CS378 Lab 3: Creating World Interactions

This lab will focus on setting up a scene containing multiple objects and using events and delegates to facilitate communication and interactions between them.

Getting Started

Upon launching UE4.25.x, create a Game Project then select “Blank” as a Template. For Project Settings, set the project to be C++ based. You should include Starter Content to simplify the scene creation, but if you feel comfortable working with UE4 lights and creating your own meshes, you do not need to include it. Name the project “CS378_Lab3” and click “Create Project”. Go ahead and connect this project to source control as you did for Lab 2 — you will be submitting this lab (and all future labs) through GitLab, so make sure you have git-lfs installed/a git ignore file/etc.

Creating a Scene

Open the “Minimal_Default” level under StarterContent->Maps and modify to your liking. We’ll be creating our own actors to interact with in the scene, so you can delete the furniture at this time if you’d like. I’d recommend keeping the lighting setup and the Player Start actor at minimum. If you are interested in learning the art/design side of Unreal, you are more than welcome to create your own level setup from scratch. There are many good tutorials on how to do that, but those go well beyond the scope of this class, so I won’t go into detail on how to work with those aspects of UE4.

Creating a Pawn with Actions

You will now create a basic Pawn class in much the same way you did in Lab 1, but now you’ll be making it from scratch. This Pawn class, called WorldPawn, will need a BoxComponent that will act as a basic hit box for your Pawn’s interactions. It optionally will need a StaticMeshComponent and a CameraComponent plus SpringArmComponent if you’d like to position the camera such that your pawn is visible (not required but may be helpful). You may notice that in the Twinstick Tutorial that the mesh is set via C++ in the Pawn class. You can do that for your pawn, or simply create the StaticMeshComponent and leave it unassigned in C++. This assumes you plan to assign it via Blueprint, and that’s totally acceptable.

This WorldPawn needs some basic player controls. Add WASD for movement plus an “Interact” button. The button you choose is not important — typical choices would be Enter, space bar, or E (really — just pick something). Much of this code will look roughly like Lab 1. Yes, you can copy sections of code over, but you will need to be careful to copy the correct parts (and update them to match your current project) rather than just wildly copying and pasting. In my personal opinion, it’s good to type things out while you’re learning what
stuff does, but a lot of this is relatively boilerplate so I won’t judge you if you’re trying to be efficient with your time (just make sure to compile frequently so you catch errors early).

Once you have created a basic Pawn and a Blueprint inheriting from your C++ class, set the Default Pawn in your Lab 3 Game Mode. You will also need to update:

Project Settings -> Project -> Maps & Modes -> Default GameMode to match your custom GameMode rather than using the Default.

Creating a Collision Actor

This task should also look pretty familiar after Lab 1. Create your own C++ class, called CollisionActor that inherits from AActor. Like the Actor you created for Lab1, the CollisionActor will have both a StaticMeshComponent and a SphereComponent to allow for interactions. Once you have the basic constructor in place, go ahead and create a Blueprint version of this Actor called CollisionActorBP and place it under Content in a new folder called Blueprints. You can set the mesh and the size of the collision volume in the Blueprint. Make the collision volume larger than the mesh so objects can enter the collision volume without colliding with the mesh. The mesh should Block on collision, and the collision volume should Overlap. This time, though, we will check for overlaps on the Pawn itself, so you don’t need to bind it to OnComponentBeginOverlap.

Creating a Pawn-Actor Interactions

Now that you have a Pawn with an Interact button, and an object to interact with, you will implement the Interact functionality such that when the pawn is overlapping with the CollisionActor. Create a function, PerformInteraction. In this function, you will now check for overlapping Actors every time the player presses the Interact button and handle the interaction based on the type of Actor. To do this, access the Pawn’s hit box’s GetOverlappingActors function. All Primitive Components have this function and one to check overlapping Components, but we’re going to check by Actor since we have not created multiple types of Components yet. If we wanted more granularity (for example, implementing hit boxes versus hurt boxes in a fighting game), we’d check GetOverlappingComponents but checking by Actor is fine for our purposes here.

Store all the overlapping Actors in a TArray and loop over them. If the Actor is of the CollisionActor type, go ahead and destroy that CollisionActor. This process will look something like this:

```cpp
TArray<AActor *> OverlappingActors;
HitboxComponent->GetOverlappingActors(OverlappingActors);
for (AActor * actor : OverlappingActors)
{
    if (actor->IsA(ACollisionActor::StaticClass()))
    {
        ....
    }
}
```
Remember that you’ll need an additional include to access information `CollisionActor` information.

Creating a Trigger Delegate

For this next type of interactive Actor, we're going to use UE4's C++ based Delegates. Start by creating a C++ class called `TriggerActor` that inherits from `AActor`. Give it the a `StaticMeshComponent` and a `SphereComponent` like you did for the `CollisionActor` and create a Blueprint version, `TriggerActorBP` as well. If you are thinking ”should these classes inherit from a shared parent class since they’re mostly the same?” that is a good observation, but a more involved architecture isn’t necessary for such a small testbed project, so you do not have to worry about the software architecture quite yet.

Next create another Actor, `ResponseActor`, that just has a `StaticMeshComponent`. Interacting with `TriggerActors` will cause changes in the associated `ResponseActors`. To do this, start by adding a Dynamic Multicast Delegate to `TriggerActor` with the macro: `DECLARE_DYNAMIC_MULTICAST_DELEGATE(FTriggerDelegate);` before the class declaration in `TriggerActor.h`. You are creating your own Delegate called `FTriggerDelegate`, which is not necessarily the best name, but it’ll be fine in this case. Somewhere in public go ahead and create an instance of this delegate:

```
FTriggerDelegate OnTriggerDelegate;
```

That’s it for `TriggerActor`. Next, add functionality to your `WorldPawn` so that when the Pawn interacts with a `TriggerActor`, it will call the `FTriggerDelegate`. You must access the delegate itself via the `TriggerActor` instance it belongs to. Once you do, the delegate will have a `Broadcast()` function it can call. You can safely call this without having a listener bound to it, because this assumes a multicast model rather than a one-to-one communication.

To finish the delegate’s communication, you now need to bind this delegate to a response function. This response function will be in `ResponseActor`, and you should use the `UFUNCTION` specifier when declaring it, so it’ll be recognized by the UE4 system. Dynamically bind this function in `BeginPlay` as you have done in Lab 1, but you’ll notice you need a pointer to the `TriggerActor` instance, so it can properly associate the calling delegate instance with the receiving object instance.

To make this work in a relatively flexible way, you’ll add a pointer in `ResponseActor` to a `TriggerActor`. Give this a UPROPERTY specifier that includes “BlueprintReadWrite” and “EditAnywhere.” We do not need to initialize this `TriggerActor` in the constructor — we’ll connect it via the `ResponseActorBP` in the level itself. Once you drop a `ResponseActorBP` into the scene, you can set its `TriggerActor` pointer to a `TriggerActor` also in the scene. This provides a way to easily assign and reassign the delegate interactions, but note (as always) that if you call on functionality from a pointer that may be null, you should always check that the pointer exists before accessing it!

Assuming this works and doesn’t crash, have the `ResponseActorBP` change its location in the scene when the delegate is called (or, more advanced, use a
Timeline if you’re feeling fancy and want proper interpolation). Try it out with multiple ResponseActorBPs and multiple TriggerActors. After you’re satisfied, collect some video footage, screenshot exciting parts of your code, and submit these files plus the project code via your GitLab account. Link to this repository as your Canvas submission.