Module 4 – Set up the First the Multiplayer Environment

COMP280-Multiplayer Game Programming

Purpose: Set up the First the Multiplayer Environment

Contents

1 Intro
1.1 Technical requirements

2 Starting your Unreal multiplayer project
2.1 Creating your project file
2.2 Creating the project game instance
2.3 Creating the Game Mode and the Game State
2.3.1 Create Game Mode
2.3.2 Create Game State

3 Adding the player classes
3.1 Compiling the source code
3.1.1 Compile the project
3.2 Creating the Character Blueprint Class
3.3 Adding the Player classes to the Game Mode
3.3.1 Chack that US_GameMode is correct

4 Summary

1. Intro

We will:

- Create an Unreal project and get things set up properly. This involves:
  - Creating the Gameplay Framework (GF) classes (so that you have access to all the necessary features needed for development)
  - Configuring the project settings to use such classes.
- Setup the foundation for the multiplayer game.

1.1. Technical requirements

- We are going to use Unreal Engine 5.4.
- Some basic knowledge about C++ programming

2. Starting your Unreal multiplayer project

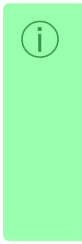
2.1. Creating your project file

Let's start by creating a blank project:

1. Open the Epic Games Launcher and launch the Unreal Editor.
2. From the Games section, select the **Blank template**.
3. In Project Defaults, select **C++** as the project type.
4. Make sure the Starter Content field is **unselected** as you won't be using it.
5. Name the project **UShadows_LOTL_{YourInitials}** or simply $US\_LTOL\_\{YourInitials\}$.
6. Click the **Create** button.
7. Once the UE project has been created, get the attached **Ch4_UnrealShadows-StarterContent.zip** file and unzip it in your computer.
8. Navigate to your project Content folder located at **[Your Project Path]** $\rightarrow US\_LTOL\_\{YourInitials\} \rightarrow$ **Content**.
9. Copy the content of the unzipped file (*ExternalActors*, Blueprints, KayKit, and Maps folders) into the Content folder to add all the needed assets to your project.
10. Check that they appear in the UE Editor and be available in your project.
11. If they do not,close the Editor and open it again to let UE update the Content folder.
12. Notice the two levels (Level_01 and Level_Boss) to the Maps folder.
13. Open the Level_01 map and check that everything is OK.
14. Open the Level_Boss map and check that everything is OK.
15. $Take\ Snapshot$

Let's add the main classes used in any UE5 project, the ones that extend the GF (GameFramework) elements.

2.2. Creating the project game instance



Definition

A **GameInstance** is a class responsible for **managing high-level** data that needs to persist across level changes or game sessions. It is essentially a globally accessible UObject that can store any data you want to keep across the entire game, such as the player score, and other information that needs to be shared across different levels or game sessions.

A class extending a GameInstance can be created as a Blueprint Class or in C++ and is instantiated when the game is started and is only destroyed when the game is shut down.

IMPORTANT NOTE

You will be working with a mix of C++ classes and Blueprints. C++ classes are located in the **All** $\rightarrow$ **C++ classes** $\rightarrow US\_LTOL\_\{YourInitials\}$ folder and can only be added there (or in a subfolder). If you can't find this folder, you have probably created a Blueprint-only project. It's OK, once the first C++ has been created (more on this in a minute), the Unreal Engine Editor will take care of it, transforming your Blueprint-only project into a C++ one and everything will be in place!

To create your project GameInstance, follow these steps:

1. On the main menu, select **Tools** $\rightarrow$ **New C++ Class**....
2. The **Add C++ Class wizard** will open, showing the **CHOOSE PARENT CLASS** section. Select the **All Classes** tab and, in the search field, type **game instance**.
3. Select the **GameInstance** class.
4. Click Next to access the **NAME YOUR NEW GAME INSTANCE** panel.
5. In the Name field, insert **US_GameInstance**. You can leave the other fields as they are.
6. Click the **Create Class** button to generate your class files.
7. If you didn't have a **All** $\rightarrow$ **C++ Classes** folder, notice that you have one now, with a US_Gameinstance item in it.
8. Double-click on the **US_GameInstance** icon;
9. In Visual Studio notice the two new files: **US_GameInstance.h** and **US_GameInstance.cpp**.

10. Pat yourself in the back! – you have just created your (maybe) first Unreal C++ class!
11. Open the header file you will see s.th. like the following code (except with your suffix, not mine):

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "Engine/GameInstance.h"
#include "US_GameInstance.generated.h"

/**
 *
 */
UCLASS()
class US_LOTL_AT_API UUS_GameInstance : public UGameInstance
{
  GENERATED_BODY()

};
```

The source file will be empty, apart from the header #include declaration:

```
// Fill out your copyright notice in the Description page of Project Settings.
#include "US_GameInstance.h"
```

Notice the following:

- **US_** prefix stands for UnrealShadows. In your future projects you can use an appropriate prefix based on your game's name.
- This class extends the base GameInstance class (i.e., UGameInstance)
- Currently does nothing apart from the macro declarations.
- As the project progresses, new features such as data collection or online services management will be added to it.
- The API is named **{YourProjectName}_API**. Note it, so you can fix issues, as, in these instructions, you will see my project name (*US_LOTL_AT_API*).
- The class name is prefixed with an extra **U** (UUS_GameInstance). Recall the meaning of this.
- The base class is also prefixed with a **U** (UGameInstance).

This game instance needs to be added to the project settings so it will be the one instantiated at runtime. To do so, follow these steps:

1. From the main menu, top-right corner, open **Settings** → **Project Settings**. Then click on the **Project** → **Maps & Modes** section.
2. In the **Game Instance** → **Game Instance Class** drop-down menu (in the bottom), select **US_GameInstance**.

Now we have a game instance assigned to the project. It is time to create a **Game Mode** for the *menu* and *lobby* levels.

2.3. Creating the Game Mode and the Game State

For more information on Game Mode and Game State see Unreal Docs - Game Mode and Game State

2.3.1 Create Game Mode

Game Mode is a class that controls the game **rules**, such as how a player joins the game, how to transition between levels, and other game-specific settings. (In FSMs it corresponts to a *StateMachine* or *Context* - The states it deals with are **EnteringMap**, **WaitingToStart**, **InProgress**, **WaitingPostMach**, **LeavingMap**, and **Aborted**)

Game Mode is paired with **Game State** class, which manages the current **state** of the game, such as the *score*, *time remaining*, and other important information. (There are other classes that can be paired with the

Game Mode class, such as *Default Pawn* class, *HUD* class, *PlayerControllers* class, *Spectator* class and *Player State* class - we may use them later).

The Game Mode is typically paired with a Game State class and together, allow developers to create complex and customized game mechanics in UE.

- Create a new C++ class just like you did for the game instance in the previous sections (click **Tools** → **New C+= Class...**).
    - From the All Classes section, select **GameMode** and click **Next**.
    - Name your class **US_GameMode** and click the **CreateClass** button.
    - Check the class has been created.
- Set it as the default Game Mode for all your levels
    - From the main menu, top-right corner, open **Settings** → **Project Settings**. Then click on the **Project** → **Maps & Modes** section.
    - Click on the **Default Modes** → **Default GameMode** drop-down menu and select **US_GameMode**

2.3.2 Create Game State

Let's create a Game State:

- Create a new C++ class just like you did for the game instance in the previous sections (click **Tools** → **New C+= Class...**).
    - From the All Classes section, select **GameState** and click **Next**.
    - Name your class **US_GameState** and click the **CreateClass** button.
    - Check the class has been created.
- Declare **Game State** inside the **Game Mode**, so that they are interconnected and fully functional in the level. To do this:
    - Open the **US_GameMode** files (.h and .cpp) by double-clicking on it.
    - Declare a constructor inside **US_GameMode.h** by adding these two lines of code:

- ```
  public:
  AUS_GameMode();
  ```

    - Implement the constructor inside **US_GameMode.cpp** by adding this piece of code:

- ```
  #include "US_GameState.h"
  AUS_GameMode::AUS_GameMode()
  {
     GameStateClass = AUS_GameState::StaticClass();
  }
  ```

    - Notice the prefix **A** in both files. Recall why is needed.

> The previous code declares the Game State class for US_GameMode to be the previously created US_GameState. This declaration can be performed also in child Blueprints; this will enable class switching through a drop-down menu in the Editor. For us, being programmers, hence code-oriented, it may be preferable this code solution.

Recap: we have created three classes (**US_GameInstance**, **US_GameMode** and **US_GameState**) and have wired them up. These classes are almost empty, but will get filled gradually in the future. In the next section, we'll be creating the classes needed to handle the character **input** and **presence** in the game.

3. Adding the player classes

We will create the following classes:

Class	Extends	Role / Description
US_PlayerController	PlayerController	Manages **input**, sends **commands** to the character
US_Character	Character	Represents the **player** in the game
US_PlayerState	PlayerState	Holds **info** about player(s) (**XPs**, **score**, etc.)

Let's create these three classes:

1. Create a C++ class extending **All Classes** → **PlayerController** and name it **US_PlayerController**.
2. Create another C++ class extending **Common Classes** → **Character** and name it **US_Character**.
3. Finally, create a C++ class extending **All Classes** → **PlayerState** and call it **US_PlayerState**.

> ⓘ You may have to click **Reload All...** in Visual Studio to see these classes there.

These three classes should be added to the Game Mode, the same way the Game State was added, but first, to get a bit more flexibility to our Character class, we'll create a Blueprint from it. To get a Blueprint out of a newly created C++ class, you need to compile the project. You are now going to compile your source code for the first time to check that everything has been properly set.

3.1. Compiling the source code

Compiling is an essential step in the development process. UE provides tools to streamline the compiling process and improve the development experience. So far we have leveraged UE's **Live Coding**:

> ⓘ **Live Coding** is a feature that enables the game's C++ code to be rebuilt and its binaries patched while the UE engine is running.

Live Coding features:

- Is enabled by default.
- You can disable/enable it via **Compile options menu** (the vertical tripple dot in the bottom-right)
- Modification of C++ classes, compilation them, and observation of changes while the Editor is running.
- Great for iterative development, especially when using C++ runtime logic such as gameplay code or frontend user interactions.
- Once enabled, you can start a Live Coding build via **Ctrl + Alt + F11**.
- To start the compilation process without Live Coding, disable it, and click the Compile button.

3.1.1 Compile the project

- Press **Ctrl + Alt + F11** or Click the **Compile** Button.
- Check for a success message (**Compile Complete**).
- Otherwise, fix the errors.
- If success, go to next section (create a Blueprint out of your Character class).

3.2. Creating the Character Blueprint Class

As you'll be working on some customization for your character later on, it's essential to have the extra flexibility provided by a Blueprint Class. Successfull compilation means,among other things, that all .generated.h files are there.

Let's test the correct working of Blueprints with these classes:

1. Navigate to the **Content** → **Blueprints** folder.
2. Create a Blueprint Class that will inherit from **US_Character** and name it **BP_Character**.

3. Save and close the Blueprint: you are not going to do anything yet on it.

This new Blueprint should be added to the Game Mode as the default Pawn to be used during a game session. Unfortunately, a Blueprint Class cannot be directly referenced in a C++ class. This means you'll have to find it through a method called **FClassFinder** available in the **ConstructorHelpers** utility class.

3.3. Adding the Player classes to the Game Mode

You are now going to declare the newly created classes to the Game Mode. Let's open again the **US_GameMode.cpp** file and add some code logic. In the declarations section, add the following block of code:

```
#include "US_PlayerController.h"
#include "US_PlayerState.h"
#include "US_Character.h"
#include "UObject/ConstructorHelpers.h"
```

This will declare all the GF classes you will be declaring, and the ConstructorHelpers utility class. Then, before the closing bracket of the constructor, add the following block of code:

```
PlayerStateClass = AUS_PlayerState::StaticClass();
PlayerControllerClass = AUS_PlayerController::StaticClass();
static ConstructorHelpers::FClassFinder<apawn> PlayerPawnBPClass(TEXT("/Game/Blueprints/BP_Character"));
if (PlayerPawnBPClass.Class != nullptr)
{
   DefaultPawnClass = PlayerPawnBPClass.Class;
}
```

As you can see, the first two lines of code declare PlayerStateClass and PlayerControllerClass in a similar way as you did in the previous section for GameStateClass. Meanwhile, retrieving a Blueprint reference (i.e., PlayerPawnBPClass) from a C++ class works differently from how it works for a regular C++ class: you need to hardcode a path to your project. This may be not an ideal solution because files can be moved around or deleted but, well... it works! Just keep in mind that my file path (i.e., "/Game/Blueprints/BP_Character") may be slightly different from yours, depending on your folder organization. Now that the Game Mode class has been modified, click the Compile button in the Unreal Editor.

3.3.1 Chack that US_GameMode is correct

1. Open the **Project Settings** → **Maps & Modes** section.
2. Locate the Selected **GameMode** field and **expand it** by clicking the small arrow next to it.
3. Check that the GF classes we have created are all correctly allocated

In this last section, you have completed the Game Mode setup by adding all the GF classes you'll be expanding in the next chapters.

4. Summary

In this Lab, we:

- Overviewed the project you'll be developing: a multiplayer stealth game involving thieves, secret treasures, and a ton of undead minions.
- Overviewed the main topics of the Unreal Engine C++ "dialect.", especially:
  - memory management: Unreal Engine flawlessly handles it (GC - UE Garbage Collector).
  - Reflection: this allows to add decorations to your classes, variables, and functions; expose them to the Blueprint system, letting your project be more flexible and accessible to non-code-oriented developers.
- We created the main classes that will be used by the game and that extend those offered by the GF.
  - US_GameInstance
  - US_GameMode
  - Other player-oriented elements.

This is a solid base to start developing the multiplayer project.

Next week, we will:

- create the player character by presenting how to manage it in a multiplayer environment.