CS378 Lab 1: Working in UE4

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 “Twin-stick Shooter” 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 UE4.27 create a Game Project then select “Twin Stick Shooter” 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 “CS378_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 UE4 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 UE4 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

Under Editor->Project Settings->Input you will see that there are axis mappings (i.e. continuous inputs) for this project. Fire right and fire forward are on right/left and up/down, but add an action mapping on space bar to fire forward and an action mapping on Enter to dash.

You will then need to adding bindings in SetupPlayerInputComponent in Lab1Pawn using BindAction. It will look something like this:

```
PlayerInputComponent->BindAction(FireBinding, IE_Pressed, this, &ALab1Pawn::FirePressed);
```

We’ll now need to make the event `FirePressed` in the .h file. We will make these functions callable from Blueprints so include the `UFUNCTION(BlueprintImplementableEvent)` specifier for both actions. This event can be called from the Blueprint’s Event Graph, so we will need to make a Blueprint that is a child of the existing C++ class. Right click CS378_Lab1Pawn and select “Create Blueprint Class”. Save it in an appropriate location (usually a folder called Blueprints) and name it something like CS378_Lab1PawnBP.

Now you can open CS378_Lab1PawnBP and add these events to your Event Graph. Click “Add New” and select the two functions you just created from the “Override Function” list. You will see them appear in the event graph space. Add a Print String statement to this by clicking on the arrow box and dragging it out. A searchable list of functions will pop up that you can choose from.
You should be compiling constantly to ensure you don’t have errors, but we’re not quite ready to test this at runtime. Go into CS378_Lab1GameMode and add something like the following:

```cpp
// set default pawn class to our character class
static ConstructorHelpers::FObjectFinder<UClass> pawnBPClass(TEXT("/Game/TwinStick/Blueprints/Lab1PawnBP.Lab1PawnBP_C"));

if (pawnBPClass.Object)
{
  GEngine->AddOnScreenDebugMessage(-1, 1.f, FColor::Red, TEXT("PawnBP Found"));
  UClass* pawnBP = (UClass*)pawnBPClass.Object;
  DefaultPawnClass = pawnBP;
}
else
{
  GEngine->AddOnScreenDebugMessage(-1, 1.f, FColor::Red, TEXT("PawnBP Not Found"));
  DefaultPawnClass = ALab1Pawn::StaticClass();
}
```

There is quite a bit happening in here, so please let me know if you need any explanation! The magical path you see on the first line can be copied in by right-clicking CS378_Lab1PawnBP and selecting “Copy Reference” ... and then adding “.C” to the path file, because Unreal reasons...

At this point, you can delete the existing ship in the scene and either 1) drag and drop the Blueprint version in or 2) place a Player Start object into the scene (the more correct way to spawn the ship).

At last we are to the part where you will explore UE4 on your own! Go ahead and make two functions in CS378_Lab1Pawn (the C++ version, not the BP child) to implement behavior for fire and dash. These functions will need to have `UFUNCTION(BlueprintCallable)` specifier so that we can call them directly from the Blueprint Event Graph. Once you create these functions, you can attach them to their respective Blueprint Event calls (why not do this all in C++ or all in Blueprint, you may ask, but the point is so that you understand how to work with both systems, which is the intended use-case of UE4).

**Adding Collision Overlaps**

Next you will create a more robust world object to intersect with the ship and projectiles. Right click within the fold where you’d like to place this new class and click “Add C++ class”. Create a new Actor and name it something reasonable (e.g. BoxActor etc). 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 either select the associated geometry through a child Blueprint or select it in the same way the Pawn and Projectile mesh are set via C++.

Your Actor class needs two components: a StaticMeshComponent and a PrimitiveComponent. This PrimitiveComponent should probably be a sphere or a box component based on the shape you pick. Create these components
in the BoxActor.h (look at the Pawn class for example code) and include the
correct component:
e.g. `#include "Components/BoxComponent.h"
You will then need to create the components as subobjects in the constructor
(again, use the Pawn 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 OnComponent-
BeginOverlap to the PrimitiveComponent instead of using OnComponent-
Hit as the Projectile class does. You will need to set up notifications so this
Component registers events (Projectile registers OnHit in the constructor, but
you may need to register BeginOverlap in BeginPlay).
Finally implement the actual callback code in a similar fashion as is present
in Projectile. If you want to test your search skills, try looking up OnCompon-
entBeginOverlap to figure out what the parameters are. If you decide to cheat,
they are as follows:

```c
(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
 UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep,
 const FHitResult &SweepResult)
```
At this point, print to screen using:
```c
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.
Feel free to add additional features to BoxActor (or the Pawn) and exper-
iment with working in C++ versus Blueprint. Make sure to test your code
by placing additional BoxActors into the scene and take video footage of your
gameplay (running in editor is fine) as well as screenshots of your BoxActor
code and additions to the Pawn class 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 (removing any
intermediate folders that are not the Source, Config, or Content folder), and
include the gameplay footage and code screenshots. If you are submitting through
a git repository, please go through GitLab and add me (thesharkcs) and the TA
(maohuaw) to the project.