Open World RPG

Engine: Unreal Engine

Focus: Gameplay systems, animation, technical iteration

Developer: Arya Singh

Feature presentation : **Inverse Kinematics**

Inverse Kinematics (IK) is a mathematical technique used in animation and robotics to calculate joint positions based on a desired end position. Instead of animating each joint in a limb individually (as in forward kinematics), IK works backward: the position of the end effector, such as a foot, is defined first, and the system computes the necessary rotations and translations of connected joints to reach that position.

Mathematical Perspective

From a mathematical standpoint, inverse kinematics solves a constrained optimization problem. Given:

- A chain of joints with known lengths and rotational limits
- A target position in 3D space

The system calculates joint angles that satisfy:

- Distance constraints (bone lengths)
- Rotation constraints (joint limits)
- Environmental constraints (ground contact, obstacles)

This typically involves vector math, trigonometry, and iterative solvers that approximate a valid solution in real time. In games, these calculations must be efficient and stable, as they are evaluated every frame during gameplay.

Impact on Game Presentation

In practical game development, inverse kinematics significantly improves visual realism and player immersion. Without IK, characters rely solely on pre-authored animations, which assume flat ground and ideal conditions. This often results in feet clipping through terrain, floating above slopes, or sliding unnaturally during movement.

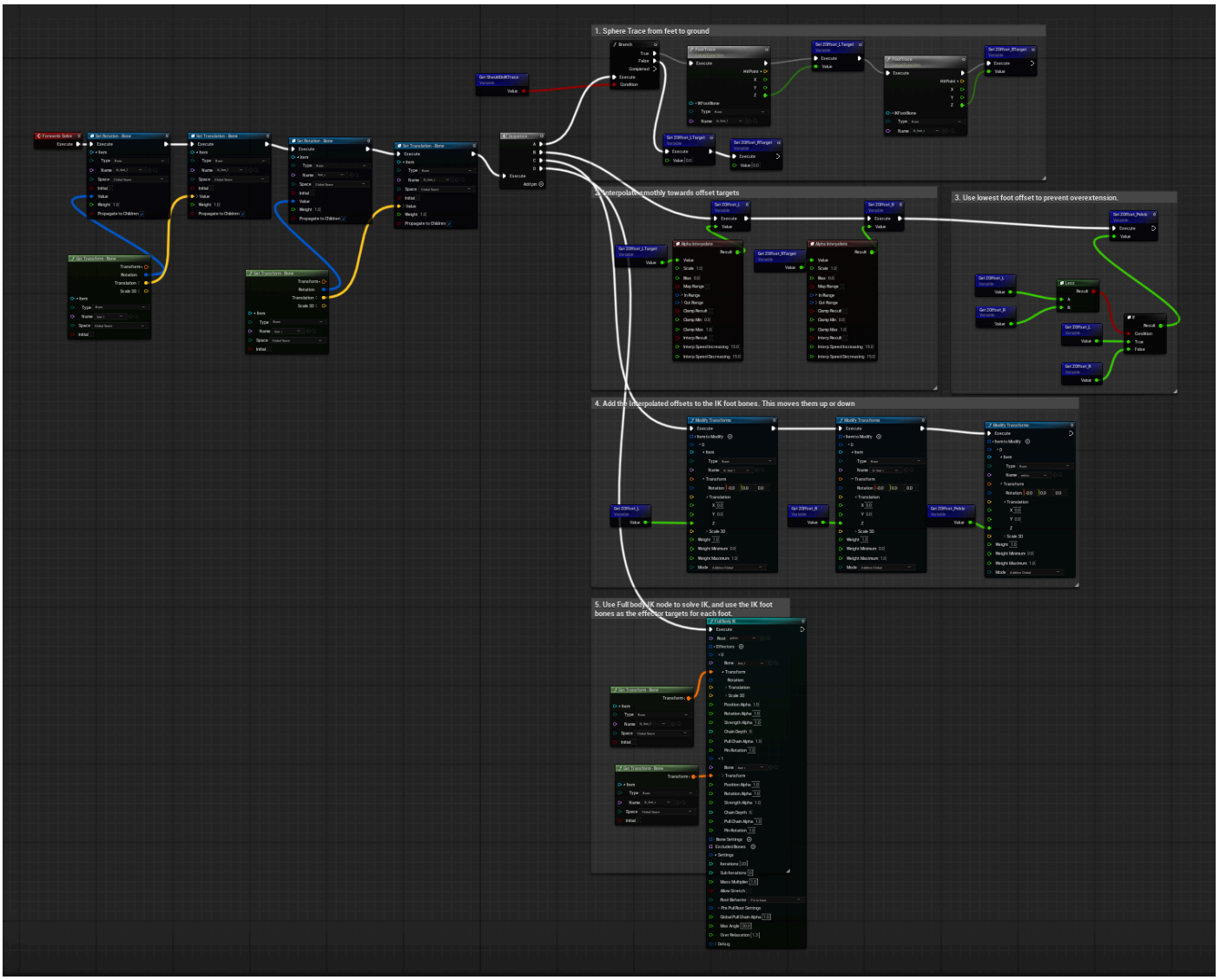
By applying IK:

- Character feet dynamically adjust to uneven terrain
- Weight distribution appears more natural
- Contact with the environment feels grounded and responsive

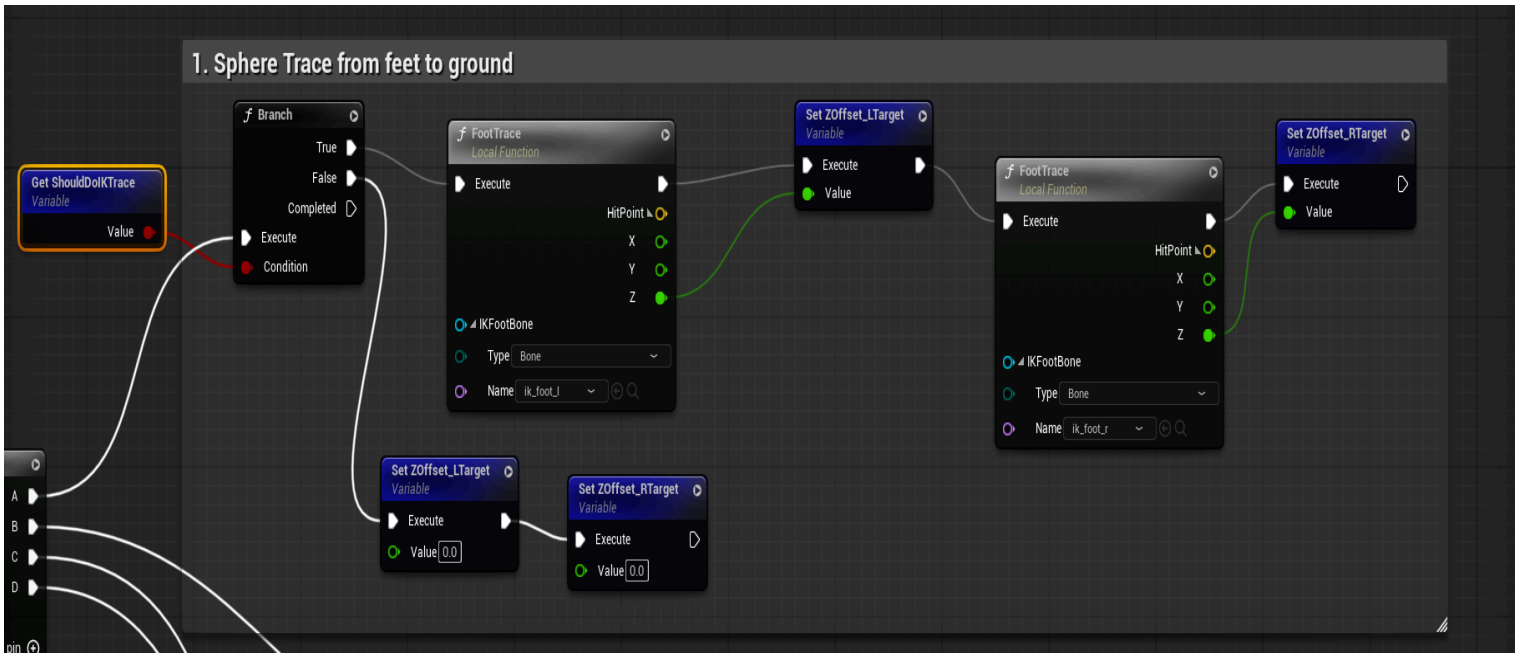
Rather than playing a fixed animation, the character’s body reacts to the game world in real time. This transforms animation from a static visual asset into a responsive system, reinforcing the illusion that the character physically inhabits the environment.

Gameplay without Inverse Kinematics

Overview of the implementation:



1. Sphere Tracing Feet to the Ground



The first step is detecting where the ground is relative to each foot.

For both the left and right foot, a **sphere trace** is performed:

- The trace starts slightly **above** the foot
- It ends **below** the foot
- A small radius is used to avoid hitting the foot itself

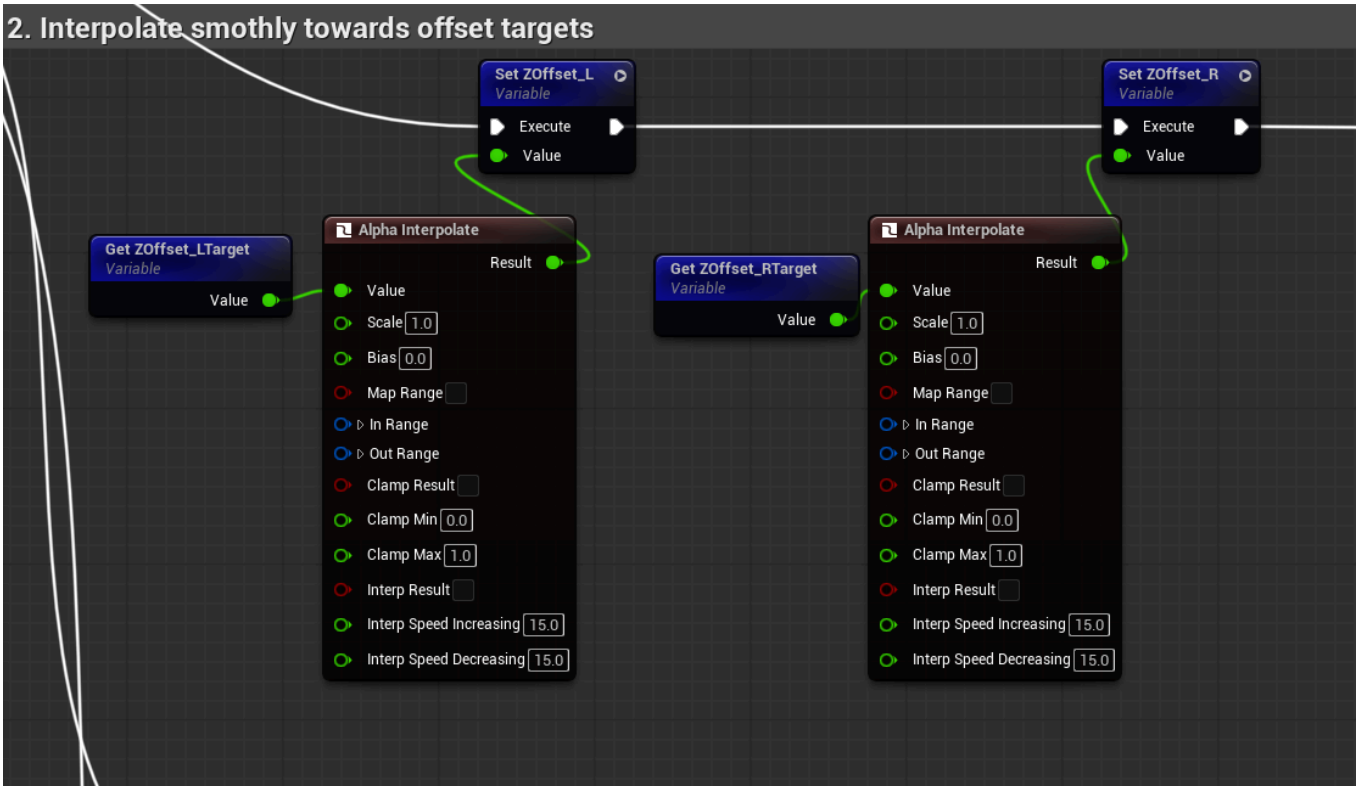
This trace detects the **exact point where the foot would contact the ground**, even on sloped or uneven surfaces. Only the **Z-value (vertical height)** of the hit location is stored, since vertical displacement is what matters for foot placement.

Each foot stores its own target:

- `Z_Offset_L_Target`
- `Z_Offset_R_Target`

This step allows the system to dynamically respond to any terrain shape without relying on predefined animations.

2. Smooth Interpolation Toward Target Offsets



Directly snapping feet to the traced ground position would look unnatural and jittery. To prevent this, the system interpolates *(*calculation of the value of a function between the values already known) toward the target offsets using float interpolation.

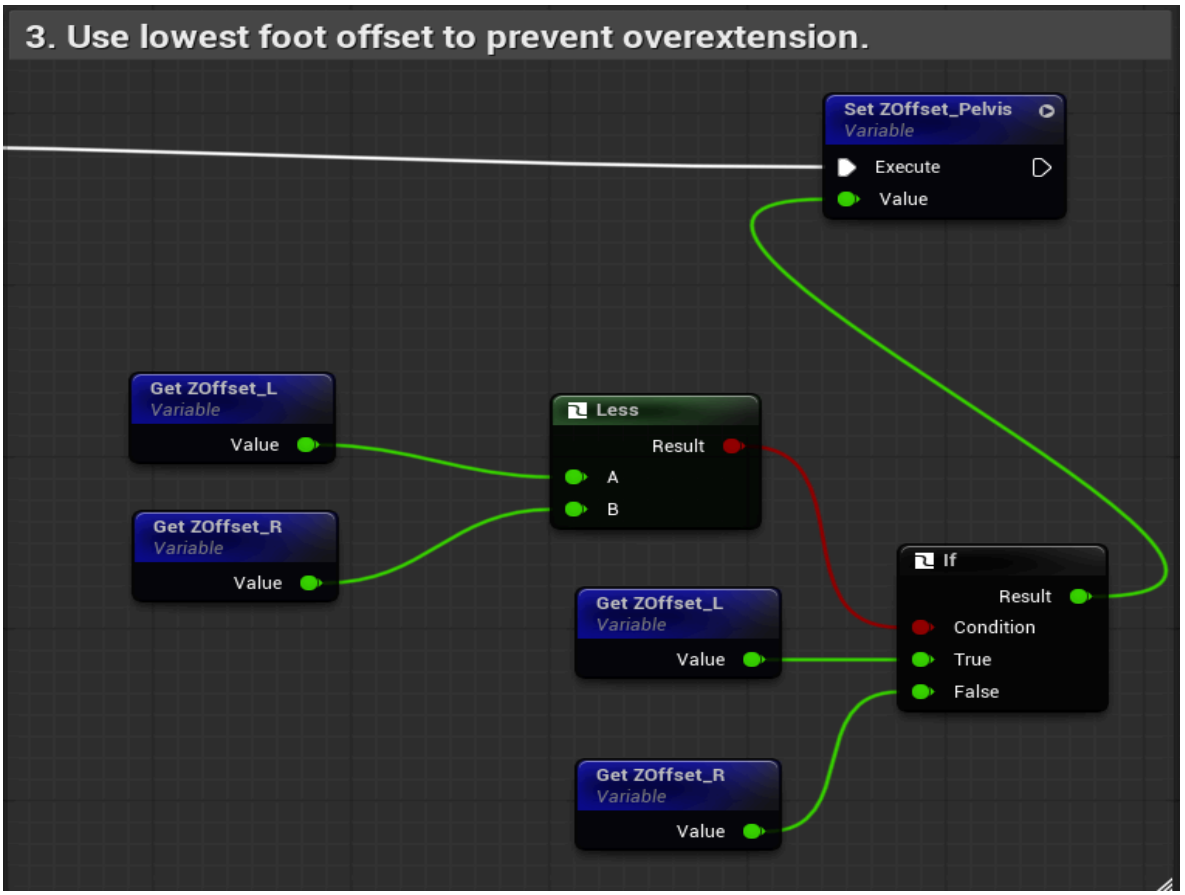
- Each frame, the current offset moves gradually toward its target
- Interpolation speeds control how responsive or soft the motion feels
- This avoids visual popping and creates natural foot motion

This produces two continuously updated values:

- `Z_Offset_L`
- `Z_Offset_R`

These values lag slightly behind the raw trace results, making movement feel grounded and realistic.

3. Determining the Pelvis Offset

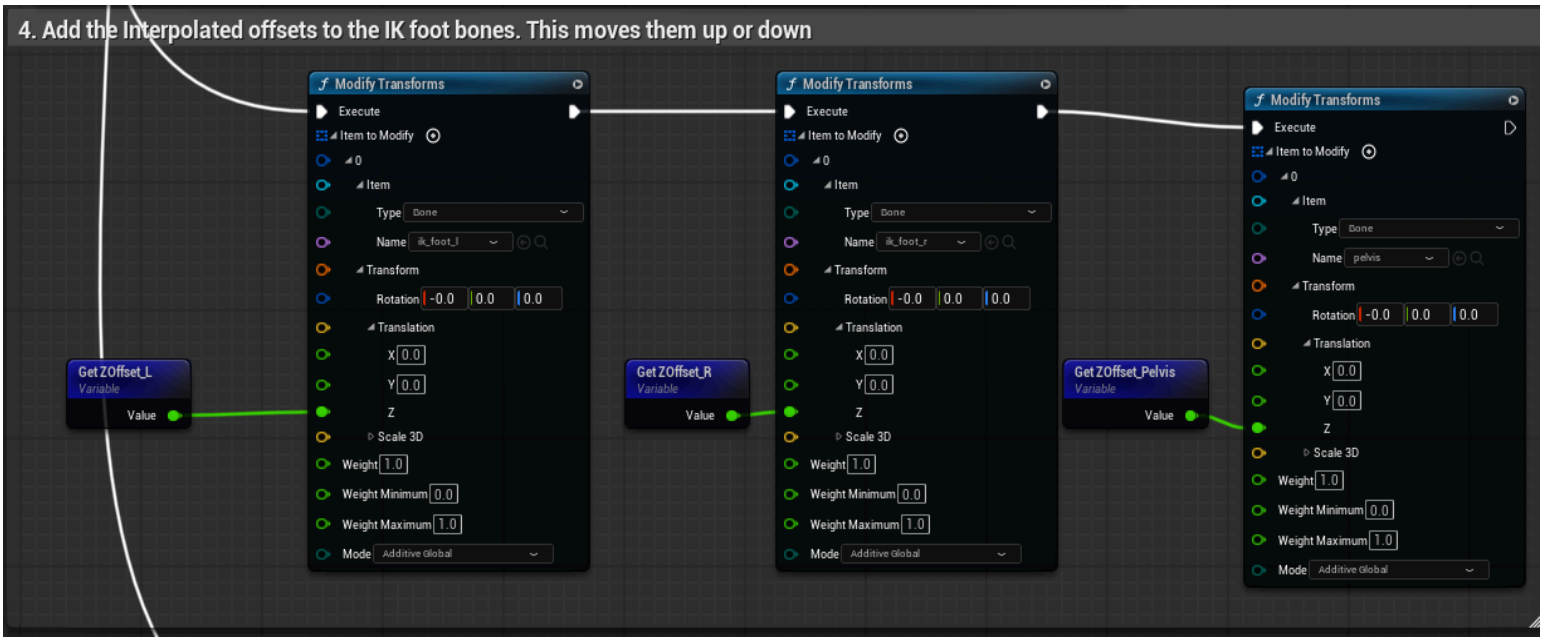


When standing on uneven terrain, one foot may be lower than the other. To prevent the legs from overextending, we select **the lowest of the two foot offsets**.

- The system compares `Z_Offset_L` and `Z_Offset_R`
- The lower value is chosen as `Z_Offset_Pelvis`
- The pelvis is lowered by this amount

By lowering the pelvis based on the lowest foot, both legs remain within a natural bending range. This prevents unrealistic stretching and ensures the character maintains believable posture and weight distribution.

4. Applying Offsets to IK Foot Bones and Pelvis

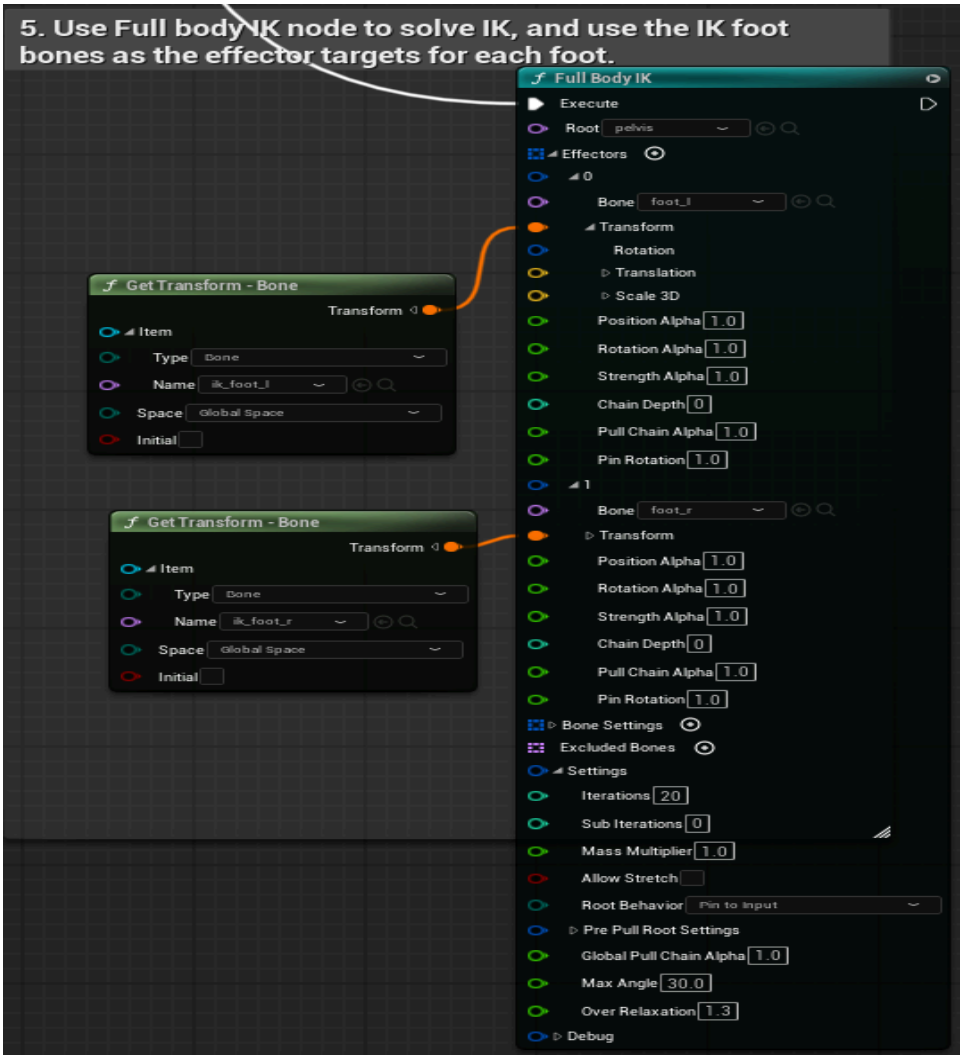


The calculated offsets are applied using **additive global transforms**:

- IK (or virtual) foot bones are shifted vertically using their respective offsets
- The pelvis bone is lowered using `Z_Offset_Pelvis`

Using IK or virtual bones allows precise positioning without directly distorting the skinned mesh. The visible skeleton remains intact while the IK system drives the adjustment.

5. Solving Leg Motion with Full Body IK



The final step uses **Full Body IK** to solve the leg chains naturally.

- Each foot acts as an **effector** (a bone that you explicitly control in an IK system)
- Effector targets come from the transformed IK foot bones
- The pelvis serves as the root of the solver

The solver automatically adjusts knees and leg joints, producing anatomically correct bending and stable foot placement on uneven surfaces.

IK System Result:

This system eliminates floating feet and ground penetration while maintaining smooth motion and realistic posture.

Feature presentation : **Trace Based Weapon Hit Detection**

Please refer to the source of this code here
https://github.com/AryaDSingh/OpenWorldRPG-Prototype/blob/main/Source/OpenWorldRPG/Private/Items/Weapons/Weapon.cpp

https://github.com/AryaDSingh/OpenWorldRPG-Prototype/blob/main/Source/OpenWorldRPG/Public/Items/Weapons/Weapon.h

Please visit this link to see the animations and the collision box, in action
https://youtu.be/scl42CsqltE

This weapon system uses a C++ trace-based approach to melee hit detection, ensuring accurate and reliable collision during fast combat animations. Instead of relying solely on overlap events, a box trace is performed between two points on the weapon each frame, preventing missed or duplicated hits.

Hit responses are handled through a C++ interface, allowing enemies and other actors to define their own reactions without coupling them to the weapon logic. Visual effects are exposed to Blueprints, maintaining a clean separation between gameplay systems and presentation.

Problem: Reliable Melee Hit Detection

Traditional melee systems in Unreal Engine 5 often rely on collision overlaps attached directly to weapon meshes. While simple, this approach introduces several problems:

- Multiple hit events can trigger during a single swing
- Fast animations can cause missed collisions
- Weapons may repeatedly damage the same target per attack
- Collision logic becomes tightly coupled to specific enemy classes

These issues reduce both gameplay consistency and system scalability.

Solution: Box Trace-Based Weapon Collision

To address these limitations, this weapon system uses a C++ box trace to detect hits between two points on the weapon during an attack.
Instead of relying on constant collision overlap events, a trace is performed along the weapon's motion path, allowing precise and controlled hit detection.

1. Weapon Collision Architecture

Rather than relying on mesh collision alone, this system separates visual representation from hit detection logic.

```
WeaponBox = CreateDefaultSubobject<UBoxComponent>(TEXT("Weapon Box"));
WeaponBox->SetupAttachment(GetRootComponent());
```

Key design choices:

- Weapon collision exists as its own component
- Collision is disabled by default to avoid unintended overlaps
- Collision responses are explicitly configured

```
WeaponBox->SetCollisionEnabled(ECollisionEnabled::NoCollision);
WeaponBox->SetCollisionResponseToAllChannels(ECR_Overlap);
WeaponBox->SetCollisionResponseToChannel(ECC_Pawn, ECR_Ignore);
```

This ensures the weapon only interacts with the world when explicitly activated during attacks.

---

2. Trace Volume Definition

To simulate the weapon's swing, two scene components define the trace boundaries:

```
BoxTraceStart = CreateDefaultSubobject<USceneComponent>(TEXT("Box Trace Start"));
BoxTraceStart->SetupAttachment(GetRootComponent());

BoxTraceEnd = CreateDefaultSubobject<USceneComponent>(TEXT("Box Trace End"));
BoxTraceEnd->SetupAttachment(GetRootComponent());
```

These components:

- Move naturally with the weapon
- Allow fine-tuning in the editor
- Define a swept volume instead of a single collision point

This avoids missed hits during fast animations.

---

3. Performing the Box Trace

Each attack frame executes a single, controlled box trace:

```
UKismetSystemLibrary::BoxTraceSingle(
    this,
    Start,
    End,
    FVector(5.5f, 5.f, 5.f),
    BoxTraceStart->GetComponentRotation(),
    ETraceTypeQuery::TraceTypeQuery1,
    false,
    ActorsToIgnore,
    EDrawDebugTrace::ForDuration,
    BoxHit,
```

```
    true
);
```

Why this matters:

- The trace accounts for motion between frames
- Debug traces visually validate hit detection
- Collision is deterministic and reproducible

This is far more reliable than raw overlap events.

---

4. Preventing Multi-Hit Exploits

A common melee bug is hitting the same enemy multiple times per swing. This system prevents that by using an ignore list.

```
TArray<AActor*> IgnoreActors;

Actors already hit during the current attack are excluded:
for (AActor* Actor : IgnoreActors)
{
    ActorsToIgnore.AddUnique(Actor);
}

ActorsToIgnore.Add(this); // prevent self-hit

After a successful hit:
IgnoreActors.AddUnique(BoxHit.GetActor());
```

Result:

- One hit per actor per attack
- Clean damage logic
- Predictable combat feel

---

5. Interface-Driven Hit Reactions

Instead of hard-coding enemy behavior, the weapon communicates through a C++ interface.

```
IHitInterface* HitInterface = Cast<IHitInterface>(BoxHit.GetActor());

If implemented:
HitInterface->Execute_GetHit(
    BoxHit.GetActor(),
    BoxHit.ImpactPoint
```

```
);
```

This allows:

- Enemies, props, and breakables to react differently
- No weapon-enemy coupling
- Easy system extension

---

6. Blueprint–C++ Hybrid Effects Pipeline

Visual feedback is exposed via a Blueprint event:

```
UFUNCTION(BlueprintImplementableEvent)
void CreateFields(const FVector& FieldLocation);
```

This allows:

- Artists or designers to iterate on VFX
- No recompiling C++ for visual changes
- Clean separation of concerns

Gameplay logic stays in C++; presentation remains flexible.

---

7. Weapon Equipping & State Control

```
Weapons are attached using socket-based transforms:
FAttachmentTransformRules TransformRules(
    EAttachmentRule::SnapToTarget,
    true
);

ItemMesh->AttachToComponent(
    InParent,
    TransformRules,
    InSocketName
);
```

When equipped:

- Weapon stops idle effects
- Sound feedback is triggered
- Item state is updated

```
ItemState = EItemState::EIS_Equipped;
```

This demonstrates understanding of:

- Actor lifecycle
- State-based gameplay logic

- Animation-ready weapon systems
