UNITY NETWORKING

MULTIPLAYER WITH STEAM

USING MIRROR AND STEAMWORKS.NET

BY FRED DUFFIELD

CONTENTS

INTRODUCTION

Networking in Unity is a task that can be approached in many ways. For this project the focus was making a game that uses the Steamworks API to connect via Steam. This will allow our finished game to use Steam servers to connect players with their friends or even with other random players in game. To connect Steamworks to our Unity game we are also going to use Mirror, which will handle the actual in game Networking. There are other options such as Proton and Unity MLAPI, which do vary slightly however the broad topics discussed here should still be relevant.

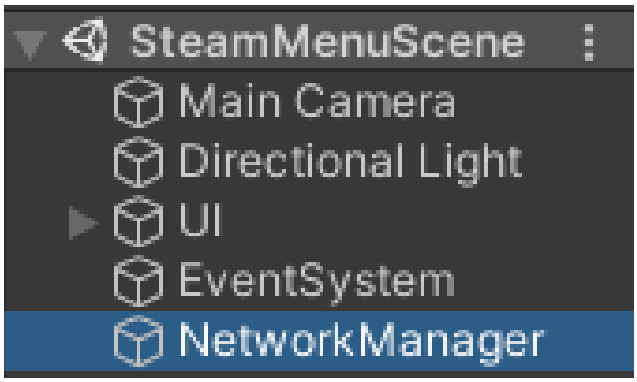
My aim for this project was to create a lobby system that connects players through Steam and allows for character customization. To do this I first focused on making a system that connects through Mirror using players IP addresses. Mirror is what handles the actual Networking of the game, and requires one player to host and the others to join as clients. Mirror is what passes the data between the different players and so this is all that is necessary for a networked game. This system worked but was not the best experience for players - having to deal with IP addresses. This is where Steam comes in. Steamworks servers can be used to find the connections between players. Once the connection is found Mirror takes over to handle the actual Networking, meaning Steamworks is only replacing the IP address input.

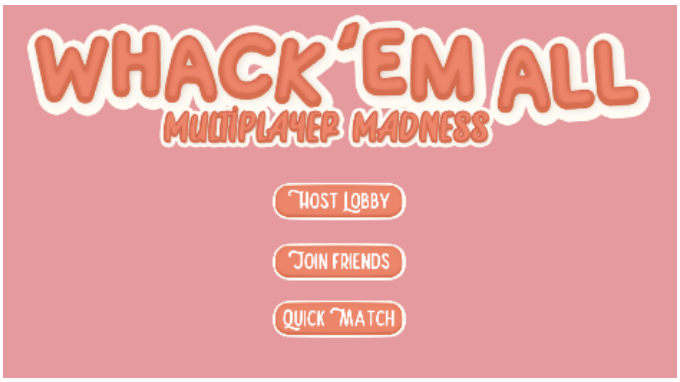
Steamworks also provides many other benefits, such as achievements, overlay integration and matchmaking. It is vital to use Steamworks to make any game that goes up on Steam, and so there's a lot to learn from the API. This devlog is of course only going to focus on the use of Steam lobbies and how to use Steamworks to create a networked game.

SETUP

One thing to consider when creating a Networked game is that most of the code will have to be written using Networking functions and syntax. Because of this it is generally easier to make a new game as opposed to trying to convert an existing game to Networking, as the gameplay programming will have to be split between being run on the Client side and being run on the Server. To set up a new project you need to import 2 things - Mirror Networking and Steamworks.API.

In the project a vital step is creating a GameObject called Network Manager. This will hold all of the important scripts concerned with Networking. You can also set up UI for interacting with the game: buttons for choosing between hosting and joining.





MIRROR CONNECTIONS

NETWORK MANAGER

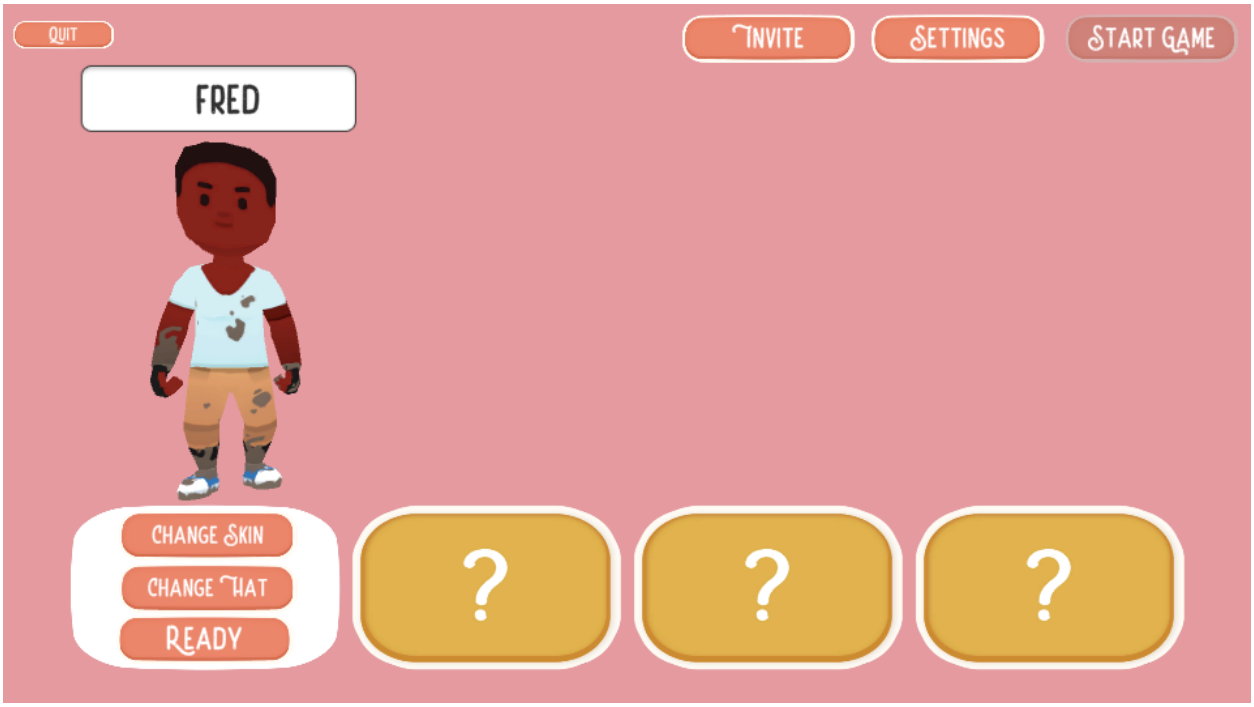
To start creating Mirror Connections you will need to create a Network Manager Override script, which will take the premade Mirror Network Manager script and allow us to add some modifications. To do this, the script needs to inherit from Network Manager as opposed to MonoBehaviour. This script will also need to use the Mirror namespace to make use of the new Mirror functions.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using System;
using System.Linq;
using Mirror;

Unity Script (2 asset references) | 17 references
public class NetworkManagerOverride : NetworkManager
{
    //name of menu scene
    [Scene][SerializeField] private string menuScene = string.Empty;
    //minimum number of players needed to start the game
    [SerializeField] private int minPlayers = 2;
    //the prefab for the player display in the lobby
    [SerializeField] private NetworkRoomPlayerLobby roomPlayerPrefab = null;

    [Header("Game")]
    //prefab for in game
    [SerializeField] private NetworkGamePlayer gamePlayerPrefab = null;
    //the player spawn system prefab
    [SerializeField] private GameObject playerSpawnSystem = null;

    //events for different cases
    //functions can be added to these events to run when called
    public static event Action OnClientConnected;
    public static event Action OnClientDisconnected;
    public static event Action<NetworkConnection> OnServerReadied;

    //in lobby list
    21 references
    public List<NetworkRoomPlayerLobby> RoomPlayers { get; } = new List<NetworkRoomPlayerLobby>();

    //in game list
    8 references
    public List<NetworkGamePlayer> GamePlayers { get; } = new List<NetworkGamePlayer>();
```

This script manages the connections between clients and the host. There are quite a few functions and scripts to this process, and I will discuss a few here, however the focus of this devlog is Steam integration. For more information about Mirror connections and what some of the more specific functions do there are some great resources linked below.

One thing to remember when working on this script (and most others when dealing with networking) is that this is going to run on both the Server and the Clients when they play the game. Because of this it is important to distinguish between which functions will be called where. For example some functions such as ***OnStartServer*** will only be called on the server when it starts, whereas ***OnClientConnect*** is only called on the clients. Some custom functions can also be restricted to only run on the server by using **[Server]**. Commands are similar, using **[Command]** restricts a function so that it can be called on clients but only run on the server.

PREFABS



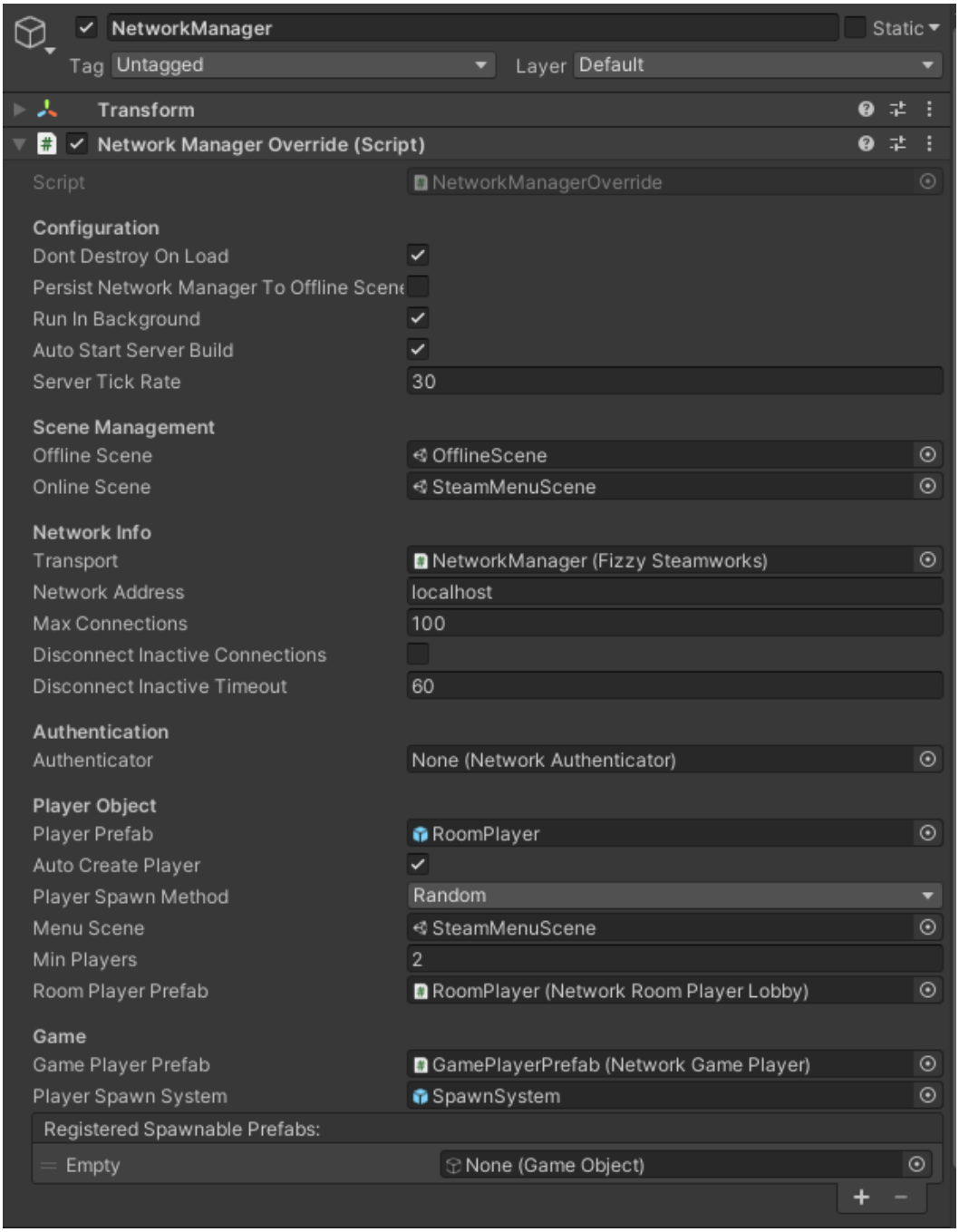
To display the players in the lobby I created a prefab to hold all the UI and character objects. This holds UI for changing the players Skin, Hat and Name. To manage this there is a separate script called CharacterLook which updates how the gameobject actually looks. The values for the characters customization is saved so when the game is started the prefabs for the game characters can be updated to match the players selection.

For prefabs to work with networking they need to be assigned to the Registered Spawnable Prefabs in the Network Override Script. One method for doing this is to add some code into the Network Override Script that automatically gets all the prefabs from a folder and assigns them to the spawnable prefabs list. This code is shown below. By doing this you can avoid the user error of not assigning new prefabs that are needed.

```
5 references
public override void OnStartServer()
{
    spawnPrefabs = Resources.LoadAll<GameObject>("SpawnablePrefabs").ToList();
}
```

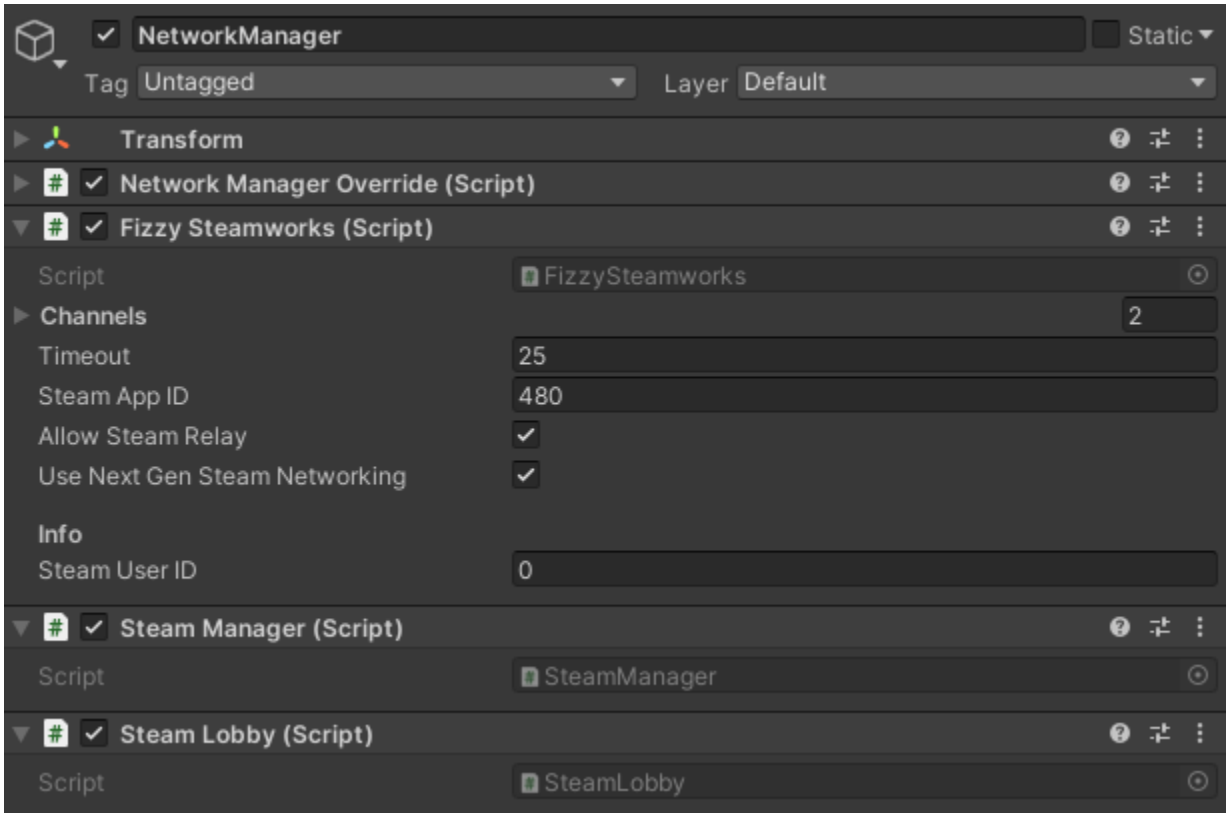
MANAGER VALUES

The Network Manager Override Script can then be added to the Network Manager Game Object. There are a lot of Public Variables shown in the inspector and these are important for getting the Networking to behave as intended. The Offline Scene is an important one, this is the scene that will be displayed when the connection is dropped. The Transport is also important - this is the actual networking transport used by the program - we will look at this more when implementing Steam.



INTEGRATING STEAM

To implement Steamworks into the game you need to add a few components to the Network Manager. Add the Fizzy Steamworks component and the Steam Manager component. To connect the Network Manager override script drag the Fizzy Steamworks into the Transport slot. You will also need to create a new script called Steam Lobby, which will manage the Steamworks API.



STEAM LOBBY SCRIPT

The SteamLobby script deals with lots of different callbacks and functions to allow the lobby system to integrate into the game. Callbacks are functions that are called from Steam when it completes a certain task. For example the game may tell Steam to retrieve data about a player and this could take a little time for Steam to complete. And so instead of the game waiting for Steam to return the data, or continuing too soon before the data is collected, Steam instead calls a new function when the task is complete. This is much more efficient and less likely to run into errors. Most of these will be covered later in this devlog, but one main function is OnLobbyCreated.

```
65      private void OnLobbyCreated(LobbyCreated_t callback)
66      {
67          //if the callback result is not ok then output error and return
68          if(callback.m_eResult != EResult.k_EResultOK)
69          {
70              Debug.Log("failed to connect");
71              return;
72          }
73
74          //otherwise start a host using Mirror
75          networkManager.StartHost();
76          //get lobby data from steam
77          lobbyID = new CSteamID(callback.m_ulSteamIDLobby);
78          //set metadata about the lobby using the lobby ID
79          //host address is the value used for other players to join the Mirror lobby
80          SteamMatchmaking.SetLobbyData(lobbyID, hostAddressKey, SteamUser.GetSteamID().ToString());
81          //This key is used to distinguish this game from any others
82          SteamMatchmaking.SetLobbyData(lobbyID, "Key", "FredsGame");
83          //hides the loading panel once it has completed loading
84          if (GameObject.Find("LoadingPanel"))
85          {
86              GameObject.Find("LoadingPanel").SetActive(false);
87          }
88      }
```

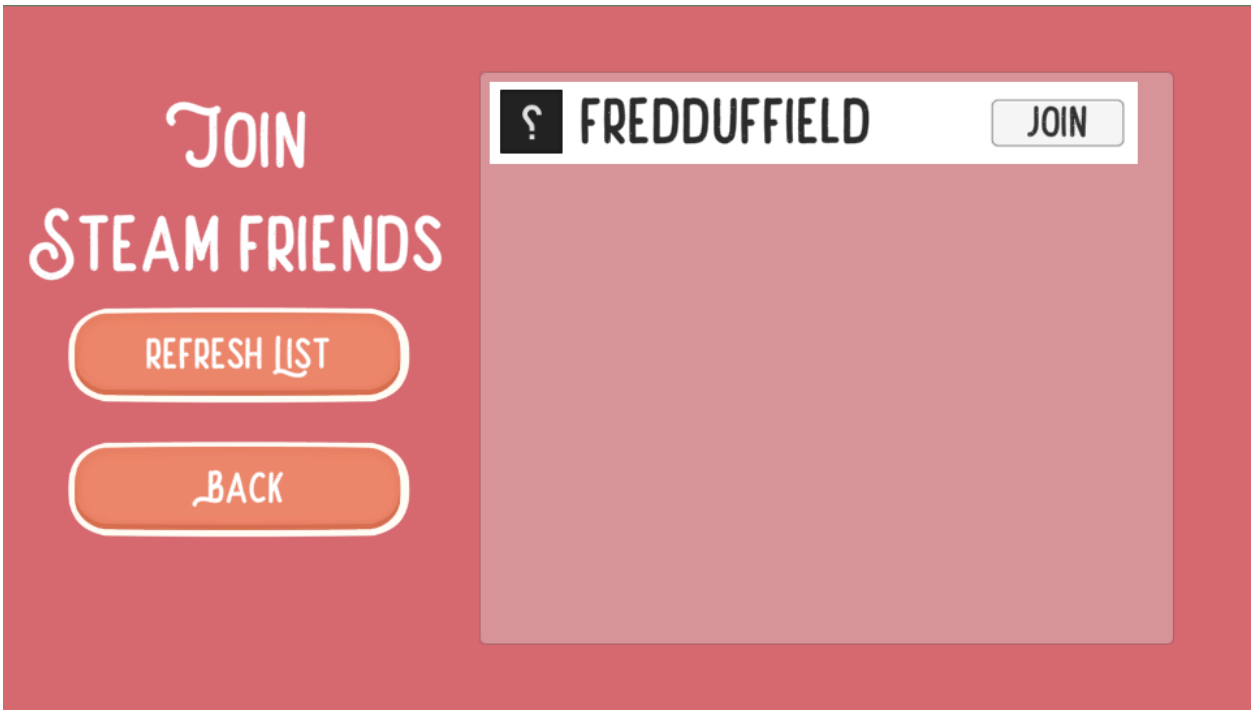
This is called when Steam has successfully created a lobby (To create a lobby simply use `SteamMatchmaking.CreateLobby(ELobbyType.k_ELobbyTypeFriendsOnly, networkManager.maxConnections);`). The OnLobbyCreated function will first check to make sure the connection was successful - see line 68 - and if so it starts a Mirror host. It then updates the metadata stored in the lobby.

Metadata can be anything and it is stored in the lobby to help all members to have some small consistent data. The first bit of metadata being set here is the hostAddress, which is the address needed for clients to connect to this Mirror host. The second bit of metadata is simply an identifier key to distinguish this game from any others that may be found when steam searches for lobbies. Metadata is important as it allows all members of the lobby to have access to the same data, and can be viewed by others outside of the lobby. This means this Metadata can be used to decide whether any one lobby is viable for a player to join by checking the data before they send a join request.

STEAM BUTTONS

This basic implementation of Steam will allow players to connect, however, they have to do so through the Steam Overlay. This works and allows the players to connect much more easily, however it could still be a lot more intuitive. To fully implement Steam Connections into the game we need to add UI. The game will have to provide the player buttons to invite or join friends, as well as showing their Steam friends in game. There are 3 different areas that we need to implement to have this working - a Join System, an Invite System and a Quick Match System. The Join System and Invite System will be similar and will allow players to connect with their Steam Friends. The Quick Match System, however, will allow players to connect to anyone else publicly playing the game.

JOIN SYSTEM



The Join System needs to search through all the user's Steam friends and return a list of those in an available lobby. We can then display this list in game and have a button for each that joins the player to that lobby. The script to find this list is shown below.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using Mirror;
using Steamworks;


Unity Script (1 asset reference) | 0 references
public class SteamFindFriendsLobbies : MonoBehaviour
{
    //display text for friends name
    public Text friendName;
    //content box to hold all list items
    public RectTransform content;
    //prefab for all list items
    public GameObject listItem;

    //list holding all spawned list items
    private List<GameObject> listOfObjects = new List<GameObject>();

    Unity Message | 0 references
    void Start()
    {
        if (!SteamManager.Initialized)
        {
            return;
        }
    }

    //refreshes the list of friends lobbies
    0 references
    public void RefreshList()
    {
        //gets the count of all player lobbies
        int friendCount = SteamFriends.GetFriendCount(EFriendFlags.k_EFriendFlagImmediate);

        //if there are more than 0 friends
        if (listOfObjects.Count > 0)
        {
            //destroys all gameobjects in list and clears the list
            foreach(GameObject obj in listOfObjects)
            {
                Destroy(obj);
            }
            listOfObjects.Clear();
        }

        // displayes no friends online if there are no
        friendName.enabled = true;
        friendName.text = "no friends online";

        //loop through all friends in the list
        for (int i = 0; i < friendCount; i++)
        {
            //gets the steam info about the friend using their index number
            FriendGameInfo_t friendInfo;
            CSteamID steamIdFriend = SteamFriends.GetFriendByIndex(i, EFriendFlags.k_EFriendFlagImmediate);

            //checks the friend is in a valid lobby
            if(SteamFriends.GetFriendGamePlayed(steamIdFriend, out friendInfo) && friendInfo.m_steamIDLobby.IsValid())
            {
                //disables the no friends online message
                friendName.enabled = false;
                //instantiates new item for in the list
                GameObject newItem = Instantiate(listItem, content);
                //adds new item to the list
                listOfObjects.Add(newItem);
                //passes data to the button
                newItem.GetComponent<JoinFriendButton>().friendNum = i;
                newItem.GetComponent<JoinFriendButton>().setName(SteamFriends.GetFriendPersonaName(steamIdFriend));
                newItem.GetComponent<JoinFriendButton>().setImage(steamIdFriend);
            }
        }
    }

}
```

The first line to receive data from Steam is line 33:

```
int friendCount = SteamFriends.GetFriendCount(EFriendFlags.k_EFriendFlagImmediate);
```

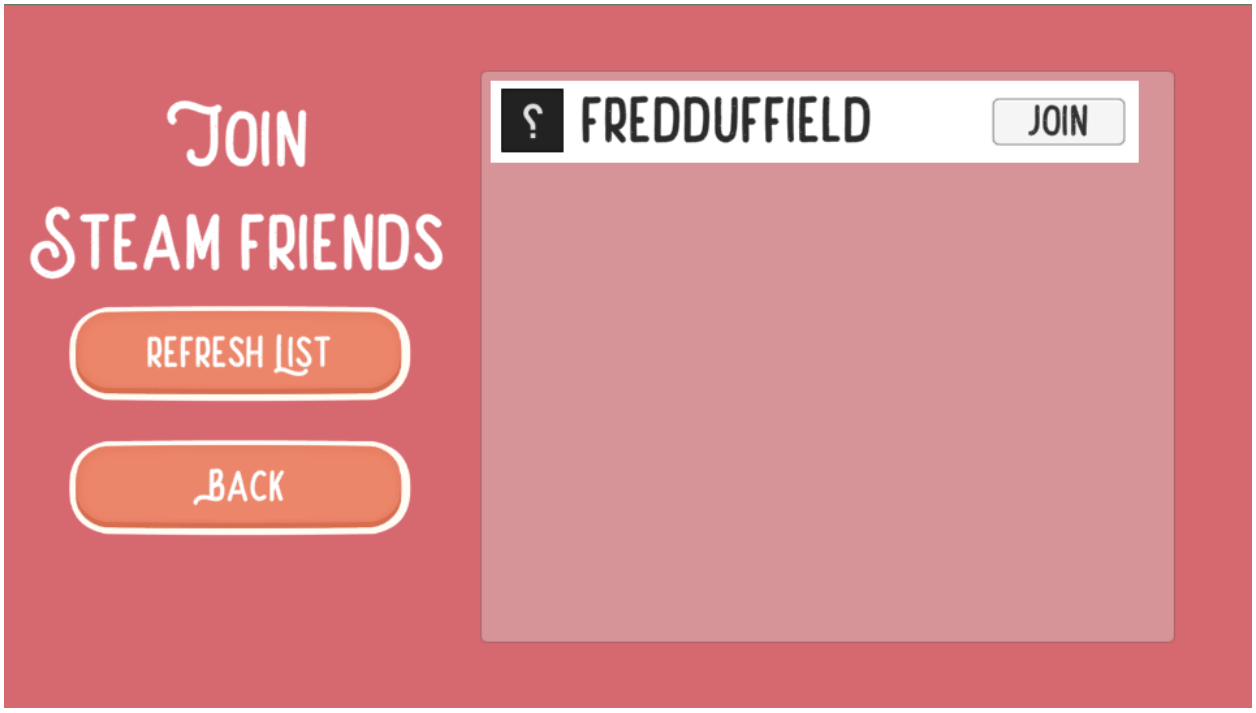
This creates a new int to store the amount of friends the player has in Steam. The parameter `EFriendFlags.k_EFriendFlagImmediate` is simply restricting the friends to the players most recent and common friends. This avoids the list being too long and full of friends the player is unlikely to play with.

This int - friendCount - is used when looping through all the friends at line 51:

```
for (int i = 0; i < friendCount; i++)
{
    //gets the steam info about the friend using their index number
    FriendGameInfo_t friendInfo;
    CSteamID steamIdFriend = SteamFriends.GetFriendByIndex(i, EFriendFlags.k_EFriendFlagImmediate);

    //checks the friend is in a valid lobby
    if(SteamFriends.GetFriendGamePlayed(steamIdFriend, out friendInfo) && friendInfo.m_steamIDLobby.IsValid())
    {
        //disables the no friends online message
        friendName.enabled = false;
        //instantiates new item for in the list
        GameObject newItem = Instantiate(listItem, content);
        //adds new item to the list
        listOfObjects.Add(newItem);
        //passes data to the button
        newItem.GetComponent<JoinFriendButton>().friendNum = i;
        newItem.GetComponent<JoinFriendButton>().setName(SteamFriends.GetFriendPersonaName(steamIdFriend));
        newItem.GetComponent<JoinFriendButton>().setImage(steamIdFriend);
    }
}
```

For each integer between 0 and friendCount we find the SteamID of that friend using `SteamFriends.GetFriendByIndex()` CSteamID's are a variable type that stores Steam's own ID's. These ID's can be used for players and lobbies so they are a valuable data type to know. Then we check whether the player is currently in a valid lobby, and if so they need to be displayed on the list. The code within this IF statement is all UI management, and so depends on how you set up your UI. In this instance I pass the basic data about the friend to another script in the gameobject so it can handle displaying this information.

For my UI I used a Scroll Box with some layout components to make sure all the items line up. A great video on how to set up a scrolling system like this is linked below. I also created a prefab for the items that are displayed in the list. This prefab has a text object to show the friends name, a raw Image to show their Steam avatar and a button to allow the player to join their lobby. There is also a Refresh button which clears the list and retrieves data from Steam again. This could be an automated process that refreshes every few seconds, however, as this game is small it seems unnecessary.



The script to manage what is displayed is attached to this prefab, meaning that each one instantiated gets its own data from Steam. This avoids having to store all the data in one place and manage that data if connections drop or lobbies change. The SetName function is incredibly straight forward:

```
46      public void setName(string name)
47      {
48          tmp.text = name;
49      }
```

However the SetImage function is a little more complex. This is because to display the Image from Steam it has to be saved and loaded into a Raw Image because it is not an asset saved in Unity.

Displaying the image is broken down into three functions: SetImage, GetSteamImage and OnAvatarImageLoaded. This is because there is a chance that the avatar image will be stored in the memory - having been loaded recently. If so, SetImage will call GetSteamImage immediately. However, if the Avatar image has not loaded yet then SetImage just returns - see line 61. This will leave the image blank for now. Then when Steam loads the image the OnAvatarImageLoaded function is called. This will then call GetSteamImage to display the Avatar. The delay taken to load this image is likely to only be frames and so the filler image will only be shown for a very short time.

```
51      //displays the steam avatar
        1 reference
52      public void setImage(CSteamID steamIdFriend)
53      {
54          //stores the friends steam ID for use later on
55          steamIdStored = steamIdFriend.m_SteamID;
56
57          //gets the friends avatar from steam
58          int imageId = SteamFriends.GetMediumFriendAvatar(steamIdFriend);
59
60          //if image hasnt loaded yet it returns -1
61          if (imageId == -1)
62          {
63              return;
64          }
65
66          //sets the texture to be the steam avatar image
67          avatar.texture = GetSteamImage(imageId);
68      }
69
        2 references
70      private Texture2D GetSteamImage(int iImage)
71      {
72          //set up new texture
73          Texture2D texture = null;
74
75          //gets the size of the image and makes sure it is valid
76          bool isValid = SteamUtils.GetImageSize(iImage, out uint width, out uint height);
77
78          if (isValid)
79          {
80              //converts image into an array of bytes - 4 bytes for each pixel: RGBA
81              byte[] image = new byte[width * height * 4];
82
83              //checks if the image can be converted to a raw image
84              isValid = SteamUtils.GetImageRGBA(iImage, image, (int)(width * height * 4));
85
86              if (isValid)
87              {
88                  //creates a new texture from the bytes
89                  texture = new Texture2D((int)(width), (int)(height), TextureFormat.RGBA32, false, true);
90                  texture.LoadRawTextureData(image);
91                  texture.Apply();
92              }
93          }
94
95          //returns the new texture
96          return texture;
97      }
98
99      //once the avatar image loads it calls GetSteamImage to display it
        1 reference
100     private void OnAvatarImageLoaded(AvatarImageLoaded_t callback)
101     {
102         if(callback.m_steamID.m_SteamID != steamIdStored)
103         {
104             return;
105         }
106
107         avatar.texture = GetSteamImage(callback.m_iImage);
108     }
109
110 }
```

The way the GetSteamImage function works is interesting. As it is taking an image that is not stored as a sprite/texture in Unity it needs to convert this image into an array of bytes and then convert it back into a texture. This is done by first taking the Steam Avatar image and checking that it is valid for use. This is almost always going to be true but it's good to have a check just in case. Then the image is converted into an array of bytes - 4 bytes for every pixel in the image to hold RGBA. As long as this is successful it passes this array into a new Texture2D and returns it so it can be shown as an image.

```
35      public void clicked()
36      {
37          //gets the friends info from steam using the index stored in friendNum
38          FriendGameInfo_t friendInfo;
39          CSteamID steamIdFriend = SteamFriends.GetFriendByIndex(friendNum, EFriendFlags.k_EFriendFlagImmediate);
40          SteamFriends.GetFriendGamePlayed(steamIdFriend, out friendInfo);
41          //joins the lobby your friend is in
42          SteamMatchmaking.JoinLobby(friendInfo.m_steamIDLobby);
43      }
```

The final functionality of this script is actually joining the room when the button is pressed. This seemed like it would be a big challenge, and during my research I struggled to find a good breakdown of the steps, however, we have already set up the actual connection code using Mirror. All this needs to do is pass some data to replace the IP address and then Mirror should be able to take it from there. So when the join button is pressed the clicked function is called. This finds the Steam info about the friend, including the Steam Lobby ID. This can then be used in `SteamMatchmaking.JoinLobby()` to join this player to their friends Steam Lobby.

This then causes a callback in the SteamLobby script: OnLobbyEntered. Being in the same Steam Lobby doesn't mean you're in the same game - remember Mirror is handling the game Networking, the Steam Lobby is just a stepping stone. As you can see in the code below taken from the SteamLobby script, the first thing that is to check if the Network Server is already running - in which case the rest of the code doesn't need to run. Then it will hide any UI that doesn't need to be there anymore. It then gets the Host Address that is stored in the lobby metadata. This is set by the host when they start the server and this is the IP address needed for other players to connect. Once this is retrieved it can be used to start a Client for the game.

```
106     private void OnLobbyEntered(LobbyEnter_t callback)
107     {
108         //if there is the network server running then return. otherwise continue
109         if (NetworkServer.active)
110         {
111             return;
112         }
113
114         //disconnect from lobby incase still connected to an old lobby
115         ClientDisconnect();
116
117         //if the join panel is active then hide it
118         if (GameObject.Find("JoinPanel"))
119         {
120             GameObject.Find("JoinPanel").SetActive(false);
121         }
122
123         //if the loading panel is active then hide it
124         if (GameObject.Find("LoadingPanel"))
125         {
126             GameObject.Find("LoadingPanel").SetActive(false);
127         }
128
129         //get the host address from the steam lobby
130         string hostAddress = SteamMatchmaking.GetLobbyData(new CSteamID(callback.m_ulSteamIDLobby), hostAddressKey);
131
132         //use host address to connect to the Mirror host
133         networkManager.networkAddress = hostAddress;
134         networkManager.StartClient();
135
136         //if title is active then hide it
137         if (GameObject.Find("Title"))
138         {
139             GameObject.Find("Title").SetActive(false);
140         }
141         //if Landing panel page is active then hide it
142         if (GameObject.Find("LandingPagePanel"))
143         {
144             GameObject.Find("LandingPagePanel").SetActive(false);
145         }
146     }
```

INVITE SYSTEM

The invite system is very very similar to the join system. Instead of being accessed on the main menu page it is only available to someone hosting a lobby. Again it uses a very similar UI with a list of all available friends in a Scroll window, each with their own button to invite them to play. When the button is pressed Steam sends the friend an invite to join the game, which pops up in the bottom corner and can be accessed in the Steam overlay.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using Mirror;
using Steamworks;

Unity Script (1 asset reference) | 0 references
public class SteamInviteFriendsList : MonoBehaviour
{
    //text used to indicate that the list is loading
    public Text loadingText;
    //the parent object for all list items
    public RectTransform content;
    //the base gameobject for the list items
    public GameObject listItem;

    //list which stores all currently instantiated items
    private List<GameObject> listOfObjects = new List<GameObject>();

    //loads all friends from steam into a list in game
    0 references
    public void RefreshList()
    {
        //gets the length of the list of all players steam friends
        int friendCount = SteamFriends.GetFriendCount(EFriendFlags.k_EFriendFlagAll);


        //if the list is not empty it destroys all gameobjects and clears the list
        //this allows for a brand new list to be formed each time
        if (listOfObjects.Count > 0)
        {
            foreach (GameObject obj in listOfObjects)
            {
                Destroy(obj);
            }
            listOfObjects.Clear();
        }

        //shows text to say there are no friends
        //this gets hidden as soon as a single friend is found
        loadingText.enabled = true;
        loadingText.text = "no friends online";

        //loops through all friends in the players steam friends
        for (int i = 0; i < friendCount; i++)
        {
            //gets the friends steam ID
            CSteamID steamIdFriend = SteamFriends.GetFriendByIndex(i, EFriendFlags.k_EFriendFlagAll);
            //hides loading text
            loadingText.enabled = false;
            //instantiates new item
            GameObject newItem = Instantiate(listItem, content);
            //adds item to list
            listOfObjects.Add(newItem);
            //passes data to the button
            newItem.GetComponent<InviteFriendButton>().friendNum = i;
            newItem.GetComponent<InviteFriendButton>().setName(SteamFriends.GetFriendPersonaName(steamIdFriend));
            newItem.GetComponent<InviteFriendButton>().setImage(steamIdFriend);


        }
    }

}
```

The code behind this is similar to that of the Join system. The script loops through all of the players Friends on Steam and instantiates a new item in the list for them. The data about each friend is passed to a script on the List items called InviteFriendButton, which sets all the UI to display the correct info for this friend. This again uses the same system to grab the friends Steam Avatar and convert it into a texture to be used in game. The main difference in this script is the clicked function, which is called when the Invite button is pressed. Instead of joining the lobby the script sends an invite to the other player. This function `SteamMatchmaking.InviteUserToLobby()` immediately returns true or false depending on whether the invite was sent successfully. This doesn't have anything to do with whether they accept or deny the invite, it just returns true if it was sent to them. To visualize this the Text changes to tell the user if they sent it successfully or not.

```
30      public void clicked()
31      {
32          //finds teh steamID for the friend using the index stored in friendNum
33          CSteamID steamIdFriend = SteamFriends.GetFriendByIndex(friendNum, EFriendFlags.k_EFriendFlagAll);
34
35          //gets the player info about this player using their steamID to get the current lobby ID
36          FriendGameInfo_t playerInfo;
37          SteamFriends.GetFriendGamePlayed(SteamUser.GetSteamID(), out playerInfo);
38
39          //invites friend to lobby - this returns true if sent successfully and false if not sent
40          if (SteamMatchmaking.InviteUserToLobby(playerInfo.m_steamIDLobby, steamIdFriend))
41          {
42              tmp.text="Sent";
43          }
44          else
45          {
46              tmp.text = "Not Sent";
47          }
48      }
```

QUICK MATCH SYSTEM

The Quick Match System is the most complex of the three, however, we can borrow heavily from the previous two Systems. The way that a quick match works is when the player presses the button the system searches for any public lobbies with available spaces in them. If one is found then the player sends a join request to that lobby and joins it. If there are no available lobbies then this player hosts a new public lobby and waits for others to join. The main difference between a Quick Match and the previous Systems is that Quick Match uses Public lobbies as opposed to Friends only. This means players will most likely be put into lobbies with people they don't know. We don't need to set up any additional UI for this, it's all done behind the scenes.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using Mirror;
using Steamworks;

Unity Script (1 asset reference) | 0 references
public class SteamQuickMatch : MonoBehaviour
{
    private NetworkManager networkManager;
    protected Callback<LobbyMatchList_t> lobbyMatchList;
    protected Callback<LobbyEnter_t> lobbyEntered;

    private const string hostAddressKey = "hostAddress";

    private int lobbyToCheck = 0;
    private int maxLobbyToCheck = 0;

    public SteamLobby steamLobby;
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        networkManager = GameObject.Find("NetworkManager").GetComponent<NetworkManager>();
        //set up callbacks
        lobbyMatchList = Callback<LobbyMatchList_t>.Create(OnLobbyMatchList);
        lobbyEntered = Callback<LobbyEnter_t>.Create(OnLobbyEntered);
    }

    //called when button clicked
    0 references
    public void clicked()
    {
        //adds a filter to the search to make sure it is only finding lobbies playing this game
        SteamMatchmaking.AddRequestLobbyListStringFilter("Key", "FredsGame", ELobbyComparison.k_ELobbyComparisonEqual);
        //requests a list of all lobbies from steam using the filter we just set above
        SteamMatchmaking.RequestLobbyList();
    }

```

```
38      //called once steam retreives the lobby list
        1 reference
39      private void OnLobbyMatchList(LobbyMatchList_t callback)
40      {
41          //if there are no lobbies then host a public lobby
42          if (callback.m_nLobbiesMatching == 0)
43          {
44              steamLobby.HostPublicLobby();
45          }
46          else
47          {
48              //if there are lobbies then call tryJoinLobby
49              lobbyToCheck = 0;
50              maxLobbyToCheck = (int)callback.m_nLobbiesMatching;
51              tryJoinLobby();
52          }
53          //loop through all lobbies in the list
54          for (int i = 0; i< callback.m_nLobbiesMatching; i++)
55          {
56              //get the lobby ID
57              CSteamID lobbyID = SteamMatchmaking.GetLobbyByIndex(i);
58
59              /// Debuging ...
72
73              //join this lobby
74              SteamMatchmaking.JoinLobby(lobbyID);
75          }
76      }
77
78      //get the lobbyID and join it
        2 references
79      private void tryJoinLobby()
80      {
81          CSteamID lobbyID = SteamMatchmaking.GetLobbyByIndex(lobbyToCheck);
82          SteamMatchmaking.JoinLobby(lobbyID);
83      }
84
85
86      //if lobby entered call back
        1 reference
87      private void OnLobbyEntered(LobbyEnter_t callback)
88      {
89          //this checks if the lobby was joined or not
90          if(callback.m_EChatRoomEnterResponse == 5)
91          {
92              //if the lobby was not entered then increase lobbyToCheck num
93              lobbyToCheck++;
94
95              //if all the lobbies have been checked and failed to join, host a new lobby
96              if (lobbyToCheck >= maxLobbyToCheck)
97              {
98                  steamLobby.HostPublicLobby();
99                  return;
100             }
101
102             tryJoinLobby();
103         }
104
105
106         if (NetworkServer.active)
107         {
108             return;
109         }
110         //hide unnessicarry UI
111         if (GameObject.Find("JoinPanel"))
112         {
113             GameObject.Find("JoinPanel").SetActive(false);
114         }
115         //join the Mirror server
116         string hostAddress = SteamMatchmaking.GetLobbyData(new CSteamID(callback.m_ulSteamIDLobby), hostAddressKey);
117
118         networkManager.networkAddress = hostAddress;
119         networkManager.StartClient();
120     }
121 }
```

Once the button is pressed the clicked function is called. This then Requests a list of all Lobbies with a certain filter - in this case the filter is to check the Metadata of the lobby and to make sure the lobby is for this game specifically.

This `SteamMatchmaking.RequestLobbyList()` returns a callback to OnLobbyMatchList once the list is received. However, this list could be empty and so the first step on line 42 is to check whether the list is empty, and if so the player must host their own lobby using `steamLobby.HostPublicLobby()`. If there are lobbies in the list the script must then loop through them and try to join each one. Failing to join will be rare, only caused by edge cases such as when the lobby fills in the time between the list being retrieved and the player trying to join, and so often the player will join the first lobby in the list. If they don't then it'll loop through all the lobbies, and if none are available it'll host a new one.

REFERENCES

To build this project I did a lot of research and found many sources that helped me to succeed. Some of the most useful references are linked here:

Steamworks API:
https://partner.steamgames.com/doc/api/ISteamMatchmaking#LobbyChatUpdate_t

Mirror Setup and linking to Steam:
https://www.youtube.com/watch?v=JJESrjLWhNM&list=PLS6sInD7ThM1aUDj8lZrF4b4lpvejB2uB&index=30

Steamworks integration explanation:
https://gemesutra.com/integration-of-your-game-in-steam-working-with-the-lobby-in-steamworks-net/

Tutorial for setting up UI layout:
https://www.youtube.com/watch?v=H9GdXiF15r8