Procedural Parkour and Traversal Animation Techniques

A thesis submitted by

RAHUL CHANDRA
s5627101

MSc Computer Animation and Visual Effects
National Centre for Computer Animation
Bournemouth University
Date of Submission: 12/08/2024

ABSTRACT

The aim of this thesis was to investigate procedural animation techniques with an emphasis on dynamic and responsive parkour movements using Unreal Engine. The first scenario explores leg, hand, and head IK: leg IK ensures precise foot placement, hand IK facilitates accurate object interaction, and head IK maintains natural gaze. The second scenario deals with the implementation of complex parkour animations, such as mantling, vaulting, and climbing. By exploring these techniques, the thesis demonstrates how characters can dynamically adapt to complex environments, such as scaling obstacles, vaulting over barriers, and executing fluid transitions between different movement types.

ACKNOWLEDGEMENT

I would like to express my gratitude to my supervisor, Professor Jon Macey, for his invaluable advice and guidance in advancing this project and igniting my passion for other areas of computer graphics.

I would also like to thank the entire teaching staff at the National Centre for Computer Animation for their direct and indirect support throughout the year, which has greatly contributed to refining my skills in various subjects within and beyond this course.

Table of Contents

INTRODUCTION

In recent years, the field of video game animation has seen substantial advancements, pushing the boundaries of realism and interactivity. Traditional animation techniques, while effective in many contexts, often struggle to capture the dynamic and spontaneous nature of complex movements such as parkour. Early examples, like the 1990s platformer Prince of Persia, used frame-by-frame animation to depict acrobatic movements, but these were limited by their pre-defined sequences. More recent games, such as Assassin’s Creed and Mirror’s Edge, have made strides in integrating parkour elements, yet they still rely heavily on scripted animations and keyframes, often lacking the adaptability needed for truly fluid interactions with diverse environments. Traditional animation methods often fall short in handling the complexity and spontaneity of parkour movements, particularly when characters interact with diverse and dynamic environments. Pre-defined sequences and keyframe-based animations can struggle with the seamless integration required for realistic climbing and climbing-related actions, such as vaulting and mantling. These methods typically lack the adaptability to respond in real time to changes in the environment or to the character’s dynamic interactions with climbable surfaces.

This thesis aims to address these limitations by exploring and implementing advanced procedural animation techniques within Unreal Engine. By integrating Control Rig, procedural Inverse Kinematics (IK), and Motion Warping, this research develops a comprehensive framework for generating dynamic and adaptive parkour movements. The focus is on creating realistic climbing locomotion and interactions, such as accurate limb positioning during climbing and precise adjustments based on surface dimensions. Through custom animation blueprint movement components, this work seeks to improve the overall realism and interactivity of parkour animations, ensuring that characters can navigate complex environments with greater flow and responsiveness.

LITERATURE REVIEW

2.1 Procedural Animation Techniques

Procedural animation has become increasingly prominent in video game development as a means to create more dynamic and responsive character movements. Traditional animation approaches often involve pre-defined sequences, which can lead to repetitive and rigid actions. Early attempts at procedural animation can be traced back to games like Spore (Maxis, 2008), where procedural methods were used to animate creatures with diverse body types and behaviours, adapting animations based on player-designed creatures. Although groundbreaking, these animations were primarily limited to simple actions and lacked the complexity required for more intricate movements like parkour and traversal.

Figure 2.1 First attempt at procedural animation in Spore (Maxis, 2008)

Another notable example is Euphoria (NaturalMotion, 2007), an engine used in games such as Grand Theft Auto IV (Rockstar Games, 2008) and Red Dead Redemption (Rockstar Games, 2010). Euphoria enabled characters to react in real-time to their environment with procedural animations, such as stumbling or bracing for impact during falls. This was a significant step forward from static animations, allowing for more natural and varied character responses. However, while Euphoria excelled in simulating physical reactions, it was still constrained by the limitations of its physics-based system, often resulting in unpredictable or exaggerated movements.

2.2 Inverse Kinematics in Game Animation

Inverse Kinematics (IK) is a fundamental technique in-game animation, allowing characters to interact more naturally with their environment by adjusting limb positions based on specific targets, such as placing feet accurately on uneven terrain or reaching for objects. Early uses of IK can be seen in games like Half-Life 2 (Valve, 2004), where it was employed to ensure that characters' feet stayed grounded on uneven surfaces. This implementation was fairly basic but laid the groundwork for more advanced uses of IK in future games.

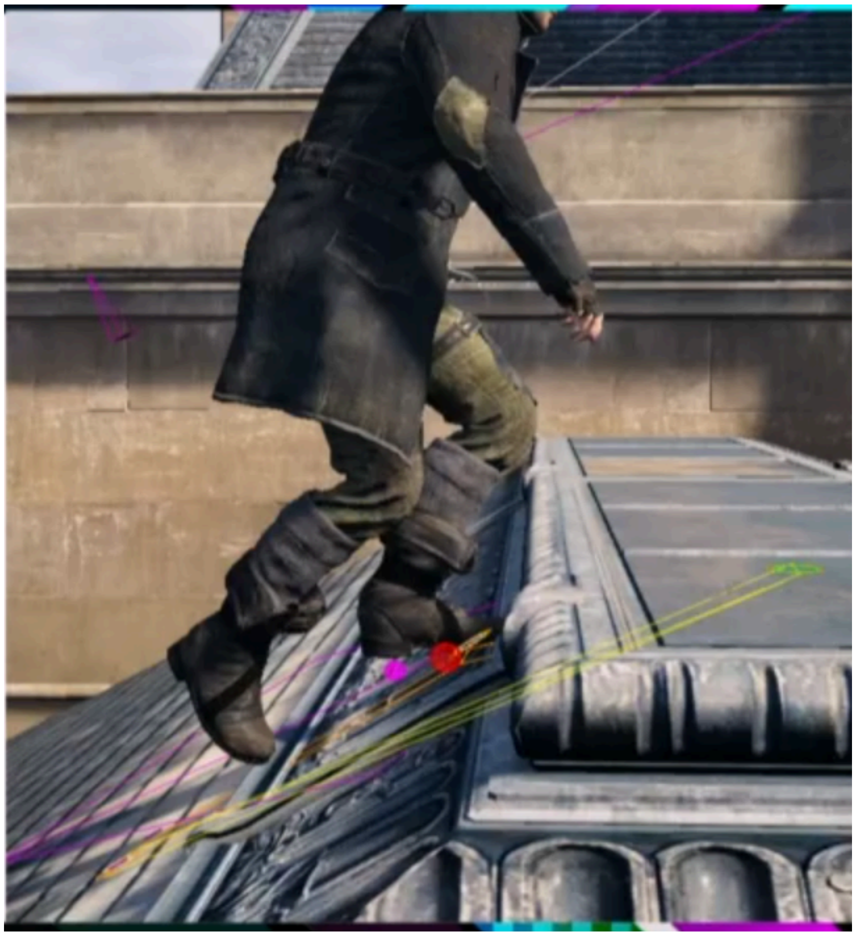

Figure 2.2 Leg IK implementation in Assassin's Creed: Syndicate(Ubisoft, 2015)

Recent titles such as Assassin's Creed: Syndicate(Ubisoft, 2015) and The Last of Us Part II (Naughty Dog, 2020) have advanced the use of IK, enabling more complex interactions like characters dynamically adjusting their posture while navigating tight spaces or accurately placing hands and feet during climbing sequences. These games showcase the potential of IK in creating more believable animations, though they still rely heavily on pre-scripted sequences, limiting the full adaptability of the movements.

2.3 Motion Warping and Advanced Climbing Animations

Motion Warping is a technique used to modify animations in real time, allowing characters to adapt their movements based on the surrounding environment. A key example is Uncharted 4: A Thief's End (Naughty Dog, 2016), where the character's animations are adjusted dynamically to interact with ledges, walls, and other obstacles, creating a more seamless and immersive climbing experience. This method allows for a greater degree of flexibility compared to traditional animation techniques, enabling characters to respond more naturally to varying terrain.

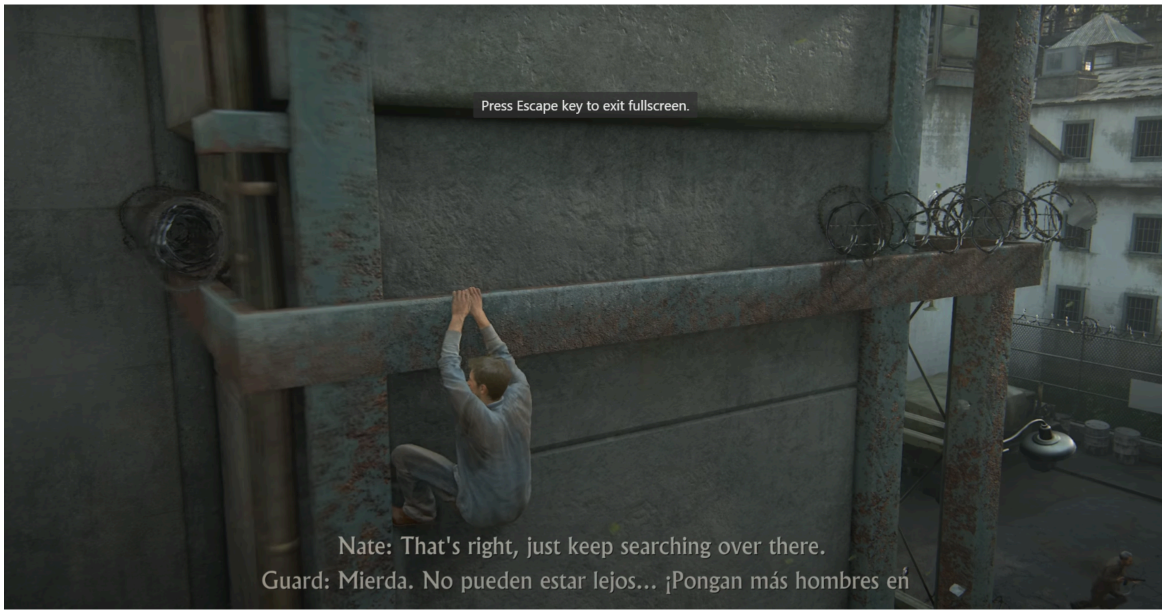
Press Escape key to exit fullscreen.
Nate: That's right, just keep searching over there.
Guard: Mierda. No pueden estar lejos... ¡Pongan más hombres en

Figure 2.3 Ledge climbing with hand IK and motion warping in Uncharted 4: A Thief's End (Naughty Dog, 2016)

Despite these advancements, many current implementations of Motion Warping and climbing animations remain somewhat rigid, with movements often restricted to pre-defined paths or animations that cannot fully adapt to unexpected environmental changes. This limitation highlights the ongoing need for more sophisticated procedural techniques that can dynamically adjust animations in real-time, ensuring a more fluid and responsive character experience in complex environments.

TECHNICAL BACKGROUND

3.1 Inverse Kinematics (IK)

Inverse Kinematics (IK) is essential for realistic limb positioning in character animation, particularly in dynamic environments. IK calculations involve determining the joint angles needed for a character's limb to reach a specific target point. One of the commonly used algorithms in Unreal Engine for IK is the **Forward And Backward Reaching Inverse Kinematics (FABRIK)** algorithm. The basic formula for calculating the position of joints using FABRIK can be described as:

1. **Backward Reaching:**

$$\mathbf{p}_i = \mathbf{p}_{i+1} + \frac{l_i}{d_i}(\mathbf{p}_i - \mathbf{p}_{i+1})$$

   - p_i is the position of joint i.
   - l_i is the length of the segment between joint i and i +1.
   - d_i is the distance between p_i and p_{i+1}
2. **Forward Reaching:**

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \frac{l_i}{d_i}(\mathbf{p}_{i+1} - \mathbf{p}_i)$$

   These steps are repeated iteratively until the end effector (e.g., hand or foot) reaches the target position within an acceptable threshold.

3.2 Sphere and Line Tracing

In this project, sphere tracing and line tracing were key techniques taht were used detecting environmental interactions. Sphere tracing casts a virtual sphere from a specific body part, like a hand or foot, to check for nearby surfaces. For most animations , this was Implemented in Blueprints using the Sphere Trace for Objects node. This method was ideal for detecting proximity and ensuring natural character interactions, such as hand placements on walls and wall in front of the player.

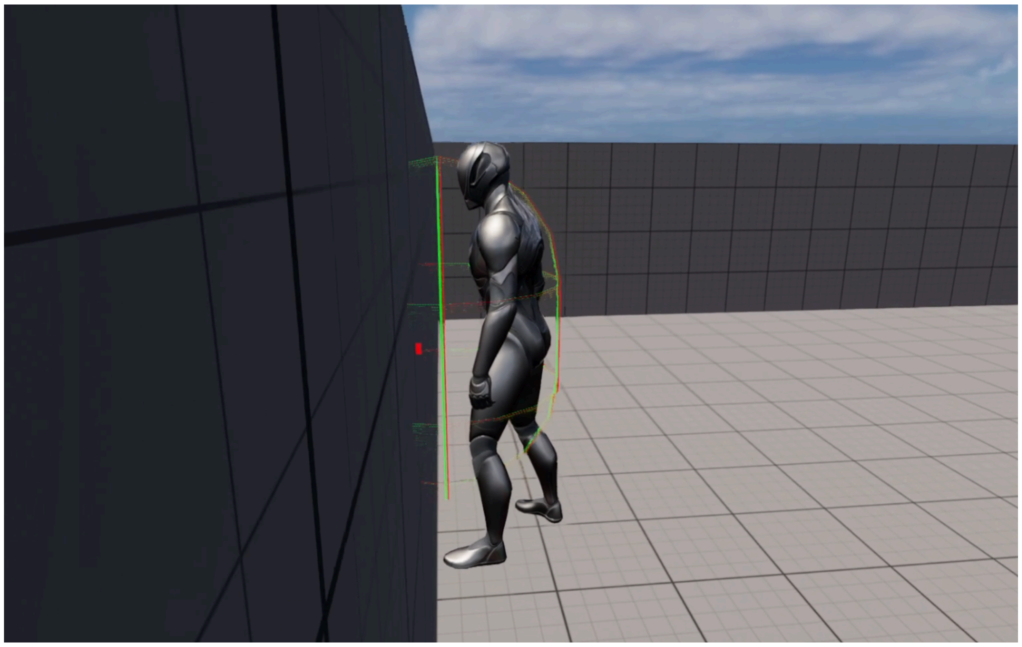

Figure 3.1 Implementation of Sphere Trace by Channel for detecting distances between wall

Line tracing involves casting a straight ray from the character's position or limb toward a direction to detect surfaces. It’s typically used to determine exact interaction points, such as where a foot should land on uneven terrain. The Line Trace by Channel node in Blueprints helps achieve this by providing data like the impact point, which is crucial for precise character placement.

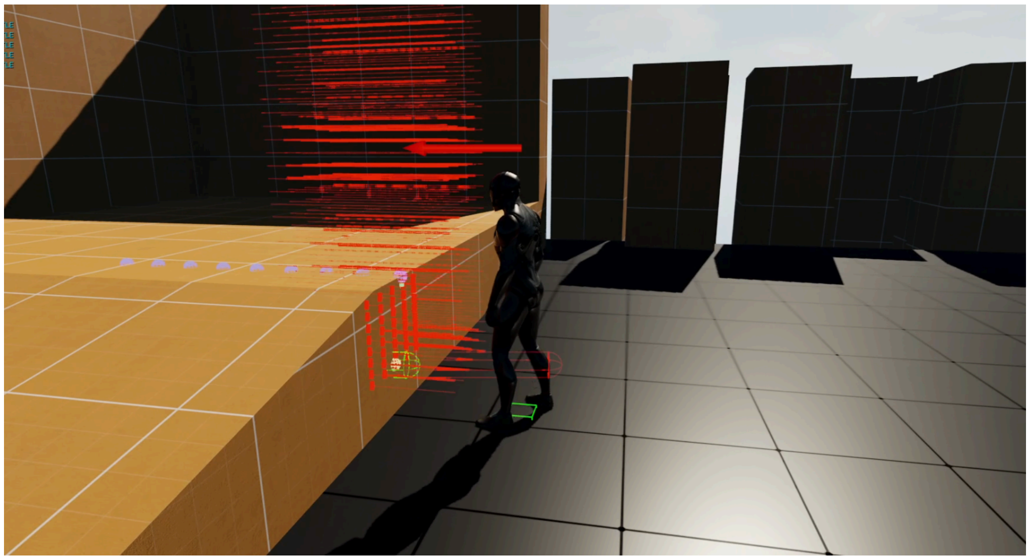

Figure 3.2 Implementation of Line Trace to detect height of obstacle

3.3 Motion Warping

Motion Warping adjusts the character’s movement path dynamically to fit the environment, ensuring seamless transitions between actions. The root motion of an animation sequence is altered based on the environment's conditions.

This technique allows for dynamic adjustments, like changing the trajectory or landing position during parkour actions, such as vaulting or jumping.

3.4 Control Rig

Control Rig in Unreal Engine 5 was utilized for setting up and manipulating character rigs dynamically. This system utilises procedural controls to adjust a character's skeletal structure during runtime, providing real-time feedback and adjustments based on interactions with the environment. For instance, during climbing, Control Rig helps to dynamically position the hands and feet by solving the IK chains, ensuring they align with the detected surfaces. The control rig can be defined with constraints such as position, orientation, and scale, which are mathematically represented by transformation matrices:

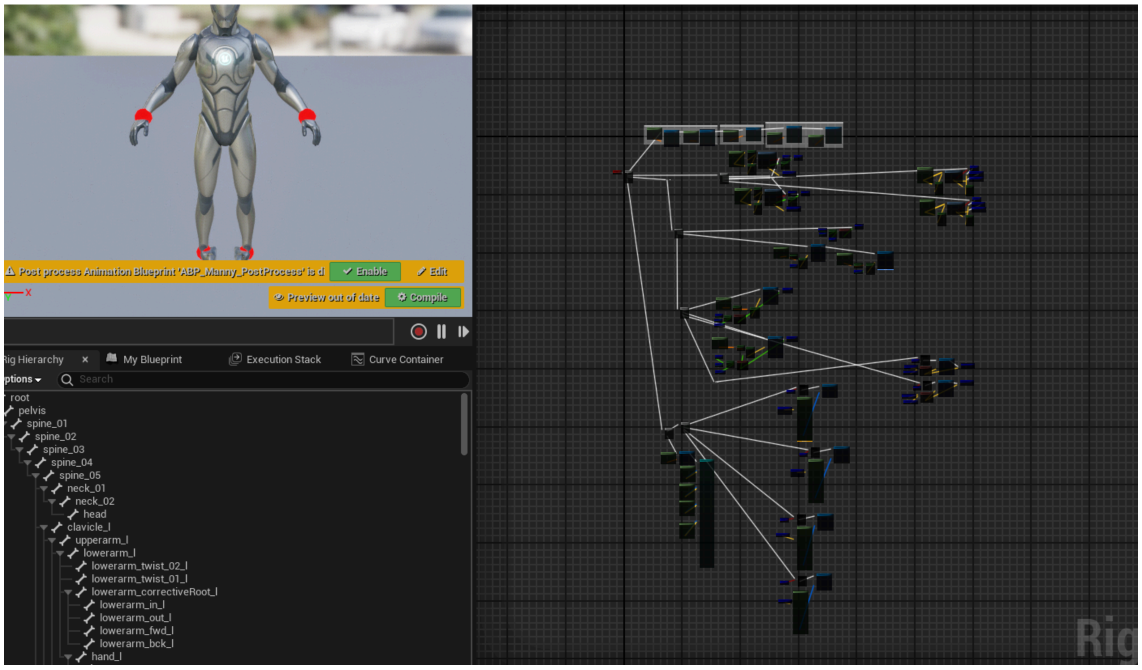

Figure 3.3 Custom Control Rig for implementing Leg and Hand IK

These transformations allow for real-time adjustment of the character’s pose, enhancing the fluidity of movements in complex animations.

SOLUTION

4.1 Scope

The primary objective of this project was to implement a dynamic and immersive parkour system in Unreal Engine. So, the system first explores full body IK using Control Rig in unreal engine . Post that, the system integrates custom character movement mechanics for climbing and vaulting, utilizing C++ for core functionality and Blueprints and motion warping for animation.

4.2 Implementation

4.2.1 IK System and Control Rig

In this project, the IK (Inverse Kinematics) System and Control Rig were implemented in Unreal Engine to create realistic and adaptive character animations, particularly focusing on the hands, legs, and head. For hand IK, IK nodes were configured within the character's animation blueprint to ensure that the hands would precisely follow designated targets, such as objects or specific locations. This setup allowed for natural hand movements during interactions and gestures, enhancing the realism of character animations in various scenarios.

The leg IK system was employed to adjust foot placement dynamically based on terrain conditions, ensuring that the character's feet aligned correctly with uneven surfaces. This was crucial for maintaining realism in walking and running animations. Additionally, head IK was used to create responsive head movements, allowing the character's head to track targets, such as other characters or points of interest.

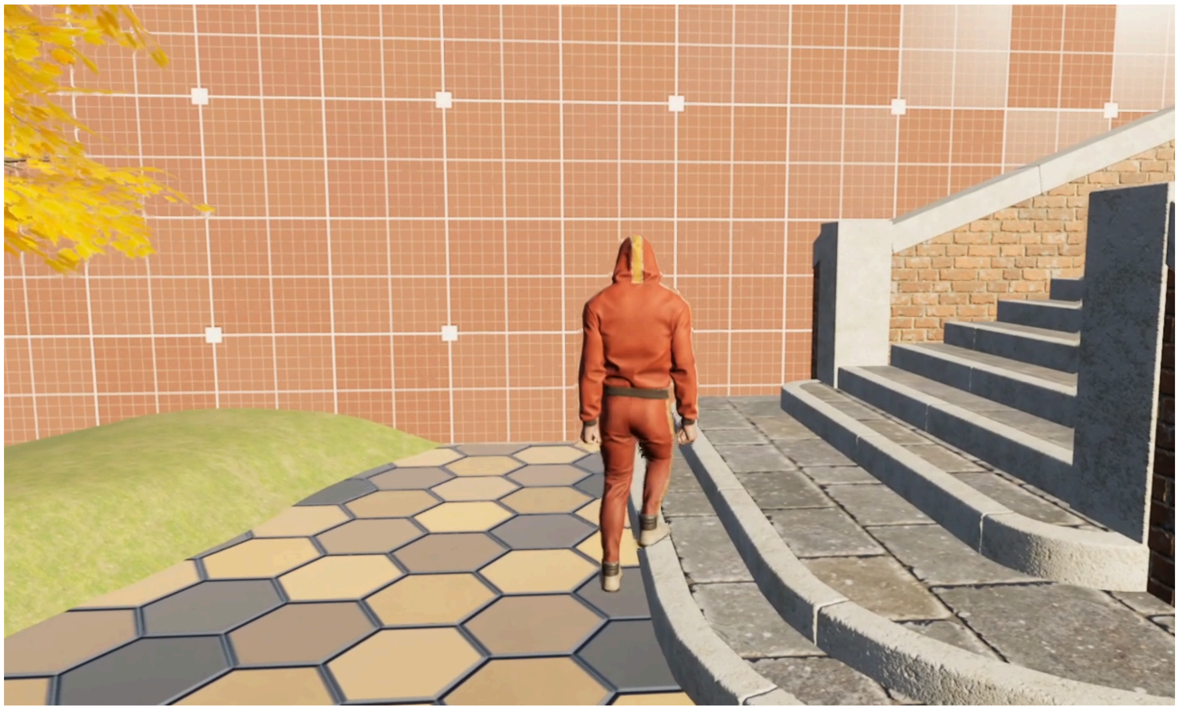

Figure 4.1 Leg IK implementation on uneven surfaces like stairs.

4.2.2 Tracing

This technique was particularly effective in determining when a character was close enough to a wall or ledge to initiate climbing or vaulting animations. By using the Sphere Trace for Objects node in the Blueprint system, accurate detection of surfaces was achieved, ensuring that parkour movements like wall runs and jumps were triggered at the right moments.

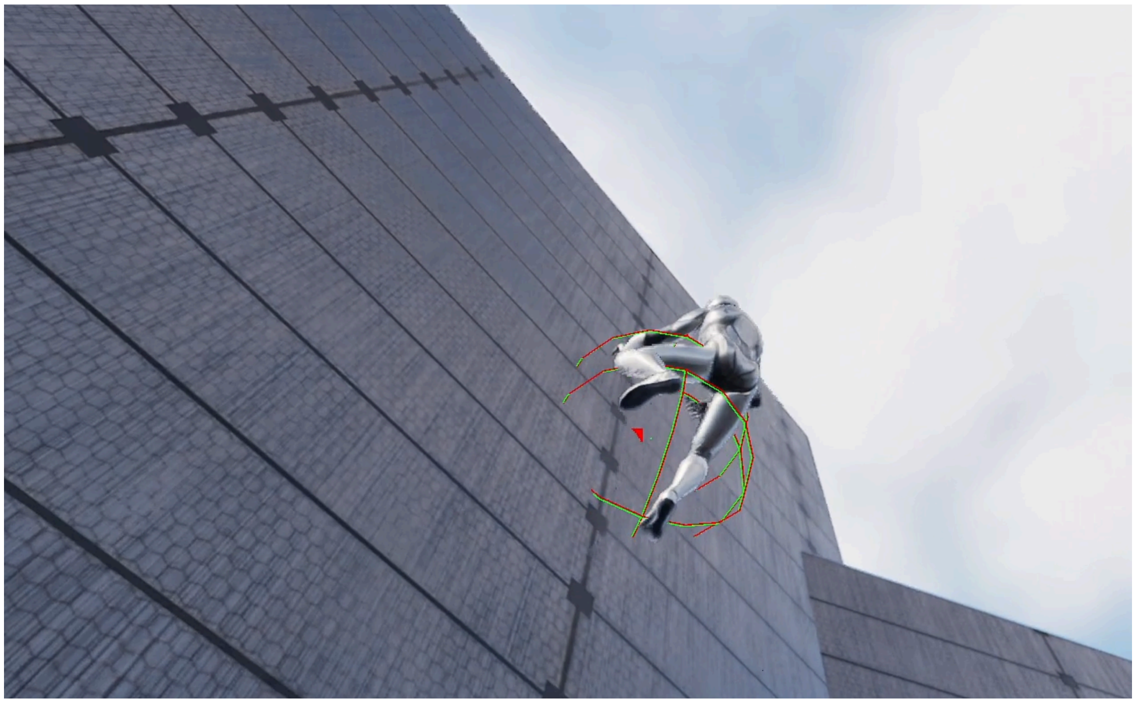

Figure 4.2 Sphere Tracing implementation to detect wall distance

Line tracing was also implemented to determine the precise points of interaction during parkour sequences. This was crucial for actions such as foot placement during wall runs or hand placement when grabbing ledges.

Together, sphere and line tracing provided a robust framework for detecting environmental interactions, enabling responsive and realistic parkour animations in the virtual environment.

4.2.3 Motion Warping

Motion warping allowed for the modification of an animation's root motion to better align with specific environmental conditions or player inputs. By integrating animation notifies, precise adjustments were made during key moments in an animation sequence, such as altering a jump's trajectory or adjusting a character's landing position to match the environment. This ensured that character movements were fluid and contextually appropriate, regardless of the initial animation data.

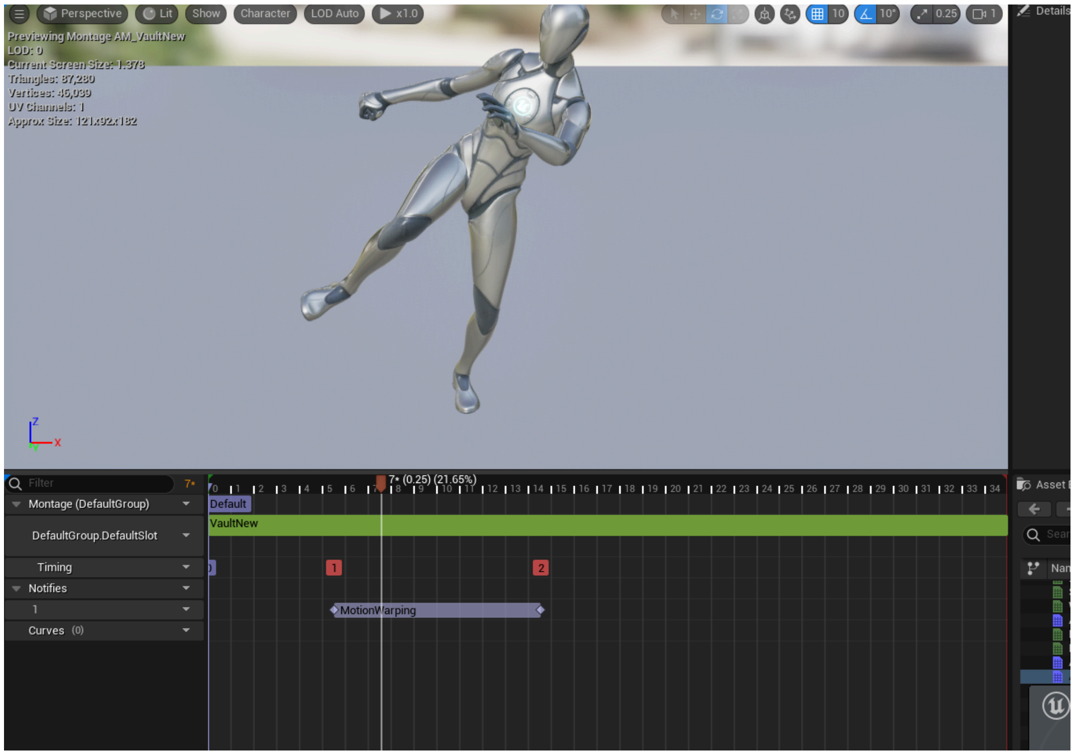

Figure 4.3 Motion warp anim notifies for vault animation

Animation notifies played a crucial role in triggering motion warping at the right moments. These notifies were strategically placed within the animation timeline to activate warping adjustments, such as scaling the distance of a leap based on the distance to the target surface or modifying a roll to better align with the terrain. The combination of motion warping and animation notifies provided a highly adaptable animation system, allowing characters to interact with their environment in a more dynamic and realistic manner.

4.2.4 Climbing and Vault System using C++

From the C++ side of the project, the climbing mechanics are primarily managed by a combination of custom classes and components, including UCharacterAnimInstance, AClimbingCharacter, and UCustomMovementComponent. The UCharacterAnimInstance class is integral to the animation system, providing real-time updates on character states such as climbing. It pulls crucial data from the AClimbingCharacter class and the UCustomMovementComponent, including ground and air speed, climbing status, and climb velocity. The NativeUpdateAnimation function ensures that animations reflect the character's current movement state by calling helper functions like GetGroundSpeed and GetIsClimbing, which update animation parameters accordingly.

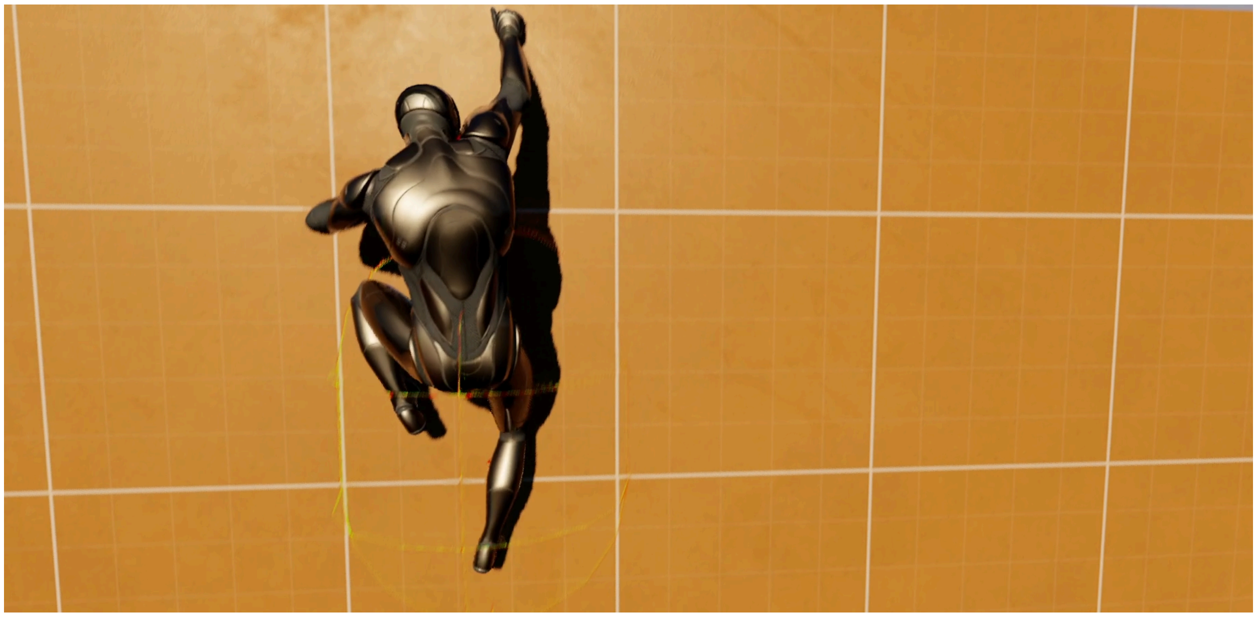
Figure 4.4 Climbing implementation using CustomMovementComponent C++ class

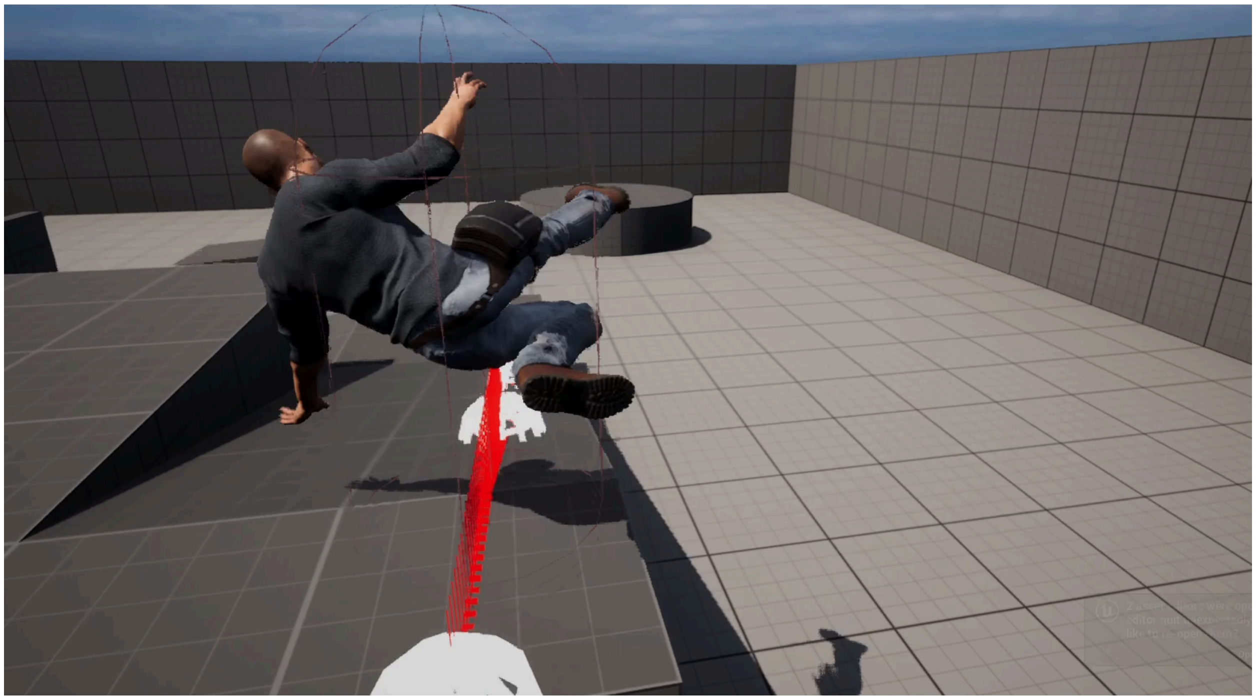
Figure 4.5 Vault area detection using C++ class

The AClimbingCharacter class is central to the character's movement and input management. This class sets up various input actions for movement, climbing, and camera control through the SetupPlayerInputComponent function. It manages the transition between ground movement and climbing via the Move function, which directs input handling to either HandleGroundMovementInput or HandleClimbMovementInput based on the current movement state. Additionally, the OnClimbActionStarted function toggles the climbing state using the ToggleClimbing method from UCustomMovementComponent, which controls whether the character engages in climbing or returns to regular movement.

The UCustomMovementComponent class extends the base movement functionality to support climbing mechanics. It handles the physics and movement logic for climbing through methods such as DoCapsuleTraceMultiByObject and TraceClimbableSurfaces, which detect potential climbing surfaces. The component also manages the climbing state transitions with functions like OnEnterClimbStateDelegate and OnExitClimbStateDelegate, which are bound to the climbing character's state changes. This setup ensures that climbing interactions are seamlessly integrated with the character's movement and animation systems, creating a cohesive and immersive climbing experience in the game.

4.2.6 Sliding and Crouching

For convenience, sliding and crouching mechanics are typically managed through Blueprints, which handle both character states and interactions with the environment. In case of sliding and crouching both, detecting obstacles and ensuring the character doesn’t stand up when under something is of key importance. So, collision traces and overlapping volumes were used to prevent this. The Blueprint can perform checks to see if there’s enough space above the character and switch to crouching or sliding if necessary, thereby preventing the character from standing in confined spaces.

Crouching is handled similarly, where a Blueprint can adjust the character’s capsulesize and collision profile when the crouch state is activated. This change allows the character to fit into lower spaces while maintaining a crouched posture.

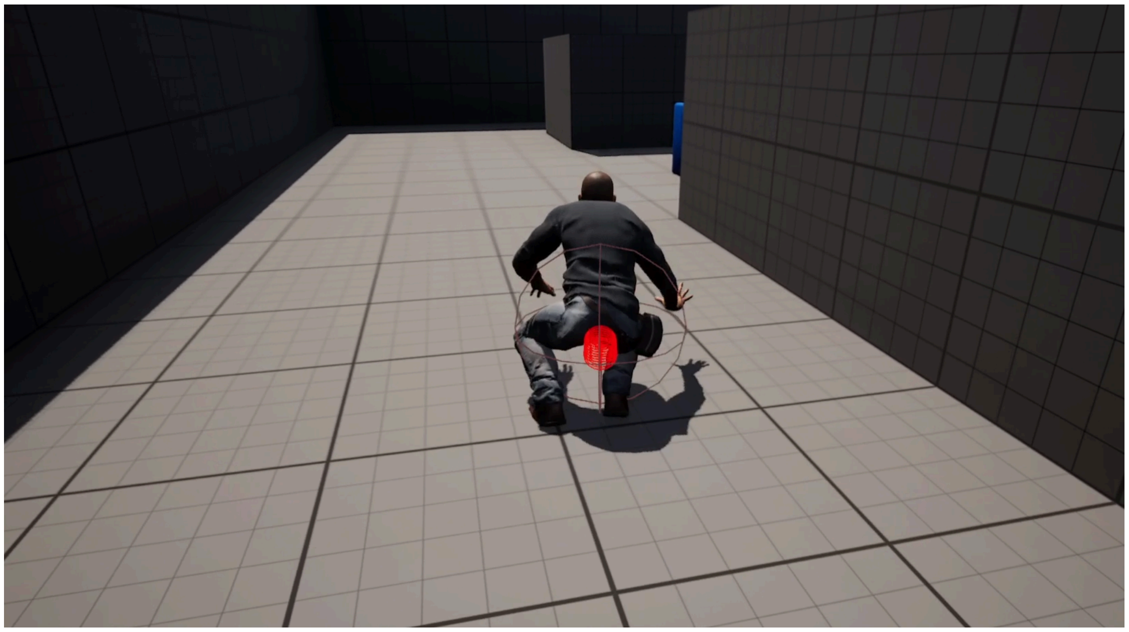

Figure 4.6 Capsule collider adjustment according to the state

4.2.7 Locomotion System

Locomotion in Unreal Engine, including walking, sprinting, and idling, is managed through a combination of Animation Blueprints and Character Movement Components. The Character Movement Component handles fundamental movement mechanics by adjusting parameters such as speed and acceleration. For walking and sprinting, distinct animations are triggered based on the character's current speed, and it involved the use of Blend Spaces. For instance, when the player initiates a sprint, the animation blueprint switches to a faster sprinting animation while adjusting the movement component’s max walk speed. Idle animations play when no movement input is detected, creating a sense of realism and responsiveness. These transitions between states—walking, sprinting, and idling—are smoothly blended in the animation blueprint using blend spaces to ensure fluid and natural character movement.

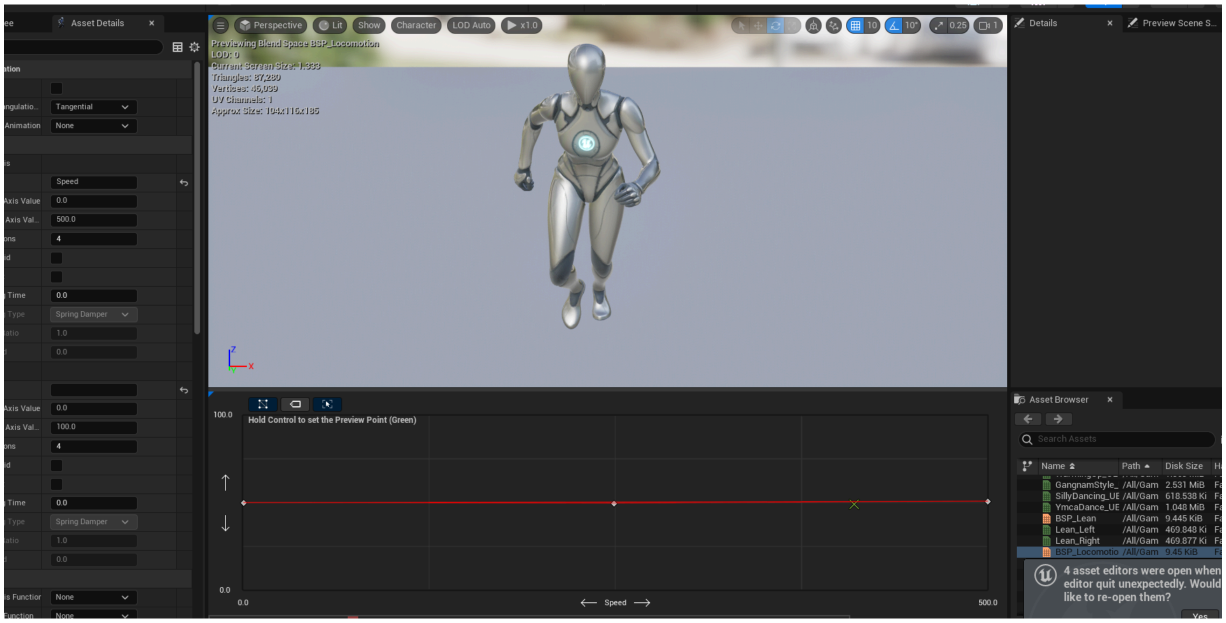
Asset Details
Perspective
Lit
Show
Character
LOD Auto
x1.0
Previewing Blend Space BSP_Locomotion
Hold Control to set the Preview Point (Green)
Speed
Details
Preview Scene S
Asset Browser
Search Assets
4 asset editors were open when editor quit unexpectedly. Would like to re-open them?

Figure 4.7 Blend Space for Walk/Sprint

CONCLUSION

This project successfully demonstrates the potential of procedural animation techniques. By integrating advanced systems such as Control Rig, Inverse Kinematics (IK), Motion Warping, and environment detection algorithms, the project achieved its objective of creating adaptive parkour animations. The framework developed in Unreal Engine allowed characters to interact with complex environments, accurately adjusting their movements based on surface conditions,dimensions and player inputs.

Despite these successes, there are areas where the project could be further improved.

One limitation observed was the occasional lack of hand and foot placements in complex geometries involving irregular surfaces or rapid changes in direction. Additionally, the current system, while highly adaptive, still relies heavily on pre-configured parameters, which could potentially limit its ability to handle entirely unforeseen scenarios dynamically.

Future work on this project could focus on several key areas:

- **Enhanced Procedural Logic**: Further development could include more sophisticated procedural logic that allows for on-the-fly adjustments to character movement paths. This could be achieved by integrating machine learning techniques that enable the system to learn and adapt from gameplay data, resulting in more organic and unpredictable character behavior.

- **Further Implementation of motion warping and environment detection using C++**: The current system relies more on blueprints instead of C++ for logic implementation. For optimsation purposes C++ is preferred method for implementing this than Blueprints.
- **Expanded Environmental Interaction**: Expanding the range of environmental interactions is another critical area for future work. This could involve developing more complex surface detection algorithms that account for a wider variety of surface types and conditions, such as slippery or unstable surfaces, to add another layer of realism to the character's movements.
- **Multiplayer and Cooperative Scenarios**: Another potential area of development is extending the framework to support multiplayer and cooperative scenarios. This would involve synchronizing the procedural animation system across multiple characters, allowing for coordinated parkour actions and interactions between players, which would enhance the overall gameplay experience.
- **Advanced AI Integration**: Incorporating more advanced AI-driven decision-making processes could further improve the adaptability of the system. For example, integrating an AI that can predict and respond to the player’s movements in real-time could create more challenging and engaging parkour sequences, pushing the boundaries of what’s possible in interactive environments.
- **Optimization for Performance**: Finally, optimizing the system for better performance, particularly in large-scale environments or on lower-end hardware, would be essential for broadening the accessibility of the developed techniques. This could involve streamlining the IK calculations and refining the algorithms used for surface detection and motion warping.

In conclusion, while the project has achieved its primary objectives, it also opens up numerous possibilities for future development.

REFERENCES

Watson, D., 2023. Creating Dynamic Parkour Animations in Unreal Engine 5. 12 May 2023. [online] Available at: https://www.youtube.com/watch?v=ofA6YWVTURU&t=1084s&ab_channel=DevinWatson

GDC Vault. 2017. Fitting the World: A Biomechanical Approach to Character Animation | GDC Vault [online] Available at: https://www.gdcvault.com/play/1023316/Fitting-the-World-A-Biomechanical

Unreal Engine Forums. Available at: https://forums.unrealengine.com

Mixamo.com. n.d. Mixamo. [online] Available at: <https://www.mixamo.com>

Epic Games. n.d. Control Rig Documentation | Unreal Engine [online] Available at: : https://dev.epicgames.com/documentation/en-us/unreal-engine/control-rig?application_version=4.27

80.lv. 2022. How to Set Up Procedural Turret Animation with Unreal Engine's Control Rig Available at: : https://80.lv/articles/how-to-set-up-procedural-turret-animation-with-unreal-engine-s-control-rig/

Unreal Engine Marketplace. n.d. Envirosense: Immersive Hand IK [online] Available at: https://www.unrealengine.com/marketplace/en-US/product/envirosense-immersive-hand-ik/questions

ContinueBreak. 2024. Procedural Walking Anim Generator UE5 Available at: : https://continuebreak.com/articles/procedural-walking-anim-generator-ue5/#google_vignette

YouTube. n.d. Unreal Engine: Full Body IK Setup Tutorial [online] Available at: https://www.youtube.com/watch?v=SM_AR-oZ-1k&ab_channel=UnrealEngine

Unity Asset Store. n.d. Final IK Available at: : https://assetstore.unity.com/packages/tools/animation/final-ik-14290

Foundry. n.d. Full Body IK Setup | Modo Help Available at: :
https://learn.foundry.com/modo/content/help/pages/animation/fullbody_ik_setup.html

Unreal Engine Forums. n.d. How Do You Set Up a Rig for Full Body IK? Available at:
https://forums.unrealengine.com/t/how-do-you-setup-a-rig-for-full-body-ik/253954

APPENDIX : C++ Source Code

CustomMovementComponent.h

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "CustomMovementComponent.generated.h"

DECLARE_DELEGATE(FOnEnterClimbState)
DECLARE_DELEGATE(FOnExitClimbState)

class UAnimMontage;
class AClimbingCharacter;

UENUM(BlueprintType)
namespace ECustomMovementMode
{
	enum Type
	{
		Move_Climb UMETA(DisplayName = "Climb Mode")
	};
}


/**
 *
 */
UCLASS()
class PROCANIMATIONS_API UCustomMovementComponent : public UCharacterMovementComponent
{
	GENERATED_BODY()
public:
	FOnEnterClimbState OnEnterClimbStateDelegate;
	FOnExitClimbState OnExitClimbStateDelegate;

private:

#pragma region ClimbTraces

	TArray<FHitResult> DoCapsuleTraceMultiByObject(const FVector& Start, const FVector& End, bool
bShowDebugShape = false, bool bDrawPersistantShapes = false);
	FHitResult DoLineTraceSingleByObject(const FVector& Start, const FVector& End, bool
bShowDebugShape = false, bool bDrawPersistantShapes = false);
#pragma endregion

#pragma region ClimbCore

	bool TraceClimbableSurfaces();
	FHitResult TraceFromEyeHeight(float TraceDistance, float TraceStartOffset = 0.f);
	bool CanStartClimbing();
	bool CanClimbDownLedge();
	void StartClimbing();
	void StopClimbing();
	void PhysClimb(float deltaTime, int32 Iterations);
	void ProcessClimbableSurfaceInfo();
	bool CheckShouldStopClimbing();
	bool CheckHasReachedFloor();
	FQuat GetClimbRotation(float DeltaTime);
	void SnapMovementToClimbableSurfaces(float deltaTime);
	bool CheckHasReachedLedge();
	void TryStartVaulting();
	bool CanStartVaulting(FVector& OutVaultStartPosition, FVector& OutVaultLandPosition);
	void PlayClimbMontage(UAnimMontage* MontageToPlay);
	UFUNCTION()
	void OnClimbMontageEnded(UAnimMontage *Montage, bool bInterrupted);
	void SetMotionWarpTarget(const FName& InWarpTargetName, const FVector& InTargetPosition);



#pragma endregion

#pragma region ClimbVariables

	TArray<FHitResult> ClimbableSurfacesTracedResults;
	FVector CurrentClimbableSurfaceLocation;
	FVector CurrentClimbableSurfaceNormal;

	UPROPERTY()
	class UAnimInstance* OwningPlayerAnimInstance;

	UPROPERTY()
	AClimbingCharacter* OwningPlayerCharacter;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	TArray<TEnumAsByte<EObjectTypeQuery> > ClimbableSurfaceTraceTypes;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float ClimbCapsuleTraceRadius = 50.f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float ClimbCapsuleTraceHalfHeight = 72.f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float MaxBreakClimbDeceleration = 400.f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float MaxClimbSpeed = 100.f;
	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float MaxClimbAcceleration = 300.f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	UAnimMontage* IdleToClimbMontage;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	UAnimMontage* ClimbToTopMontage;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float ClimbDownWalkableSurfaceTraceOffset = 15.f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	float ClimbDownLedgeTraceOffset = 25.f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	UAnimMontage* ClimbDownLedgeMontage;

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Character Movement: Climbing", meta=
(AllowPrivateAccess = true))
	UAnimMontage* VaultMontage;

#pragma endregion

#pragma region OverridenMethods
	protected:
		virtual void BeginPlay() override;
		virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;
		virtual void OnMovementModeChanged(EMovementMode PreviousMovementMode, uint8
PreviousCustomMode) override;
		virtual void PhysCustom(float deltaTime, int32 Iterations) override;
		virtual float GetMaxSpeed() const override;
		virtual float GetMaxAcceleration() const override;
		virtual FVector ConstrainAnimRootMotionVelocity(const FVector& RootMotionVelocity, const
FVector& CurrentVelocity) const override;
#pragma endregion


public:

	void ToggleClimbing(bool bEnableClimb);
	bool IsClimbing() const;
	FORCEINLINE FVector GetClimbableSurfaceNormal() const {return CurrentClimbableSurfaceNormal;}
	FVector GetUnrotatedClimbVelocity() const;
};
```

CustomMovementComponent.cpp

```
// Fill out your copyright notice in the Description page of Project Settings.


#include "ClimbingSystem/CustomMovementComponent.h"

#include "MotionWarpingComponent.h"
#include "Components/CapsuleComponent.h"
#include "Kismet/KismetMathLibrary.h"
#include "ClimbingSystem/ClimbingCharacter.h"
#include "Kismet/KismetSystemLibrary.h"
#include "ProcAnimations/DebugHelper.h"

void UCustomMovementComponent::BeginPlay()
{
	Super::BeginPlay();

	OwningPlayerAnimInstance = CharacterOwner->GetMesh()->GetAnimInstance();

	if(OwningPlayerAnimInstance)
	{

OwningPlayerAnimInstance->OnMontageEnded.AddDynamic(this,&UCustomMovementComponent::OnClimbMont
ageEnded);

OwningPlayerAnimInstance->OnMontageBlendingOut.AddDynamic(this,&UCustomMovementComponent::OnClim
bMontageEnded);
	}
	OwningPlayerCharacter = Cast<AClimbingCharacter>(CharacterOwner);
}

void UCustomMovementComponent::TickComponent(float DeltaTime, ELevelTick TickType,
				FActorComponentTickFunction* ThisTickFunction)
{
	Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

	CanClimbDownLedge();
}

void UCustomMovementComponent::OnMovementModeChanged(EMovementMode PreviousMovementMode,
uint8 PreviousCustomMode)
{

	if(IsClimbing())
	{
		bOrientRotationToMovement = false;
		CharacterOwner->GetCapsuleComponent()->SetCapsuleHalfHeight(48.f);

		OnEnterClimbStateDelegate.ExecuteIfBound();
	}

	if(PreviousMovementMode == MOVE_Custom && PreviousCustomMode ==
ECustomMovementMode::Move_Climb)
	{
			bOrientRotationToMovement = true;
			CharacterOwner->GetCapsuleComponent()->SetCapsuleHalfHeight(96.f);

			const FRotator DirtyRotation = UpdatedComponent->GetComponentRotation();
			const FRotator CleanStandRotation = FRotator(0.f,DirtyRotation.Yaw,0.f);
			UpdatedComponent->SetRelativeRotation(CleanStandRotation);

			StopMovementImmediately();

			OnExitClimbStateDelegate.ExecuteIfBound();

		}
		Super::OnMovementModeChanged(PreviousMovementMode, PreviousCustomMode);
}

void UCustomMovementComponent::PhysCustom(float deltaTime, int32 Iterations)
{
		Super::PhysCustom(deltaTime, Iterations);

		if(IsClimbing())
		{
			PhysClimb(deltaTime,Iterations);
		}
}

float UCustomMovementComponent::GetMaxSpeed() const
{
		if(IsClimbing())
		{
			return MaxClimbSpeed;
		}
		else
		{
			return Super::GetMaxSpeed();
		}


}

float UCustomMovementComponent::GetMaxAcceleration() const
{
		if(IsClimbing())
		{
			return MaxClimbAcceleration;
		}
		else
		{
			return Super::GetMaxAcceleration();

		}
}

FVector UCustomMovementComponent::ConstrainAnimRootMotionVelocity(const FVector& RootMotionVelocity,
		const FVector& CurrentVelocity) const
```

```
{
	const bool bIsPlayingRMMontage =
	IsFalling() && OwningPlayerAnimInstance && OwningPlayerAnimInstance->IsAnyMontagePlaying();

	if(bIsPlayingRMMontage)
		return RootMotionVelocity;
	else
	{
		return Super::ConstrainAnimRootMotionVelocity(RootMotionVelocity, CurrentVelocity);
	}
}

#pragma region ClimbTraces

	TArray<FHitResult> UCustomMovementComponent::DoCapsuleTraceMultiByObject(const FVector& Start,
const FVector& End,

									bool bShowDebugShape, bool bDrawPersistantShapes)
	{
		TArray<FHitResult> OutCapsuleTraceHitResults;
		EDrawDebugTrace::Type DebugTraceType = EDrawDebugTrace::None;

		if(bShowDebugShape)
		{
			DebugTraceType = EDrawDebugTrace::ForOneFrame;
			if(bDrawPersistantShapes)
			{
				DebugTraceType = EDrawDebugTrace::Persistent;
			}
		}
		UKismetSystemLibrary::CapsuleTraceMultiForObjects(
			this,
			Start,
			End,
			ClimbCapsuleTraceRadius,
			ClimbCapsuleTraceHalfHeight,
			ClimbableSurfaceTraceTypes,
			false,
			TArray<AActor*>(),
			DebugTraceType,
			OutCapsuleTraceHitResults,
			false
		);

		return OutCapsuleTraceHitResults;

	}

	FHitResult UCustomMovementComponent::DoLineTraceSingleByObject(const FVector& Start, const
FVector& End,
		bool bShowDebugShape, bool bDrawPersistantShapes )
	{
		FHitResult OutHit;
		EDrawDebugTrace::Type DebugTraceType = EDrawDebugTrace::None;
```

```
        if(bShowDebugShape)
        {
            DebugTraceType = EDrawDebugTrace::ForOneFrame;
            if(bDrawPersistantShapes)
            {
                DebugTraceType = EDrawDebugTrace::Persistent;
            }
        }
        UKismetSystemLibrary::LineTraceSingleForObjects(
            this,
            Start,
            End,
            ClimbableSurfaceTraceTypes,
            false,
            TArray<AActor*>(),
            DebugTraceType,
            OutHit,
            false
        );
        return OutHit;
    }
#pragma endregion

#pragma region ClimbCore

bool UCustomMovementComponent::TraceClimbableSurfaces()
{
    const FVector StartOffset = UpdatedComponent->GetForwardVector() * 30.f;
    const FVector Start = UpdatedComponent->GetComponentLocation() + StartOffset;
    const FVector End = Start + UpdatedComponent->GetForwardVector();
    ClimbableSurfacesTracedResults = DoCapsuleTraceMultiByObject(Start,End,true);

    return !ClimbableSurfacesTracedResults.IsEmpty();
}

FHitResult UCustomMovementComponent::TraceFromEyeHeight(float TraceDistance, float TraceStartOffset)
{
    const FVector ComponentLocation = UpdatedComponent->GetComponentLocation();
    const FVector EyeHeightOffset = UpdatedComponent->GetUpVector() *
(CharacterOwner->BaseEyeHeight + TraceStartOffset);
    const FVector Start = ComponentLocation + EyeHeightOffset;
    const FVector End = Start + UpdatedComponent->GetForwardVector() * TraceDistance;

    return DoLineTraceSingleByObject(Start,End,true);
}


bool UCustomMovementComponent::CanStartClimbing()
{
    if(IsFalling()) return false;
    if(!TraceClimbableSurfaces()) return false;
    if(!TraceFromEyeHeight(100.f).bBlockingHit) return false;
```

```
	return true;
}

bool UCustomMovementComponent::CanClimbDownLedge()
{
	if(IsFalling()) return false;

	const FVector ComponentLocation = UpdatedComponent->GetComponentLocation();
	const FVector ComponentForward = UpdatedComponent->GetForwardVector();
	const FVector DownVector = -UpdatedComponent->GetUpVector();

	const FVector WalkableSurfaceTraceStart =
		ComponentLocation + ComponentForward * ClimbDownWalkableSurfaceTraceOffset;
	const FVector  WalkableSurfaceTraceEnd =
		WalkableSurfaceTraceStart + DownVector * 100.f;

	FHitResult WalkableSurfaceHit = DoLineTraceSingleByObject(WalkableSurfaceTraceStart,
WalkableSurfaceTraceEnd, false);

	const FVector LedgeTraceStart = WalkableSurfaceHit.TraceStart + ComponentForward *
ClimbDownLedgeTraceOffset;
	const FVector LedgeTraceEnd = LedgeTraceStart + DownVector * 200.f;

	FHitResult LedgeTraceHit = DoLineTraceSingleByObject(LedgeTraceStart, LedgeTraceEnd, false);

	if(WalkableSurfaceHit.bBlockingHit && !LedgeTraceHit.bBlockingHit)
		return true;

	return false;
}

void UCustomMovementComponent::StartClimbing()
{
	SetMovementMode(MOVE_Custom,ECustomMovementMode::Move_Climb);
}

void UCustomMovementComponent::StopClimbing()
{
	SetMovementMode(MOVE_Falling);
}

void UCustomMovementComponent::PhysClimb(float deltaTime, int32 Iterations)
{
	if (deltaTime < MIN_TICK_TIME)
	{
		return;
	}

	//Process all the climbable surfaces info
	TraceClimbableSurfaces();
	ProcessClimbableSurfaceInfo();
```

```
        //check if we should start climbiung
        if(CheckShouldStopClimbing() || CheckHasReachedFloor())
        {
                StopClimbing();
        }
        RestorePreAdditiveRootMotionVelocity();

        if( !HasAnimRootMotion() && !CurrentRootMotion.HasOverrideVelocity() )
        {
                //define the max climb speed and acceleration
                CalcVelocity(deltaTime, 0.f, true, MaxBreakClimbDeceleration);
        }

        ApplyRootMotionToVelocity(deltaTime);


        FVector OldLocation = UpdatedComponent->GetComponentLocation();
        const FVector Adjusted = Velocity * deltaTime;
        FHitResult Hit(1.f);

        //handle climb rotation
        SafeMoveUpdatedComponent(Adjusted, GetClimbRotation(deltaTime), true, Hit);

        if (Hit.Time < 1.f)
        {
                //adjust and try again
                HandleImpact(Hit, deltaTime, Adjusted);
                SlideAlongSurface(Adjusted, (1.f - Hit.Time), Hit.Normal, Hit, true);
        }

        if( HasAnimRootMotion() && !CurrentRootMotion.HasOverrideVelocity() )
        {
                Velocity = (UpdatedComponent->GetComponentLocation() - OldLocation) / deltaTime;
        }

        //snap movement to climbable surfaces
        SnapMovementToClimbableSurfaces(deltaTime);
        if(CheckHasReachedLedge())
        {
                PlayClimbMontage(ClimbToTopMontage);
        }

}

void UCustomMovementComponent::ProcessClimbableSurfaceInfo()
{
        CurrentClimbableSurfaceLocation = FVector::ZeroVector;
        CurrentClimbableSurfaceNormal = FVector::ZeroVector;

        if(ClimbableSurfacesTracedResults.IsEmpty()) return;

        for(const FHitResult& TracedHitResult:ClimbableSurfacesTracedResults)
        {
                CurrentClimbableSurfaceLocation += TracedHitResult.ImpactPoint;
                CurrentClimbableSurfaceNormal += TracedHitResult.ImpactNormal;
        }

        CurrentClimbableSurfaceLocation /= ClimbableSurfacesTracedResults.Num();
        CurrentClimbableSurfaceNormal = CurrentClimbableSurfaceNormal.GetSafeNormal();

}

bool UCustomMovementComponent::CheckShouldStopClimbing()
{
        if(ClimbableSurfacesTracedResults.IsEmpty()) return true;

        const float DotResult = FVector::DotProduct(CurrentClimbableSurfaceNormal, FVector::UpVector);
        const float DegreeDiff = FMath::RadiansToDegrees(FMath::Acos(DotResult));

        if(DegreeDiff <= 60.f)
                return true;

        return false;
}

bool UCustomMovementComponent::CheckHasReachedFloor()
{
        const FVector DownVector = -UpdatedComponent->GetUpVector();
        const FVector StartOffset = DownVector * 50.f;

        const FVector Start = UpdatedComponent->GetComponentLocation() + StartOffset;
        const FVector End = Start + DownVector;

        TArray<FHitResult> PossibleFloorHits = DoCapsuleTraceMultiByObject(Start, End, false);

        if(PossibleFloorHits.IsEmpty()) return false;

        for(const FHitResult& PossibleFloorHit : PossibleFloorHits)
        {
                const bool bFloorReached =
                FVector::Parallel(-PossibleFloorHit.ImpactNormal, FVector::UpVector) &&
                        GetUnrotatedClimbVelocity().Z < -10.f;

                if(bFloorReached)
                        return true;
        }

        return false;
}

FQuat UCustomMovementComponent::GetClimbRotation(float DeltaTime)
{
         const FQuat CurrentQuat = UpdatedComponent->GetComponentQuat();

        if(HasAnimRootMotion() || CurrentRootMotion.HasOverrideVelocity())
        {
                return CurrentQuat;
```

```
	}

	const FQuat TargetQuat = FRotationMatrix::MakeFromX(-CurrentClimbableSurfaceNormal).ToQuat();
	return FMath::QInterpTo(CurrentQuat, TargetQuat, DeltaTime, 5.f);

}

void UCustomMovementComponent::SnapMovementToClimbableSurfaces(float deltaTime)
{
	const FVector ComponentForward = UpdatedComponent->GetForwardVector();
	const FVector ComponentLocation = UpdatedComponent->GetComponentLocation();

	const FVector ProjectedCharacterToSurface =
		(CurrentClimbableSurfaceLocation-ComponentLocation).ProjectOnTo(ComponentForward);

	const FVector SnapVector = -CurrentClimbableSurfaceNormal * ProjectedCharacterToSurface.Length();


UpdatedComponent->MoveComponent(SnapVector*deltaTime*MaxClimbSpeed,UpdatedComponent->GetCompo
nentQuat(),true);
}

bool UCustomMovementComponent::CheckHasReachedLedge()
{
	FHitResult LedgeHitResult = TraceFromEyeHeight(100.f,50.f);

	if(!LedgeHitResult.bBlockingHit)
	{
		const FVector WalkableSurfaceTraceStart = LedgeHitResult.TraceEnd;
		const FVector DownVector = -UpdatedComponent->GetUpVector();
		const FVector WalkableSurfaceTraceEnd = WalkableSurfaceTraceStart + DownVector * 100.f;

		FHitResult WalkableSurfaceHitResult =
		DoLineTraceSingleByObject(WalkableSurfaceTraceStart,WalkableSurfaceTraceEnd, true);

		if(WalkableSurfaceHitResult.bBlockingHit && GetUnrotatedClimbVelocity().Z > 10.f)
			return true;

	}

	return false;
}

void UCustomMovementComponent::TryStartVaulting()
{
	FVector VaultStartPosition;
	FVector VaultLandPosition;
	if(CanStartVaulting(VaultStartPosition,VaultLandPosition))
	{
		SetMotionWarpTarget(FName("VaultStartPoint"), VaultStartPosition);
		SetMotionWarpTarget(FName("VaultEndPoint"), VaultLandPosition);

		StartClimbing();
		PlayClimbMontage(VaultMontage);
	}

}

bool UCustomMovementComponent::CanStartVaulting(FVector& OutVaultStartPosition, FVector&
OutVaultLandPosition)
{
	if(IsFalling()) return false;

	OutVaultStartPosition = FVector::ZeroVector;
	OutVaultLandPosition = FVector::ZeroVector;
	const FVector ComponentLocation = UpdatedComponent->GetComponentLocation();
	const FVector ComponentForward = UpdatedComponent->GetForwardVector();
	const FVector UpVector = UpdatedComponent->GetUpVector();
	const FVector DownVector = -UpdatedComponent->GetUpVector();

	for (int i = 0; i< 5; i++)
	{
		const FVector Start = ComponentLocation + UpVector*100.f + ComponentForward * 100.f * (i+1);

		const FVector End = Start + DownVector * 100.f * (i+1);

		FHitResult VaultTraceResult = DoLineTraceSingleByObject(Start,End);

		if(i == 0 && VaultTraceResult.bBlockingHit)
		{
			OutVaultStartPosition = VaultTraceResult.ImpactPoint;
		}
		if(i == 3 && VaultTraceResult.bBlockingHit)
		{
			OutVaultLandPosition = VaultTraceResult.ImpactPoint;
		}
	}

	if(OutVaultStartPosition != FVector::ZeroVector && OutVaultLandPosition != FVector::ZeroVector)
	{
		return true;
	}
	else
	{
		return false;
	}

}

void UCustomMovementComponent::PlayClimbMontage(UAnimMontage* MontageToPlay)
{
	if(!MontageToPlay) return;
	if(!OwningPlayerAnimInstance) return;
	if(OwningPlayerAnimInstance->IsAnyMontagePlaying()) return;

	OwningPlayerAnimInstance->Montage_Play(MontageToPlay);
}
```

```
void UCustomMovementComponent::OnClimbMontageEnded(UAnimMontage* Montage, bool bInterrupted)
{
	if(Montage == IdleToClimbMontage || Montage==ClimbDownLedgeMontage)
	{
		StartClimbing();
		StopMovementImmediately();
	}

	if(Montage == ClimbToTopMontage || Montage == VaultMontage)
	{
		SetMovementMode(MOVE_Walking);
	}
}

void UCustomMovementComponent::SetMotionWarpTarget(const FName& InWarpTargetName, const FVector&
InTargetPosition)
{
	if(!OwningPlayerCharacter) return;

	OwningPlayerCharacter->GetMotionWarpingComponent()->AddOrUpdateWarpTargetFromLocation(
	InWarpTargetName,
	InTargetPosition
	);
}


void UCustomMovementComponent::ToggleClimbing(bool bEnableClimb)
{
	if(bEnableClimb)
	{
		if(CanStartClimbing())
		{
			PlayClimbMontage(IdleToClimbMontage);
		}
		else if(CanClimbDownLedge())
		{

			PlayClimbMontage(ClimbDownLedgeMontage);
		}
		else
		{
			TryStartVaulting();
		}

	}
	if(!bEnableClimb)
	{
		StopClimbing();
	}
}

bool UCustomMovementComponent::IsClimbing() const
{
	return MovementMode == MOVE_Custom && CustomMovementMode ==
ECustomMovementMode::Move_Climb;
}

FVector UCustomMovementComponent::GetUnrotatedClimbVelocity() const
{

	return UKismetMathLibrary::Quat_UnrotateVector(UpdatedComponent->GetComponentQuat(),Velocity);
}


#pragma endregion
```

CharacterAnimInstance.h

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "Animation/AnimInstance.h"
#include "CharacterAnimInstance.generated.h"

class AClimbingCharacter;
class UCustomMovementComponent;
/**
 *
 */
UCLASS()
class PROCANIMATIONS_API UCharacterAnimInstance : public UAnimInstance
{
	GENERATED_BODY()

public:
	virtual void NativeInitializeAnimation() override;
	virtual void NativeUpdateAnimation(float DeltaSeconds) override;

private:
	UPROPERTY()
	AClimbingCharacter* TraversalMechCharacter;

	UPROPERTY()
	UCustomMovementComponent* CustomMovementComponent;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Reference", meta= (AllowPrivateAccess =
true))
	float GroundSpeed;
	void GetGroundSpeed();

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Reference", meta= (AllowPrivateAccess =
true))
	float AirSpeed;
	void GetAirSpeed();
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Reference", meta= (AllowPrivateAccess =
true))
	bool bShouldMove;
	void GetShouldMove();

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Reference", meta= (AllowPrivateAccess =
true))
	bool bIsFalling;
	void GetIsFalling();

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Reference", meta= (AllowPrivateAccess =
true))
	bool bIsClimbing;
	void GetIsClimbing();

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Reference", meta= (AllowPrivateAccess =
true))
	FVector ClimbVelocity;
	void GetClimbVelocity();
};
```

CharacterAnimInstance.cpp

```
// Fill out your copyright notice in the Description page of Project Settings.


#include "ClimbingSystem/CharacterAnimInstance.h"
#include "ClimbingSystem/CustomMovementComponent.h"
#include "Kismet/KismetMathLibrary.h"
#include "ClimbingSystem/ClimbingCharacter.h"

void UCharacterAnimInstance::NativeInitializeAnimation()
{
	Super::NativeInitializeAnimation();

	TraversalMechCharacter = Cast<AClimbingCharacter>(TryGetPawnOwner());
	if(TraversalMechCharacter)
	{
		CustomMovementComponent = TraversalMechCharacter->GetCustomMovementComponent();
	}
}

void UCharacterAnimInstance::NativeUpdateAnimation(float DeltaSeconds)
{
	Super::NativeUpdateAnimation(DeltaSeconds);

	if(!TraversalMechCharacter || !CustomMovementComponent) return;
	GetGroundSpeed();
	GetAirSpeed();
	GetShouldMove();
	GetIsFalling();
	GetIsClimbing();
	GetClimbVelocity();
}

void UCharacterAnimInstance::GetGroundSpeed()
{
	GroundSpeed = UKismetMathLibrary::VSizeXY(TraversalMechCharacter->GetVelocity());
}

void UCharacterAnimInstance::GetAirSpeed()
{
	AirSpeed = TraversalMechCharacter->GetVelocity().Z;
}

void UCharacterAnimInstance::GetShouldMove()
{
	bShouldMove = CustomMovementComponent->GetCurrentAcceleration().Size() > 0 && GroundSpeed >
0.5f && !bIsFalling;
}

void UCharacterAnimInstance::GetIsFalling()
{
	bIsFalling = CustomMovementComponent->IsFalling();
}

void UCharacterAnimInstance::GetIsClimbing()
{
	bIsClimbing = CustomMovementComponent->IsClimbing();
}

void UCharacterAnimInstance::GetClimbVelocity()
{
	ClimbVelocity = CustomMovementComponent->GetUnrotatedClimbVelocity();
}
```

<u>ClimbingCharacter.h</u>

```
// Copyright Epic Games, Inc. All Rights Reserved.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "InputActionValue.h"
#include "ClimbingCharacter.generated.h"

class UCustomMovementComponent;
class UMotionWarpingComponent;

UCLASS(config=Game)
class AClimbingCharacter : public ACharacter
{
	GENERATED_BODY()
```

```
public:
	AClimbingCharacter(const FObjectInitializer& ObjectInitializer);

	UPROPERTY(BlueprintReadWrite, VisibleAnywhere)
	bool bCanPickTorch;

private:


	/** Default TP Camera */
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera, meta = (AllowPrivateAccess =
"true"))
	class USpringArmComponent* DTPCameraBoom;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Camera, meta = (AllowPrivateAccess =
"true"))
	class UCameraComponent* DTPFollowCamera;


	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess =
"true"))
	UCustomMovementComponent* CustomMovementComponent;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Movement, meta = (AllowPrivateAccess =
"true"))
	UMotionWarpingComponent* MotionWarpingComponent;

	void OnPlayerEnterClimbState();
	void OnPlayerExitClimbState();

	/** MappingContext */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputMappingContext* DefaultMappingContext;

	/** Jump Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputAction* JumpAction;

	/** Move Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputAction* MoveAction;

	/** Called for movement input */
	void Move(const FInputActionValue& Value);

	/** Look Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputAction* LookAction;

	/** Called for looking input */
	void Look(const FInputActionValue& Value);

	/** Climb Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputAction* ClimbAction;
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputAction* TraversalAction;

	/** Called for climbing input */
	void OnClimbActionStarted(const FInputActionValue& Value);

	void HandleGroundMovementInput(const FInputActionValue& Value);
	void HandleClimbMovementInput(const FInputActionValue& Value);

	/** Run Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input, meta = (AllowPrivateAccess = "true"))
	class UInputAction* RunAction;

	/** Called for movement input */
	void Run(const FInputActionValue& Value);


	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Input, meta = (AllowPrivateAccess = "true"))
	float WalkSpeed = 500.f;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Input, meta = (AllowPrivateAccess = "true"))
	float RunSpeed = 700.f;

protected:


	// APawn interface
	virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;

	// To add mapping context
	virtual void BeginPlay();

public:

	/** Returns CameraBoom subobject **/
	FORCEINLINE class USpringArmComponent* GetCameraBoom() const { return DTPCameraBoom; }
	/** Returns FollowCamera subobject **/
	FORCEINLINE class UCameraComponent* GetFollowCamera() const { return DTPFollowCamera; }

	FORCEINLINE class UCustomMovementComponent* GetCustomMovementComponent() const { return
CustomMovementComponent; }

	FORCEINLINE UMotionWarpingComponent* GetMotionWarpingComponent() const {return
MotionWarpingComponent;}
};
```

<u>ClimbingCharacter.cpp</u>

```
// Copyright Epic Games, Inc. All Rights Reserved.


#include "ClimbingSystem/ClimbingCharacter.h"
```

```
#include "ClimbingSystem/ClimbingCharacter.h"

#include "ProcAnimations/DebugHelper.h"
#include "Camera/CameraComponent.h"
#include "Components/CapsuleComponent.h"
#include "Components/InputComponent.h"
#include "ClimbingSystem/CustomMovementComponent.h"
#include "GameFramework/Controller.h"
#include "GameFramework/SpringArmComponent.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "MotionWarpingComponent.h"



AClimbingCharacter::AClimbingCharacter(const FObjectInitializer& ObjectInitializer)
	:
Super(ObjectInitializer.SetDefaultSubobjectClass<UCustomMovementComponent>(ACharacter::CharacterMoveme
ntComponentName))
{
	// Set size for collision capsule
	GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);

	// Don't rotate when the controller rotates. Let that just affect the camera.
	bUseControllerRotationPitch = false;
	bUseControllerRotationYaw = false;
	bUseControllerRotationRoll = false;

	CustomMovementComponent = Cast<UCustomMovementComponent>(GetCharacterMovement());

	// Configure character movement
	GetCharacterMovement()->bOrientRotationToMovement = true; // Character moves in the direction of
input...
	GetCharacterMovement()->RotationRate = FRotator(0.0f, 500.0f, 0.0f); // ...at this rotation rate

	// Note: For faster iteration times these variables, and many more, can be tweaked in the Character
Blueprint
	// instead of recompiling to adjust them
	GetCharacterMovement()->JumpZVelocity = 700.f;
	GetCharacterMovement()->AirControl = 0.35f;

	GetCharacterMovement()->MinAnalogWalkSpeed = 20.f;
	GetCharacterMovement()->BrakingDecelerationWalking = 2000.f;

	// Create a camera boom (pulls in towards the player if there is a collision)
	DTPCameraBoom =
CreateDefaultSubobject<USpringArmComponent>(TEXT("DefaultThirdPersonCameraBoom"));
	DTPCameraBoom->SetupAttachment(RootComponent);
	DTPCameraBoom->TargetArmLength = 400.0f; // The camera follows at this distance behind the character
	DTPCameraBoom->bUsePawnControlRotation = true; // Rotate the arm based on the controller
	DTPFollowCamera =
CreateDefaultSubobject<UCameraComponent>(TEXT("DefaultThirdPersonCamera"));
	DTPFollowCamera->SetupAttachment(DTPCameraBoom, USpringArmComponent::SocketName); // Attach
the camera to the end of the boom and let the boom adjust to match the controller orientation
```

```
	DTPFollowCamera->bUsePawnControlRotation = false; // Camera does not rotate relative to arm

	MotionWarpingComponent =
CreateDefaultSubobject<UMotionWarpingComponent>("MotionWarpingComp");
}

void AClimbingCharacter::BeginPlay()
{
	// Call the base class
	Super::BeginPlay();

	GetCharacterMovement()->MaxWalkSpeed = WalkSpeed;
	GetCustomMovementComponent()->MaxWalkSpeed = WalkSpeed;

	//Add Input Mapping Context
	if (APlayerController* PlayerController = Cast<APlayerController>(Controller))
	{
		if (UEnhancedInputLocalPlayerSubsystem* Subsystem =
ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer()))
		{
			Subsystem->AddMappingContext(DefaultMappingContext, 0);
		}
	}

	if(CustomMovementComponent)
	{
		CustomMovementComponent->OnEnterClimbStateDelegate.BindUObject(this,
&ThisClass::OnPlayerEnterClimbState);
		CustomMovementComponent->OnExitClimbStateDelegate.BindUObject(this,
&ThisClass::OnPlayerExitClimbState);


	}
}

//////////////////////////////////////////////////////////////////////////
// Input




void AClimbingCharacter::SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent)
{
	// Set up action bindings
	if (UEnhancedInputComponent* EnhancedInputComponent =
CastChecked<UEnhancedInputComponent>(PlayerInputComponent)) {

		//Jumping
		EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Triggered, this,
&ACharacter::Jump);
		EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Completed, this,
&ACharacter::StopJumping);

		//Moving
```

```
			EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this,
&AClimbingCharacter::Move);

			EnhancedInputComponent->BindAction(RunAction, ETriggerEvent::Started, this,
&AClimbingCharacter::Run);
			EnhancedInputComponent->BindAction(RunAction, ETriggerEvent::Completed, this,
&AClimbingCharacter::Run);

			//Looking
			EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Triggered, this,
&AClimbingCharacter::Look);


			EnhancedInputComponent->BindAction(ClimbAction, ETriggerEvent::Started, this,
&AClimbingCharacter::OnClimbActionStarted);

	}

}

void AClimbingCharacter::OnPlayerEnterClimbState()
{

}

void AClimbingCharacter::OnPlayerExitClimbState()
{

}

void AClimbingCharacter::Move(const FInputActionValue& Value)
{
	if(!CustomMovementComponent) return;
	if(CustomMovementComponent->IsClimbing())
	{
		HandleClimbMovementInput(Value);
	}
	else
	{
		HandleGroundMovementInput(Value);
	}

}

void AClimbingCharacter::Run(const FInputActionValue& Value)
{
	if(!CustomMovementComponent) return;

	if(Value.GetMagnitude() > 0.f)
		GetCustomMovementComponent()->MaxWalkSpeed = RunSpeed;
	else
		GetCustomMovementComponent()->MaxWalkSpeed = WalkSpeed;
}
```

```
void AClimbingCharacter::Look(const FInputActionValue& Value)
{
	// input is a Vector2D
	FVector2D LookAxisVector = Value.Get<FVector2D>();

	if (Controller != nullptr)
	{
		// add yaw and pitch input to controller
		AddControllerYawInput(LookAxisVector.X);
		AddControllerPitchInput(LookAxisVector.Y);
	}
}

void AClimbingCharacter::OnClimbActionStarted(const FInputActionValue& Value)
{

	if(!CustomMovementComponent)return;

	if(!CustomMovementComponent->IsClimbing())
	{

		CustomMovementComponent->ToggleClimbing(true);
	}
	else
	{
		CustomMovementComponent->ToggleClimbing(false);
	}
}

void AClimbingCharacter::HandleGroundMovementInput(const FInputActionValue& Value)
{
	// input is a Vector2D
	const FVector2D MovementVector = Value.Get<FVector2D>();

	if (Controller != nullptr)
	{
		// find out which way is forward
		const FRotator Rotation = Controller->GetControlRotation();
		const FRotator YawRotation(0, Rotation.Yaw, 0);

		// get forward vector
		const FVector ForwardDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);

		// get right vector
		const FVector RightDirection = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);

		// add movement
		AddMovementInput(ForwardDirection, MovementVector.Y);
		AddMovementInput(RightDirection, MovementVector.X);
	}
}

void AClimbingCharacter::HandleClimbMovementInput(const FInputActionValue& Value)
{
    // input is a Vector2D
    const FVector2D MovementVector = Value.Get<FVector2D>();

    // get right vector
    const FVector ForwardDirection = FVector::CrossProduct(
        -CustomMovementComponent->GetClimbableSurfaceNormal(),
        GetActorRightVector()
    );

    // get forward vector
    const FVector RightDirection = FVector::CrossProduct(
    -CustomMovementComponent->GetClimbableSurfaceNormal(),
    -GetActorUpVector()
        );

    // add movement
    AddMovementInput(ForwardDirection, MovementVector.Y);
    AddMovementInput(RightDirection, MovementVector.X);
}
```