CS354p Lab 1: Working in UE5

This lab is to help familiarize you with the Unreal Engine system and see existing code/coding best practices. While we will work from the "Top Down" template for this lab to see it in action, we will be building from blank templates for the remaining labs and projects. That said, it's worth looking through the example projects to see how to get started and what is expected of your code.

Getting Started

Upon launching UE5.2.1, create a Game Project then select "Top Down" as a Template. For Project Settings, set the project to be C++ based. You do not need to include Starter Content, but you can if you would like. Name the project "Lab1" and click "Create Project".

Now is a good time to go through the editor walkthrough if you have not done so already and look through all the existing code via the associated IDE (Visual Studio in the case of Windows machines).

Once you are done, you will explore UE5 by adding and modifying the existing code to give you an idea of how the system functions. For this lab, I will be giving relatively detailed instructions as working with UE5 for the first time can be quite mystifying, but the point is not just to follow directions but to understand how they work. If at any time, you feel confused, please ask me or the rest of the class for assistance.

Adding Controls

Open the Content Drawer at the bottom to reveal the Content Browser (I like to pin mine to the workspace, but it depends on how much screen real estate you have). Go to Content->TopDown->Input, and you will see an asset called "IMC_Default." This is an InputMappingContext object that contains all of the input-handling for this project. You can make your own InputMappingContext by right-clicking in the Content Browser and selecting Input->Input Mapping Context, but for now we will use IMC_Default for convenience.

There are currently action mapping for moving the character using mouse and touch, but we'll add an action mapping on Enter to dash. To do this, we first add two Input Actions under Content->TopDown->Input->Actions by right-clicking the Browser and selecting Input->Input Action. Create IA_Dash. Add this Input Action to IMC_Default by clicking the + button next to Mappings. Dash should map to Enter. This is the only property you need to set. Save before continuing.

You will then need to adding bindings between the input handler, the character's C++ implementation and the character's BP implementation. To do this, we'll need to open `Lab1PlayerController.h` and add a UProperty and a callback function. The UProperty is a UInputAction that will allow us to bind between the Input Mapping Context and the character controller. Create a UInputAction for Dash that mirrors the input actions for `SetDestinationClickAction`

and `SetDestinationTouchAction`. Also create a **callback function** that will be called when the input is pressed: `OnDashStarted()`.

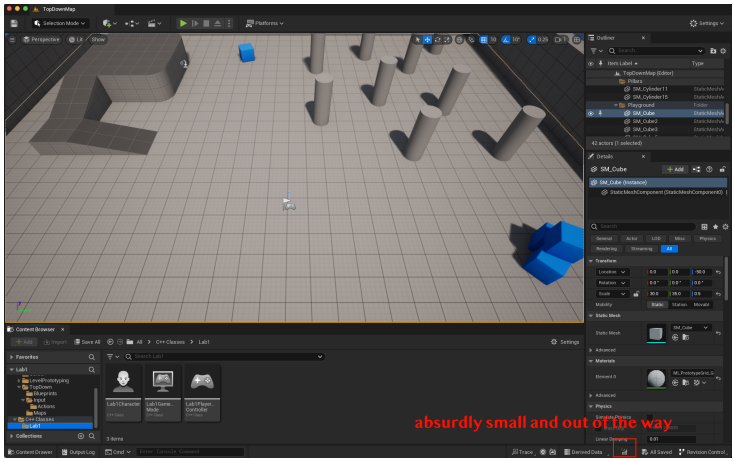
In `Lab1PlayerController.cpp`, we're going to bind this input to the callbacks in `SetupPlayerInputComponent()` using `BindAction`. They will look something like this:

```
EnhancedInputComponent->BindAction([DashInputBinding],
ETriggerEvent::Started, this,
&ALab1PlayerController::OnDashStarted);
```

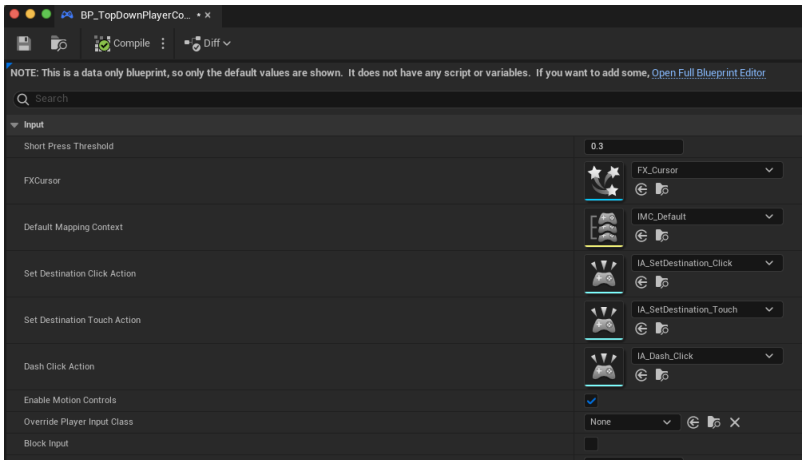
We'll also now need to create the function definition for `OnDashStarted` in the .cpp file. In this function, include a screen debug print statement so we can check if the system is hooked up correctly before we actually implement the functionality:

```
if(GEngine)
GEngine->AddOnScreenDebugMessage(-1, 2.0f, FColor::Yellow, TEXT("Dash
Started"));
```

If you click the absurdly small and out of the way compile button from UE5, you will (hopefully!) see your code compile successfully.



We are not quite done, though, because we still need to tie this functionality into Blueprint. We will go over what this function is doing in more detail later, but in `Lab1GameMode.cpp`, we are building out our playable character from a Blueprint built on top of the C++ code, and the same is true for the Controller class, which we've been modifying. Thus, we need to tie our inputs into the Controller Blueprint itself. Open `BP_TopDownPlayerController`, which is under Content->TopDown->Blueprints. Since you added an Input Action to your C++ files, you will now see that property mirrored here. Assign it to the action you created in IMC_Default. It will look something like this:



You will not need to recompile the code, but be sure to recompile/save the Blueprint you just modified. This should be much faster, and now when you run the game, you should see your print statements appear on the screen when you press the corresponding inputs!

Implementing Dash Functionality

You will experiment with adding the dash functionality on your own, but as a hint, you'll want to take the character's forward vector then apply motion along that axis for some short amount of time. It's possible to do this using an impulse force (launching the character) or using the `AddMovementInput()` function (which is used in the tutorial code). If you want to experiment with impulse forces, you'll need to look into the API for the `ACharacter` class (and `UCharacterMovementComponent` if you want a deeper dive).

Adding Collision Overlaps

Next you will create a more robust world object that will intersect with the player character to allow for callback rather than physics interactions. Right click within the folder where you'd like to place this new class (probably with the other C++ classes) and click "Add C++ class". Create a new Actor and name it something reasonable based on the shape you want (e.g. BoxActor, or SphereActor etc). Note I will be creating a Box and calling it BoxActor accordingly for the rest of this tutorial.

In general, you will create the base functionality and building block pieces in C++ then add any "design considerations" via Blueprint. For this lab, we'll implement collision detection and response in C++, and you can set the associated geometry through a child Blueprint similar to some of the functionality added via BP_TopDownCharacter.

Your Actor class needs two components: a StaticMeshComponent and a PrimitiveComponent. Note that the PrimitiveComponent can be either a sphere

or a box component! It does not need to match the shape of your mesh, as this component is invisible and for checking overlaps. I will be using a Sphere Component, but you're welcome to use a Box! Regardless, create your mesh and primitive components in the BoxActor.h (look at the Lab1Character class for example code in the .h).

In BoxActor.cpp, include the correct component:
e.g. `#include "Components/BoxComponent.h"`

You will then create the components you defined in the .h as subobjects in the constructor (use the Lab1Character class as a reference but if you have any questions, please ask!).

To add collision checks to the BoxActor, we're going to add the OnComponentBeginOverlap to the PrimitiveComponent. You will need to set up notifications so this Component registers events in BeginPlay. To do this, first create a function named something slightly meaningful like `BeginOverlap`. The name itself doesn't matter, so long as the method signature is correct as we'll see in a moment. Be sure to include the `UFUNCTION()` macro as well, and make sure it has the following as its method signature:

```
(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
const FHitResult &SweepResult)
```

At this point, print to screen using:

```
if (GEngine)
GEngine->AddOnScreenDebugMessage
```

(Syntax nicely provided here `https://docs.unrealengine.com/en-US/API/Runtime/Engine/Engine/UEngine/AddOnScreenDebugMessage/1/index.html`)

Then go ahead and destroy the instance if it overlaps with the playable character. As a final step, create a Blueprint based on this C++ class by right-clicking the C++ by selecting"Create Blueprint class." Give it a name (convention is BP_NameOfC++Class) and choose where to save it (probably with the other Blueprints in the project) then hit Create Class. Add a Static Mesh to the mesh component then change the size of the trigger volume so that it extends beyond the mesh shape. Place a few in the scene then test that functionality runs when the player character overlaps with the object.

Feel free to add additional features to BoxActor (or the Pawn) and experiment with working in C++ versus Blueprint. Once you are happy with your results, take video footage of your gameplay (running in editor is fine) as well as screenshots of your BoxActor code and additions to any of the existing C++ classes so the TA can quickly glance over what you are doing without having to run the project/search through the codebase. If you are submitting via Canvas, please zip up the full project after removing any intermediate folders that are not the Source, Config, or Content folder and including the gameplay footage and code screenshots. Note: do not remove the .uproject!! Both the .sln and the .xworkspace can be safely deleted but you do not need to so. If you are submitting through a git repository, please go through GitLab and add me (thesharkcs) and the TA (bosun1) to the project. We must have **Developer** access or higher to your repo to grade the project.