Activity - Character

Marco Antonio García Rodríguez

08/06/2024

For this project, I initially followed the technical implementations taught in class to achieve a third-person character character, including its animation logic and components, using both C++ and Blueprints. However I did the necessary modifications so this implementation could work with a character different to Paragon's Belica. In my case, I decided to use Yin as my character.

For the implementation of the character's jump, using the input actions and the movement component in the Character blueprint I was able to work on the animation blueprint's state machine by checking if the character is in the air; if it is, the jump animation is executed, and when it detects that the character has returned to the ground, a landing animation is played (Image 1).

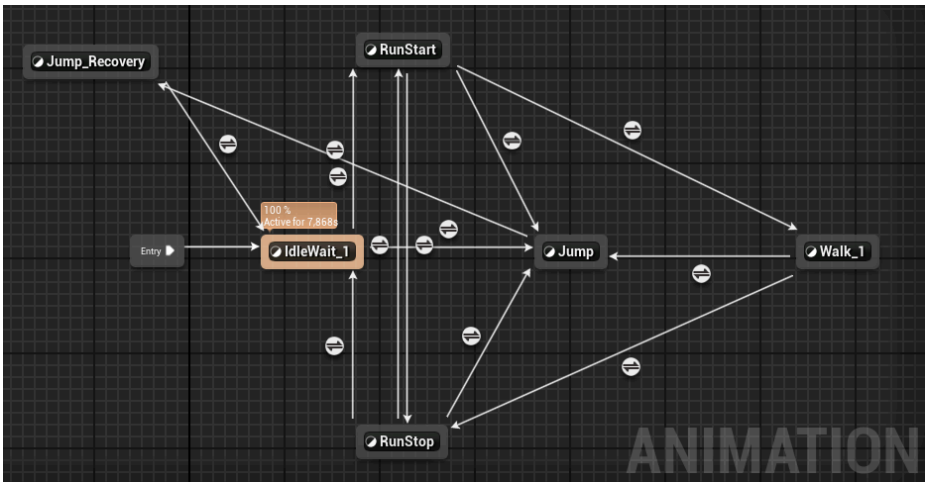

Image 1. Animation State Machine.

Regarding the additional player actions, I utilized the Ability System to implement the shooting ability covered in class, which I am using as a "primary attack." I created a Blueprint class called "BP_Destructible" capable of taking damage from this player "ability."

Upon receiving damage, it triggers a particle effect and, after a brief delay, is destroyed. To activate this ability, the InputAction "IA_PrimaryAttack" must be executed by pressing the right mouse button.

I also developed an ability called "primary interaction" to interact with certain types of objects in the world, which I named "interactables." These objects inherit from a C++ class I created called Interactable. This class essentially includes a mesh, a collider, and a "BlueprintImplementableEvent" named "OnInteraction" (Image 2).

```cpp
AInteractable::AInteractable()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    DefaultSceneRoot = CreateDefaultSubobject<USceneComponent>(TEXT("DefaultSceneRoot"));
    RootComponent = DefaultSceneRoot;

    Widget = CreateDefaultSubobject<UWidgetComponent>(TEXT("Widget"));
    Widget->AttachToComponent(RootComponent, FAttachmentTransformRules::KeepRelativeTransform);
    Widget->SetWidgetSpace(EWidgetSpace::Screen);

    Sphere = CreateDefaultSubobject<USphereComponent>(TEXT("Sphere"));
    Sphere->InitSphereRadius(100.0f);
    Sphere->AttachToComponent(RootComponent, FAttachmentTransformRules::KeepRelativeTransform);

}
```

Image 2. Interactable class construction method.

The two objects I created based on this class for player interaction are a door and an item that restores the player's health. Both objects are Blueprints that implement the "OnInteraction" event. Using the door as an example, it runs a timeline to interpolate one of its angles to animate the "opening" and "closing" process (Image 3). In the case of the health item, it takes the character as an argument and can call a "heal" method within the character, which updates the character's health via the attribute component.

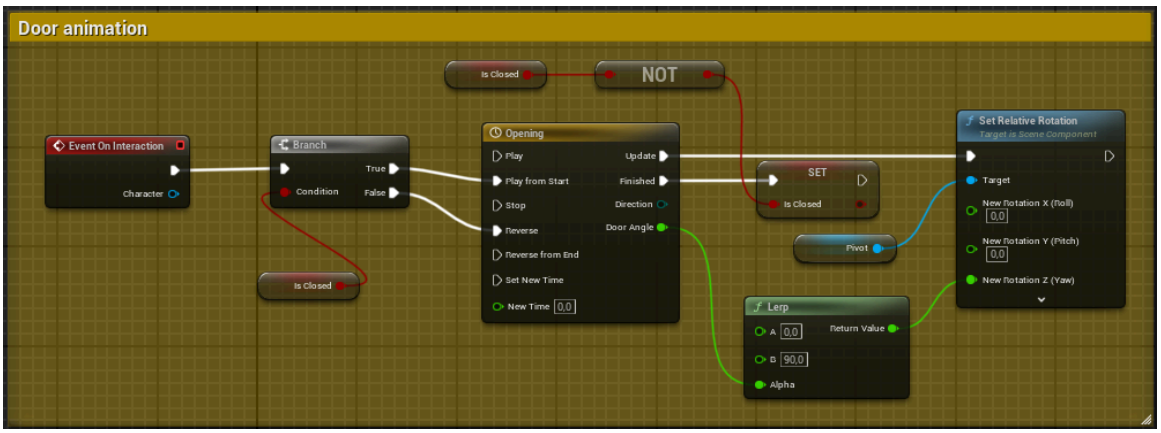

Image 3. Door "OnInteraction" Implementable event.

To interact with these objects, the InputAction "IA_PrimaryInteraction" is called by pressing the "E" key. When within the detection range of these objects, a message will be displayed on the screen.

I used gameplay tags to prevent the player from performing two actions simultaneously. I created the CharacterTags container in C++ with references to the tags "Status.PrimaryAttack" and "Status.PrimaryInteraction." To perform any of these actions, the base character class must ensure that no tag for a different ability is active. It checks this with an if condition. If the condition is met, the desired action is executed using the ability component (Image 4).

Once the abilities were executed, I called a timed method that waits for a duration equal to the cooldown time. When this duration is reached, it ends the ability and removes the gameplay tag associated with the ability from the tag container.

```cpp
void AD3DCharacterBase::PrimaryInteraction() {

    if (!CharacterTags.HasTag(AttackTag)) {
        CharacterTags.AddTag(InteractionTag);
        UE_LOG(LogTemp, Warning, TEXT("I have Primary Interaction tag"));
        AbilityComponent->TryActiveAbilityByName("PrimaryInteraction");
    }

    if (CharacterTags.HasTag(AttackTag)) {
        UE_LOG(LogTemp, Warning, TEXT("I have Attack tag"));
    }
}

void AD3DCharacterBase::PrimaryAttack()
{
    if (!CharacterTags.HasTag(InteractionTag)) {
        CharacterTags.AddTag(AttackTag);
        UE_LOG(LogTemp, Warning, TEXT("I have Primary Attack tag"));
        AbilityComponent->TryActiveAbilityByName("PrimaryAttack");
    }
}
```

Image 4. Gameplay tag validation before ability execution.

Finally, I used the health component to enable the player to receive damage. I created a simple HUD that displays a health bar, which starts completely full. A basic Blueprint that inflicts damage upon contact with the player causes the player's health component to modify the player's health attributes. In Yin's Blueprint, my character, I take the current health value and use it to update the health bar in the HUD (Image 5).

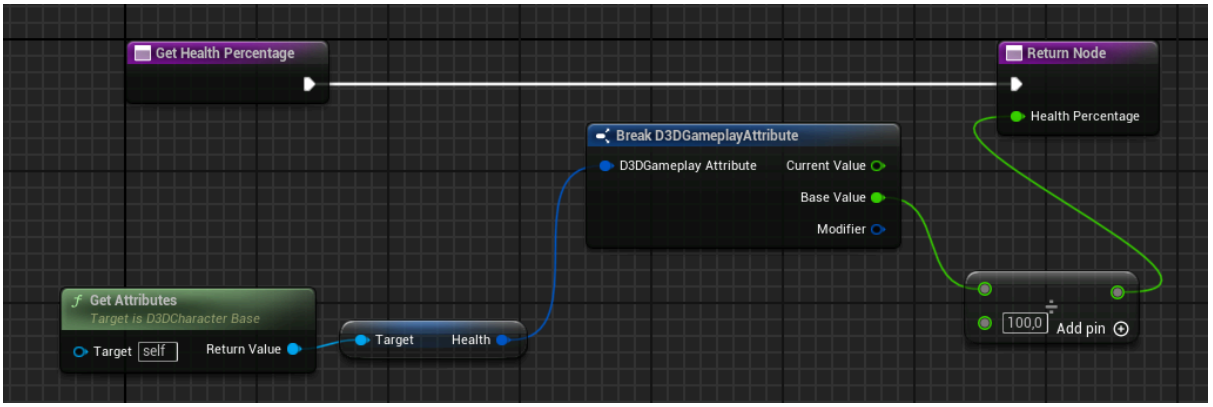

Image 5. Calculation of Health percentage to modify the HUD health bar.