NETWORKING LAB 01

HANDS-ON

In the hands-on networking unit the student will start off looking at a HLAPI (High-Level API) which falls within layers 4 and 5 on the TCP/IP generic model. The student will be mostly protected from networking errors, and will get a grasp on how connections can be created between computers. The foundational knowledge of a server authorative networking state is also established, as well as client-side prediction and latency compensation.

The student will then transition into a LLAPI (Low-Level API) which resides within layer 2 and 3 on the TCP/IP generic model. The student will lose a significant portion of the protection provided from the HLAPI as they begin to work directly with socketing and commands.

Section One: I Can See Clearly Now

In this section of the networking unit, the student will create a very basic networked game using a HLAPI (High level API). This will allow:

- Seamless scene transitions
- Menu navigation using a dictionary state machine
- Multiple spawn points in a world
- Simple manual movement interpolation
- Client and Server commands

The basic knowledge gained in this unit will be the foundation of the students learning as the students begins to work in a LLAPI (Low-Level API) where the student will directly handle sockets and other networking tasks.

STEP ONE: SETUP

For the setup of this lab you have been provided with a template game. The game includes two scenes: A menu, and the main game world.

You are also supplied with multiple scripts: MenuTransitions, Script_MenuState, Script_Latency, and Script_PlayerSync. Most of these scripts contain empty functions that you will fill in during this lab.

This template game **will not run** in it's current state. As you progress through this lab you will enter checkpoints where the game will be runnable and you can test your results.

Look at all of the scripts provided, and become familiar with the MenuTransitions script.

STEP TWO: BECOMING FAMILIAR WITH THE MENU

Before you can actually begin creating the networked game, you must complete some tidying tasks. The first of these tasks is going to be to create the menu system.

In the scene labeled Main Menu, you should see four objects nested in the Canvas. The objects named Main_Scene, Setup_Scene, and Connect_Scene will be the various menu's that the player is shown when starting the game.

Open the Script_MenuState script.

The menu is controlled by a very basic state machine. Each menu, as well as each button on the menu, is considered a **state.** If you look at the enum MenuStates, you can see the various states that the menu can be in:

1. MENU_MAIN (when the main menu is shown on the screen)
2. MENU_CONNECT (when the connect menu is shown on the screen)
3. MENU_SETUP (when the setup menu is shown on the screen)
4. MENU_QUITTING (when the user presses the exit game button)
5. SETUP_STARTING_HOST (when the user clicks the start host button on the setup screen)
6. SETUP_STARTING_SERVER (when the user presses the start server button on the setup screen)
7. CONNECT_CONNECTING_TO_SERVER (when the user presses the connect to server button on the connect screen).

In addition to each state, there is included **transitions**. A transition represents a transition from one state to another state. For example, moving from the main menu to the connection menu. In the menu's state machine, a transition is triggered by a command. Look at the enum MenuCommands. Each command will trigger a transition in the engine.

In order to connect each state with a transition, you will need some Dictionaries. Remember, dictionaries connect two types together, with one type being the key and another type being the value. Dictionaries work very much like an enum does, except instead of each key having to be a string and each value an int you can use any two types.

```
        enum ColourOfDogs
        {
            Black,
            Green,
            Orange
        }

        Dictionary<string, int> colourOfDogs = new Dictionary<string, int>
        {
            {"Black", 0 },
            {"Green", 1 },
            {"Orange", 2 }
        };
```

This particular dictionary links the string "Black" with the int 0, just like the enum links the ColourOfDogs.Black to the int 0.

STEP THREE: DICTIONARIES

Now you need to actually connect each state with a transition. To do this, you will utilize both a dictionary and the MenuTransitions class provided. The MenuTransitions class stores two things: A menu state, and a command (the thing that will start the transition). This means that your dictionary will store a MenuTransitions (which in turn stores a state and a command) as the "key". The "value" will be another state.

This will make each item in your dictionary look like this:

{new MenuTransition(if the menu is in this current state , if this command is issued) , move into this state }

{ new MenuTransitions (MenuStates ifCurrentState, MenuCommands ifThisCommandIsIssued), MenuStates moveIntoThisState }

key　　　　　　　　　　　　　　　　value

{ new MenuTransitions (MenuStates ifCurrentState, MenuCommands ifThisCommandIsIssued), MenuStates moveIntoThisState }

In this menu, there are 8 transitions. They are as follows:

1. If current state is MENU_MAIN, and the GOTO_CONNECT command is issued, move into the MENU_CONNECT state.
2. If current state is MENU_MAIN, and the GOTO_SETUP command is issued, move into the MENU_SETUP state.
3. If current state is MENU_MAIN, and the QUIT_APPLICATION command is issued, move into the MENU_QUITTING state.
4. If current state is MENU_CONNECT, and the GOTO_MAIN command is issued, move into the MENU_MAIN state.
5. If current state is MENU_CONNECT, and the CONNECT_CLIENT command is issued, move into the CONNECT_CONNECTING_TO_SERVER state.
6. If current state is MENU_SETUP, and the GOTO_MAIN command is issued, move into the MENU_MAIN state.
7. If current state is MENU_SETUP, and the SETUP_HOST command is issued, move into the SETUP_STARTING_HOST state.
8. If current state is MENU_SETUP, and the SETUP_SERVER command is issued, move into the SETUP_STARTING_SERVER state.

See if you can code this yourself. There is a declared dictionary in the Start() function named allTransitions that you can use.

The code for this dictionary is as follows:

```
        //Create the dictionary
        allTransitions = new Dictionary<MenuTransitions, MenuStates>
        {
            {new MenuTransitions(MenuStates.MENU_MAIN, MenuCommands.GOTO_CONNECT), MenuStates.MENU_CONNECT },
            {new MenuTransitions(MenuStates.MENU_MAIN, MenuCommands.GOTO_SETUP), MenuStates.MENU_SETUP },
            {new MenuTransitions(MenuStates.MENU_MAIN, MenuCommands.QUIT_APPLICATION), MenuStates.MENU_QUITTING },
            {new MenuTransitions(MenuStates.MENU_CONNECT, MenuCommands.GOTO_MAIN), MenuStates.MENU_MAIN },

            {new MenuTransitions(MenuStates.MENU_CONNECT, MenuCommands.CONNECT_CLIENT),
MenuStates.CONNECT_CONNECTING_TO_SERVER },

            {new MenuTransitions(MenuStates.MENU_SETUP, MenuCommands.GOTO_MAIN), MenuStates.MENU_MAIN },
            {new MenuTransitions(MenuStates.MENU_SETUP, MenuCommands.SETUP_HOST), MenuStates.SETUP_STARTING_HOST },
            {new MenuTransitions(MenuStates.MENU_SETUP, MenuCommands.SETUP_SERVER), MenuStates.SETUP_STARTING_SERVER },
        };
```

There is one more dictionary to create before you can continue on. This dictionary is a lot simpiler. You will use this next dictionary to issue commands to your state machine from Unity buttons. Because Unity buttons cannot send enum's to scripts (they can only send Object's, int's, float's, and string's) you will want to key string to your enum values for commands. This way, when you setup the onClick for your button, you can send a string through the button and use the dictionary to tie it to the string.

NOTE: You may be asking yourself why not just use enumValue.ToString(). While you CAN use the .ToString() method, it is a very slow process. Dictionaries are much more optimized!

The dictionary that you will use for this is the enumParse dictionary, where the key is a string and the value is a MenuCommand.

The following pairs (key to value) is as follows:

1. "goto connect menu" should link to GOTO_CONNECT
2. "goto setup menu" should link to GOTO_SETUP
3. "goto main menu" should link to GOTO_MAIN
4. "quit application" should link to QUIT_APPLICATION
5. "setup host" should link to SETUP_HOST
6. "setup server" should link to SETUP_SERVER
7. "connect to server" should link to CONNECT_CLIENT

Try and code this dictionary yourself. There is a declared dictionary in the Start() function named enumParse that you can use.

The code for this dictionary is as follows:

```
         //Create the dictionary where
         //{string that is passed by the button, command the string represents}
        enumParse = new Dictionary<string, MenuCommands>
        {
            {"goto connect menu", MenuCommands.GOTO_CONNECT },
            {"goto setup menu", MenuCommands.GOTO_SETUP },
            {"goto main menu", MenuCommands.GOTO_MAIN },
            {"quit application", MenuCommands.QUIT_APPLICATION },
            {"setup host", MenuCommands.SETUP_HOST },
            {"setup server", MenuCommands.SETUP_SERVER },
            {"connect to server", MenuCommands.CONNECT_CLIENT}
        };
```

STEP FOUR: TRANSITIONING

The next step in creating the state machine is creating the function that will actually move the machine from one state to the next (for example, move the machine state from MenuStates.MAIN_MENU to MenuStates.CONNECT_MENU).

You will do this through two functions: GetNext and MoveNext.

The GetNext function will determine what the next state the machine **should** be in, and the MoveNext function will actually move the machine into that state (in this case by just setting the CurrentState equal to a new state!).

GETNEXT()

The MenuStates GetNext(MenuCommands command) function is responsible for determining what the next state of the machine should be, if the machine were to be issued a command.

For example, if you are in the MAIN_MENU state and issue the command GOTO_CONNECT, then the new state should be CONNECT_MENU (you defined this relationship in your dictionary, remember?).

GetNext will use the key and value from the dictionary you created earlier to determine what state the machine should move into.

So, if you remember the key for your dictionary was a MenuTransitions that represented the current state of the machine and the command that was issued. The value was the resulting state of the machine if the command was issued.

If current state is MENU_MAIN, and the GOTO_CONNECT command is issued, move into the MENU_CONNECT state.

`{new MenuTransitions(MenuStates.MENU_MAIN, MenuCommands.GOTO_CONNECT), MenuStates.MENU_CONNECT }`

This means that you will need a MenuTransitions and a MenuState to look into the dictionary (MenuTransitions being the key, MenuState being the resulting value).

Your MenuTransitions will hold the current state that the machine is in, and the command that is being issued to transition the machine into a new state.

```
    MenuStates GetNext(MenuCommands command)
    {
         //Construct the new transition based on the machines current state, and the supplied transition/command
        MenuTransitions newTransition = new MenuTransitions(CurrentState, command);
```

Your new MenuState will hold the resulting state of the transition (If you are in MENU_MAIN, and issue the command GOTO_CONNECT, then your new MenuState will hold MENU_CONNECT)

```
    MenuStates GetNext(MenuCommands command)
    {
        //Construct the new transition based on the machines current state, and the supplied transition/command
        MenuTransitions newTransition = new MenuTransitions(CurrentState, command);

        //Location to store the new state for the machine to go into
        MenuStates newState;
```

With your variables set up, you can now look into the direction and search for the key, and then get the resulting value (continuing with our example, you would be looking for a MenuTransitions(MENU_MAIN, GOTO_CONNECT) and getting the resulting value (MENU_CONNECT).

In order to look something up in a dictionary, you use the method Dictionary.TryGetValue(Type key, out Type value). This will return a bool that indicates if they key was found. Meaning if you were to search for MenuTransitions(MENU_MAIN, GOTO_MAIN) it would return false (you never created a declaration in the dictionary for this type of transition).

You can use this bool value to your advantage, capturing if it returns false to inform the user that they are trying to make an invalid transition (such as from the main menu to the main menu).

```
MenuStates GetNext(MenuCommands command)
{
    //Construct the new transition based on the machines current state, and the supplied transition/command
    MenuTransitions newTransition = new MenuTransitions(CurrentState, command);

    //Location to store the new state for the machine to go into
    MenuStates newState;

    //Make sure that the transition is valid, using the dictionary lookup
    if (!allTransitions.TryGetValue(newTransition, out newState))
        throw new UnityException("Invalid transition " + CurrentState + " -> " + command);
```

Lastly, you can return the resulting MenuState.

```
MenuStates GetNext(MenuCommands command)
{
    //Construct the new transition based on the machines current state, and the supplied transition/command
    MenuTransitions newTransition = new MenuTransitions(CurrentState, command);

    //Location to store the new state for the machine to go into
    MenuStates newState;

    //Make sure that the transition is valid, using the dictionary lookup
    if (!allTransitions.TryGetValue(newTransition, out newState))
        throw new UnityException("Invalid transition " + CurrentState + " -> " + command);

    //If at this point we have not broken anything, return the new state
    return newState;
}
```

Keep in mind that at this point, you have not actually changed your menu's state machine's state. The menu is still in the same state as it was prior to this function being run. You have only determined what the next state **should** be if a given command was issued to the machine.

MOVENEXT()

Moving the state machine from one state to the next is rather simple: just assign the new state to the supplied state! You will want to make sure to record what state you are moving from, however, so that you can create the transition later on.

However, because the commands are going to be passed in by buttons (which therefore means via strings) you will need to reference the actual command from the enumParse dictionary.

```
/// <summary>
/// Moves the state machine into the next state given a specific command/transition and triggers the transition
/// </summary>
/// <param name="command">The command/transition to run the machine through</param>
public void MoveNextAndTransition(string command)
{
    //Record the previous state for transition purposes
    PreviousState = CurrentState;

    //Location for the new command
    MenuCommands newCommand;

    //Try to get the value
    if (!enumParse.TryGetValue(command, out newCommand))
        throw new UnityException("Invalid command " + command);

    //Setup the next state
    CurrentState = GetNext(newCommand);

    //Transition to the next state
}
```

TRANSITION!

Finally, you can transition your state machine. To do this, you will want to call the Transition() function at the end of the MoveNextAndTransition function.

```
/// <summary>
/// Moves the state machine into the next state given a specific command/transition and triggers the transition
/// </summary>
/// <param name="command">The command/transition to run the machine through</param>
public void MoveNextAndTransition(string command)
{
    //Record the previous state for transition purposes
    PreviousState = CurrentState;

    //Location for the new command
    MenuCommands newCommand;

    //Try to get the value
    if (!enumParse.TryGetValue(command, out newCommand))
        throw new UnityException("Invalid command " + command);

    //Setup the next state
    CurrentState = GetNext(newCommand);

    //Transition to the next state
    Transition();
}
```

The transition function is the meat of this entire script. It's job is to make sure that the menu items appear appropriately depending on which state the menu is in. For example, making sure to show the MainMenu object when the state of the machine is MENU_MAIN.

This means that you will need to check what state you were previously in, and what state you are in now. Therefore, if you are trying to transition from MENU_MAIN to MENU_CONNECT, you will know to disable all of the UI elements associated with the main menu, and enable all of the UI elements associated with the connect menu.

You can do this a variety of ways (nested if statements, nested switch statements, splitting into functions). In this lab you will use a nested if statement inside of a switch statement. The switch statement will switch based on the previous state, and the if statements will control what to do for the new state transitions.

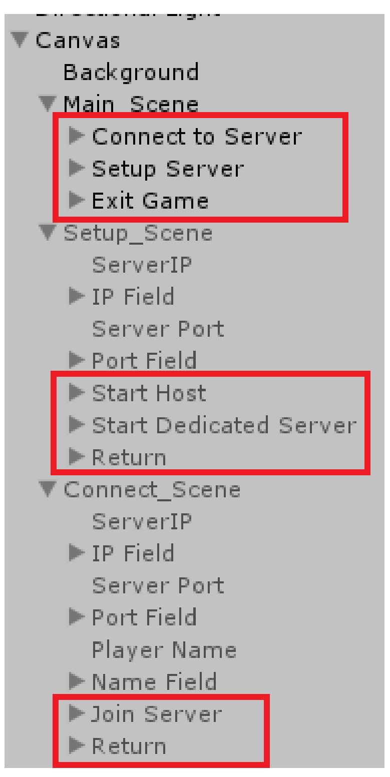
```
/// <summary>
/// Will run any neccessary code for the transition from one state to the next
/// </summary>
void Transition()
{
    switch (PreviousState)
    {
        //If my previous state is MAIN
        case MenuStates.MENU_MAIN:
            //If my NEW state is CONNECT
            if (CurrentState == MenuStates.MENU_CONNECT)
            {

            }
            break;
    }
}
```

You should be familiar enough with toggling UI elements to complete this portion on your own. Try it (just use Debug.Log when attempting to transition into your START_SERVER, START_HOST, and CONNECT_CLIENT states), and then check your code against the solution.

```
/// <summary>
/// Will run any neccessary code for the transition from one state to the next
/// </summary>
void Transition()
{
    switch (PreviousState)
    {
        //If my previous state is MAIN
        case MenuStates.MENU_MAIN:
            //If my NEW state is CONNECT
            if (CurrentState == MenuStates.MENU_CONNECT)
            {
                Debug.Log("Transition from Main to Connect");
                mainMenu.SetActive(false);
                connectMenu.SetActive(true);
            }
            //If my NEW state is SETUP
            else if (CurrentState == MenuStates.MENU_SETUP)
            {
                Debug.Log("Transition from Main to Start");
                mainMenu.SetActive(false);
                setupMenu.SetActive(true);
            }
            //If my NEW state is QUIT
            else if (CurrentState == MenuStates.MENU_QUITTING)
            {
                Debug.Log("Transition from Main to Quit");
                Application.Quit();
            }
            break;

        //If my previous state is CONNECT
        case MenuStates.MENU_CONNECT:
            //If my NEW state is MAIN
            if (CurrentState == MenuStates.MENU_MAIN)
            {
                Debug.Log("Transition from Connect to Main");
                connectMenu.SetActive(false);
                mainMenu.SetActive(true);

            }
            else if (CurrentState == MenuStates.CONNECT_CONNECTING_TO_SERVER)
            {
                Debug.Log("Transition from Connect to Connecting to Server");
            }
            break;

        //If my previous state is SETUP
        case MenuStates.MENU_SETUP:
            //If my NEW state is MAIN
            if (CurrentState == MenuStates.MENU_MAIN)
            {
                Debug.Log("Transition from Start to Main");
                setupMenu.SetActive(false);
                mainMenu.SetActive(true);
            }
            else if (CurrentState == MenuStates.SETUP_STARTING_SERVER)
            {
                Debug.Log("Transitioning from Start to Server Starting");
            }
            else if (CurrentState == MenuStates.SETUP_STARTING_HOST)
            {
                Debug.Log("Transition from Start to Host Starting");
            }
            break;
    }
}
```

STEP FIVE: BUTTONS

Before you can test if this is working, you will need to make the buttons actually call these functions. The Main Camera should contain the Script_MenuState. Setup each buttons onClick() to call the MoveNextAndTransition(string command) function from the Script_MenuState script on the main camera. Make sure to type in the appropriate command (the string key in the enumParse dictionary you made). You should do this for 8 buttons:

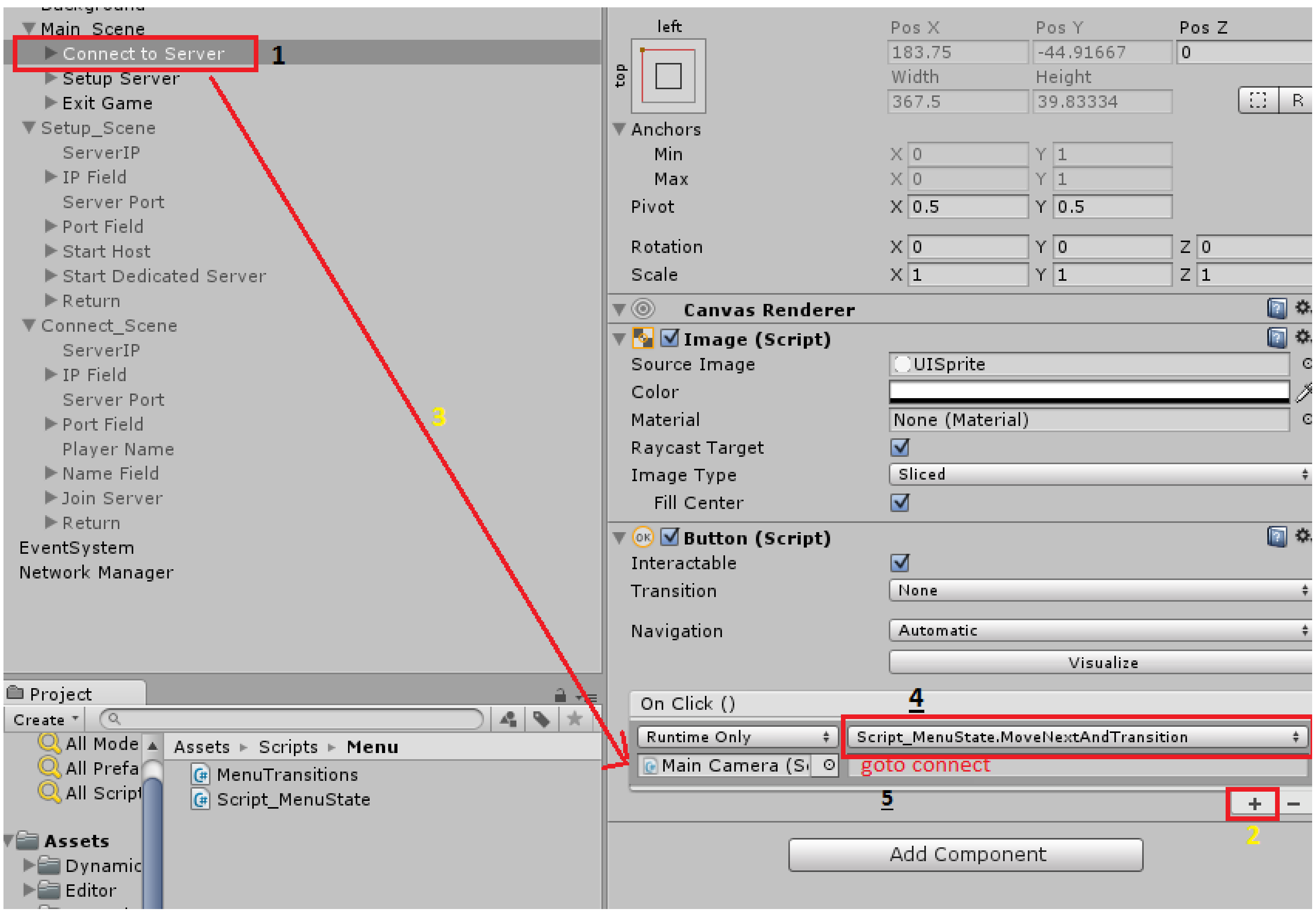


```
▼ Canvas
    Background
  ▼ Main_Scene
    ▶ Connect to Server
    ▶ Setup Server
    ▶ Exit Game
  ▼ Setup_Scene
      ServerIP
    ▶ IP Field
      Server Port
    ▶ Port Field
    ▶ Start Host
    ▶ Start Dedicated Server
    ▶ Return
  ▼ Connect_Scene
      ServerIP
    ▶ IP Field
      Server Port
    ▶ Port Field
      Player Name
    ▶ Name Field
    ▶ Join Server
    ▶ Return
```

For example, to tie the Connect to Server button into the engine, you would:

1. Click on Connect to Server in the Hierarchy

2. Press the + for the On Click()
3. Drag the Main Camera in for the object
4. Set the function to Script_MenuState > MoveNextAndTransition(string command)
5. Type in "goto connect menu" without the quotes.



```
▼ Main_Scene
    ▶ Connect to Server
    ▶ Setup Server
    ▶ Exit Game
▼ Setup_Scene
      ServerIP
    ▶ IP Field
      Server Port
    ▶ Port Field
    ▶ Start Host
    ▶ Start Dedicated Server
    ▶ Return
▼ Connect_Scene
      ServerIP
    ▶ IP Field
      Server Port
    ▶ Port Field
      Player Name
    ▶ Name Field
    ▶ Join Server
    ▶ Return
  EventSystem
  Network Manager
```

```
Pos X 183.75   Pos Y -44.91667   Pos Z 0
Width 367.5    Height 39.83334
▼ Anchors
    Min    X 0    Y 1
    Max    X 0    Y 1
Pivot      X 0.5  Y 0.5
Rotation   X 0    Y 0    Z 0
Scale      X 1    Y 1    Z 1
▼ Canvas Renderer
▼ Image (Script)
Source Image     UISprite
Color
Material         None (Material)
Raycast Target   ✓
Image Type       Sliced
  Fill Center    ✓
▼ Button (Script)
Interactable     ✓
Transition       None
Navigation       Automatic
                 Visualize
On Click ()
Runtime Only     Script_MenuState.MoveNextAndTransition
Main Camera (S   goto connect
                                   + −
Add Component
```

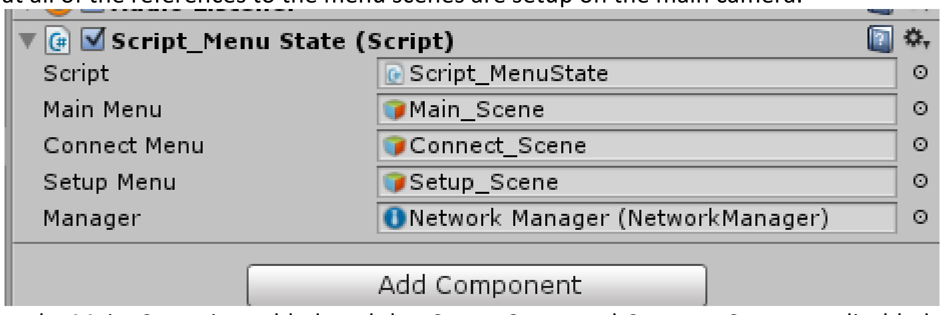
```
Project
Create
All Models   Assets ▸ Scripts ▸ Menu
All Prefabs    MenuTransitions
All Scripts    Script_MenuState
▼ Assets
  ▶ Dynamic
  ▶ Editor
```

Repeat this process with each button and the appropriate string command setup in the enumParse dictionary. You will use goto main menu multiple times!

CHECKPOINT!

To test that what you have done so far is working:

1. Verify that all of the references to the menu scenes are setup on the main camera.



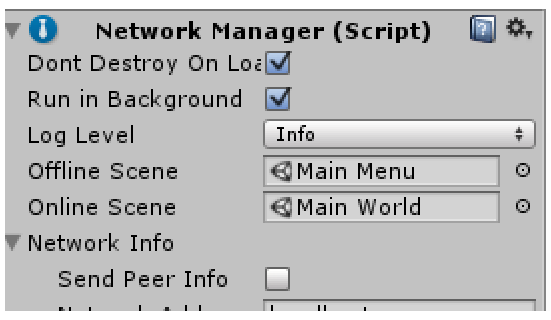
```
▼ Script_Menu State (Script)
Script          Script_MenuState
Main Menu       Main_Scene
Connect Menu    Connect_Scene
Setup Menu      Setup_Scene
Manager         Network Manager (NetworkManager)
                Add Component
```

    a.
2. Verify that the Main_Scene is enabled, and that Setup_Scene and Connect_Scene are disabled.
3. Click the play button to enter runtime.
4. Click Connect to Server
    a. You should get a debug: Transition from Main to Connect
5. Click Return to Main Menu
    a. You should get a debug: Transition from Connect to Main
6. Click Setup Server
    a. You should get a debug: Transition from Main to Start
7. Click Return to Main Menu:
    a. You should get a debug: Transition from Start to Main
8. Click Exit Game
    a. You should get a debug: Transition from Main to Quit
9. If you click any buttons now, you should get an error. This is because there is no transition setup from main to quit (quit –should- quit the game, but since the editor can't be "quit" it just kills your state machine!)

STEP SIX: SIMPLE CONNECTION

Next you can create a simple connection to the networked game. To do this, first verify that the "Online Scene" in the Network Manager is set to Main World, and the Offline scene is set to Main Menu.

Remember, both of these scenes must be in the build settings to be placed there.



```
▼ Network Manager (Script)
Dont Destroy On Loa  ✓
Run in Background    ✓
Log Level            Info
Offline Scene        Main Menu
Online Scene         Main World
▼ Network Info
    Send Peer Info   ☐
```

Next, return to the Script_MenuState.

You will be starting the server/host, or connecting to a server/host in the Transitions() function. Make sure you are working in the appropriate transition.

When working in the HLAPI, it is very easy to start and connect networked games.

To connect to an already running server, you use the function NetworkManager.StartClient(). This will pull the IP and Port from the network manager, and attempt to start a client connection to that particular IP and Port. In the Script_MenuState you already have a reference to the NetworkManager (in a variable named manager at the top of the script). This means that in order to connect to a running server, you only need to call manager.StartClient()!This will also immediately transfer the client to the online scene if the connection was successful.

To start a hosting server (which is a server where the server and a single client are on the same machine, in the same instance) you would use NetworkManager.StartHost(). Again, this will automatically pull the IP and Port from the network manager, and attempt to start a hosting connection for that IP and Port. This will also immediately transfer the client to the online scene if the hosting server was started successfully.

To start a dedicated server (where the client and server are not in the same instance) you would use NetworkManager.StartServer().Again, this will automatically pull the IP and Port from the network manager, and attempt to start a hosting connection for that IP and Port. This will also immediately transfer the server to the online scene if the dedicated server was started successfully.

MOAR BUTTONS!

Something nice that you can add is the ability for the player to change the IP and Port that they will attempt to start a server on (or connect to). This functionality is easily implemented using InputFields and a few clever functions.

At the bottom of the Script_MenuState you will notice three functions: UpdateIP, UpdatePort, and UpdateName. They all tabe a parameter of type Object. This is because InputFields cannot pass the strings they contain into a function. However, they can pass the actual text object into the function. After being supplied with the text object, you can get the text component off of it and then parse the information. After parsing the information you can assign it to the network manager using NetworkManager.networkAddress and NetworkManager.networkPort. There is no function for a player name, so you can worry about that later.

So, inside of UpdateIP your first task is to pull the Text component off of the object passed in. Now, you should be comfortable using GameObject.GetComponent at this point. However the parameter calls for an Object, not a GameObject. This means that you will have to caste the Object into a GameObject in order for you to GetComponent.

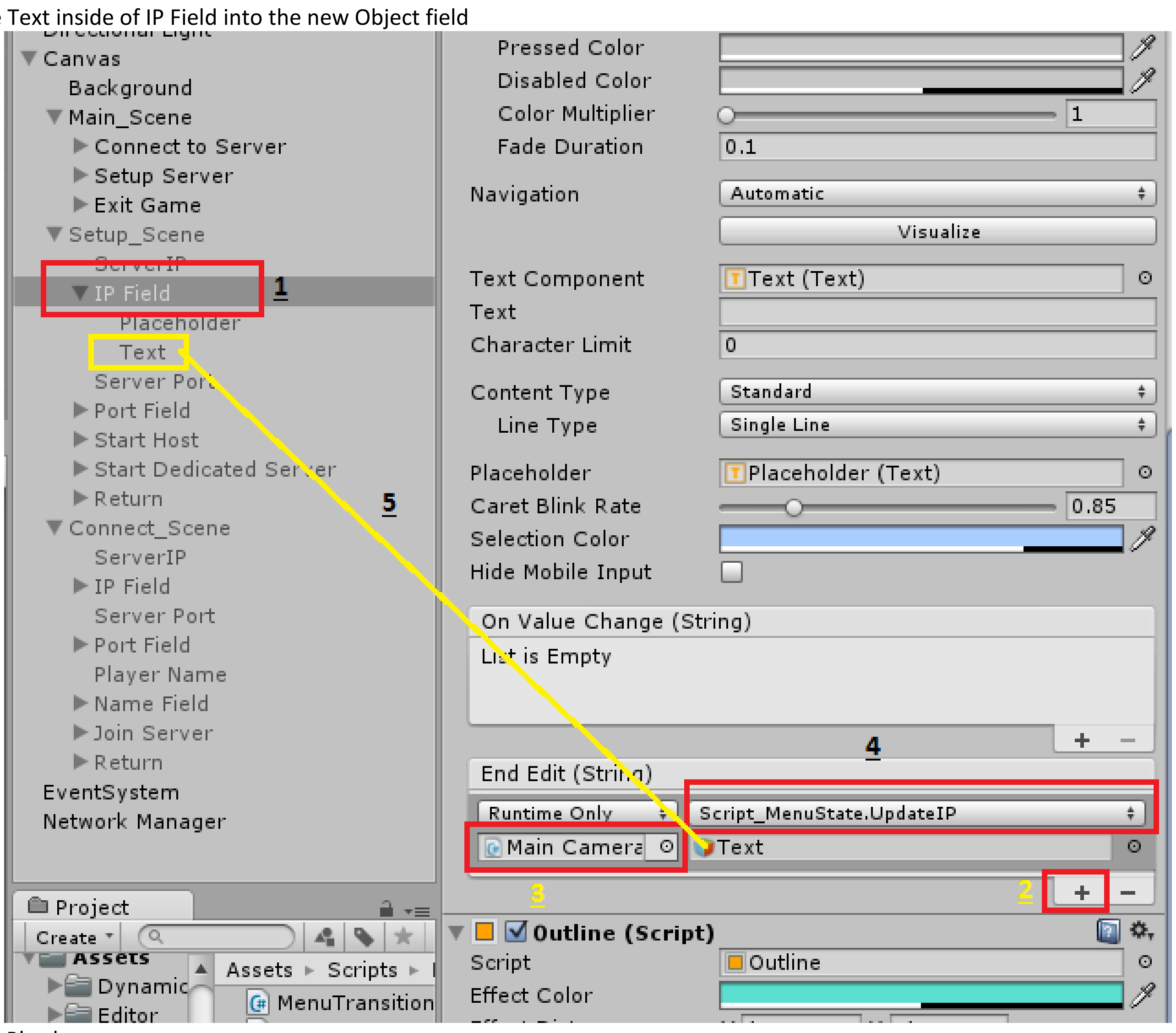
After you have gotten the text component, you want to make sure that it isn't empty. If it is empty, default the IP to "localhost". If it isn't empty, you can set the IP to whatever string the text component contains.

```
/// <summary>
/// Updates the IP on the manager
/// </summary>
/// <param name="message">The IP to update to</param>
public void UpdateIP(Object message)
{
    Text textObj = ((GameObject)message).GetComponent<Text>();
    if (textObj.text == "")
        manager.networkAddress = "localhost";
    else
        manager.networkAddress = textObj.text;

    Debug.Log("Updating address to... " + textObj.text);
}
```

Following the same idea, you can do the UpdatePort.

```
Text textObj = ((GameObject)message).GetComponent<Text>();
int newPort;

if (int.TryParse(textObj.text, out newPort))
    manager.networkPort = newPort;
else
    manager.networkPort = 7777;

Debug.Log("Updating port to... " + newPort.ToString());
```

The last thing to do is assign these functions on each of the InputFields as appropriate. For example:

1. Click on IP Field
    a. Inside of the Setup_Scene
2. Click the + corresponding to the End Edit (String)
3. Drag the Main Camera into the object area
4. Change the function to Script_MenuState.UpdateIP()
5. Drag the Text inside of IP Field into the new Object field
    a. 

Pressed Color
Disabled Color
Color Multiplier: 1
Fade Duration: 0.1
Navigation: Automatic
Visualize
Text Component: Text (Text)
Text:
Character Limit: 0
Content Type: Standard
Line Type: Single Line
Placeholder: Placeholder (Text)
Caret Blink Rate: 0.85
Selection Color
Hide Mobile Input
On Value Change (String)
List is Empty
End Edit (String)
Runtime Only | Script_MenuState.UpdateIP
Main Camera | Text
Script: Outline
Effect Color
6. Click the Play button
7. Click Connect to Server
8. Change the IP address
9. Click the Network Manager in the Hierarchy
10. Look at the IP address and verify it has changed!

STEP SEVEN: SCENE CONTROL

The next thing to do is set up a spawning location for the player in the Main World.

When you open the Main World scene, the first thing you should notice is that the Game view says No Camera. Do not add a camera to the Game View, unless you wish to manually delete these camera for every client that attempts to connect to the scene.

1. Inside of the scene, add a new empty game object. Name it Spawn Positions. Change its position to 0,0,0 with no rotation.
2. Add a child empty game object to the Spawn Position object, and name it Spawn 1.
3. Add the Network Start Position script to the child empty game object.
4. Change it's position to X: -119, Y: 165.08, X: -924
5. Duplicate Spawn 1 and place it somewhere else. Repeat this two or three times.

When the NetworkManager transitions the client to a new scene, it will automatically scan for any game object with the Network Start Position component. It will collect all of the spawn positions, and then randomly (by default) pick one to spawn the player at.

CHECKPOINT!

You can verify the spawn locations are working by:

1. Returning to the Main Menu scene
2. Verifying the player prefab is located in the Network Manger
    a. If it is not, you want to use the player prefab located in StaticAssets > Prefabs > Common
3. Start the Game
4. Setup Server
5. Start Host
6. Look at where you spawned (players transform)

7. Repeat 2-3 times to verify you spawn in different locations

STEP EIGHT: CONTROL YOURSELF!

Now to get to the meat of the multiplayer networking aspect of the game. You should at this point have a functional menu, and sem-functional world scene that can spawn the player.

Open up the Script_PlayerSync script and familiarize yourself with the various variables that are declared at the beginning of the script.

Currently, if you were to attempt to run this as a multiplayer game, you would have control over both players. Go ahead and test this for yourself.

This is because you are running a player controller script on all connected clients. Instead, what you really want to do is run the script only on the player that represents your character. In order to do this, you can use a very simple bool named isLocalPlayer. This is a bool that is built into the UnityEngine.Networking namespace, and you have access to it via the NetworkBehaviour that this script extends.

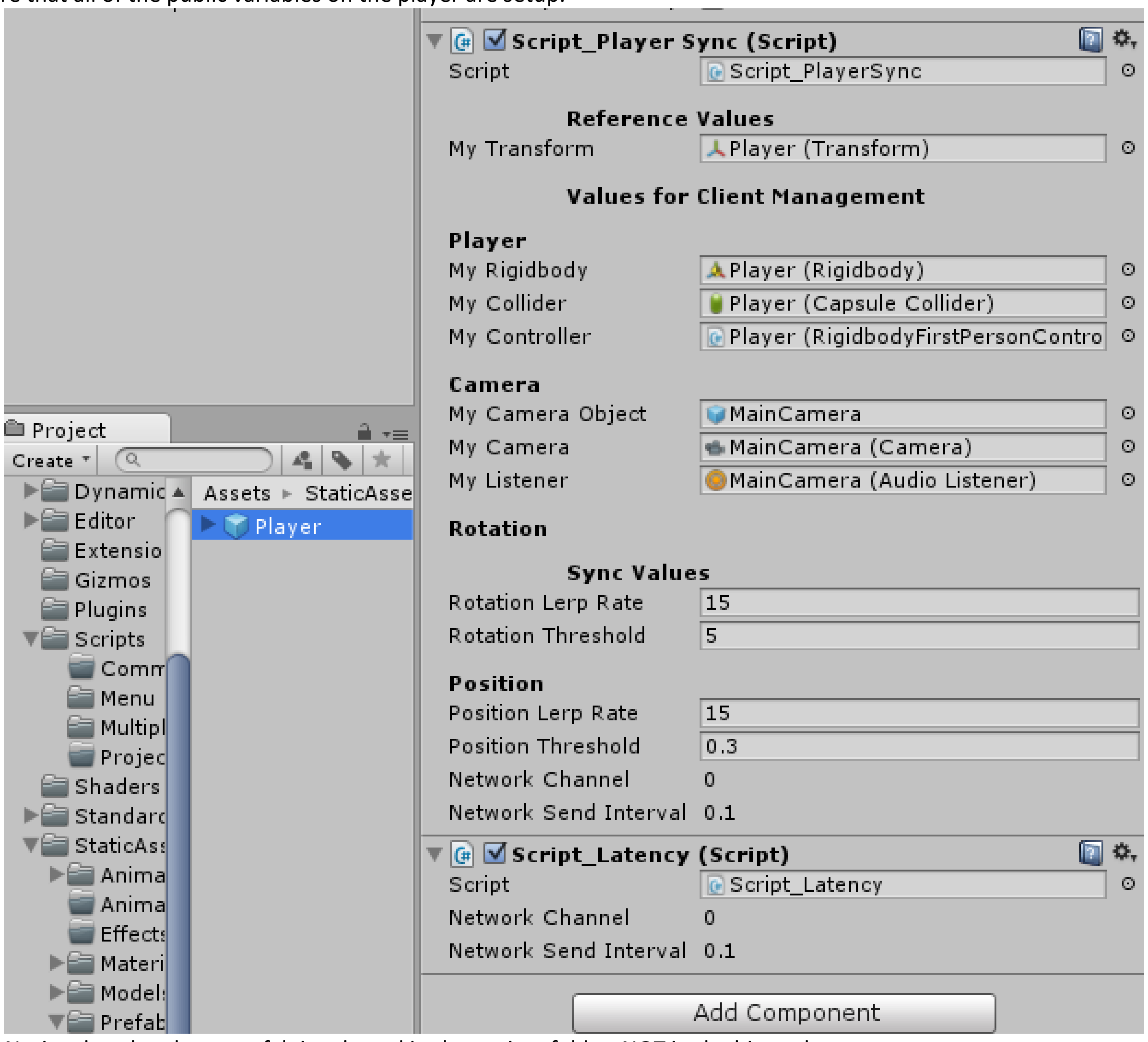
So, when this script starts what you want to do is destroy all of the scripts, rigid bodies, controllers, cameras, and anything else on a player other than its transform and visual appearance (such as a capsule in this case) if the script is on a player that does not represent your personal character.

Since you already have access to the controller, rigidbody, collider, and camera object through public variables, you simply want to check if the script is running on the local player (aka the player that represents your personal character). If it is not, then destroy everything (mostly!).

```
void Start()
{
    if (!isLocalPlayer)
    {
        Destroy(myController);
        Destroy(myRigidbody);
        Destroy(myCollider);
        Destroy(myCameraObject);
    }
}
```

CHECKPOINT!

You can verify this is working by:

1. Make sure that all of the public variables on the player are setup.
   a. 

   ```
   Script_Player Sync (Script)
   Script                  Script_PlayerSync
           Reference Values
   My Transform            Player (Transform)
           Values for Client Management
   Player
   My Rigidbody            Player (Rigidbody)
   My Collider             Player (Capsule Collider)
   My Controller           Player (RigidbodyFirstPersonContro
   Camera
   My Camera Object        MainCamera
   My Camera               MainCamera (Camera)
   My Listener             MainCamera (Audio Listener)
   Rotation
           Sync Values
   Rotation Lerp Rate      15
   Rotation Threshold      5
   Position
   Position Lerp Rate      15
   Position Threshold      0.3
   Network Channel         0
   Network Send Interval   0.1
   Script_Latency (Script)
   Script                  Script_Latency
   Network Channel         0
   Network Send Interval   0.1
   ```
   b. Notice that the player prefab is selected in the project folder, NOT in the hierarchy.
2. Open the Main World scene
3. Go to Window > Lighting
4. Disable Precomputed Realtime Gi and Baked GI
5. Close the Lighting window
6. Go to the Main Menu scene
7. Go to Window > Lighting
8. Disable Precomputed Realtime Gi and Baked GI
9. Close the Lighting window
10. Build and Run your game
11. Start a Host on the built game
12. Run the game in the editor
13. Connect to the host on the game
14. Look in the hierarchy for the character that is not your personal character (one of the Player(Clone)s)
15. Verify that it is missing the appropriate scripts, and that when you move you are only moving one player.

Of course, this is not flawless. If you run around on one of the running games, it is not updating on the other. However, you should see two players on each version of the game you are running (two players on the built game, two players in the editor including yourself).

STEP NINE: CREATING MOVEMENT

In order for each connected player to see each other move, you need to start sending information about yourself to the server. The server can then relay this information to other clients, and allow them to see your position update.

To do this, you will periodically send to the server your position and your rotation. However, again you only want the script to do this if it is running on a player that represents your person character, aka if it is on a local player.

In the FixedUpdate function, check to make sure that the script is running on a local player. If it is running on a local player, then call the TransmitRotation() function.

```
void FixedUpdate()
{
    if(isLocalPlayer)
    {
        TransmitRotation();
    }
}
```

Inside of the TransmitRotation function, you will want to issue the command “SendRotationToServer”.

There are some very important things to know about these two function. The TransmitRotation function has an attribute [Client] above it. Remember that this script is running over a network, but more importantly it is also running on the server. Although you can’t “see” the server, the server has a player character on it that represents your personal character. When you have the [Client] attribute on a function, you are telling the server to ignore the function. You ONLY want to run the function on the client’s version of the script.

The “SendRotationToServer” function has the [Command] attribute. What the [Command] attribute does it allow a client version of a script to call a function on only the server. This means that the function will never be called on the client’s version of the script, but it will be called on the server’s version of the script. Any function with the [Command] attribute must begin with the ‘Cmd’ prefix. This is why the function is actually called CmdSendRotationToServer.

What you want to do, is record the position/rotation of your person character on your client, and then send that position/rotation to the server so the server can inform all of the other client’s where you are.

Inside of TransmitRotation, record your characters current rotation into the lastPlayerRotation variable.

```
[Client]
void TransmitRotation()
{
    lastPlayerRotation = myTransform.rotation;
}
```

Then call the CmdSendRotationToServer function so the server can update it’s variables with the data you are going to supply (in this case, blast your rotation to the server so it can inform all of the other clients).

```
[Client]
void TransmitRotation()
{
    lastPlayerRotation = myTransform.rotation;
    CmdSendRotationToServer(myTransform.rotation);
}
```

Inside of the CmdSendRotationToServer, the only thing you want to do is update the server’s idea of the rotation so it can tell all of the other client’s your rotation. If you look at the variable declarations, you can see that there are two variables with the attribute [SyncVar]. What this attribute does is send any changes (on the server version’s of the script) to any client’s connected to the server. So, if the server’s syncedRotation variable changes on the script that represents your player character then the server will inform all clients that the player that represents your player character has a new value in syncedRotation.

To utilize this, inside of the CmdSendRotationToServer (which is a [Command] and therefore only called on the server) assign syncedRotation to the rotation supplied via the parameter (rotationToSend). When the server receives this command, it will update the syncedRotation and then blast the update out to each client (including yourself!).

```
[Command]
void CmdSendRotationToServer(Quaternion rotationToSend)
{
    syncedRotation = rotationToSend;
}
```

Now all clients are receiving updates about your character’s rotation every update (Update is calling TransmitPosition, which is called CmdRotationToServer).

However, this is only just storing a variable somewhere. You still need to actually do something with these variables to make the characters appear to move on the non-local connections. One thing you can do is just set the rotation of every non-local player to the value of the syncedRotation. However, this will cause very “jumpy” movement. The non-local player will appear to be looking one direction, and then all of a sudden appear to be looking another direction when it gets the update to the syncedRotation from the server.

To solve this “jumpiness” you can simply interpolate between the current rotation of the non-local player, and the syncedRotation.

Inside of the Update() function, make sure that the player that the script is running on is not a local player, and if it is not a local player then call the LerpRotation() function.

Inside of the LerpRotation() function, lerp the object from its current position to its synced position, with t = Time.deltaTime * rotationLerpRate. (Lerping rotations is the same as lerping position, except instead of using Vector3.Lerp you use Quaternion.Lerp).

```
void LerpRotation()
{
    myTransform.rotation = Quaternion.Lerp(myTransform.rotation, syncedRotation, Time.deltaTime * rotationLerpRate);
}
```

CHECKPOINT!

To verify this is working, start up a host and client, and use the mouse to rotate them.

Keep in mind at this point positions are not synced!

STEP TEN: BE EFFEICIENT

At this point in time, you are syncing the updates every single update. This can lead to a large amount of network traffic. It would make sense to only blast out the position of your player if you moved enough for it to matter. At the top of the script there are Threshold variables. These variables represent the amount of movement a player needs to make before it will inform the server that it has changed location/rotation.

Implementing this threshold is as easy as determining the distance between the last location that was blasted to the server and the players current position.

You can do this by implementing Quaterntion.Angle(from, to) or Vector3.Distance(from, to).

```
#region rotation
[Client]
void TransmitRotation()
{
    if (Quaternion.Angle(myTransform.rotation, lastPlayerRotation) > rotationThreshold)
    {
        CmdSendRotationToServer(myTransform.rotation);
        lastPlayerRotation = myTransform.rotation;
    }

}
```

Note: You may have to play with the threshold a bit!

STEP TEN: LATENCY

A good thing to include in any networked game is the ability for the player to see their latency. Latency is the time it takes for information to travel from the client to the server, and then back from the server to the client. This is also called Round Trip Time (RTT). It is typical for RTT to be acceptable if it is below 200 ms, decent if it is between 200-300, and bad if it is over 300ms.

You can show the player their latency by using NetworkClient.GetRTT.

Open the Script_Latency script. Familiarize yourself with the variables inside of the script.

In the Start() function, check to make sure that this is a local player (you don't want to be displaying latencies if this isn't a local player). If it isn't a local player, then just destroy the script. It is not needed!

However, if it is a local player, then you need to somehow get access to two things: the client information (stored on the network manager) and the Text from the canvas to display numbers.

```
    void Start () {
        if (isLocalPlayer)
        {
            client = GameObject.Find("Network Manager").GetComponent<NetworkManager>().client;
            latencyText = GameObject.Find("Latency_Text").GetComponent<Text>();
        }
        else
            Destroy(this);
       }
```

Next, in every Update() you want to UpdateLatency().

Inside of UpdateLatency(), you will first want to get the RTT of the client. After getting the RTT, you can colour the text depending on how large the RTT time is, and finally display it to the client.

```
    void UpdateLatency()
    {
        latency = client.GetRTT();

        if (latency < 200)
            latencyText.color = Color.green;
        else if (latency < 300)
            latencyText.color = Color.yellow;
        else
            latencyText.color = Color.red;

        latencyText.text = latency.ToString() + " ms";
    }
```

CHECKPOINT!

You can test this by running the game and checking the lower left hand corner. As a direct connection from the same computer, your latency should be zero.

WRAP UP

You now have the tools to create a very basic networked game with a simple HLAPI.

THINK ABOUT IT!

1. What is the difference between a client, a server, and a host?
2. If you are running a host, and use [Client] and [Command] will it run both or just one on your host? Why?
3. What does the [Client] attribute mean when you use it?
4. What does the [Command] attribute mean when you use it?

ON YOUR OWN!

1. Finish the position syncing between client and server.
2. Add a new script, named player health
   a. Every time the player takes damage/healing, it should inform the server
   b. Every 2 seconds it should debug.log the player name and how much health it has
3. Add a new button in the main world scene named "Take Damage" and "Take Healing"
   a. When the player clicks take damage, it should remove health from the player
   b. When the player clicks take healing, it should add health to the player
4. Add a new Text in the main world scene named "Player Name"
   a. Put the player name (created in the main menu) inside the text