CS354p Lab 8: Building a Data-Driven GUI

This lab will explore the basics of building out a GUI in a data-driven way. This will involve bringing in .csv data, working with the Game Instance for persistent storage and data handling, then connecting all of this information to a UMG Widget to display it to screen. There are a lot more software design considerations you'll want to account for in a larger game, but this will provide the requisite knowledge to get started!

Getting Started (Complete Before Class)

Create a blank C++ Unreal project called `Lab8` as usual. You won't need any starter content, and you won't need to see anything in the actual game, but I'd recommend creating a simple level for loading purposes. All the level needs is a plane to place your Player Start capsule and a directional light to make it easier to work with in the editor. Save this map and make it the default.

Next, go to the Canvas page and download all the assets under Files ->Lab Material ->Lab 8. Create a folder in your Unreal project under Content called NonAssets and place these files (a .jpg and two .csvs) there. If the Unreal project asks you if you want to import them, for now, say no — we're going to handle it during the lab itself.

Make sure to set up git to include these assets (the .jpg can be committed using lfs but the .csvs should be treated as regular text files), but otherwise, you're ready for the lab!

Importing String Tables

The first thing we need to do is modify the `Lab8.h` and `Lab8.cpp` to handle loading and unloading String Tables, which is where we store all the text in the game for localization. In this case, we have one .csv called `Lab8.csv` which contains all the strings we'll need for our dialogue demo. If you open up this file, you'll see it contains header information for two columns, `Key` and `SourceString` and one row of information that will act as our test dialogue.

To do this, we will first add our own class definition to `Lab8.h`, which will have two methods, `StartupModule()` and `ShutdownModule()`. It should look something like this:

```
class FLab8 : public FDefaultGameModuleImpl
{
public:
    virtual void StartupModule() override;
    virtual void ShutdownModule() override;
};
```

Create the method definitions in `Lab8.cpp`. You will also need the following include:

```
#include "Internationalization/StringTableRegistry.h"
```

And the parent function call will look something like this:
`FDefaultGameModuleImpl::StartupModule();`

Next, you'll need to load in your table data using the `LOCTABLE_FROMFILE_GAME` macro. This macro registers a string table using a table key, a table namespace, and a path to the .csv with the values you're bringing into the table. Ours will look something like this:
`LOCTABLE_FROMFILE_GAME("Lab8Strings", "Lab8Strings", "NonAssets/Lab8Strings.csv");`

This must be done on module startup. We'll unregister any tables we've loaded via the table key during shutdown:
`FStringTableRegistry::Get().UnregisterStringTable("Lab8Strings");`

Finally, we must update IMPLEMENT_PRIMARY_Game_Module to look like this:
`IMPLEMENT_PRIMARY_GAME_MODULE( FLab8, Lab8, "Lab8" );`

i.e. we want to use the module we just defined rather than the default. Note that this macro must run AFTER the functions above have been defined! Assuming this compiles, we'll start on the next part, but we won't be able to confirm this is fully working until a bit later...

Creating Data Handlers

Next, we're going to create some files to assist us in data handling. We're going to create two files: `Lab8Structs`, and `Lab8DataAssets`. These will help us with managing the data within `Lab8Dialogue.csv`, which includes how we will pair who is talking with the dialogue they're saying with the portrait we want to display. Note that we need to also setup a UMG Widget to accomplish this, but for now, let's just focus on bringing in the data.

Structs

We will first create a .h to hold all the structs we'll be using in our game (one in this case). To do this, create a C++ class that inherits from UObject called `Lab8Structs`. We can just ignore the .cpp. For `Lab8Structs.h`, go ahead and remove all the class information.

We need to add the following include:
`#include "Engine/DataTable.h"`

Next, create a struct that will handle our `Lab8Dialogue.csv` info. Since this .csv essentially contains dialogue information, we'll name our struct `FDialogueStruct`, and it will inherit from type `FTableRowBase`. We want our USTRUCT to be a BlueprintType so it is accessible viable Blueprints, and we also need to include

the macro `GENERATED_USTRUCT_BODY()` so that it will play nicely in the UObject runtime system.

Next, we need to create UProperties for each of the columns in the .csv excluding the first RowName column (which is used for searching through the data). You MUST create a UProperty for each column, and the name must exactly match the name in the .csv. Types can vary (e.g. you can load this column data as booleans, ints, floats, etc), but in our case we're going to treat all the column data as "String" data, which means we will import it as FName data.

As an example, the first column's field can look something like this (though the UProperty specifies can vary):

```
UPROPERTY(EditAnywhere, BlueprintReadOnly)
FName Dialogue;
```

The last step for completing `Lab8Structs.h` is to create a constructor for the struct. You can assign all field values to "None" (i.e. `FName(TEXT("None"))`).

Now that we have a data structure to reference when we bring in our .csv, we can drag the `Lab8Dialogue.csv` into the editor's Content Browser. It will ask what type we'd like to bring the CSV in as, so we'll select `DialogueStruct`. Once that data's imported, you can open it to see the contents within the editor. If you update this csv, the editor will automatically ask if you want to reimport the data, and when you do, you'll see these updates reflected in the editor.

Data Assets

Data Assets provide a clean way to bring in additional information and manage project assets without directly referring to the file in the directory by path (which is extremely fragile in practice). We're going to try this feature out by creating a small "portrait handler" which will associate portrait data (in this case, just Dracula's) with the FName associated with it via the Portrait column in our `Lab8Dialogue.csv`.

We will create a C++ class of type `UDataAsset` called `Lab8DataAsset`, and all we need to do is declare any data structures we want this asset to hold. Since all we need this time are the portrait images and their associated keys, the only thing we need to add for now is a `TMap` that contains an `FName` key associated with a `UTexture2D*` value.

Once this is compiled, you can now create a DataAsset in the editor and make it of type `Lab8DataAsset`. We'll name it DialogueAsset (since that's what it contains). You can now open up this asset and add a key/value pair. The key should exactly match the value(s) listed in the Portrait column of `Lab8Dialogue.csv`, and you can drag and drop the .jpg provided for this project into the editor's Content Browser to create a texture of it. Once that's in the project, you can add this as the value to your key.

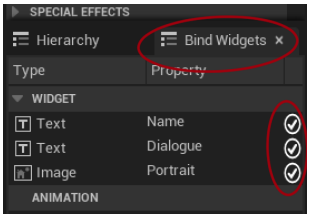
Creating a User Widget

We're not *quite* ready to bring in the data, but we're getting there...let's take a break from that to mock up a Widget first, though. First, create `Lab8DialogueWidget`, which inherits from `UserWidget` in C++. This widget must contain any child widgets that we want to hook up via C++, which will include two TextBlocks to hold the Speaker and Dialogue respectively as well as an Image to display the speaker's portrait.

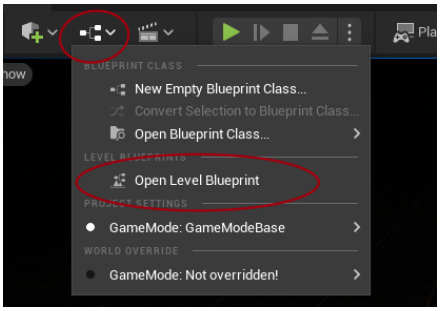
Thus, we'll declare three properties: Name, Dialogue, and Portrait in `Lab8DialogueWidget.h`. In order to bind these correctly to the Blueprint UMG, we must include the BindWidget specifier in our UProperty macro. It will look something like this:

```
UPROPERTY(BlueprintReadWrite, meta = (BindWidget))
UTextBlock * Name;
```

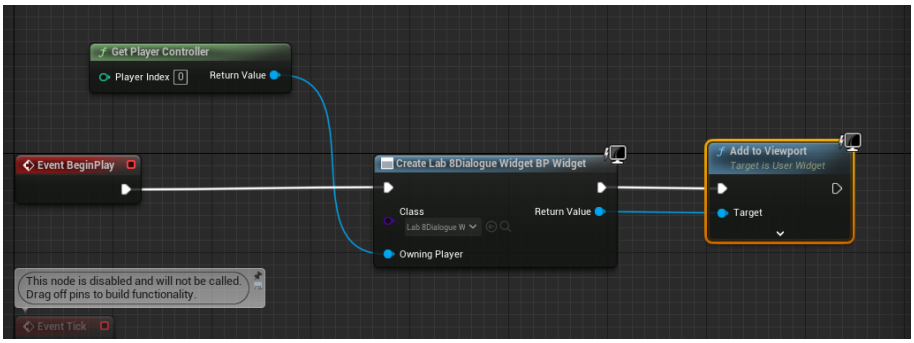
In practice, we'd want to make RichTextBlocks rather than TextBlocks, but that requires additional setup time we definitely don't have for this lab. Once that's done, we will switch back to the editor and create a new Widget Blueprint from the drop down under the User Interface tab. When it asks you for the type, choose `Lab8DialogueWidget` and name it `Lab8DialogueWidgetBP`. Add in the correctly named and typed widgets to match what's in the BP. You'll be able to check if they're set up correctly by looking at this bindings tab under the Palette tab for the UMG Designer.



Once that's done, we'll hook up the widget to the controller via the Blueprint level for simplicity. Open the level BP by clicking the little graph button at the top. It will look something like this:



Once that's in place, add the following BP nodes to the BeginPlay exec node:



You should now see your Widget displayed on the screen when you hit play!

Creating a Game Instance

You might be thinking at this point, "surely we're almost done, right?" and by some definition of "almost" that's probably true! But no, not really. We're finally to the important stuff. Namely, now it's time to create our own Game Instance so we can handle asynchronous loading and persistent data storage (both essential things to do correctly on larger projects).

The last C++ class we need to create is `Lab8GameInstance`, which will inherit from GameInstance. Once you do this, you will need to go into Maps and Modes under Project Settings and set the Game Instance default class to `Lab8GameInstance`.

We want to load our dialogue dynamically at runtime, so we're going to override `Init()` (and usually `Shutdown()` to handle any clean up, but we don't need to worry about that for this project). We're also going to create a function `void LoadDialogueData(FSoftObjectPath DialogueToLoad)` and we're going to create a variable `DialogueData`, which is of type `UDataTable *`, and it's where we're going to actually store the .csv data at runtime.

The `LoadDialogueData` function will follow a very boilerplate form. It will look like this:

```
void ULab8GameInstance::LoadDialogueData(FSoftObjectPath DialogueToLoad)
{
    TSharedPtr<FStreamableHandle> AssetHandle =
    UAssetManager::GetStreamableManager().RequestAsyncLoad(DialogueToLoad);

    if (AssetHandle)
    {
        UDataTable * ReturnedTable = Cast<UDataTable>(AssetHandle->GetLoadedAsset());
        if (ReturnedTable)
        {
            DialogueData = ReturnedTable;
        }
    }
}
```

What this does is create a streamable handle to allow asynchronous loading of the data table. If the Asset Manager is able to retrieve the data, it then

pulls out the data and, in our case, assigns it to the `DialogueData` pointer we created earlier. Note that the code within the `if (AssetHandle)` block is an **asynchronous** callback, so we don't know exactly when `AssetHandle` will be populated. Thus, if you call `LoadDialogueData` in `Init` (which we will do!) you cannot this pointer will be populated before the next call in `Init`. Asynchronous function calls are extremely common in event driven systems, so developing a good mental model of concurrency is essential.

You will also need to include `Engine/AssetManager.h` and `Engine/DataTable.h` to get it to compile. You can now call on this function in `Init`. You can create the Soft Object Path (e.g. a soft reference to an asset) via `FSoftObjectPath DataPath(TEXT("DataTable'/PathToData/Lab8Dialogue.Lab8Dialogue'"));` where Lab8Dialogue is the Data Table you created in the editor.

Connecting the Data to Widgets

At long last, we're ready to actually populate our widget! We need to populate both the text and the texture, but let's start with the text just to confirm it's working (though just fyi, there will be a bug with our approach the first time we load in the data, but we'll come back to fix it after we get this part connected).

Let's start by creating a `BlueprintCallable` function called `SetupDialogue()` in `Lab8DialogueWidget`. We're doing this so we can kick the dialogue loading off via the level BP for convenience (so make it public), but there are many other ways to hook this up depending. This will be the actual function that populates the three widgets, so we'll need to get the correct `FDialogueStruct` in order to get that information. Note that there is only one row in our example, but there will be many more in practice. Since there is only one row, the Row Name we're looking for is 1.

Regardless, we need to search through the `DialogueData` variable stored in `Lab8GameInstance` to find the row that matches to this Row Name key. Let's create a helper method in `Lab8GameInstance` to assist with that process. For the sake of simplicity, our method signature will be `FDialogueStruct GetDialogueStruct()`, and we'll hardcode our Row Name into this method rather than passing it in as an argument. This is a pretty bad idea, but we'll do it for now, since this lab already has enough going on.

All we need to do is call `FindRow` from `DialogueData` then return it to whoever is querying for that data. The function implementation will look like this:

```
FDialogueStruct ULab8GameInstance::GetDialogueStruct()
{
    if (DialogueData)
    {
        FDialogueStruct * DialogueStruct =
        DialogueData->FindRow<FDialogueStruct>(FName(TEXT("1")),
            FString(TEXT("Get Dialogue Struct Called")));
        if (DialogueStruct)
            return *DialogueStruct;
    }
    return FDialogueStruct();
}
```

Once you get that compiled (be sure to include `Lab8Structs.h`, let's access this function from `Lab8DialogueWidget`'s `SetupDialogue` function to retrieve our data!

Displaying Text

First, we need to include `"Kismet/GameplayStatics.h"` and `Lab8GameInstance.h` so we can access the GameInstance. We're going to access the GameInstance by calling `GameplayStatics::GetGameInstance(GetWorld())` and then cast that value to our `Lab8GameInstance`. We can then access the `FDialogueStruct` that's returned from `GetDialogueStruct()` to pull out the FNames associated with the individual values.

You can call `SetText` from the TextBlock widgets to set them to FText values (not FNames, so you need to call the function `FText::FromName()`). You'll also notice the dialogue isn't correct (since the value stored in `Lab8Dialogue.csv` is actually a key into the string table. Thus we need to access the string table using the function `FText::FromStringTable()`. All together, the function will look like this:

```
void ULab8DialogueWidget::SetupDialogue()
{
    ULab8GameInstance * GameInstance =
        Cast<ULab8GameInstance>(UGameplayStatics::GetGameInstance
        (GetWorld()));
    if (GameInstance)
    {
        FDialogueStruct DialogueStruct =
            GameInstance->GetDialogueStruct();
        FName DialogueSpeaker = DialogueStruct.Speaker;
        FName Key = DialogueStruct.Dialogue;
        FText DialogueText = FText::FromStringTable("Lab8Strings",
            Key.ToString());

        Name->SetText(FText::FromName(DialogueSpeaker));
        Dialogue->SetText(DialogueText);
    }
}
```

Don't forget the necessary includes: `Lab8Structs.h` and `"Components/TextBlock.h"`, and then hook this into the Level blueprint after creating the widget to test that it's working. The last step is to add the image data!

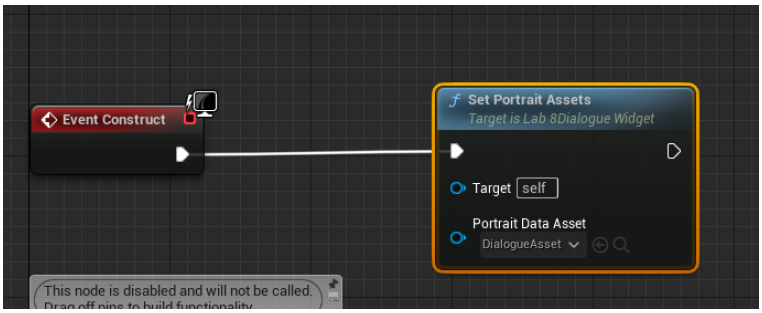
Displaying Images

We've come this far using reasonable-ish code practices, so let's handle the images in a mildly intelligent way as well. In this kind of system, we can probably assume we'll display any given image several times over the course of a level, so let's start by caching our image data in TMap `PortraitMap` in `Lab8DialogueWidget`. Note that TMaps aren't garbage collected in Unreal, so you should implement `NativeDestruct()` to remove elements from the TMap.

We'll also create a function `void SetPortraitAssets(ULab8DataAsset * PortraitDataAsset)`, which will bring in `DialogueAsset` and pull out the data into `PortraitMap`. That functionality will just look like this:

```
for (auto& Elem :  PortraitDataAsset->Portraits)
{ PortraitMap.Emplace(Elem.Key, Elem.Value); }
```

Now that you've loaded the FNames and Textures into `PortraitMap`, you can access the correct image to display in `SetupDialogue` using the `FindRef` function for `PortraitMap`. This should return a `UTexture2D *` which you can then set to be the image displayed for the `Portrait` binding by calling `SetBrushFromTexture`. You can call this function within `Lab8DialogueWidgetBP`'s Event Graph or from the Level BP. Either way is fine given our current setup! If you do it from your `Construct` event in in the widget directly, it will look something like this:



If everything's working, congratulations! You're done with your whirlwind tour of data management in Unreal! There are a couple caveats I've ignored, but this should be enough to start digging into additional data/UI functionalities!

Submission

After you're satisfied, take a screenshot of your GUI displaying the correct information. 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.