CS378 Lab 4: Building a Player Class

This lab will focus on creating a playable character using the Character class provided by UE4. You will also explore the concepts of player state and separating input handling from the playable character.

Getting Started

Upon launching the current version of UE4, create a Game Project then select “Blank” as a Template. For Project Settings, set the project to be C++ based. You do not need to include Starter Content as we will not be creating interactables in the scene, so the basic level and lighting setup will be sufficient for us. Name the project “CS378_Lab4” and click “Create Project”. Connect this project to source control (using git ignore and git lfs).

Take the provided base map and modify to your liking (one big plane is perfectly fine for what this lab requires) then save it into a folder you create called “Maps” in the Content folder. You don’t need to worry about lighting etc — so long as there is some basic geometry to move along, enough lighting to see, and a Player Start actor in the scene, you are good.

Creating a Character

For the next step, you will create Lab4Character, which will inherit from ACharacter not APawn. This is very important. The Character class has a bunch of additional functionality that Pawn does not include including access to the CharacterMovementComponent, which we are not modifying at this time, but we will be using to add in jump functionality.

Since we’ll be in first-person view mode for this project, you don’t need to add any geometry to Lab4Character, but you will need to add basic input handling under Project Settings -> Engine -> Input. Please include WASD controls for movement (WS will be move forward and backward, and AD will be strafe left and right), Interactions mapped to E, and Jump mapped to space bar.

You can go ahead and make some of the necessary functionality as you’ve done before such as:

- Create BlueprintImplementableEvents as you’ve done in previous labs to handle input on the BP side of things
- Create your Lab4CharacterBP file in a folder you create called “Blueprints” under Content
- Modify both the Lab4GameModeBase.cpp to set the DefaultPawn to Lab4CharacterBP and update Project Settings -> Project -> Maps & Modes -> Default GameMode to match your custom GameMode rather than using the Default.
At this point we will completely diverge from the previous labs so stop copying-and-pasting!

Creating a Player Controller

We are now going to explicitly create our own Player Controller rather than relying on the default interface via our Pawn/Character class. While for many games, accessing the inputs directly from the Playable Character is fine, as soon as we introduce any complexity (e.g. networking, dynamically swapping between characters, a large roster of characters with asymmetric controls, etc), treating the controller (e.g. the interface for the player into the game) as a separate abstraction from a character (e.g. an object the player controls) greatly benefits us (and may be essential in the case of certain networking scenarios).

To do this, we first need to create a custom Lab4PlayerController, which inherits from PlayerController. Create a constructor method for this class as well as a protected override of the virtual function SetupInputComponent(). You can go ahead and create some private functions as well, which you’ll bind the axis/actions to. Note that we’re not calling on the BlueprintImplementableEvents directly here. Instead we’ll bind the inputs to our functions in the Lab4PlayerController class and then from there call on the BlueprintImplementableEvents.

It will look something like this in SetupInputComponent():
Super::SetupInputComponent();
InputComponent->BindAxis("Move", this, &ALab4PlayerController::Move);
...

And in a separate function, Move(float value) it will look like this:
ALab4Character * character = Cast<ALab4Character>(this->GetCharacter());
if (character)
{
    character->MoveBPEvent(value);
}

Where MoveBPEvent is the BlueprintImplementableEvent called from the Character itself.

At this point, you may be thinking “why all this overhead for something I could do in one class just as easily?” but again — the point is to see an architecture that more gracefully handles increasing amounts of complexity.

Interlude: Some Non-Trivial Scenarios

Imagine we have multiple playable characters the player can swap between in a puzzle game (e.g. Lost Vikings, Trine, etc). In this case, the same buttons on the controller may match to a completely different move on the character. An inheritance-style structure won’t necessarily solve this problem, but a component-based approach allows us to change things quickly and efficiently in one location.
Now think about games with DLC characters. The separation of the controller for the character will speed up the process of building these characters, and will allow devs to make big changes in the design of the character controls without breaking already existing characters.

Finally, think about networked games where you can respawn and/or change characters. Having a Player Controller that is associated with one player throughout the game versus a bunch of Pawns that are continually spawned and destroyed helps with managing state.

And Back to Coding...

Once you have hooks in Lab4PlayerController connecting the input bindings to the BlueprintImplementable calls in Lab4Character, make sure you add this line to your GameMode, so that the default Player Controller is updated to yours:

```cpp
PlayerControllerClass = ALab4PlayerController::StaticClass();
```

At this time you should be able to access player inputs within Lab4Character’s Event Graph. You should create movement and strafe functionality at this time and confirm that the Character successfully moves.

Working with Character State

We are now going to introduce the idea of action states to our Lab4Character class. Eventually you may want to create an entire Finite-State-Machine component to connect to your Character, but for now, we’ll keep the functionality within Lab4Character itself. We’ll do so using a UENUM:

```cpp
ECharacterActionStateEnum
```

This enum will be of type `uint8` so it is accessible via Blueprints and should have the macro `UENUM(BlueprintType)`. It will look something like this:

```cpp
UENUM(BlueprintType)
enum class ECharacterActionStateEnum : uint8 {
    IDLE UMETA(DisplayName = "Idling"),
    MOVE UMETA(DisplayName = "Moving"),
    JUMP UMETA(DisplayName = "Jumping"),
    INTERACT UMETA(DisplayName = "Interacting")
};
```

To create our very basic FSM functionality, we need to keep track of the state: `ECharacterActionStateEnum CharacterActionState` and we need at least two functions: `CanPerformAction(ECharacterActionStateEnum updatedAction)` and `UpdateActionState(ECharacterActionStateEnum newAction)`. These functions should be BlueprintCallable so that we can check if an updated action is allowed before performing the action and updating the current action state to
our new action. This flow of actions can be laid out in BP, but the actual functions that determine the logic should be in C++.

The logic to implement is as follows:

- All actions are available from Idle
- Player cannot Interact while in Jump or Move states
- Player cannot perform any other action while in Interact
- An interaction takes n seconds (by default n = 3) to complete, at which point the player returns automatically to Idle
- Once the player jumps, they remain in the Jump state until they land (touching the ground triggers the Character’s OnLanded Event), at which point the player returns automatically to Idle
- The difference between Idle and Move states are the player’s current velocity (accessible in Character via GetVelocity())

As you’re implementing all of this logic in the Lab4Character class, please include a couple print-to-screen calls when interactions begin and end, so we can verify that’s working without implementing a full interaction system.

Submission

After you’re satisfied, collect some video footage showing the system in action. Include some print statements to screen when an action is not allowed, so the TA can verify actions are available (and unavailable) when expected. Also screenshot exciting parts of your code, and submit these files plus the project code via your GitLab account and include a link to your video via Youtube. Link your repository as your Canvas submission.