COMP280-Multiplayer Game Programming

Lab 3 - Test the Multiplayer System with a Project Prototype

Contents

1. Purpose: Creating a Project Prototype to Test the Multiplayer System

By th eend of the Lab you will have an introduction to the main UE multiplayer framework features and how to test them on a single computer. By the end, you will have created your first multiplayer prototype and will be ready for the next step, which is creating a fully working networked game from scratch.

1.1. Technical requirements

We are going to use Unreal Engine 5.4.

2. Creating a Multiplayer Game Prototype

2.1. The Game Rules

- Each player should control their character
- The server will spawn item pickups at random positions
- Players will capture pickups and gain points from that
- The game will go on indefinitely

Note: In the following whenever you see *{YourInitials}*, substitute for your initials; for example for me it would be *AT*. **Note2:** In the following try to take snapshots in relavant places (have a word .docx ready on the side, named Lab3_Snapshots_{YourInitials}.docx).

2.2. Setting Up the project from a template

- Launch UE 5.4
- Select the template **Games->TopDown**
- Select **Blueprint**
- Name the project **TopDown_MultiPlayer_{YourInitials}**
- Leave other settings to their default values.
- Click the **Create** button.

> **Note**
>
> The template comes with a map (level) with some props (cylinders, ramps, cubes that form walls and blocks, quarter-cylinders that form rounded part of platforms).

- *Take Snapshot*

2.3. Obtain some assets for the props

- You could use the **Quixel Bridge** (if logged in with Epic Account you have free access)...:

    - Open Quixel Bridge by clicking *Quickly add to the project → Quixel Bridge*.
    - From the 3D Assets section, look for some *fruit or vegetables* – or anything that will spark your imagination!
    - Download the assets and add them to your project by clicking the *Download* button.
- Or use **built-in** assets (cubes / spheres etc — we are not interested at this point at polish)...:
    - Go to *Quickly add to the project → Shapes → Cube* to add a Cube in the current map.
    - Experiment with other shapes as well (*Sphere, Cone, Cylinder and/or Plane*).
- *Take Snapshot*

2.4. Resizing Assets

Once obtained, some assets may need resizing. To do so:

- Create an Empty Level: **File->New Level**, Select **Basic** template, Press **Create** button.

> **Refresher**
>
> Notice in the *Outliner* that the new *Basic* level comes with *Lighting* (folder), *Floor* and *PlayerStart* reference point (Capsule Collider). The *Lighting* folder comes completed with *DirectionalLight*, *ExponentialHeightFog*, *SkyAtmosphere*, *SkyLight*, *SkySphere* and *Volumetric Cloud*. Whew!

- Add a reference character: Go to **Content → Characters → Mannequins → Meshes** and drag'n'drop SKM_Quinn in the level.
- Add your assets (cubes, spheres etc) nearby the reference character.
- Scale them to about a third of the height of the reference character.
- *Take Snapshot*

2.5. Baking (in Unity was *Creating Prefabs*)

- Go to *Selection Mode → Modeling* or press *Shift + 5*.
- Select the model.
- Select *XForm → Bake TransformRS to activate the Rotation and Scale baking tool.*
- *In the New Asset Location drop-down menu at the bottom of the tool, select **AutoGen Folder (Global)**.*
- *Click the blue **Accept** button in the bottom of the page to start the baking process.*
- *Repeat the above for each model you want to *bake*.*
- Discard the new level as we don't need it anymore.

2.6. Generate Collisions for the Models (in Unity were colled “Colliders”)

- Open the **TopDown...** level.
- Press **Shift + 1** to go to *Selection Mode*.
- In the outline double-click on the model you want to generate collisions.
- In the top **Collisions** drop-down select **Add 26DOP Simplified Collision** to add a collision area to the mesh.
- Save the modified asset to apply the changes.
- *Take Snapshot*

2.7. Modifying the Player Controller

Character movement is controlled by a single click on a point on the map (UNreal speak for Unity's level/scene). If we want players to be able to move their character by keeping the mouse button pressed and moving it around the level (better control), we need to follow these steps to modify the Player Controller Blueprint:

1. Navigate to **Content** → **TopDown** → **Blueprints** and open the **BP_TopDownController** Blueprint.
2. Then, open the Event Graph by clicking the **Event Graph** tab.
3. Delete the **Set Destination Input – Touch** group as you won't be using it.
4. In the **Set Destination Input – Gamepad and Mouse** group, delete all the nodes connected to the **Canceled** and **Completed** execution pins.
5. Connect the **Ongoing** execution pin to the same **Branch** node as the **Triggered** execution pin.
6. Playtest (you should be able to move your character around whenever you keep the left mouse button pressed)
7. *Take Snapshot*

3. Testing a multiplayer game locally

UE allows playtesting of Multiplayer games on a single computer, making it much easier for developers to create and test these games. In this case, the editor is used as a "listen server" and a few other instances of the game are launched locally.

3.1. Playing as a Listen Server

We'll use the UE's **Net Mode* feature:**

1. **Open the **Change Play Mode and Play Settings** menu by clicking the vertical tripple dot button next to the Play button.
2. In the **Number of Players** field, enter the number of players you want to simulate; for instance, **3**.
3. Then, select **Net Mode** → **Play As Listen Server**
4. Playtest. You'll see three windows: 2 clients and a server.

ⓘ *Listen Sever* is analogous to a *Host Mode* in unity, that is, it serves as a client and a server simultaneously. Activating each of the windows, you see that you can control a specific avatar in each of them but the movement is replicated in all the windows (instances).

Note

If you check the **Outliner** window while in Play Mode, you will notice that there are three **BP_TopDownCharacter** instances (one for each player), but just one **BP_TopDownController** – this is the one you will need for the local player.

- *Take Snapshot*

3.2. Updating over the network

While we have not done anything to enable the above synchronization, the answer is **replication**: Character Actors are replication-enabled, so some of their properties, such as **Location** and **Rotation**, are updated across clients during gameplay. We can check this like so:

1. Open the **BP_TopDownCharacter** Blueprint by going to **Content** → **TopDown** → **Blueprints**.
2. Open the **Details** panel by clicking the Class Defaults tab.
3. Find the **Replication** category and notice that the **Replicates** attribute has been selected. Additionally, notice that **Replicate Movement** is selected

> **Note**
>
> It is this that allows automatic replication over the network!

- *Take Snapshot*

3.3. Adding additional character spawn points

We see that in all the windows the players are instantiated close to each other. We need more spawn points to remedy this:

1. Add several **Player Start** objects, up to the number of players you want to test – click the **Quickly Add To The Project** button, then select **Basic** → **Player Start**.
2. Place them anywhere on the map that you deem suitable for your game.
3. Playtest.
4. *Take Snapshot*

This works better, but you'd still see random spawning of two players at the same spawn point. To fix this we need to check positions and not reuse the ones already taken on new players. We do this as follows:

1. Open the **BP_TopDownGameMode** Blueprint by going to **Content** → **TopDown** → **Blueprints**.
2. Then, open the Event Graph.
3. In **My Blueprints** → **Functions**, add an override to the **ChoosePlayerStart** function by clicking the Override option.
4. Add a **Get All Actors Of Class** node and connect its *incoming execution pin* to the *execution pin* of the **Choose Player Start** node. Then, set the **Actor Class** drop-down attribute to **Player Start**.
5. Add a **For Each Loop** node to cycle through all the **Out Actor** properties you found in the previous node.
6. Connect the **Loop Body** execution pin to a **Branch** node.
7. Click and drag from the **Array Element** pin for the loop to get a **Player Start Tag** node and connect its outgoing pin to a not equal (!=) node. Assign the comparison value of this node to **Used**. Connect the outcome of this check to the **Condition** pin of the **Branch** node.
8. Connect the **True** execution pin of the **Branch** node to a **Set Player Start Tag** node with a value equal to **Used**. The **Target** pin should be connected to the **Array Element** area of the loop.
9. Connect the outgoing execution pin of the **Set** node to the graph's **Return Node**.
10. The **Return Value** pin of **Return Node** should be set to the **Array Element** property of the loop.
11. Playtest. You'll notice that this works better; no places with two players.
12. *Take Snapshot*

Note - In pseudo code, the meaning is:

```
Loop the array of Player_Start  objects:
  If not tagged Used:
    tag as Used and return value so it can be used
```

4. Updating properties over the network

We will work on the following features:

- Creating the pickup Blueprint
- Adding pickup variants
- Adding a points counter to the characters

4.1. Creating the pickup Blueprint

We are going to create a pickable item that will grant points to the character that picks it up by sending them a message. To create this type of communication, we'll need to create an interface:

1. In the Blueprint folder, right-click and select **Blueprints** → **Blueprint Interface**.
2. Name the interface **PointsAdder**.
3. Open the **Blueprint Interface**.
4. Rename the default function **AddPoints**.
5. Add an Input parameter called **Value** that is of the **Integer** type.
6. *Take Snapshot*

Now let's create the pickup Blueprint that will use it:

1. In your **Blueprints** folder, add a Blueprint Class that inherits from StaticMeshActor, and name the Blueprint **BP_BasePickup**.
2. Open the Blueprint. Then, select the **Class Defaults** tab and add a mesh of your choice to the **Static Mesh** field.
3. In the **Physics** section, enable the **Simulate Physics** attribute and check that the **Enable Gravity** attribute has been enabled.
4. Add a **SphereCollision** component to the Blueprint components hierarchy.
5. Name the component **Trigger** and ensure that the **Generate Overlap Events** attribute has been enabled.
6. Set the **Sphere Radius** attribute of the **SphereCollision** component to a value that is a little bigger than the static meshes you'll be using (for instance, **50**).
7. *Take Snapshot*

Let's add the code logic to the Blueprint. First, let's add a points value for the picking character:

1. Open the Event Graph.
2. Add a variable of the **Integer** type and call it **Points**.
3. Make it **Instance Editable** by clicking the **eye button** next to the variable type.
4. After compiling, set the variable's **Default Value** to **1**.
5. *Take Snapshot*

Now let's set the overlap event behavior for the Blueprint:

1. Delete the **Event BeginPlay** and **Event Tick** nodes as we won't be using them.

2. Add a **Cast To Character** node and connect its incoming execution pin to the outgoing pin of **Event ActorBeginOverlap**, to check that the actor is of the required type (that is, a Character).
3. If the check succeeds, then add an **AddPoints (Message)** node: this is the function we previously declared in the interface.
4. Connect the **As Character** pin of the cast to the **Target** pin of the function node.
5. Add a **Get Points** node to the graph and connect the pin to the **Value** pin of the **Add Points** function node.
6. 6. Finally, connect the outgoing execution pin of the **Add Points** node to a **Destroy Actor** node to remove the pickup once it has been taken.
7. *Take Snapshot*

The meaning of the above is that when an actor overlaps with the pickup, the pickup sends an **AddPoints** message and destroys itself.

Let us now enable **replication** !!!:

1. In the **Components** tab, select the **BP_BasePickup (self)** element.
2. Then, in the **Details** panel, look for the **Replication** category and enable the **Replicates** attribute.

The Pickup is now ready. We can add a few variants.

4.2. Adding pickup variants

To create a variant do the following:

1. Right-click on your **BP_BasePickup** item in the Content Browser.
2. Select **Create Child Blueprint Class**, give your new pickup a name, and open it.
3. Open the **Class Defaults** tab. Then, assign a mesh to the **Static Mesh** field.
4. Assign a value of your choice to the **Points** attribute (ex. 5, 10, etc...)
5. *Take Snapshot*
6. Repeat the above steps with few more variants.
7. Add the variants in the map
8. Playtest.
9. *Take Snapshot*

> **Note: Replication of Rotations**
>
> If your variants are not replication rotations, make sure that fields **Replicates** and **Replicat Movement** are checked.

4.3. Adding a points counter to the characters

Now that we know how to replicate Actors across the network, it is time to learn how to **replicate single variables** and how to **intercept changes at runtime (RepNotify)**. You'll be doing this by keeping track of the points that have been gained by each player by displaying them next to the gaming Actor. Follow these steps:

1. Open **Blueprints** → **BP_TopDownCharacter**.
2. Add a **Points** variable of the **Integer** type.
3. In the **Details** panel of the **Points** property, look for the **Replication** field and, from the drop-down menu, select **RepNotify**.

> **Note: RepNotify and OnRep_XYZ**
>
> Once the **RepNotify** field has been selected, a function named **OnRep_Points** has been added to your Blueprint – this function will be called on the clients every time the variable is updated by the authoritative Actor.

- *Take Snapshot*

> **Note: The difference between the RepNotify and Replicated ...**
>
> ... is that in the second case, the variable will be updated over the network without executing any notification function. Also note that the **OnRep_XYZ** function is called from the server on each client, and will not be executed on the server itself.

Let's add a text component to the character to display the points they have gained during the match:

1. Add a **TextRenderer** component to the character and name it **PointsLabel**.
2. Place the component anywhere you deem appropriate. For example: **Location (−120, 0, −80)** and **Rotation (0, 90, 180)**.
3. Enhance the characteristics of the components according to your wishes
4. *Take Snapshot*

Let's implementing now the interface we defined some time ago:

1. With **BP_TopDownCharacter** open, select the **Class Settings** tab.
2. In the **Details** panel, click the **Add** drop-down button on the **Implemented Interfaces** field and select the **PointsAdded** interface.
3. A new function named **AddPoints** will be added to the **Interfaces** section of the **MyBlueprint** tab. Right-click on the function's name and select **Implement Event** – this will add the corresponding node to the Event Graph and select it.
4. Drag the **Points** variable into the Event Graph and select the **Set** option.
5. Drag the **Points** variable again, this time selecting the **Get** option.
6. Add the outgoing **Value** pin from the event to the **Get Points** node by using an **Add (+)** node.
7. Connect the execution pin of the **Event** node to the **Set** node.
8. Connect the result pin of the **Add** node to the **Points** pin of the set node.
9. *Take Snapshot*

> **Note: Get and Set w/Notify**
>
> Notice that both **Set** and **Get** nodes now have an icon in the top-right corner. In addition, the **Set** node is decorated with text **w/Notify*: this means that the **Points** variable is replicated with a function notification.

Lastly, we need to implement the notification function so that we can update the points that are displayed to the character:

1. Double-click on the **OnRep_Points** function to open it.
2. Drag a **Get** node from the **PointsLabel** component in the Event Graph.
3. From its outgoing pin, add a **Set Text** node, and connect its incoming execution pin to the outgoing execution pin of the **On Rep Points** node.
4. Drag a **Get** node from the **Points** variable in the Event Graph and connect its pin to the **Value** pin of the **Set Text** node. Unreal will automatically add a **To Text** conversion node.
5. *Take Snapshot*
6. Playtest and see that all clients update the points.

7. *Take Snapshot*

5. Executing functions over the network

Here we will learn:

- how to properly call functions over the network, and
- what the word “authority” really means for the UE multiplayer system.

5.1. Spawning Actors

Let's spawn pickups ar runtime via a Spawn Area Blueprint that should do the following:

- Choose a random position every time it spawns something
- Spawn random pickups at predefined intervals
- And behave correctly over the network (replication)!

5.1.1 Choosing a random position

1. Create a new Actor Blueprint and call it **BP_Spawner**.
2. Add a **Box Collision** component, name it **SpawnArea**, and make it the **Scene Root** component by dragging it onto the default one.
3. Add an **Array** variable of the **Actor (Class Reference)** type and name it **SpawnableObjects**, making it **Instance Editable**.
4. *Take Snapshot*

Compile and Save the Blueprint. Open it's Event Graph. Then do the following:

1. Create a function named **Spawn** and open it.
2. Connect the execution node of the function to a **SpawnActor from Class** node.
3. Add a **Get** node for the **SpawnableObjects** variable and connect its outgoing pin to a **Random Array Item** node.
4. Connect the outgoing **Actor Class Reference** pin of **Random Node** to the **Class** pin of the **Spawn** node.
5. *Take Snapshot*

To get a position for the spawned item, we will get a random location inside the **Box Collision** component:

1. Right-click on the **Spawn Transform** pin of the **SpawnActor** node and select **Split Struct Pin**.
2. Drag the **SpawnArea** component into the function graph and connect its pin to a **Get Scaled Box Extent** node.
3. Add a **Get Actor Location** node and connect its outgoing pin to a **Random Point in Bounding Box** node.
4. Connect the outgoing pin of the **Get Scaled Box Extent** node to the **Half Size** pin of the **Random Point in Bounding Box** one.
5. Connect the **Return Value** pin of the **Random Point in Bounding Box** node to the **Spawn Transform Location** pin of the **SpawnActor** one.
6. *Take Snapshot*

The **Spawn** function selects a random Blueprint Class from a given list of elements and generates an instance of it at a random position within a defined area.

> **Tip**
>
> The analoguous in Unity would be a Spawn script that picks a random prefab from a list of prefabs and spawns it in a random position in a given area.

5.1.2 Spawn random pickups at predefined intervals

We'll add a timer to spawn pickups at predefined intervals:

1. Open the Event Graph section of the Blueprint and delete the **Actor BeginOverlap** and **Tick** events.
2. Add a **Set Timer by Event** node and connect its incoming execution pin to the outgoing one of the **BeginPlay** event.
3. Set the **Time** value equal to **1** and check the box in **Looping**.
4. Connect the **Event** pin to a **Custom Event** node and call it **OnTimer**.
5. Connect the execution pin of the custom event to the **Spawn** function.
6. *Take Snapshot*

Deploy the solution to the level (a slightly wrong approach - we'll fix it in a minute):

1. Delete all the pickups you may have previously added to your game level.
2. Add an instance of **BP_Spawner** to the level.
3. Place the instance approximately at the center of the scene and change the **Box Extent** values of the **SpawnArea** component so that the box will cover the entire play area; ex. **(1300, 1600, 32)**.
4. Place the **SpawnArea** component above the ground – pickups should be dropped from above and fall to the ground.
5. Add all the pickups we have created to the **Spawnable Objects** array.
6. Run the multiplayer simulation.
7. *Take Snapshot*

Notice that things will seem totally out of sync between clients: in particular, the UE instance (that is, the server) will spawn a single pickup at each interval, while the additional client will be spawning two.

Why? All clients are spawning locally and the server is spawning across the network. We need to fix this.

5.1.3 Using Actor autority to correctly spawn pickups

1. Add a **Switch Has Authority** node in between the execution pin of the **BeginPlay** event and **Set Timer by Event**.
2. Connect the **Authority** execution pin to the incoming pin of the **Timer** node.
3. *Take Snapshot*
4. Playtest and see that now it should behave correctly
5. *Take Snapshot*

5.2. Skinning characters

> **Optional**
>
> This section is Optional

Let's create some colorized materials and assign them to the characters as soon as they are spawned. First we create Material instances:

1. In the Content Browser, navigate to **Content** → **Characters** → **Mannequins** → **Materials** → **Instances** → **Manny**.
2. Duplicate **MI_Manny_01** several times to equal the number of connections we set in the *Play as a Listen Server* section (that is, **3**).
3. Name the new materials **MI_Manny_01_[ColorName]_{YourInitials}** (let [ColorName] be f.e. Red, Green and Blue).
4. Open each new instance and change the **Tint** property to the color *[ColorName]* above Red, Green, Blue).
5. Save all the material instances and close them.
6. *Take Snapshot*

Next, in the blueprint add a variable named **SkinMaterial**:

1. Open **BP_TopDownCharacter**.
2. Add a new variable called **SkinMaterial** of the *Material Interface Object Reference* type and make it **Instance Editable**.
3. Set the drop-down menu for the **Replication** field to **RepNotify**. This will create a function called **OnRep_SkinMaterial**.
4. *Take Snapshot*

Next, write the logic in **OnRep_SkinMaterial**:

1. Drag a reference for *Mesh* from the *Components* panel in the Event Graph.
2. Drag a **Get** node for the **SkinMaterial** variable.
3. Connect the function execution pin to a **Set Material** node.
4. Connect the **Mesh** reference to the **Target** pin.
5. Connect the **SkinMaterial** reference to the **Material** pin.
6. Notice that, whenever the **SkinMaterial** variable is changed, **OnRep_SkinMaterial** will take care of assigning it to the first material of the character.
7. *Take Snapshot*

Next, change the material for each character once it has been added to the level:

1. In the **Blueprints** folder, open **BP_TopDownGameMode**.
2. Add an **Array** variable of the **Material Interface Object Reference** type; name it **SkinMaterials**.
3. Once we have compiled the Blueprint, add all the materials we previously created to the **Default Value** field in the **Details** panel.
4. Add a variable of the **Integer** type named **SkinCount**; this variable will be used as an index counter for selecting the skins.
5. *Take Snapshot*

> ⚠ **Warning: SkinCount is not replicated**
>
> Knowing what variables to replicate or not is a valuable skill for a multiplayer programmer. *Chapter 6, Replicating Properties Over the Network* has more on this.

Next, let's get the count of the used skins, so, every time a new connection is created, we'll assign the next available skin to the character:

1. In the Event Graph, add an **Event OnRestartPlayer** node.
2. From **New Player**, connect a **Get Controlled Pawn** node.
3. Cast its outgoing pin to a **BP_TopDownCharacter** node.
4. Connect the event execution pin to the **Cast** node.

5. Connect the outgoing **As BP Top Down Character** to a **Set Skin Material** node (note the **w/Notify** label that was added to indicate the notification call when the value is changed).
6. Connect the successful execution pin of the cast node to the **Set Skin Material** node.
7. Add a **Get** node for the **SkinMaterials** array and a **Get** node for the **SkinCount** variable.
8. Connect the outgoing **Skin Materials** pin to the **Get (a copy)** node.
9. Connect the **Skin Count** pin to the **Get** index.
10. Connect the outgoing pin of the **Get** node to the **Skin Material** pin of the **Set Skin Material** node.
11. Finally, connect the outgoing execution pin of **Set Skin Material** to an **Increment (++)** node. The **Skin Count x** variable should be incremented; this will keep track of the selected skin in the array.
12. *Take Snapshot*

Prototype is done. Time to playtest.

5.3. Testing the Game

Playtest! Notice that players can run around and get the falling pickups, ready to gain points. - *Take Snapshot*

6. Summary

Here, we:

- created a prototype of our first multiplayer game, and
- gained knowledge about synchronizing Actors and variables across the network.
- tested the prototype through the UE system, which emulates multiple connections concurrently
- used of Blueprints.

Next, we will:

- start creating a game from scratch.
- develop it in C++ with all the advantages that come with this kind of development.