CS354p 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. You will also be setting up a project from the blank template to better understand the “boilerplate” setup that should be done before all future projects.

Getting Started

Upon launching UE5.2, 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 “Lab3” and click “Create”. 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 .gitignore/a .gitattributes 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 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 Unreal.

Creating a Pawn with Actions

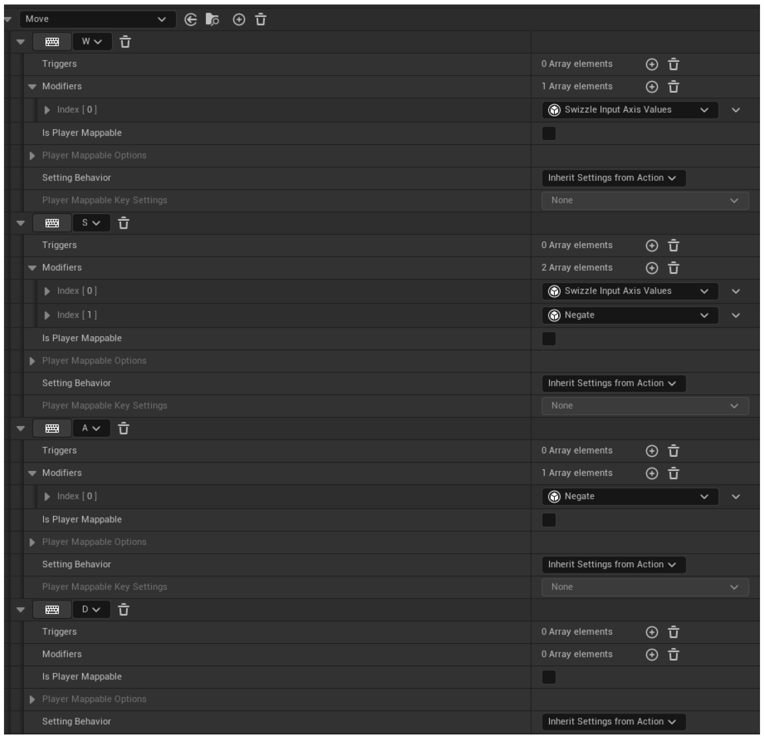
You will notice you have basic controls. If you look under Edit -> Project Settings -> Maps and Modes, you’ll see the default Pawn is ADefaultPawn. We will build our own Pawn in a similar way to what was built for you in Lab 1, but this time, you’ll be making it from scratch. Our Pawn class, called `WorldPawn`, 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 are encouraged to create these components in C++ then modify them via Blueprint.

This `WorldPawn` needs some basic player controls, and we are going to implement a slightly simplified way of connecting controls to a Pawn that bypasses extending the PlayerController. Note that there’s no right or wrong way to approach this — it just depends on the system you’re implementing. To get started, we first need to add the “Enhanced Input” module to our included

`PublicDependencyModuleNames` in `Lab3.Build.cs`. This gives us access to Enhanced Input functionality which UE5 assumes to be the preferred way of handling player inputs.

Next, we will add WASD controls 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), but we'll need to create an InputMappingContext and two Data Assets – Interact and Move — in our Content as we did in Lab 1. Interact will stay the default bool as its Value Type. Move needs a Value Type of Axis2D. This allows us to take in input across two axes (in this case, forward/backward and left/right or along the X and Y axes).

You now need to set your button inputs in the InputMappingContext. This is a little more complex than last time, since we'll be mapping WASD to Move. This one input can handle all of these buttons, because it's an Axis2D and takes inputs as a range between -1.0 to 1.0. W and D will be positive, while S and A will have one of their modifiers set to "Negate." We also need to handle inputs along separate axes (i.e. WS will be on the Y axis while AD will be on the X axis) so add the "Swizzle Input Axis Values" Modifier to W and S. It will look something like this:



```
Move
  W
    Triggers                      0 Array elements
    Modifiers                     1 Array elements
      Index [ 0 ]                 Swizzle Input Axis Values
    Is Player Mappable
    Player Mappable Options
    Setting Behavior              Inherit Settings from Action
    Player Mappable Key Settings  None
  S
    Triggers                      0 Array elements
    Modifiers                     2 Array elements
      Index [ 0 ]                 Swizzle Input Axis Values
      Index [ 1 ]                 Negate
    Is Player Mappable
    Player Mappable Options
    Setting Behavior              Inherit Settings from Action
    Player Mappable Key Settings  None
  A
    Triggers                      0 Array elements
    Modifiers                     1 Array elements
      Index [ 0 ]                 Negate
    Is Player Mappable
    Player Mappable Options
    Setting Behavior              Inherit Settings from Action
    Player Mappable Key Settings  None
  D
    Triggers                      0 Array elements
    Modifiers                     0 Array elements
    Is Player Mappable
    Player Mappable Options
    Setting Behavior              Inherit Settings from Action
```

We can now create our `UInputMappingContext` and our `UInputActions` in the C++ WorldPawn directly rather than going through a Controller. It will look very similar to what is in Lab 1 but now in the Pawn directly. Create two events, `MoveEvent` and `InteractEvent`, that will be the callbacks when a Move button or Interact button is pressed. You will need to add some functionality to `BeginPlay` that handles the input through the default PlayerController (as we are no longer extending a PlayerController to build it out there), and bind our actions to callbacks in `SetupPlayerInputComponent`. Finally, include the movement handling in the `MoveEvent` and some placeholder print in the `InteractEvent` to ensure they're both connected.

Make sure you add the InputMappingContext, and the Move and Interact inputs to your Blueprinted Pawn then set the Default Pawn in `Lab3GameMode` class, which inherits from `AGameModeBase`. You will also need to update:
`Project Settings -> Project -> Maps & Modes -> Default GameMode` to match your custom GameMode rather than using the Default. If everything is working, you should be able to move the Pawn (in a character relative way, so it may be a bit awkward depending on your character and camera orientation) and see a print statement on the screen when you press Interact.

I've uploaded the code for WorldPawn, Lab3GameModeBase, and the Lab3 build to Canvas for reference to help with setup, but please try to write the code yourself rather than copy and pasting! It's critical that you understand all the lines of code we've written even if you're referencing a working solution while doing so.

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 `BP_CollisionActor` and place it under Content in your Blueprints folder. 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 that runs when the pawn is overlapping with the `CollisionActor`. In your function, `InteractEvent`, 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:

```
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 `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 has a `StaticMeshComponent`. This Actor doesn't need a hitbox, because Interacting with a `TriggerActor` will affect 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 `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. Note that this is per instance not at the class level! 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 (if it does, check that you included the correct specifiers etc), 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.