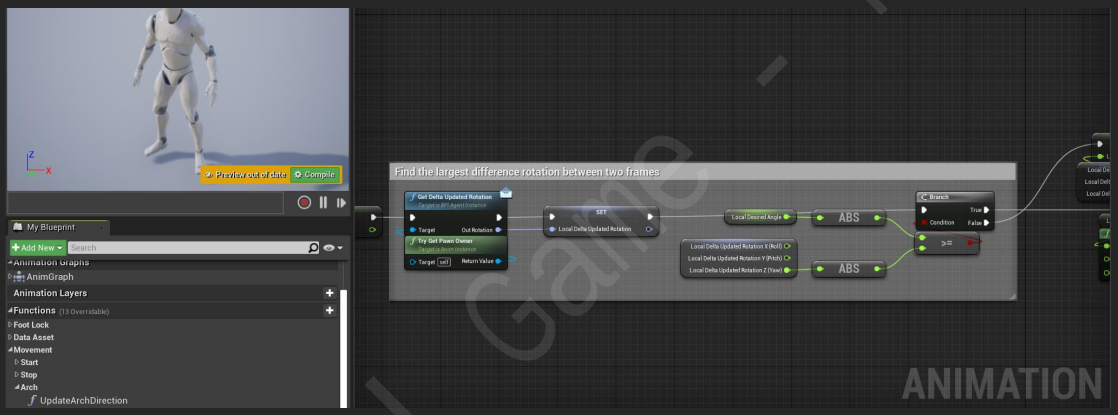
```
/** Struct of locomotion gait animations data */
USTRUCT(BlueprintType)
struct FMotionGaitAnimsData
{
    GENERATED_USTRUCT_BODY()

    /** Description for the current gait animations */
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FName Tag;

    /** The angle by which the Circle rotates per frame */
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    float CircleIncrementalAngle;

    /** Whether to ignore the specified movement direction, it will be an invalid asset */
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TArray<TEnumAsByte<EMotionCardinalDirection::Type>> IgnoreCarinalDirections;

    /** Movement start animations, 0 - 9 is forward start animation, 10 - 12 is other cardinal direction */
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TArray<FMotionAnimParamsData> StartAnims;

    /** Movement loop animations, cardinal direction * (moving + left arch + right arch) */
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TArray<FMotionAnimParamsData> MovingAnims;
```

Data is stored in GaitParams

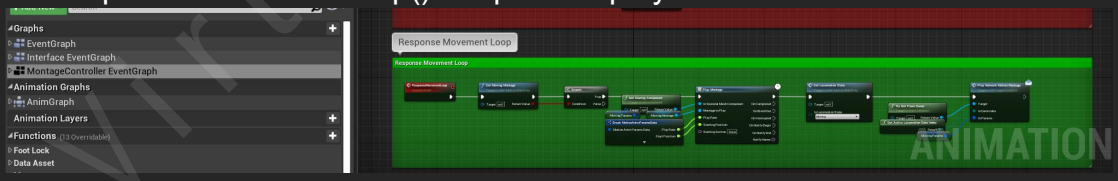
```
#pragma region Animation

protected:

    /** Idle animations. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    UAnimSequence* BasePose;

    /** Gait animation in place, we can customize the index, Start, Moving, Stop, Pivot, Jump etc.. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    TArray<FMotionGaitAnimsData> GaitParams;
```

(3) Functions

a. UpdateArchDirection(): get the current turn value, which is the difference Angle from the Controller rotation minus the current character's positive rotation. The maximum rotation Angle is obtained by comparing this Angle with the value rotated in the previous frame.



b. ResponseMovementLoop(): respond to play the movement animation



Answer the question: Why not adapt to mixed space?

(1) To simplify state machine

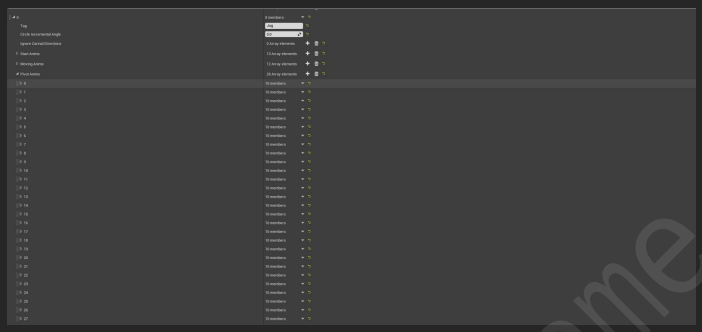
(2) In order to obtain the movement data of root bone, the root bone of the state machine cannot be used in multiplayer networking mode (in fact, the movement data of the root bone can be obtained from the state machine, but the engine needs to be modified. If you are interested, please contact me for further discussion)

(3) The multi-direction blending and arc-moving blending will come later, as it is a bit more code intensive and is an integrated process (i.e. all logical

programming is done in C++), and will be rolled out separately later to better explain how it works

8. Movement Pivot

(1) The array index corresponds to Angle Direction and Cardinal Direction turn in the front direction
0-19：Forward To 0-180 + LF/RF
20-21：Forward To Backward + LF/RF
22-23：Backward To Forward + LF/RF
24-25：Left To Right + LF/RF
26-27：Right To Left + LF/RF
(If there is no corresponding animation, we will use Start + Stop mode to simulate the pivot turn effect)



(2) Parameter
bResponsePivotStep: whether to respond pivot turn

bResponseContinuousPivot: whether to respond continuous pivot turn, if True, can continue the turn during turning

ResponsePivotAngle: respond the angle of the pivot

ResponsePivotIntervalTime: respond interval time between pivot turns

ResponsePivotByStartAnimPos: in the start state, only when the playing position of the start animation is greater than or equal to this value, can it respond to the pivot turn

ResponsePivotByStartAnimWeight: in the start state, the start animation player weight is greater than or equal to this value to respond to the pivot turn

ResponsePivotByStartAnimRotAlpha: in the start state, only when the weight of the rotation curve of the start animation is greater than or equal to this value can it respond to the pivot turn

ResponsePivotByStopAnimTime: in the stop state, only when the playing position of the stop animation is greater than or equal to this value can it respond to the pivot turn

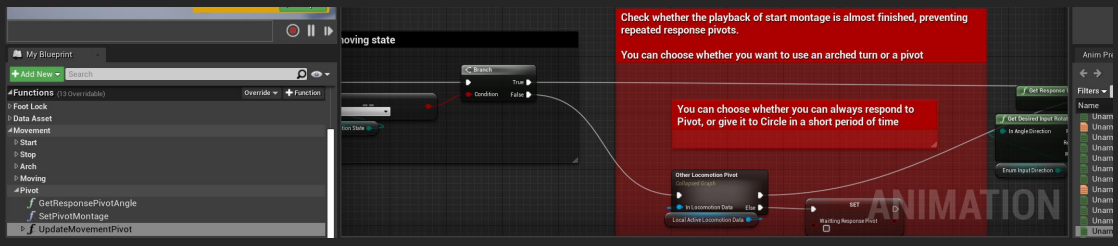
```
    /** Defines whether to perform a pivot animation in response */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    bool bResponsePivotStep;

    /** If it is true, the pivot animation will be played every time the condition for playing the pivot is reached, otherwise the Circle animati
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    bool bResponseContinuousPivot;

    /** Minimum angle(view + input) for responding to pivot animations */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    float ResponsePivotAngle;

    /** Minimum interval for responding to pivot animations, Only used in tick events, in order to get higher weight animation wait time */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    float ResponsePivotIntervalTime;

    /** If the start animation is being played, we can only execute pivot if the animation position is greater than this value. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    float ResponsePivotByStartAnimPos;

    /** If the start animation is being played, we can only execute pivot if the animation weight is greater than this value. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    float ResponsePivotByStartAnimWeight;

    /** If the start animation is being played, we can only execute pivot if the animation rotation alpha is greater than this value. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    float ResponsePivotByStartAnimRotAlpha;

    /**If the pivot animation is not played, but the stop-start pivot mode is used, we will stop the animation and execute the start animation af
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Pivot)
    float ResponsePivotByStopAnimTime;
```

```
public:

    /** Return the response pivot animation condition */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE bool CanResponsePivotStep() const { return bResponsePivotStep; }

    /** Return the response continuous pivot animation condition */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE bool CanResponseContinuousPivot() const { return bResponseContinuousPivot; }

    /** Return the response pivot animation minimum angle */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE float GetResponsePivotAngle() const { return ResponsePivotAngle; }

    /** Return the response pivot animation minimum interval time */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE float GetResponsePivotIntervalTime() const { return ResponsePivotIntervalTime; }

    /** Return the response pivot animation by start animation position */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE float GetResponsePivotByStartAnimPos() const { return ResponsePivotByStartAnimPos; }

    /** Return the response pivot animation by start animation weight */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE float GetResponsePivotByStartAnimWeight() const { return ResponsePivotByStartAnimWeight; }

    /** Return the response pivot animation by start animation rotation alpha */
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE float GetResponsePivotByStartAnimRotAlpha() const { return ResponsePivotByStartAnimRotAlpha; }

    /**If the pivot animation is not played, but the stop-start pivot mode is used, we will stop the animation and execute the start animation af
    UFUNCTION(BlueprintPure, Category = Pivot)
    FORCEINLINE float GetResponsePivotByStopAnimTime() const { return ResponsePivotByStopAnimTime; }
```

Data is stored in GaitParams

```
#pragma region Animation

protected:

    /** Idle animations. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    UAnimSequence* BasePose;

    /** Gait animation in place, we can customize the index, Start, Moving, Stop, Pivot, Jump etc.. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    TArray<FMotionGaitAnimsData> GaitParams;
```

(3) Functions

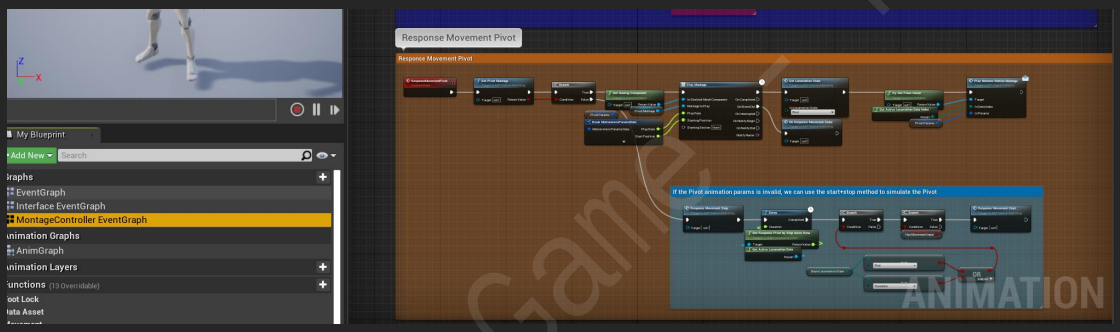


a. GetResponsePivotAngle(): obtain response Angle of the pivot turn

b. SetPivotMontage(): cache the desired pivot turn animation

c. UpdateMovementPivot(): detect each frame whether the pivot turn should be performed (when the Controller rotation changes)

d. ResponseMovementPivot(): response to play the movement pivot animation



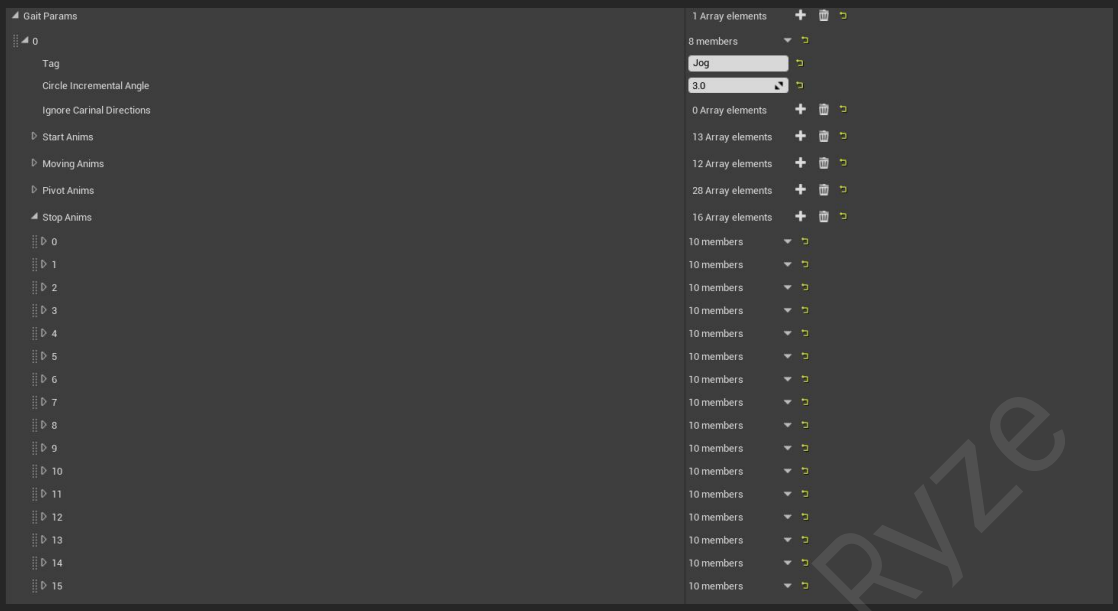
9. Movement Stop

(1) The array index corresponds to the cardinal direction left and right foot stop type * stop types
0-3: Forward Stop LF/ Forward Stop RF/ Backward Stop LF/ Backward Stop RF
4-7: Left Stop LF/ Left Stop RF/ Right Stop LF/ Right Stop RF
Every 8 data loops are of a Stop Section type (type: default slow Stop, quick emergency Stop)

```
/** Types of virtual motion stop section */
UENUM(BlueprintType)
namespace EMotionStopSection
{
    enum Type
    {
        Normal,
        Immediate,
        Max UMETA(Hidden)
    };
}
```

(V1 only needs to configure the first 4 animations to be configured. Forward

Stop LF/RF - Normal/Immediate)



(2) Parameter

bResponseStopStep: whether to respond stop steps

ResponseMovementInputIntervalTime: respond interval time between movement input, to prevent the animation from repeating over a short period of time

ResponseStopByStartAnimPos: in the start state, only when the start animation playing position is greater than or equal to this value can respond the stop

ResponseStopByStartAnimWeight: in the start state, only when the start animation playing weight is greater than or equal to this value can respond the stop

ResponseStopByStartAnimRotAlpha: in the start state, only when the start animation rotation weight is greater than or equal to this value can respond the stop

ResponseStopByPivotAnimPos: in pivot turn state, only when the pivot turn animation playing position is greater than or equal to this value can respond to the stop

ResponseStopByPivotAnimWeight: in pivot turn state, only the pivot turn animation playing weight is greater than or equal to this value can respond the stop

ResponseStopByPivotAnimRotAlpha: in pivot turn state, only the pivot turn animation rotation weight is greater than or equal to this value can respond the stop

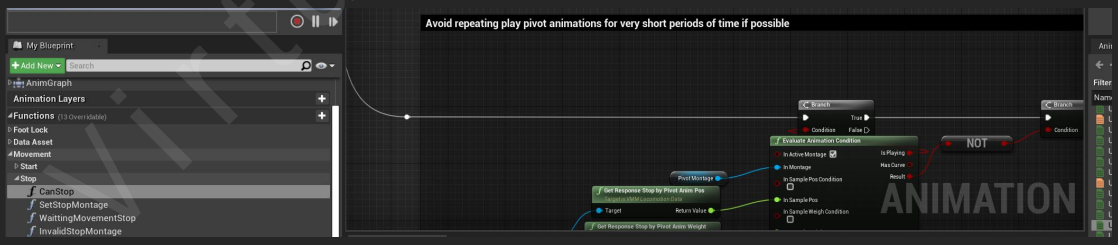
```
    /** Return stop animation params from input data. */
    UFUNCTION(BlueprintCallable, Category = Moving, meta = (DisplayName = "Get Stop Params"))
    FMotionAnimParamsData K2_GetStopParams(const int32 InGaitAxis, const int32 InFootIndex, const EMotionCardinalDirection::Type InDirectionType, EMotionStopSectio
    const FMotionAnimParamsData* GetStopParams(const int32& InGaitAxis, const int32& InFootIndex, const EMotionCardinalDirection::Type& InDirectionType, EMotionSto

public:

    /** Return the response pivot animation condition */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE bool CanResponseStopStep() const { return bResponseStopStep; }

    /** Return the response movement input interval time from last stop record time */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseMovementInputIntervalTime() const { return ResponseMovementInputIntervalTime; }

    /** Return the response stop animation by start animation position */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseStopByStartAnimPos() const { return ResponseStopByStartAnimPos; }

    /** Return the response stop animation by start animation weight */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseStopByStartAnimWeight() const { return ResponseStopByStartAnimWeight; }

    /** Return the response stop animation by start animation rotation alpha */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseStopByStartAnimRotAlpha() const { return ResponseStopByStartAnimRotAlpha; }

    /** Return the response stop animation by pivot animation position */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseStopByPivotAnimPos() const { return ResponseStopByPivotAnimPos; }

    /** Return the response stop animation by pivot animation weight */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseStopByPivotAnimWeight() const { return ResponseStopByPivotAnimWeight; }

    /** Return the response stop animation by pivot animation rotation alpha */
    UFUNCTION(BlueprintPure, Category = Stop)
    FORCEINLINE float GetResponseStopByPivotAnimRotAlpha() const { return ResponseStopByPivotAnimRotAlpha; }
```

Data is stored in GaitParams

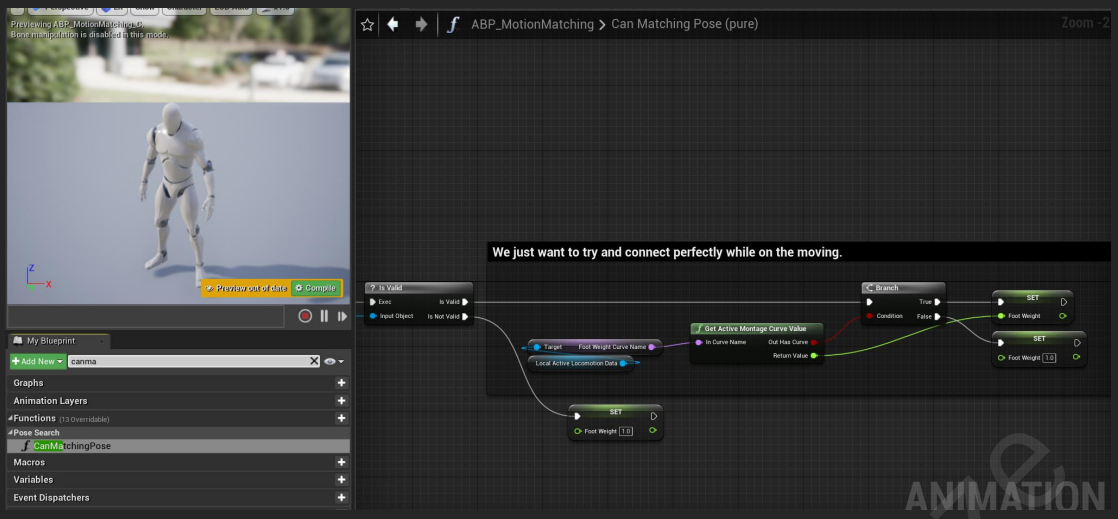
```
#pragma region Animation

protected:

    /** Idle animations. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    UAnimSequence* BasePose;

    /** Gait animation in place, we can customize the index, Start, Moving, Stop, Pivot, Jump etc.. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    TArray<FMotionGaitAnimsData> GaitParams;
```

(3) Functions

a. CanStop(): Return whether a stop can be performed



b. CanMatchingPose(): return to current pose, whether can match the animation pause pose (see VAT Foot Weight Tool)

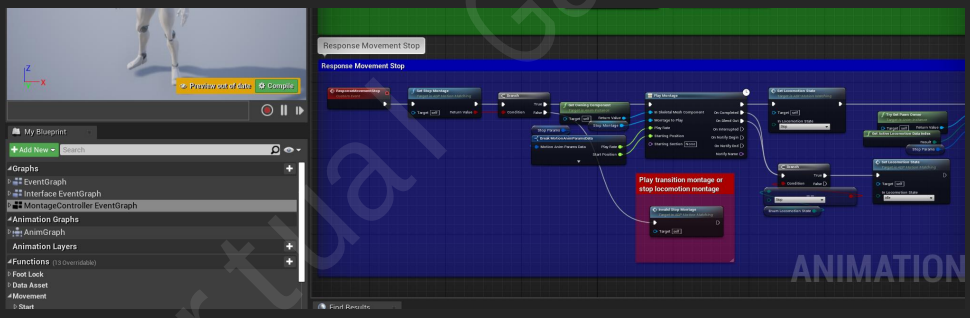


c. SetStopMontage(): cache expected stop animations

d. WaitingMovementStop(): wait respond stops (if the current position does not match and so on)

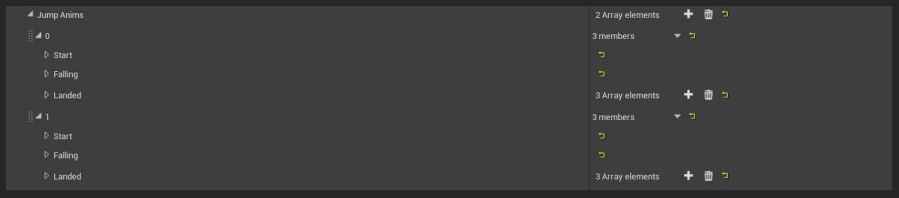
e. InvalidStopMontage(): if it is an invalid stop animation (state mismatch, etc.), we can choose to use the way of program pose transition to stop the animation or directly stop the animation

f. ResponseMovementStop(): respond to play stop animations



10. Jump

(1) The array index corresponds to Angle Direction and Cardinal Direction turn in the front direction
0-3: Forward Jump LF/ Forward Jump RF/ Backward Jump LF/ Backward Jump RF
4-7: Left Jump LF/ Left Jump RF/ Right Jump LF/ Right Jump RF

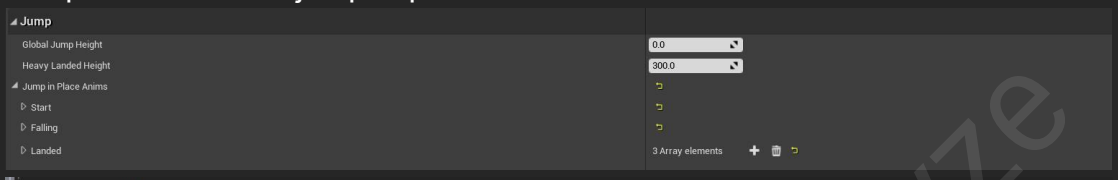


(V1 only needs to configure 2 animations, Forward Jump LF/RF)

(2) Parameter
GlobalJumpHeight: custom jump height will be available in version V2

HeavyLandedHeight: when the fall height is greater than this value, we will use a different landing animation

JumpInPlaceAnims: jump in place animation data

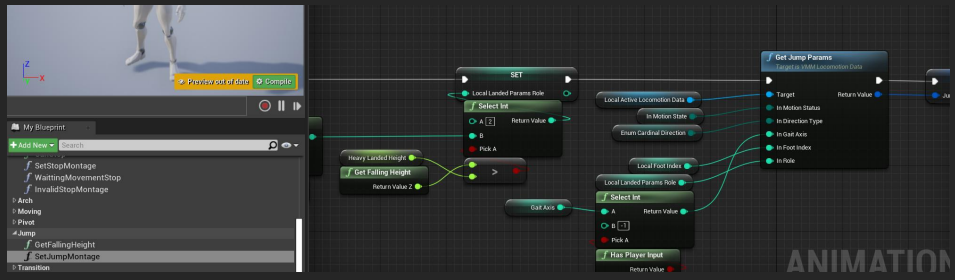



```
#pragma region Jump

protected:

    /** The default global jump height */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Jump)
    float GlobalJumpHeight;

    /** If the height of the fall is greater than this value, heavy landed will be executed */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Jump)
    float HeavyLandedHeight;

    /** Movement jump InPlace animations */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Jump)
    FMotionJumpAnimsData JumpInPlaceAnims;

public:

    /** Return jump animation params from input data. */
    UFUNCTION(BlueprintCallable, Category = Moving, meta = (DisplayName = "Get Jump Params"))
        FMotionAnimParamsData K2_GetJumpParams(const EVirtualMotionStatus::Type& InMotionStatus, const EMotionCardinalDirection::Type& InDirectionType
            , const int32& InGaitAxis, const int32& InFootIndex, const int32 InRole = 0);
    const FMotionAnimParamsData* GetJumpParams(const EVirtualMotionStatus::Type& InMotionStatus, const EMotionCardinalDirection::Type& InDirectionType
        , const int32& InGaitAxis, const int32& InFootIndex, const int32 InRole = 0);

public:

    /** Return the global jump height */
    UFUNCTION(BlueprintPure, Category = Jump)
    FORCEINLINE float GetGlobalJumpHeight() const { return GlobalJumpHeight; }

    /** Return the heavy landed height */
    UFUNCTION(BlueprintPure, Category = Jump)
    FORCEINLINE float GetHeavyLandedHeight() const { return HeavyLandedHeight; }

#pragma endregion
```

Data is stored in GaitParams

```
#pragma region Animation

protected:

    /** Idle animations. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    UAnimSequence* BasePose;

    /** Gait animation in place, we can customize the index, Start, Moving, Stop, Pivot, Jump etc.. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    TArray<FMotionGaitAnimsData> GaitParams;
```

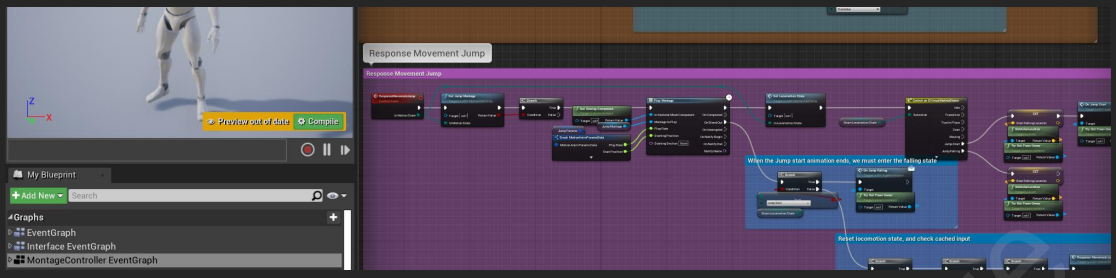
(3) Functions



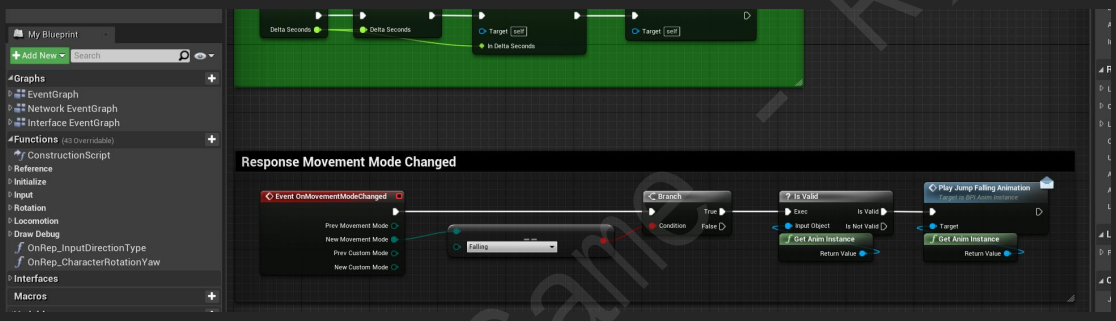
a. GetFallingHeight(): get the falling height

b. SetJumpMontage(): cache expected jump animations

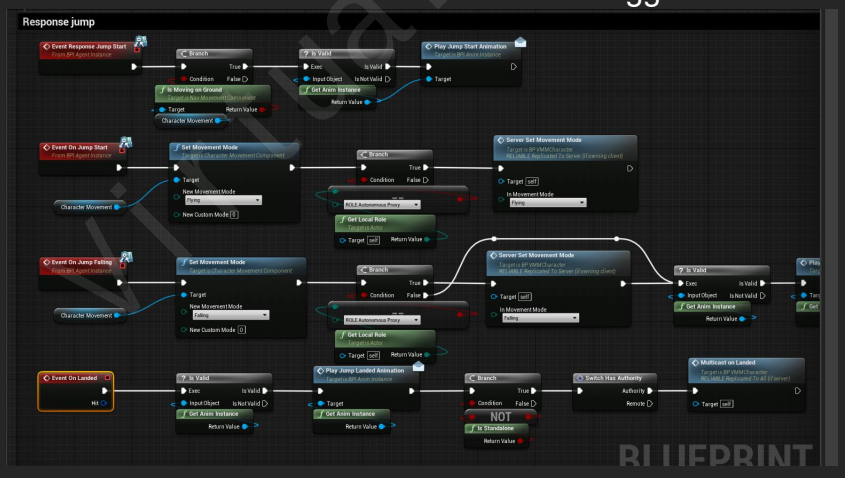
c. Response Movement Jump(): respond to play stop animations



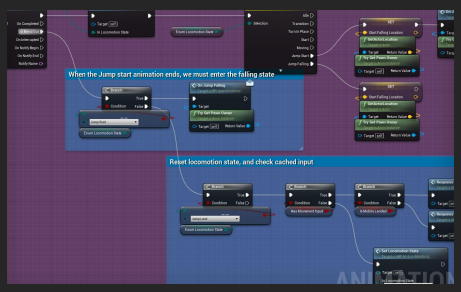
d. ResponseMovementModeChanged(): once the movement mode is switched to Falling, we should notify the animation blueprint to play the corresponding falling animations



e. OnLanded(): once the landing callback is triggered, we should inform the animation blueprint to play the corresponding landing animations. Only the local Controller and the server can trigger this animation, so we can inform the simulation side to land when the server is triggered

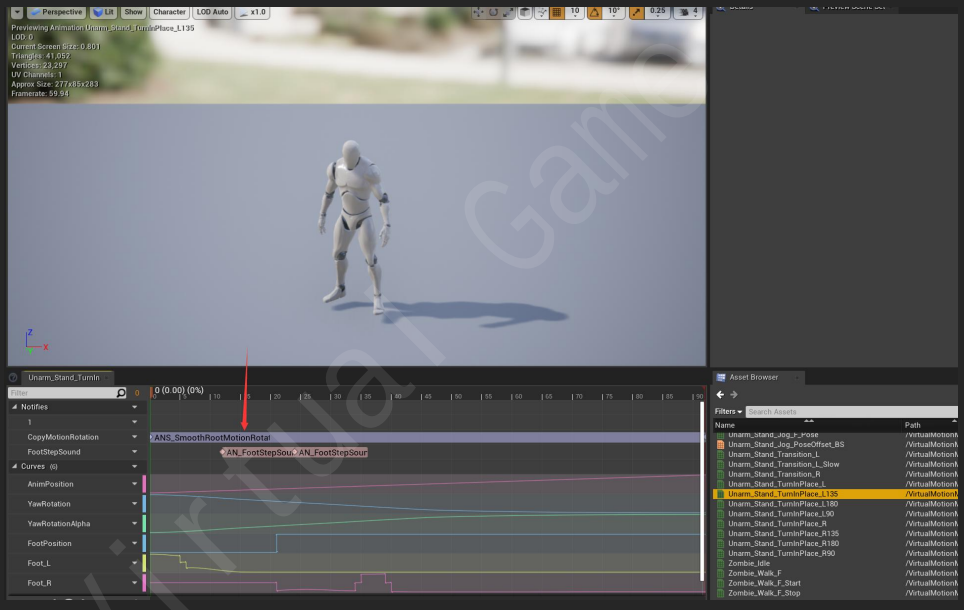


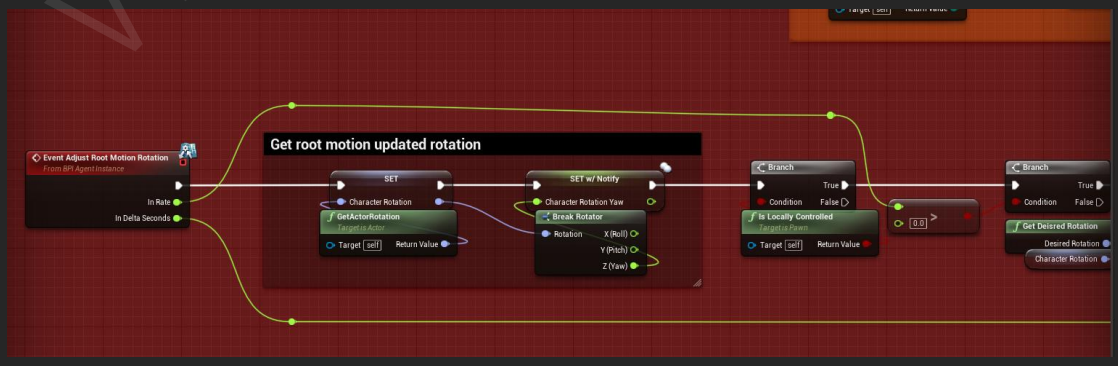
f. OnJumpStart(): the current jump mode is set to fly, and the falling mode is set to execute the engine's original flow when the animation blends out



11. Root Motion and Custom Blend Rotation

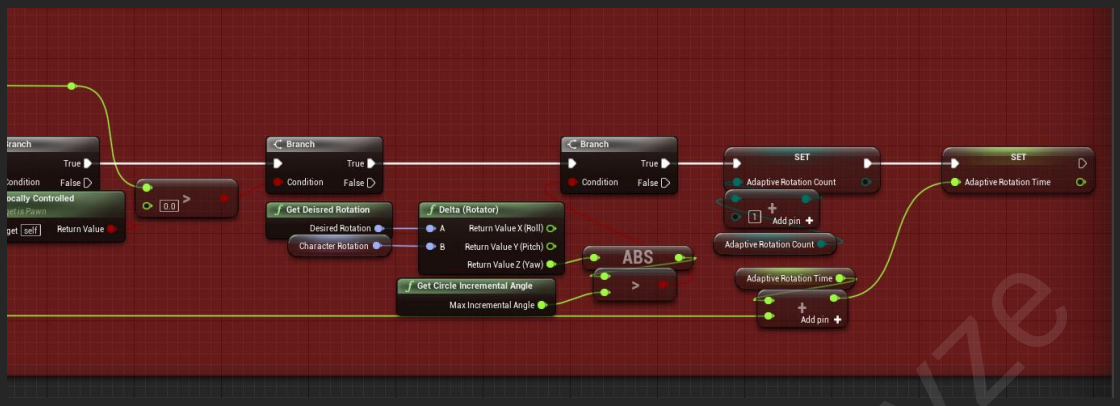
(1) We need to configure on each root bone animation with rotation data ANS_SmoothRootMotionRotation Animation notify status (use VAT tools to batch configuration)



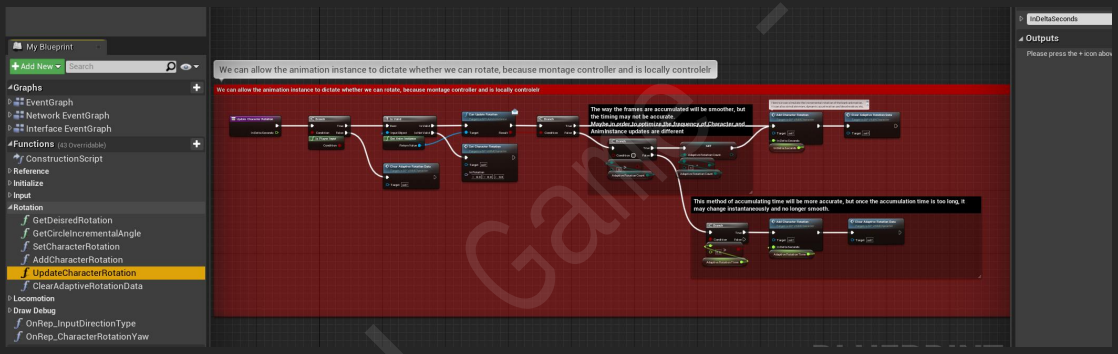


This animation notify caches the current character to a custom CharacterRotation variable and synchronizes the Yaw values on the server

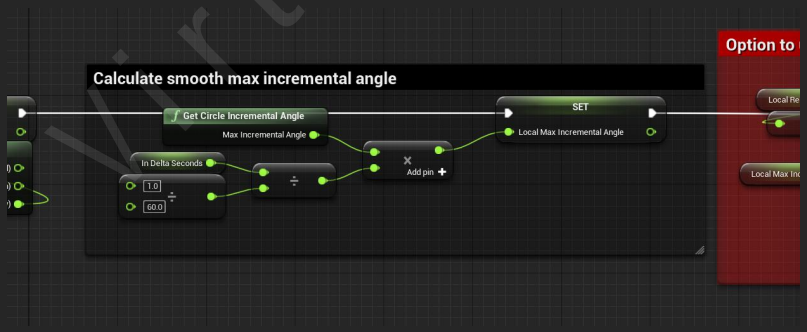
(2) If the animation notify has incremental rotation, we should delay the rotation until the next frame because of the timing of component updates.
Tick update order: CharacterMovement→Character→AnimInstance



(3) UpdateCharacterRotation
This function is responsible for managing character rotation updates

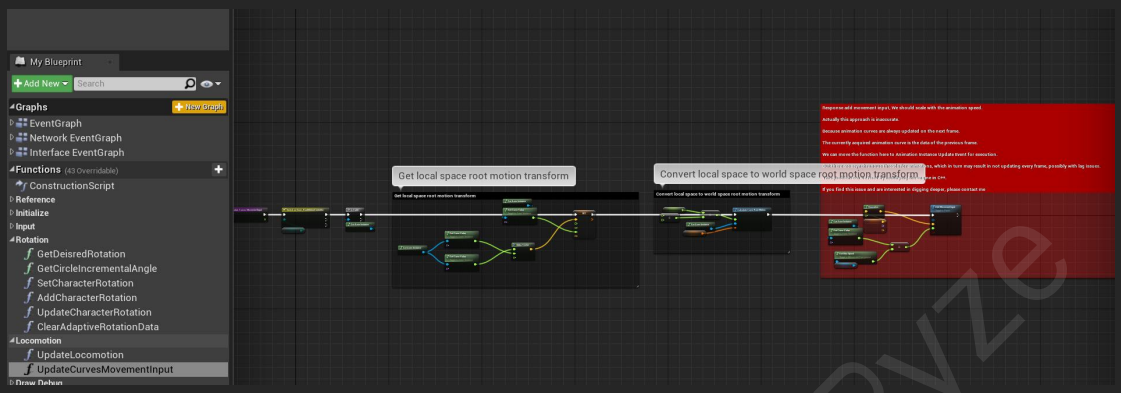


(4) AddCharacterRotation
This function is responsible for the incremental rotation of each frame (here we should convert the rotation Angle of each frame to the default animation data of 60 frames, keeping the rotation value constant at different frame rates)



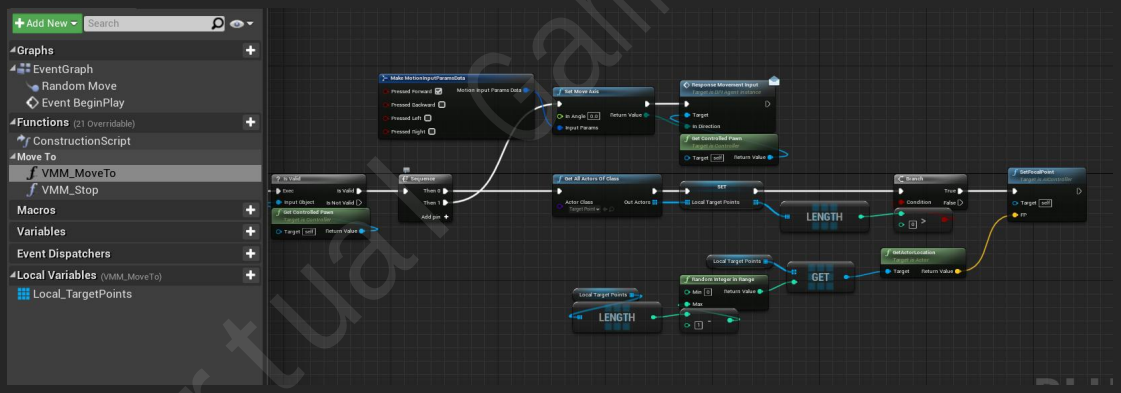
12. Curves Controller

Since this method has deviation and the plug-in already implements a more accurate montage controller, there is only a simple code framework (deprecated later)



13. AI

Currently AI only has simple movement stop behavior, there will be a full case of 3A-AI using motion matching



14. Game Framework curves

Bilibili:
(1) footstep IK:
https://www.bilibili.com/video/BV1744y1V7CS/?spm_id_from=333.788
(2) footstep lock:
https://www.bilibili.com/video/BV1R3411J7h6/?spm_id_from=333.788
(3) footstep weight:
https://www.bilibili.com/video/BV1qT4y1Y7to/?spm_id_from=333.788
(4) footstep position:
https://www.bilibili.com/video/BV1fa411i7Pq/?spm_id_from=333.788

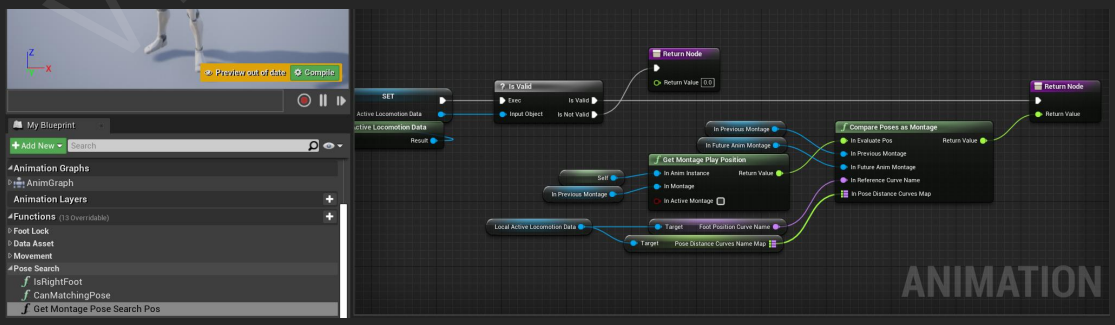
YouTuBe:
(1) footstep IK:
https://www.youtube.com/watch?v=NbeXMBoWbf0&list=PLYP0ozRez418TgeOFVmyJSd87FlEBTPrg&index=30
(2) footstep lock:
https://www.youtube.com/watch?v=ztq9zZu-6XA&list=PLYP0ozRez418TgeOFVmyJSd87FlEBTPrg&index=31
(3) footstep weight:
https://www.youtube.com/watch?v=67RhPq5cVUw&list=PLYP0ozRez418TgeOFVmyJSd87FlEBTPrg&index=32
(4) footstep postion
https://www.youtube.com/watch?v=IIYhyh2mm5c&list=PLYP0ozRez418TgeOFVmyJSd87FlEBTPrg&index=33

15. Pose Search

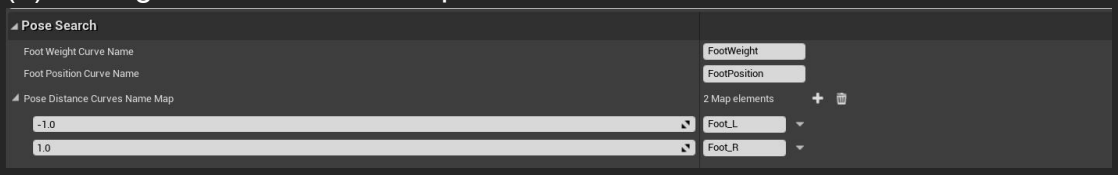
Bilibili:
https://www.bilibili.com/video/BV1rY4y1H7bx/?spm_id_from=333.788
YouTuBe:
https://www.youtube.com/watch?v=x5KjB79fKJA&list=PLYP0ozRez418TgeOFVmyJSd87FlEBTPrg&index=28
(1) IsRightFoot(): Judge the footstep range of the current pose and return to left or right foot (refer to VAT footstep position tool)

(2) CanMatchingPose(): judge whether the pose is matched (refer to VAT footstep weight tool)

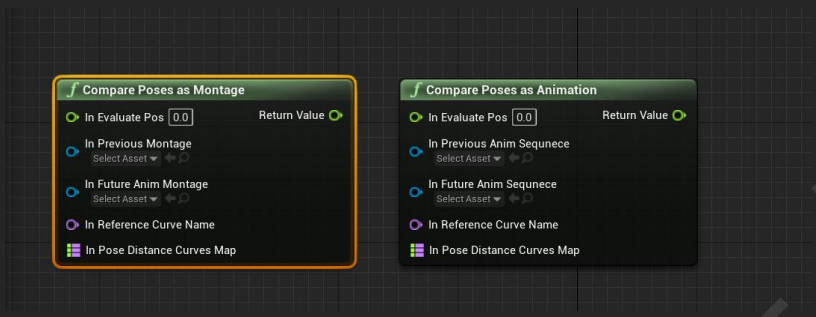
(3) Get Montage Pose Search Pos: get the jump position of the current montage matching the next montage



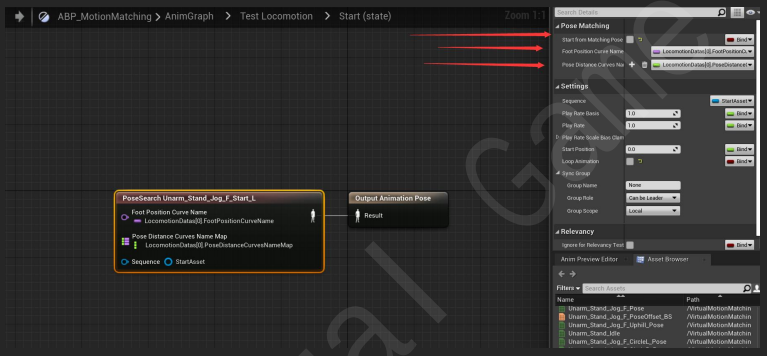
(4) Configure data needed for pose search in Locomotion Data



(5) You can use these functions to perform pose search in two animation assets

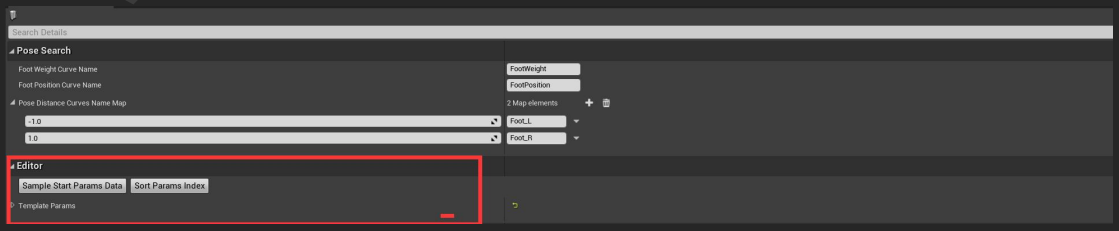


(6) Pose search can be used in the state machine (currently only used in 4.27)



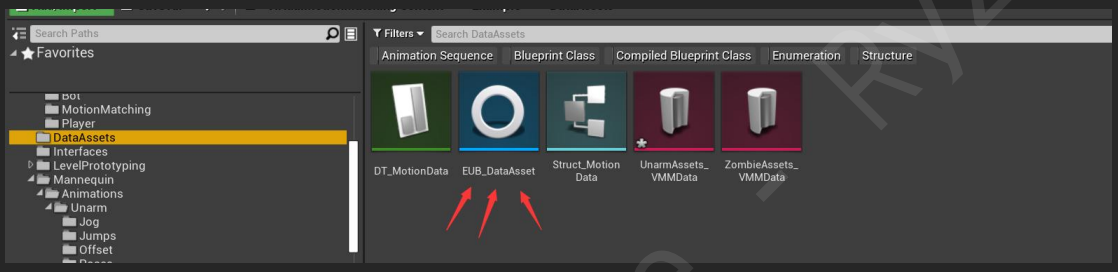
16. Editor Script

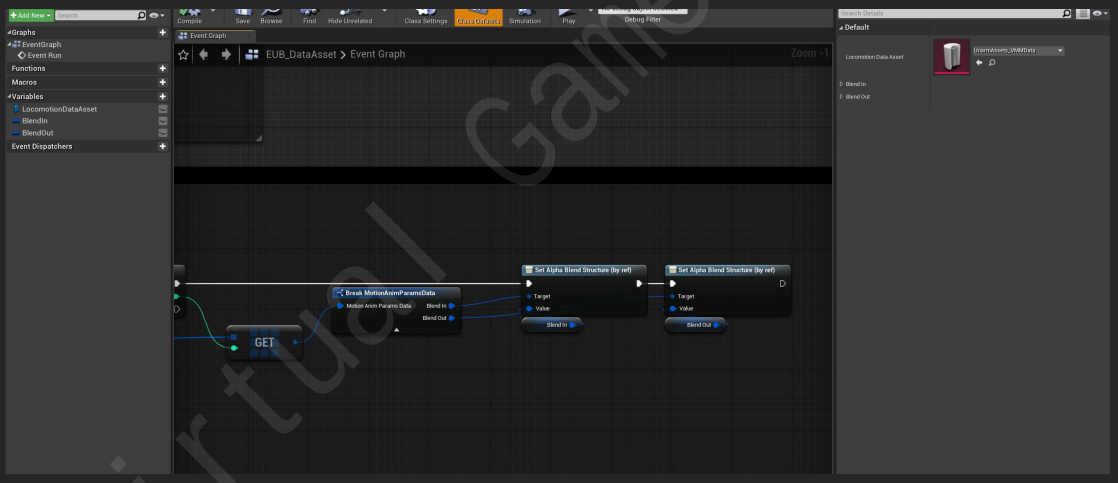
(1) Script functions that can be used in C++ custom editor



```cpp
#if WITH_EDITOR
protected:

    /** An index that sorts all data to support fast queries. */
    UFUNCTION(CallInEditor, Category = Editor)
    void SortParamsIndex();

    /** Change start params data from template params. */
    UFUNCTION(CallInEditor, Category = Editor)
    void SampleStartParamsData();

#endif // WITH_EDITOR
```

(2) Batch processing of animation data with blueprint script (batch setting of blending in, blending out animation data)

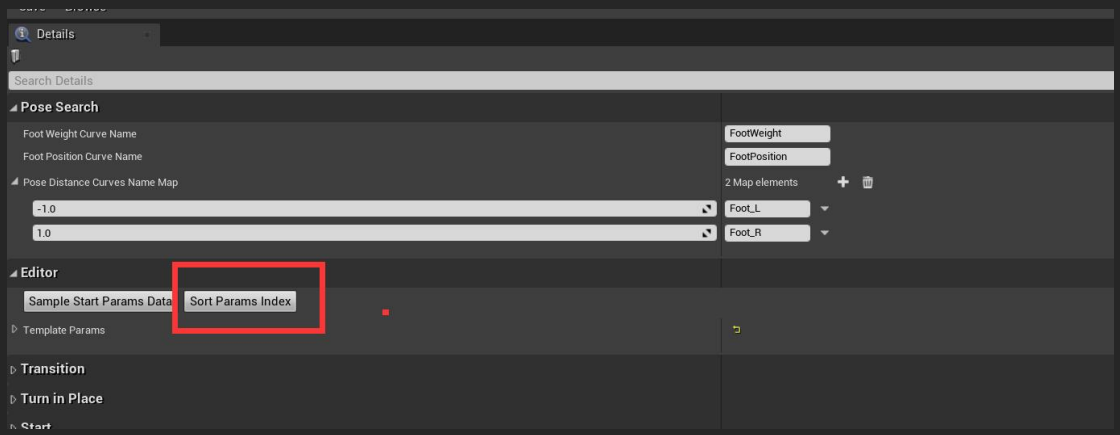




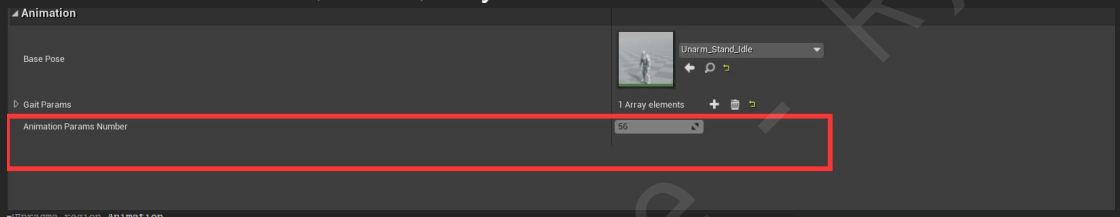
17. Animation Data and Network Optimization

(1) The reason why we use arrays instead of TMap or other data structures is for editor visualization and code performance optimization

(2) After each configuration, click this button for index sorting before running. This is for optimization of network packet sending. We only need to send the corresponding DataIndex and ParamsIndex to quickly search the corresponding animation data

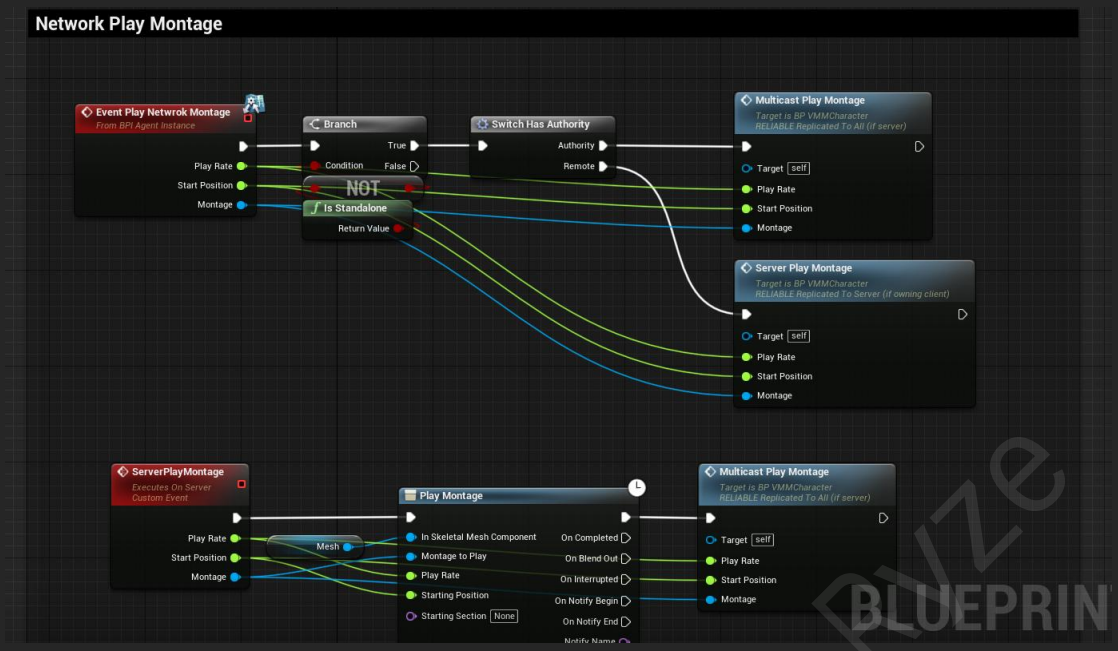


(3) In order to prevent the editor from getting stuck due to too much data, only the number of sorted animations is shown here. Currently, the maximum amount of data is 256, that is, 1 byte

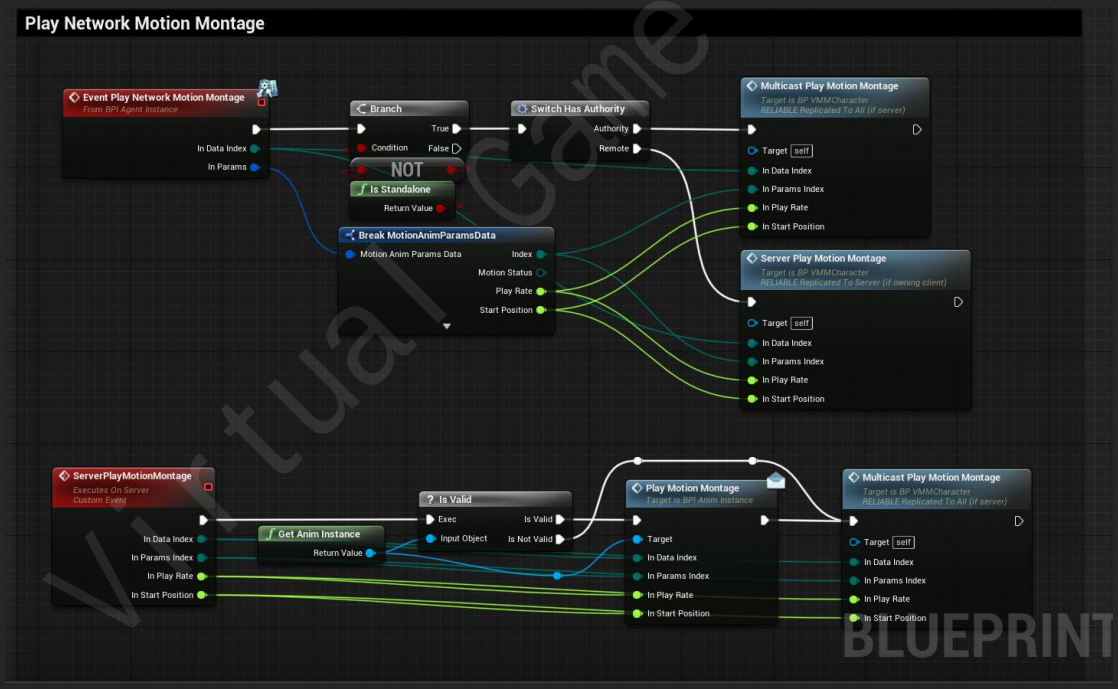



```
#pragma region Animation

protected:

    /** Idle animations. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    UAnimSequence* BasePose;

    /** Gait animation in place, we can customize the index, Start, Moving, Stop, Pivot, Jump etc.. */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Animation)
    TArray<FMotionGaitAnimsData> GaitParams;

    /** Number of animations currently collected */
    UPROPERTY(VisibleDefaultsOnly, Category = Animation)
    int32 AnimationParamsNumber;

    /*
    * Collect all animation params so that you can quickly get the specified params according to the index
    * Not displayed in the editor, because it may cause lag
    */
    UPROPERTY(/*VisibleAnywhere, Category = Animation*/)
    TArray<FMotionAnimParamsData> MotionParams;

public:

    /** Return valid gait animation data index from input data */
    UFUNCTION(BlueprintPure, Category = Animation)
    const int32 GetValidGaitAnimsDataIndex(int32 InGaitAxis, const EMotionCardinalDirection::Type& InCardinalDirection);

    /** Return gait animation data from gait axis */
    FORCEINLINE const FMotionGaitAnimsData* GetGaitAnimsData(const int32& InGaitAxis)
    {
        return GaitParams.IsValidIndex(InGaitAxis) ? &GaitParams[InGaitAxis] : nullptr;
    }

    /** Return the motion params from animations array. */
```

(4) Use montage object packet, the first packet will have extra amount of data (dynamically created montage should not be used in this way, or need to set dynamically created montage to network object in C++)



(5) Use DataIndex and ParamsIndex to send packets greatly reduces the amount of packet data sent



18. Motion Debug and Parameter Debug

(1) Motion debug will be introduced in V2, with visual debug such as motion matching and pose search

(2) Parameter debug, we can use TAB key to call out the corresponding interface and modify the corresponding parameters in real time in the game (you can define your favorite template class)