Whiskey Buisness/ Ragdoll

Architecture/Design Document

Table of Contents

Contents

Change History

Version: 0.1

Modifier: Gabriel Hernandez

Date: 28 / 02 / 2025

Description of Change: Module Design Document started.

1Introduction

This document delineates the architectural framework and design specifics of the ragdoll physics mechanics integrated within the AQuetzalMultiplayerCharacter class. This class is pivotal not only for managing player inputs and conventional movement but also for orchestrating complex ragdoll physics.

The ragdoll system is designed to respond to various gameplay dynamics, such as combat impacts and environmental hazards, by simulating a lifeless body that reacts according to the laws of physics.

For developers, the documentation outlines the technical implementation and integration points, enabling them to understand and contribute to the system efficiently.

Project managers will benefit from insights into the system's modularity and performance considerations, facilitating effective task assignments and resource planning.

Maintenance programmers are provided with the necessary information to ensure the system remains robust, efficient, and easy to extend or modify in response to future requirements.

2 Design Goals

The design of the ragdoll mechanics class is driven by several key objectives aimed at enhancing gameplay and ensuring technical robustness:

- **Realism and Immersion**: The primary goal is to provide a highly realistic response of the character’s body to impacts and forces within the game environment, contributing to a deeper sense of immersion for players. The implementation must convincingly handle the transition from controlled character movement to passive ragdoll states triggered by gameplay events such as explosions, falls, or being knocked out.

- **Modularity and Reusability**: The ragdoll system is designed to be modular, allowing for easy integration and reuse across various character classes and types within the game.
- **Ease of Maintenance and Extension**: The system is structured to be easily maintainable and extensible by other developers.

3 System Behavior

The ragdoll system within the AQuetzalMultiplayerCharacter class is intricately designed to simulate realistic physical behavior of the character's body in response to various in-game stimuli. This section outlines the functional behavior of the ragdoll mechanics from activation to deactivation, detailing the transitions and interactions at play.

Activation:

The ragdoll effect is triggered under specific circumstances such as severe impacts from enemy attacks, environmental hazards, or fatal falls. The transition from an animated state to ragdoll physics involves several steps:

- **Detection of Impact**: The system first detects a significant impact or health threshold breach.
- **Disabling Animations**: Upon triggering, animated control over the character’s body is disabled, and the skeletal mesh's animation blueprint transitions to a passive state.
- **Enabling Physics Simulation**: The character's skeletal mesh then enables physics simulation.

  Physics Simulation:

- **Collision Handling**: As the character hits the ground or other objects, collision responses are calculated to produce natural body movements and interactions based on the materials and surfaces involved.
- **Continuous Update**: The physics simulation continuously updates the position and orientation of each bone in the skeletal mesh

  Recovery:

- **Condition Check**: The system periodically checks whether conditions for recovery are met, such as the absence of ongoing harmful impacts or player input signaling an attempt to stand up.

- **Deactivation of Physics**: Once recovery conditions are satisfied, physics simulation on the skeletal mesh is disabled, and control is handed back to the game's animation system.
- **Animation Blending**: The character transitions out of the ragdoll state through a blending process that smoothly integrates pre-defined "get up" animations. These animations are chosen based on the character's final ragdoll position (e.g., lying face up or face down).

System Integration:

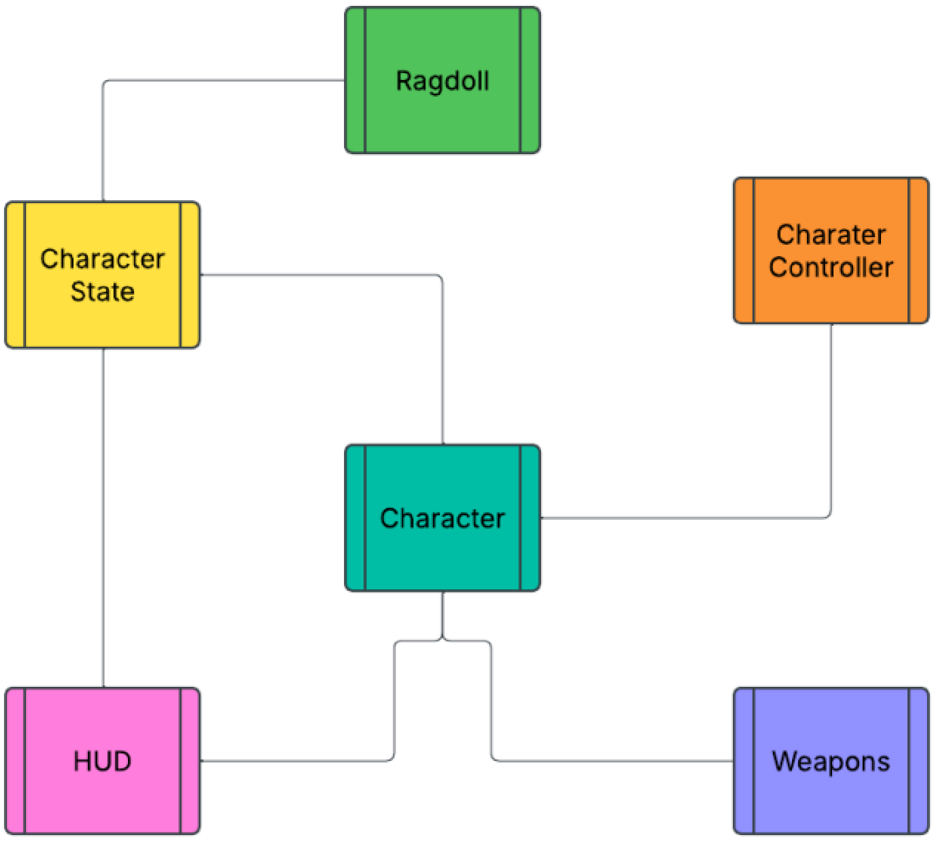
- **Feedback Loops**: Throughout the ragdoll state, the system provides feedback to other game systems, such as the health management and player control systems, ensuring that gameplay mechanics like damage calculation and movement control are accurately maintained in accordance with the character's state.
- **Event Handling**: The system is capable of handling multiple triggers and transitions between different states, allowing for complex sequences of interactions, such as being knocked down repeatedly or transitioning between different types of impacts.

Error Handling and Constraints:

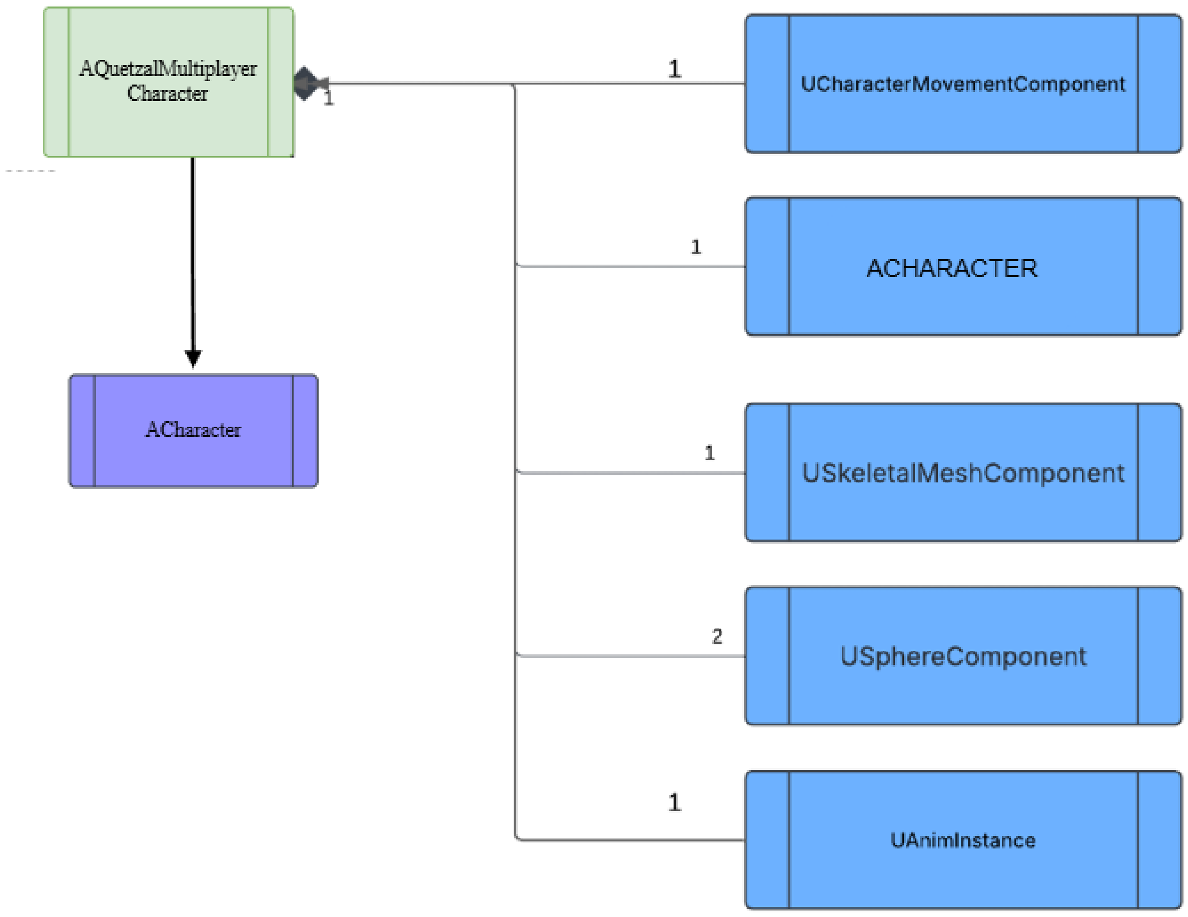
- **Stability Measures**: To maintain stability and performance, the system implements constraints on joint movements and collision responses, preventing unnatural bending or breaking of the character model.
- **Error Recovery**: In cases where the simulation encounters errors, such as unexpected changes or loss of collision data, the system has mechanisms to reset the character's position or adjust parameters to regain stability.

4 Logical View

4.1 High-Level Design


Ragdoll
Character State
Charater Controller
Character
HUD
Weapons

4.2 Mid-Level Design of the Ragdoll



High-Level Design of varying systems in the game.

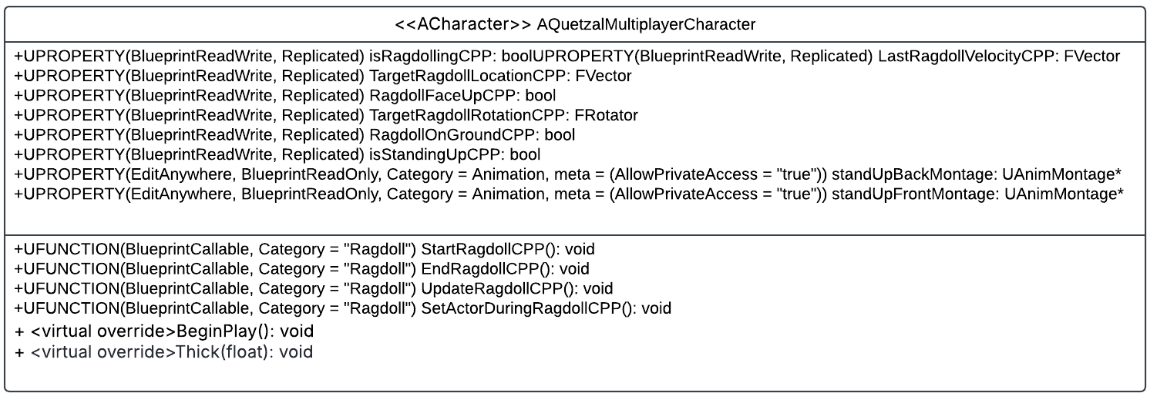
The Character Controller will handle the input from the user that will control the player. The HUD will keep the players informed of the current health and score of the players using values from Character and Character State.

Weapons will interact with the player by activating equipped weapons or receiving damage from weapons.

Depending on the Character State the player can enter a Ragdoll state.

At the center is the Character interacting with all systems in some varying ways.

4.3 Detailed Class Design of the Ragdoll



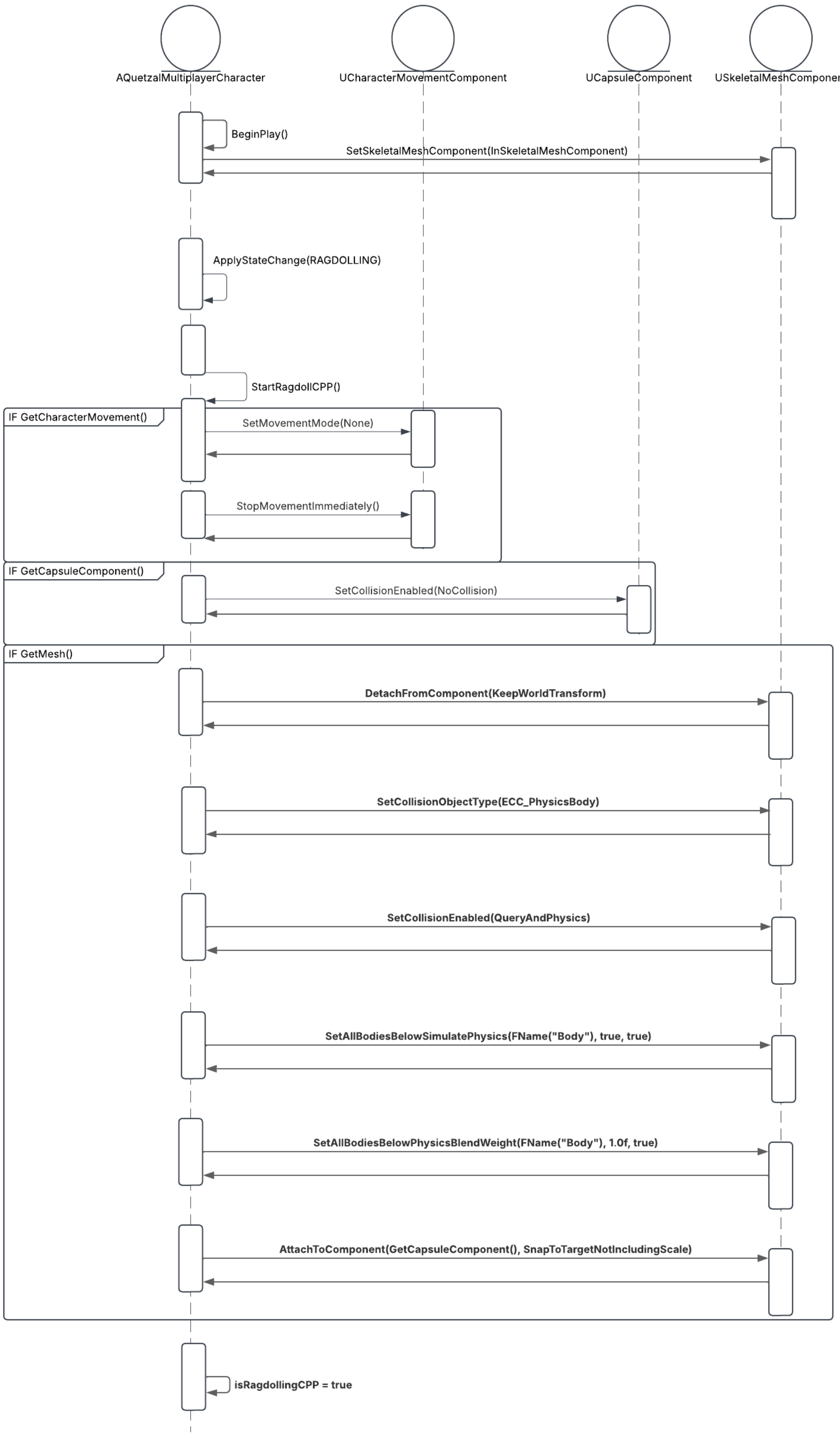
```
<<ACharacter>> AQuetzalMultiplayerCharacter
+UPROPERTY(BlueprintReadWrite, Replicated) isRagdollingCPP: boolUPROPERTY(BlueprintReadWrite, Replicated) LastRagdollVelocityCPP: FVector
+UPROPERTY(BlueprintReadWrite, Replicated) TargetRagdollLocationCPP: FVector
+UPROPERTY(BlueprintReadWrite, Replicated) RagdollFaceUpCPP: bool
+UPROPERTY(BlueprintReadWrite, Replicated) TargetRagdollRotationCPP: FRotator
+UPROPERTY(BlueprintReadWrite, Replicated) RagdollOnGroundCPP: bool
+UPROPERTY(BlueprintReadWrite, Replicated) isStandingUpCPP: bool
+UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Animation, meta = (AllowPrivateAccess = "true")) standUpBackMontage: UAnimMontage*
+UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Animation, meta = (AllowPrivateAccess = "true")) standUpFrontMontage: UAnimMontage*

+UFUNCTION(BlueprintCallable, Category = "Ragdoll") StartRagdollCPP(): void
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") EndRagdollCPP(): void
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") UpdateRagdollCPP(): void
+UFUNCTION(BlueprintCallable, Category = "Ragdoll") SetActorDuringRagdollCPP(): void
+ <virtual override>BeginPlay(): void
+ <virtual override>Thick(float): void
```

5 Process View of the Ragdoll

StartRagdoll

Ragdoll Start

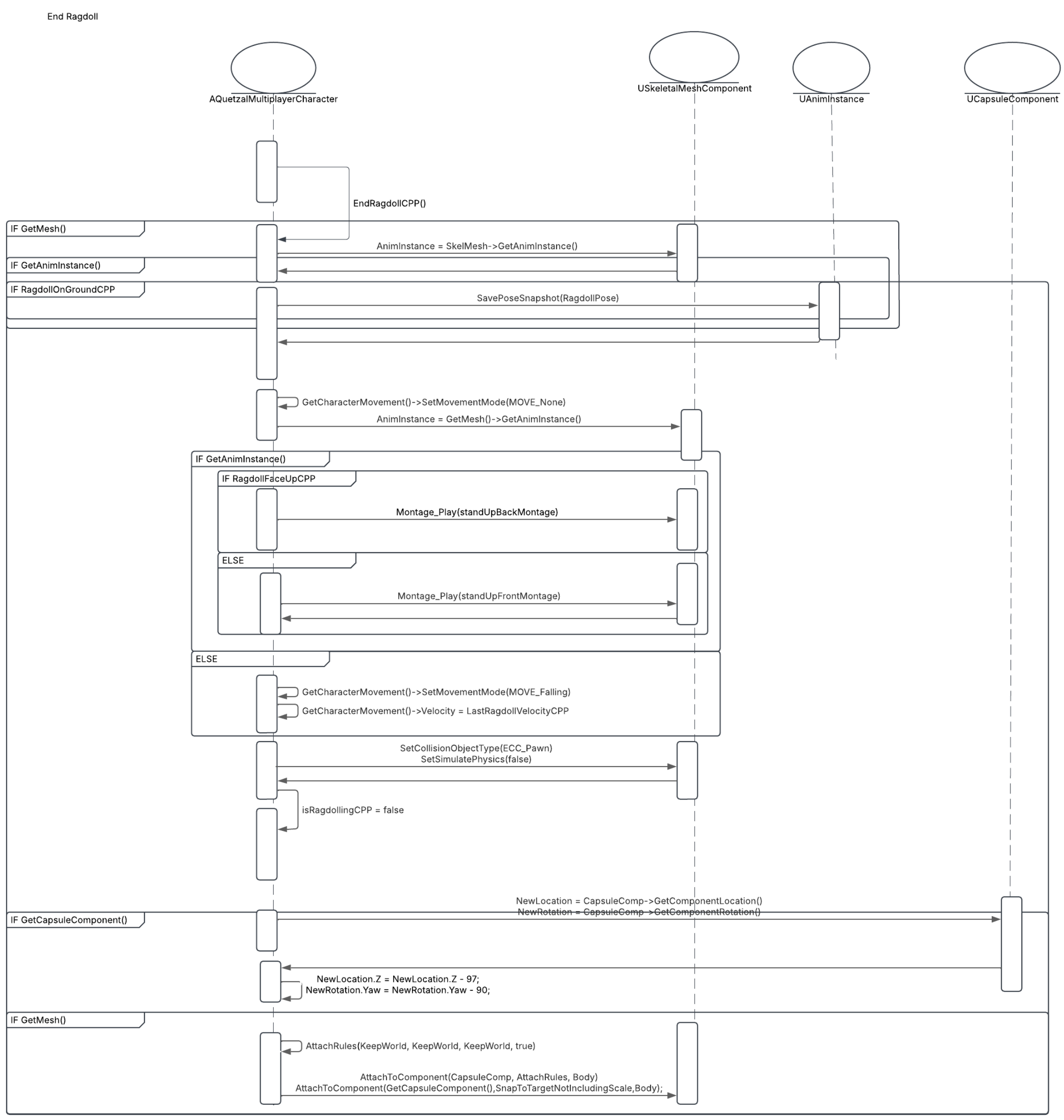

AQuetzalMultiplayerCharacter
UCharacterMovementComponent
UCapsuleComponent
USkeletalMeshComponent
BeginPlay()
SetSkeletalMeshComponent(InSkeletalMeshComponent)
ApplyStateChange(RAGDOLLING)
StartRagdollCPP()
IF GetCharacterMovement()
SetMovementMode(None)
StopMovementImmediately()
IF GetCapsuleComponent()
SetCollisionEnabled(NoCollision)
IF GetMesh()
DetachFromComponent(KeepWorldTransform)
SetCollisionObjectType(ECC_PhysicsBody)
SetCollisionEnabled(QueryAndPhysics)
SetAllBodiesBelowSimulatePhysics(FName("Body"), true, true)
SetAllBodiesBelowPhysicsBlendWeight(FName("Body"), 1.0f, true)
AttachToComponent(GetCapsuleComponent(), SnapToTargetNotIncludingScale)
isRagdollingCPP = true

The StartRagdollCPP() method transitions from a controlled state to a ragdoll state where physics take over, simulating realistic responses to impacts or falls.

Process Steps:

1. **Disable Character Movement**:
    - Movement is halted to prevent influence from ongoing player inputs, with the movement mode set to MOVE_None.
2. **Disable Capsule Collision**:
    - The capsule collider is disabled to avoid interference with the ragdoll physics, allowing the skeletal mesh to react freely to environmental collisions.
3. **Configure Skeletal Mesh for Physics**:
    - The skeletal mesh is detached and prepared for physics interactions:
        - Set as a ECC_PhysicsBody to participate in physics simulations.
        - Physics and collision are enabled to interact with the game world.
        - Physics simulation is activated from a specific bone downwards, fully controlling the mesh's movement by physics.
4. **Reattach Skeletal Mesh**:
    - After setup, the skeletal mesh is reattached to the capsule component to maintain its association with the character's main component.
5. **Update State Variable**:
    - The state variable isRagdollingCPP is set to true, marking the character's transition into the ragdoll state.

EndRagdoll


End Ragdoll
AQuetzalMultiplayerCharacter
USkeletalMeshComponent
UAnimInstance
UCapsuleComponent
EndRagdollCPP()
IF GetMesh()
AnimInstance = SkelMesh->GetAnimInstance()
IF GetAnimInstance()
IF RagdollOnGroundCPP
SavePoseSnapshot(RagdollPose)
GetCharacterMovement()->SetMovementMode(MOVE_None)
AnimInstance = GetMesh()->GetAnimInstance()
IF GetAnimInstance()
IF RagdollFaceUpCPP
Montage_Play(standUpBackMontage)
ELSE
Montage_Play(standUpFrontMontage)
ELSE
GetCharacterMovement()->SetMovementMode(MOVE_Falling)
GetCharacterMovement()->Velocity = LastRagdollVelocityCPP
SetCollisionObjectType(ECC_Pawn)
SetSimulatePhysics(false)
isRagdollingCPP = false
NewLocation = CapsuleComp->GetComponentLocation()
NewRotation = CapsuleComp->GetComponentRotation()
IF GetCapsuleComponent()
NewLocation.Z = NewLocation.Z - 97;
NewRotation.Yaw = NewRotation.Yaw - 90;
IF GetMesh()
AttachRules(KeepWorld, KeepWorld, KeepWorld, true)
AttachToComponent(CapsuleComp, AttachRules, Body)
AttachToComponent(GetCapsuleComponent(),SnapToTargetNotIncludingScale,Body);

The EndRagdollCPP() method is responsible for concluding the ragdoll state of the character, transitioning back to a controlled state with animation-driven movements.

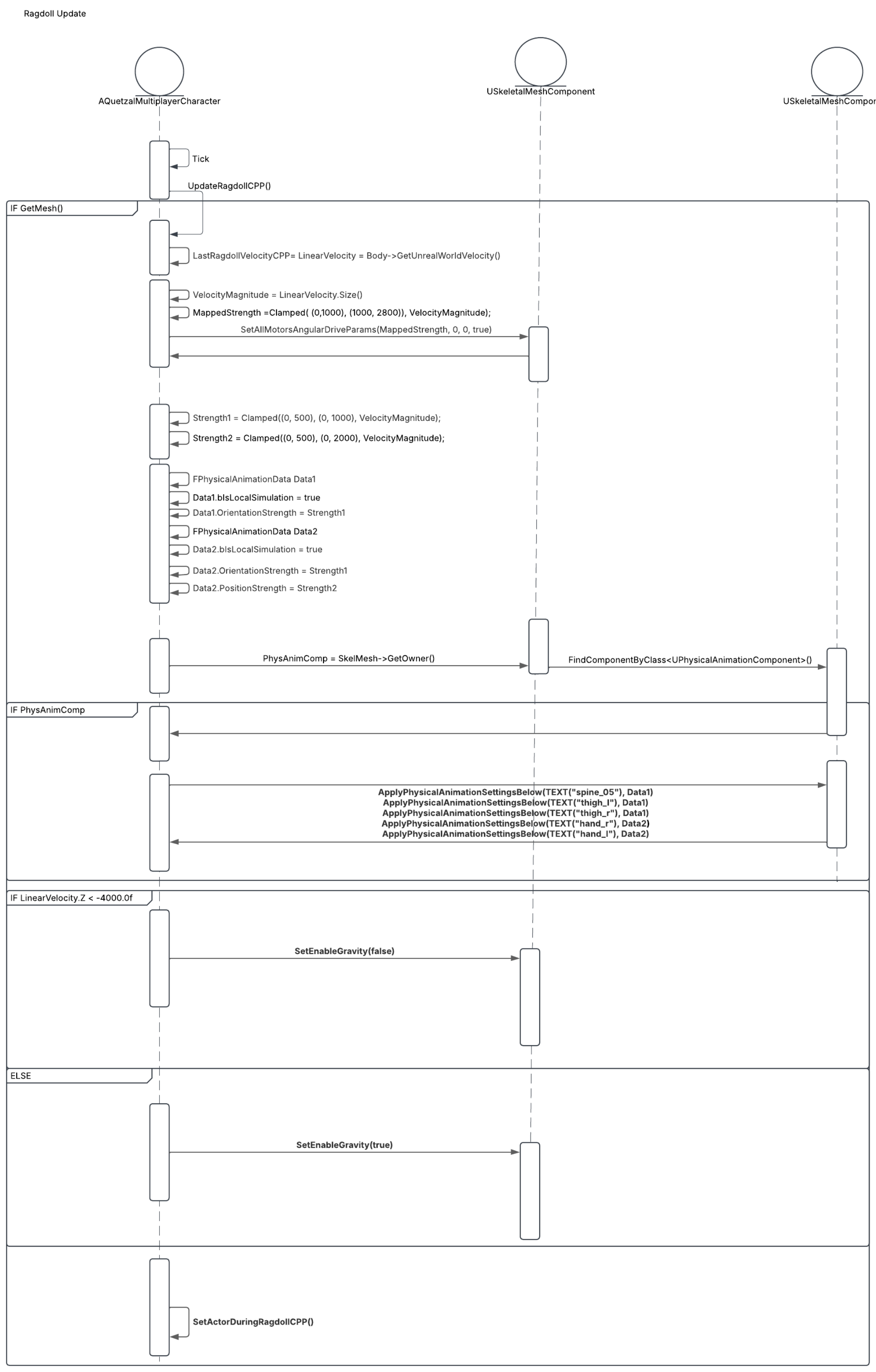
Process Steps:

1. **Retrieve Animation Instance**:
   - The method starts by retrieving the animation instance from the skeletal mesh to access and manipulate the character's animations.
2. **Save Ragdoll Pose**:

- A snapshot of the last ragdoll pose is saved, which can be used for blending into recovery animations, ensuring a smooth transition from the ragdoll state.

3. **Check Ragdoll Position**:
    - The method checks if the character is on the ground (RagdollOnGroundCPP). If true, it sets the character movement mode to MOVE_None to stabilize the character before transitioning to animations.
4. **Play Recovery Animation**:
    - Depending on the character's orientation (RagdollFaceUpCPP), the appropriate "get up" animation is played to visually transition the character from lying down to standing.
5. **Handle Aerial Ragdoll**:
    - If the character is not on the ground, the movement mode is set to MOVE_Falling, and the character's movement speed is adjusted to match the last known velocity (LastRagdollVelocityCPP) from the ragdoll state, preparing for a landing sequence.
6. **Reset Collisions and Physics**:
    - Collision settings are adjusted:
        - The skeletal mesh's collision type is set back to ECC_Pawn to resume normal interactions with game elements.
        - Physical simulation on the skeletal mesh is disabled, ceasing the ragdoll physics.
    - The capsule component's collision is potentially reactivated to resume its role in character collisions and interactions.
7. **Reposition and Reattach Skeletal Mesh**:
    - The skeletal mesh is repositioned slightly if needed to align with the capsule component.
    - Finally, the skeletal mesh is firmly reattached to the capsule component to ensure that it moves correctly with the rest of the character model as controlled movements resume.
8. **Update State Variable**:
    - The state variable isRagdollingCPP is set to false, marking the end of the ragdoll state and resuming normal gameplay mechanics.

UpdateRagdoll


Ragdoll Update
AQuetzalMultiplayerCharacter
USkeletalMeshComponent
USkeletalMeshComponent
Tick
UpdateRagdollCPP()
IF GetMesh()
LastRagdollVelocityCPP= LinearVelocity = Body->GetUnrealWorldVelocity()
VelocityMagnitude = LinearVelocity.Size()
MappedStrength =Clamped((0,1000), (1000, 2800)), VelocityMagnitude);
SetAllMotorsAngularDriveParams(MappedStrength, 0, 0, true)
Strength1 = Clamped((0, 500), (0, 1000), VelocityMagnitude);
Strength2 = Clamped((0, 500), (0, 2000), VelocityMagnitude);
FPhysicalAnimationData Data1
Data1.bIsLocalSimulation = true
Data1.OrientationStrength = Strength1
FPhysicalAnimationData Data2
Data2.bIsLocalSimulation = true
Data2.OrientationStrength = Strength1
Data2.PositionStrength = Strength2
PhysAnimComp = SkelMesh->GetOwner()
FindComponentByClass<UPhysicalAnimationComponent>()
IF PhysAnimComp
ApplyPhysicalAnimationSettingsBelow(TEXT("spine_05"), Data1)
ApplyPhysicalAnimationSettingsBelow(TEXT("thigh_l"), Data1)
ApplyPhysicalAnimationSettingsBelow(TEXT("thigh_r"), Data1)
ApplyPhysicalAnimationSettingsBelow(TEXT("hand_r"), Data2)
ApplyPhysicalAnimationSettingsBelow(TEXT("hand_l"), Data2)
IF LinearVelocity.Z < -4000.0f
SetEnableGravity(false)
ELSE
SetEnableGravity(true)
SetActorDuringRagdollCPP()

The UpdateRagdollCPP() is crucial for dynamically updating the physics and animations of a character in a ragdoll state based on ongoing physical interactions and movements.

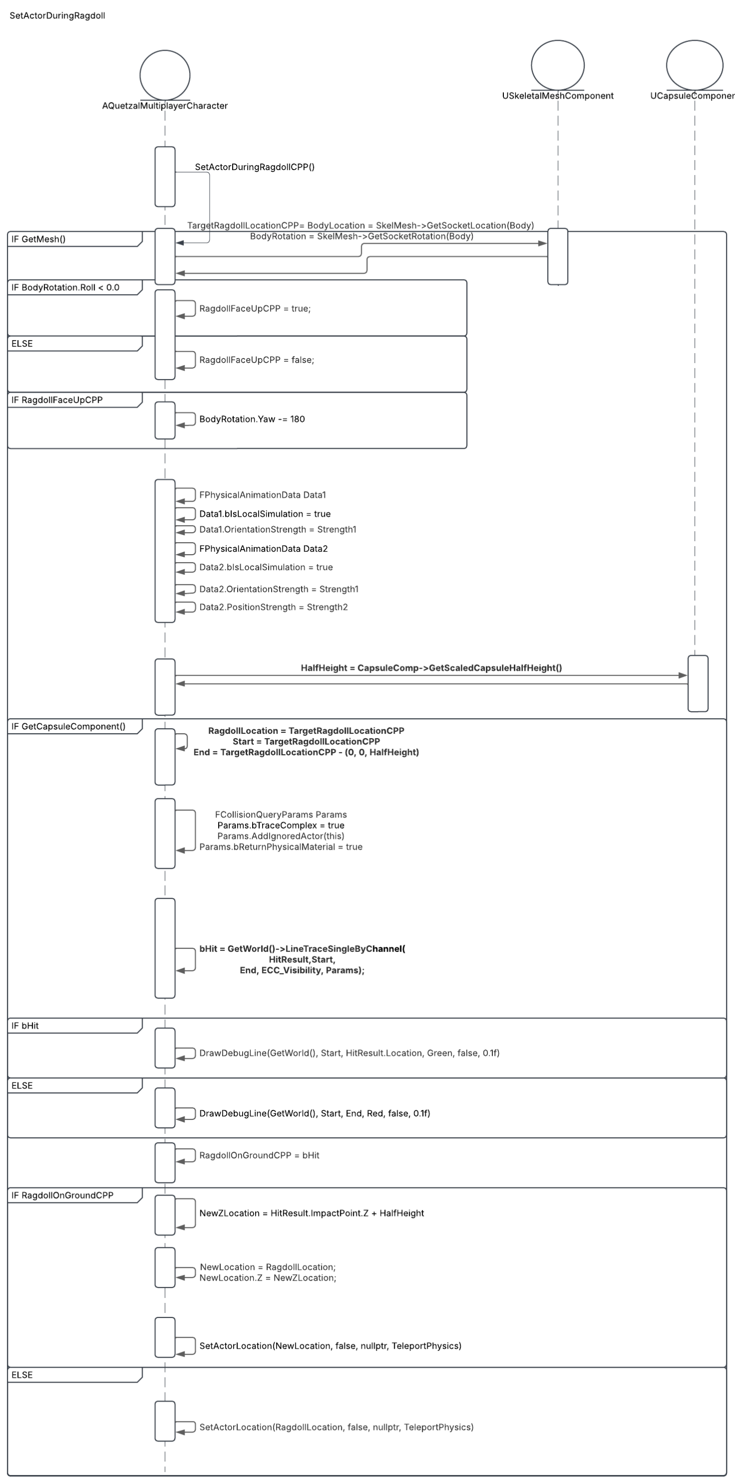
Process Steps:

1. **Velocity Analysis**:
   - Retrieve the linear velocity of a central bone (Body) in the skeletal mesh to understand how the character is moving in the ragdoll state.
   - Store this velocity in LastRagdollVelocityCPP for potential use in other calculations or recovery scenarios.
2. **Dynamic Strength Mapping**:
   - Calculate the magnitude of the velocity to understand the intensity of the movement.
   - Map this magnitude to a new range that will determine the strength applied to the physical animation motors, adjusting how joints and limbs react to ongoing forces.
3. **Physical Animation Adjustment**:
   - Set parameters for the motors controlling angular movements in the skeletal mesh based on the mapped strength, enhancing the realism of the ragdoll's reactions.
   - Create and apply two sets of physical animation data:
     - **Data1**: Controls basic joint strength and is applied to major body parts like the spine and thighs.
     - **Data2**: Adjusts the position and orientation strength for more detailed control, applied to extremities like hands.
4. **Component-Specific Adjustments**:
   - Utilize the PhysicalAnimationComponent to apply the animation settings below specific bones, allowing for localized physical responses that reflect the character’s current physical state.
5. **Gravity Management**:
   - Check if the character is falling too quickly (indicated by a Z-component of velocity less than -4000). If so, disable gravity to prevent unnatural acceleration.

   - If the velocity is not excessively high, ensure that gravity is enabled, allowing natural falling dynamics.

6. **Update Actor State**:
   - Call SetActorDuringRagdollCPP() to adjust the character's position or other properties based on the ragdoll simulation, ensuring the character's representation in the game world remains accurate and believable.

SetActorDuringRagdollCPP


SetActorDuringRagdoll
AQuetzalMultiplayerCharacter
USkeletalMeshComponent
UCapsuleComponent
SetActorDuringRagdollCPP()
IF GetMesh()
TargetRagdollLocationCPP= BodyLocation = SkelMesh->GetSocketLocation(Body)
BodyRotation = SkelMesh->GetSocketRotation(Body)
IF BodyRotation.Roll < 0.0
RagdollFaceUpCPP = true;
ELSE
RagdollFaceUpCPP = false;
IF RagdollFaceUpCPP
BodyRotation.Yaw -= 180
FPhysicalAnimationData Data1
Data1.bIsLocalSimulation = true
Data1.OrientationStrength = Strength1
FPhysicalAnimationData Data2
Data2.bIsLocalSimulation = true
Data2.OrientationStrength = Strength1
Data2.PositionStrength = Strength2
HalfHeight = CapsuleComp->GetScaledCapsuleHalfHeight()
IF GetCapsuleComponent()
RagdollLocation = TargetRagdollLocationCPP
Start = TargetRagdollLocationCPP
End = TargetRagdollLocationCPP - (0, 0, HalfHeight)
FCollisionQueryParams Params
Params.bTraceComplex = true
Params.AddIgnoredActor(this)
Params.bReturnPhysicalMaterial = true
bHit = GetWorld()->LineTraceSingleByChannel(
HitResult,Start,
End, ECC_Visibility, Params);
IF bHit
DrawDebugLine(GetWorld(), Start, HitResult.Location, Green, false, 0.1f)
ELSE
DrawDebugLine(GetWorld(), Start, End, Red, false, 0.1f)
RagdollOnGroundCPP = bHit
IF RagdollOnGroundCPP
NewZLocation = HitResult.ImpactPoint.Z + HalfHeight
NewLocation = RagdollLocation;
NewLocation.Z = NewZLocation;
SetActorLocation(NewLocation, false, nullptr, TeleportPhysics)
ELSE
SetActorLocation(RagdollLocation, false, nullptr, TeleportPhysics)

The SetActorDuringRagdollCPP is designed to continuously update and adjust the character's position and orientation based on the ragdoll physics simulation during gameplay. This method ensures that the character remains accurately represented in the game world relative to the physics interactions it is undergoing.

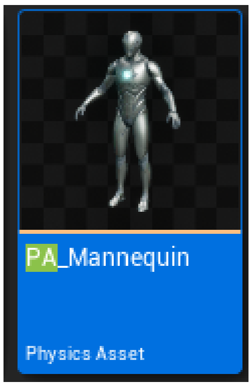
Process Steps:

1. **Retrieve Skeletal Mesh Component**:
   - Access the skeletal mesh component which represents the character's body in the game to manipulate its position and orientation.
2. **Obtain Body Location and Rotation**:
   - Fetch the current location and rotation of the 'Body' socket, a central point in the ragdoll simulation, to understand the character's current positioning in the game world.
3. **Update Internal State**:
   - Store the retrieved location in TargetRagdollLocationCPP for potential use in further calculations or adjustments.
   - Determine whether the character is facing up or down based on the roll component of the body rotation. Adjust the yaw rotation if the character is facing up to align correctly with game world orientations.
4. **Ground Collision Check**:
   - Conduct a line trace directly downward from the body location to check if the character is lying on the ground. This is crucial for deciding how to manage recovery animations and other interactions.
   - Adjust the character's Z position to align precisely with the ground level if the trace hits the ground, ensuring that the character appears properly placed relative to terrain and other objects.
5. **Update Actor Position**:
   - If the line trace indicates the character is not on the ground, update the character's position to reflect the ongoing ragdoll simulation without making Z-axis adjustments. This ensures the character remains visually coherent in aerial or falling states.
6. **Set Ragdoll Grounded State**:
   - Update RagdollOnGroundCPP based on whether the trace detected the ground. This state is crucial for other methods that might rely on whether the character is grounded to perform further logic.

6 Set Up Guide

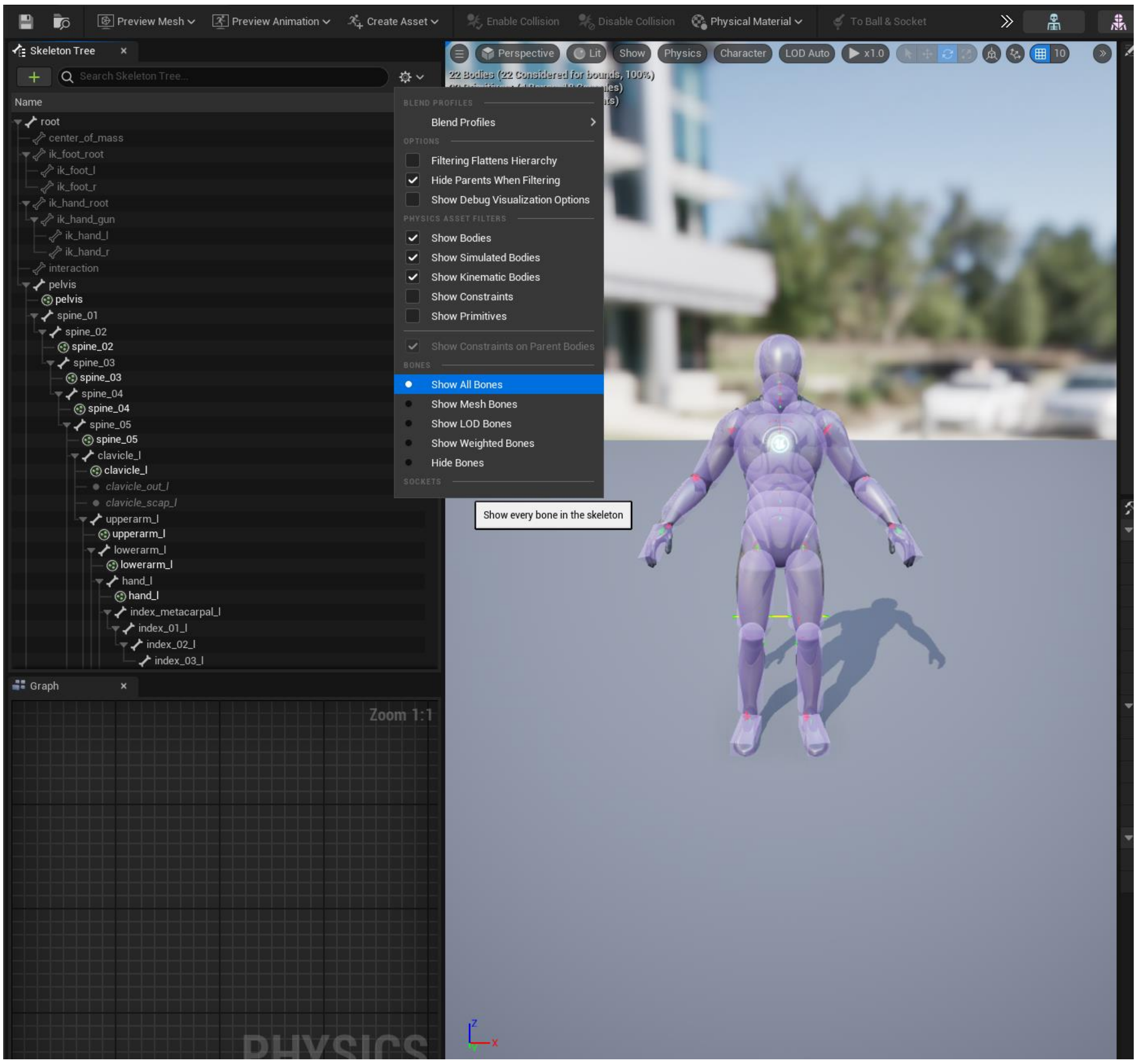
Physical Asset SetUP

Locate the Physical Asset of the character



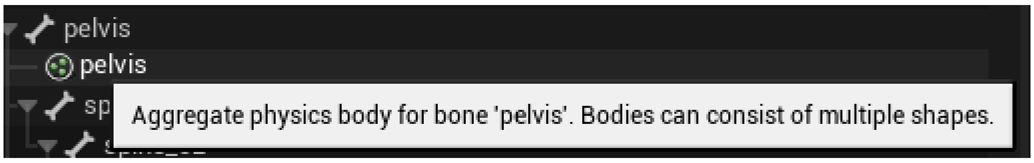
Double click it and get into the Physics view

Change the settings to Show all bones so you can see the root and the skeletal mesh attached to the PA



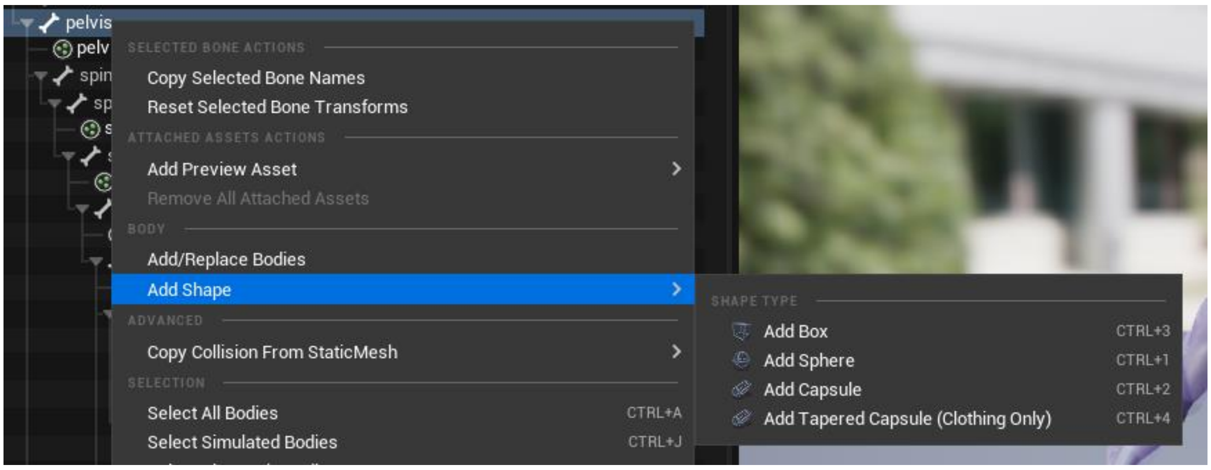
The skeleton tree is divided into

Bones(From the skeletal mesh), Shapes(the physical assets) and constraints (connections between the PA/articulations)



Aggregate physics body for bone 'pelvis'. Bodies can consist of multiple shapes.

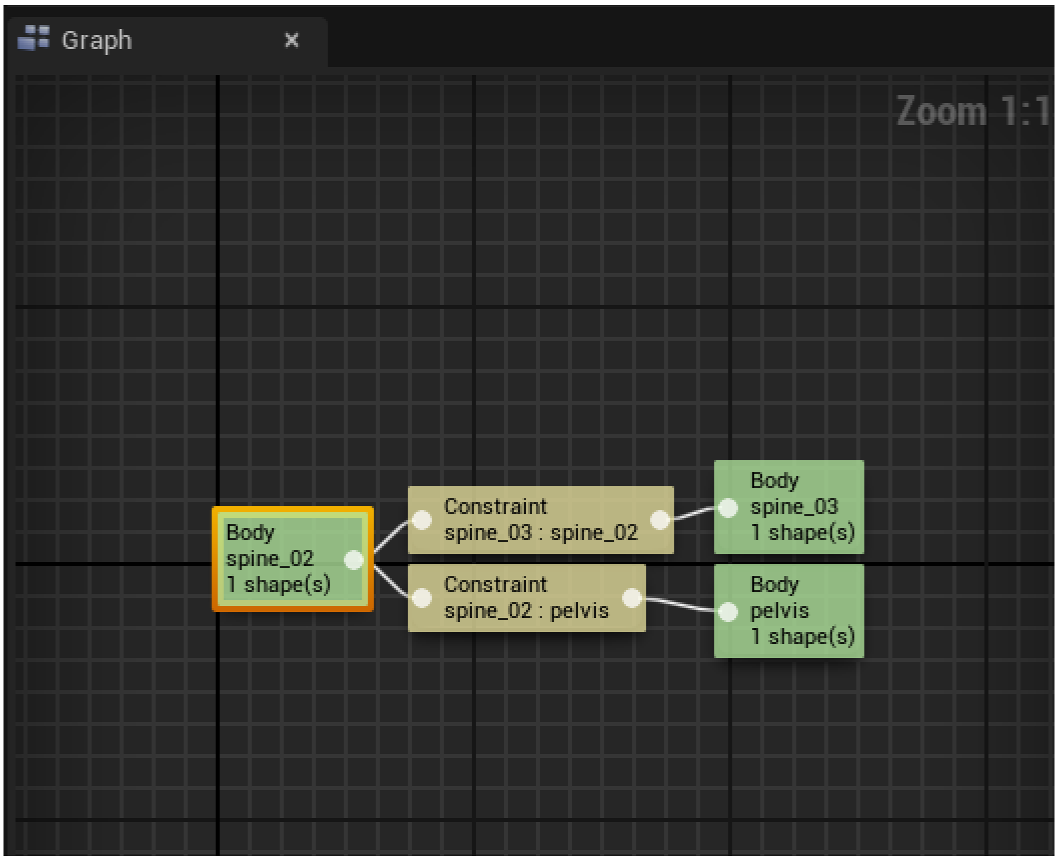
For all the bones you think they need a PA Right click the bone and add a shape



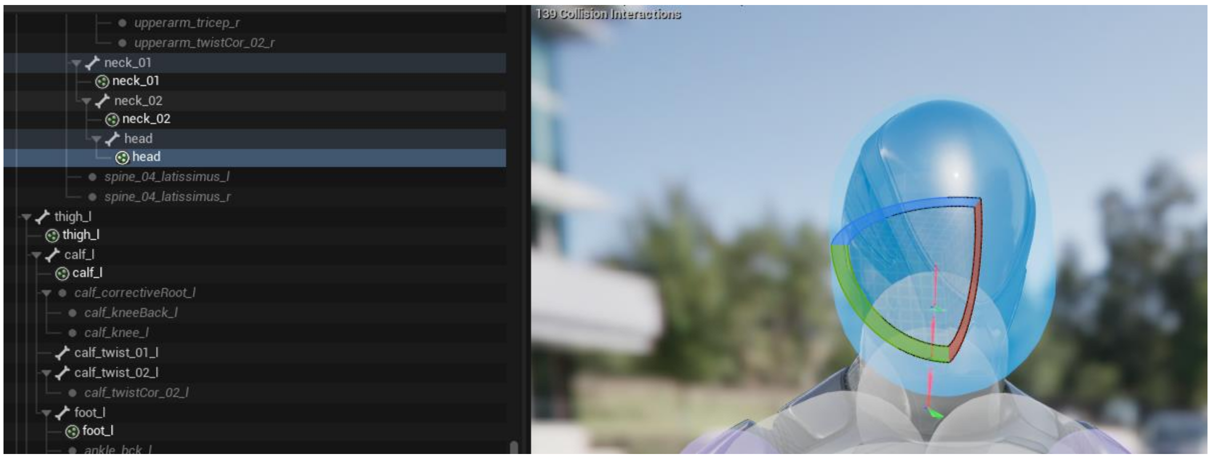
You need to edit the Size and rotaton of each PA,

You can see those bones and constraints

Also you can see them in the physics graph

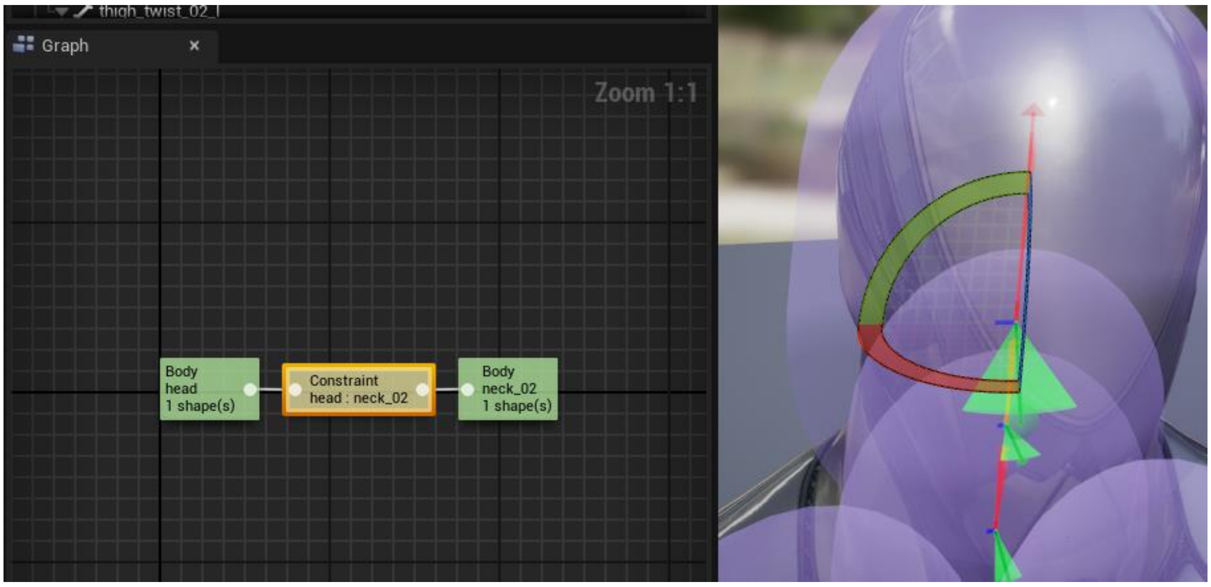


Select Bodies: Click on different colliders in the viewport to select and view corresponding bones.

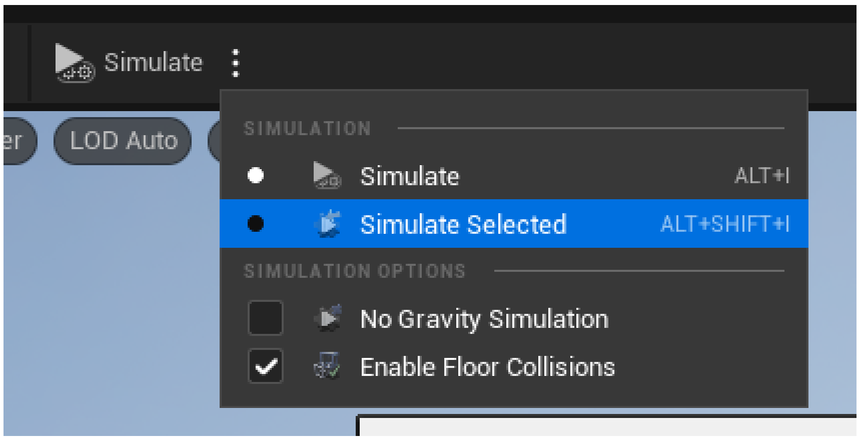


Select Constraint: click on the corresponding node or the arrows that show the constraints

Constraints limit how far each bone can rotate or twist.



Test Simulation: Use the Simulate button to see how the character collapses under physics.

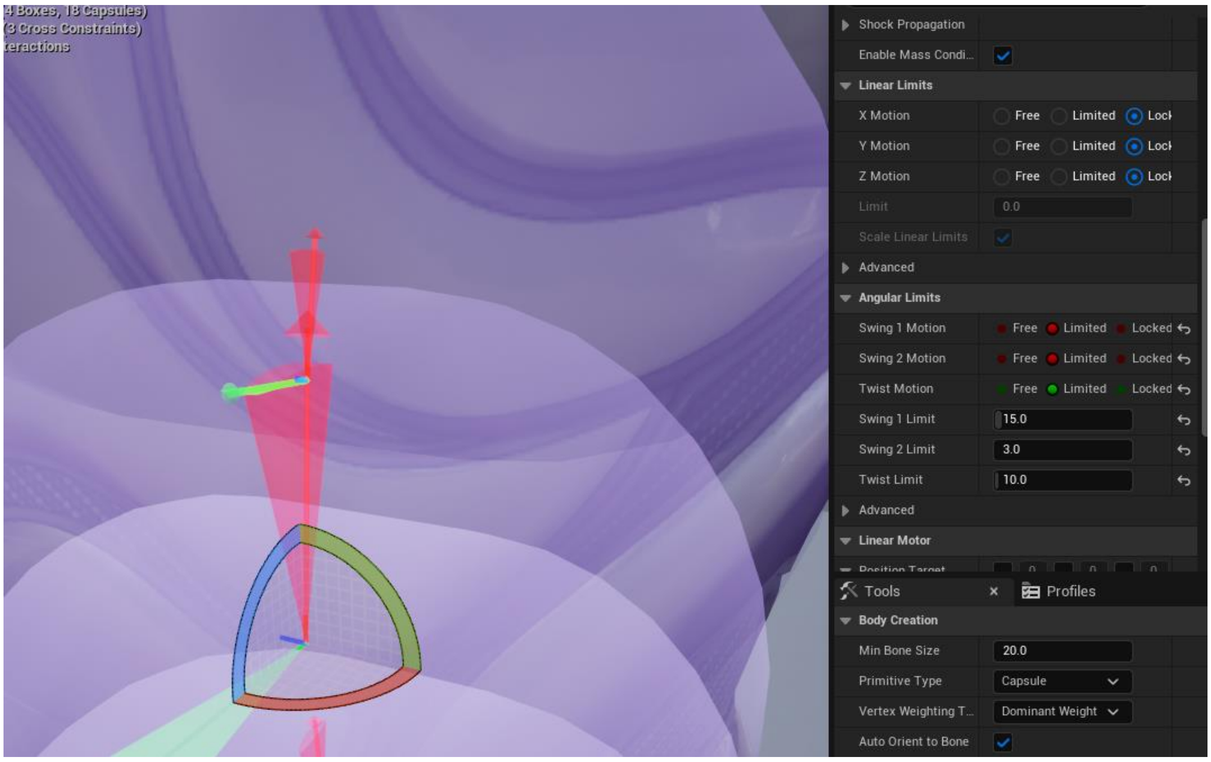


You can simulate all the body or just a group of them

Physics Setup

Constraints are important because they determine how each body part will react with the others.

For that we need to edit the Angular Limits



Shock Propagation	
Enable Mass Condi...	✓
Linear Limits	
X Motion	Free Limited [✓ Lock]
Y Motion	Free Limited [✓ Lock]
Z Motion	Free Limited [✓ Lock]
Limit	0.0
Scale Linear Limits	✓
Advanced	
Angular Limits	
Swing 1 Motion	Free [✓ Limited] Locked
Swing 2 Motion	Free [✓ Limited] Locked
Twist Motion	Free [✓ Limited] Locked
Swing 1 Limit	15.0
Swing 2 Limit	3.0
Twist Limit	10.0
Advanced	
Linear Motor	

Tools Profiles

Body Creation	
Min Bone Size	20.0
Primitive Type	Capsule
Vertex Weighting T...	Dominant Weight
Auto Orient to Bone	✓

We have Swing 1 Motion, Swing 2 motion and Twist.

Swing 1 and Swing 2: These limits specifically manage the arm's movement within vertical and horizontal planes.

Swing 1 handles the forward and backward motions, akin to swinging the arm towards or away from the body.

Swing 2 controls the lateral movements, allowing the arm to swing side to side across the body. Setting these parameters accurately ensures that the arm's movements remain within natural human limits, thus preventing any awkward or exaggerated swings.

Twist:
This limit is focused on the rotation's movement around the arm's longitudinal axis. It regulates how much the arm can twist.

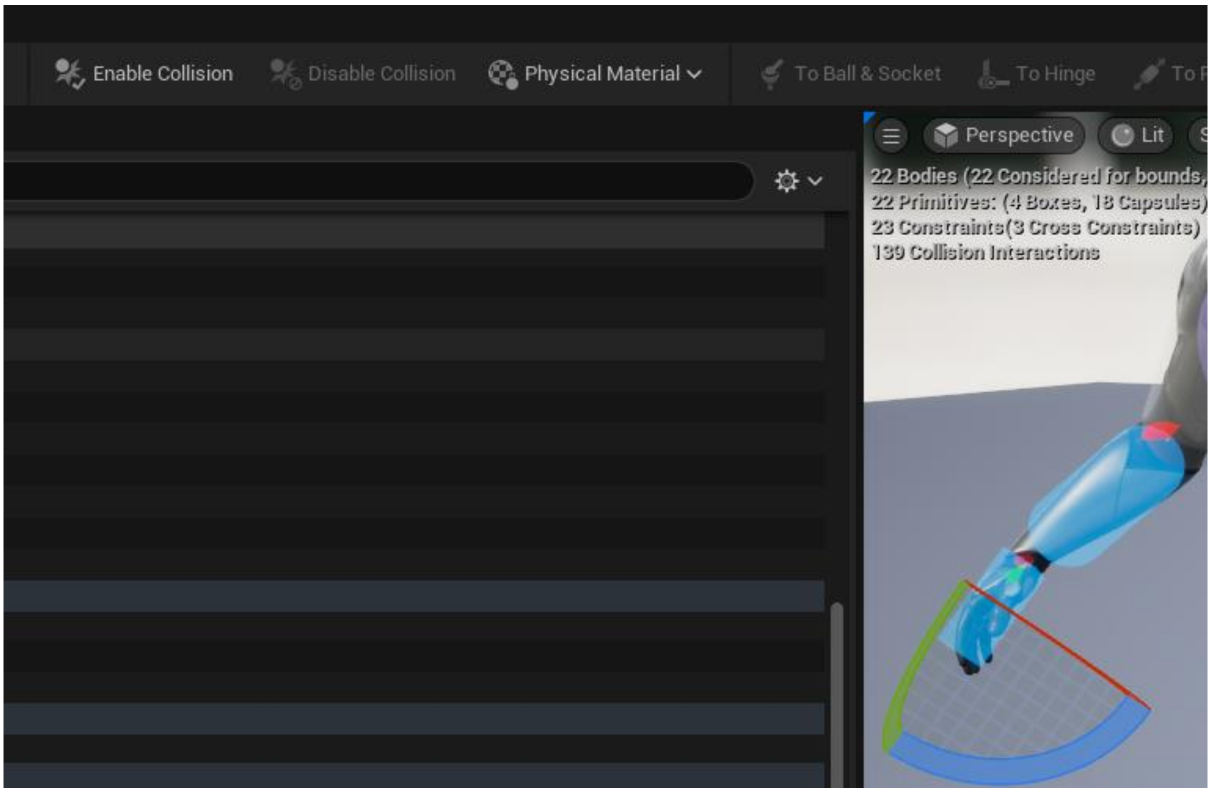
Linear Limits control movement along an axis, allowing settings for free, limited, or locked motion.

Angular Limits manage joint rotation via Twist and Swing parameters.

Drive Settings apply spring-like forces to maintain alignment of physics bodies.

Before starting to edit the Constraints we need to make sure that exist a collision Between others

You need to select 2 shapes and enable collision to them



22 Bodies (22 Considered for bounds,
22 Primitives: (4 Boxes, 18 Capsules)
23 Constraints(3 Cross Constraints)
139 Collision Interactions

Not all of them need a collision enable between.

Its important to have a Root bone, is crucial in ragdoll physics because it retains velocity information when the character's capsule may not move, serving as a reliable reference for movement and animation transitions in Unreal Engine, where actor velocity isn't accessible during physics simulations.

Constraints Setup

Use the constraints of the PA Mannequin that its already edited to work properly