Overview of Game:
For my Master's project, I created a 3D multiplayer virtual reality game using Unity, a C# framework, where the player's goal is to collect more points than the opponent. The players start in the center of the ground and move around the playing area to collect items, which are each worth either 2 or 3 points. Initially, in order to move, the players use the W,A,S, D, E, and C keys for moving forward, left, backward, right, up, and down respectively. The player would also use J and L to rotate their camera left and right respectively. After integrating the game with the Oculus system from the lab, the players use the joysticks on the Oculus devices to move and rotate. When other lab members are able to get the quadcopters flying, moving the ball will be determined by the inputs from the quadcopter interface as well.

My Game vs Roll-a-Ball:
My project is based on the Roll-a-Ball game shown in this tutorial https://learn.unity.com/project/roll-a-ball?uv=2019.4; however there are some differences between Roll-a-Ball and Collect. To begin with, Roll-a-Ball is a single player game, whereas Collect is multiplayer. Furthermore, Collect uses the Mirror Unity package as well as the ParrelSync editor extension to simulate a multiplayer setting and ensure consistency in the state of the game amongst the players whereas Roll-a-Ball doesn't. The player in Roll-a-Ball can move in only 2 dimensions while the players in Collect can move in 3 dimensions. Gravity is enabled in Roll-a-Ball but not Collect. Collect is playable on Oculus VR devices whereas Roll-a-ball isn't. Roll-a-Ball uses only one type of item that can be collected, each worth one point whereas Collect has two types of items each worth either two or three points. The items in Roll-a-Ball are in preset locations on a 2D plane, whereas in my game, the items are spawned at randomly generated locations in a 3D space when both players have joined the game. The dimensions of the ground plane in Roll-a-Ball are 20x20 whereas the dimensions of the ground plane in my game are 30x30. On a successful collect, Roll-a-ball only updates the scoreboard since the scoreboard is in screen space and is visible at all times. In Collect, in addition to a scoreboard update, I implemented visual and audio effects in my game since the scoreboard is in world space and may not be visible if the player is facing away from it. The scoreboard isn't visible to players when in screen space and the game is played on an Oculus device.

Implementation Details:
Scene:
I began by implementing the Roll-a-Ball game from the above tutorial with my intended differences. I first created the plane for which the items will be found and the players can roll their ball around in order to collect those items. The ball that the players each roll is a prefab created from a sphere GameObject. The items that the players collect are also prefabs, and they are created from cube GameObjects. I used a TextMeshProUGUI to display the current score for the players during the game. I also added a Network Manager to allow playing the game as multiplayer. Following this implementation, I had a meeting with my professor for feedback. I then implemented the feedback that I was given. I made my game 3D instead of 2D by disabling gravity and allowed players to move up and down. I made the camera view a cockpit view as well as allowed players to rotate their view. Steps to make this work on Oculus devices are explained later.

Scripts:
The PlayerMovement.cs script is attached to the prefab for the player sphere. This script is responsible for detecting player input from the keyboard as well as ensuring that players do not go outside of the playing area. The MoveCamera.cs script ensures that for all players, the camera is always at the same coordinate and rotation as the player and the view is always a cockpit view. This script also allowed rotating the camera using keyboard inputs before integrating with the Oculus. The BasicMovement.cs script was added when integrating with Oculus. This script when attached to the player prefab takes input from Oculus devices and moves the player accordingly. This script was provided in the original template; however, I made a few changes. I changed the class from MonoBehaviour to NetworkBehaviour in order to use Mirror. XRRig is the camera for the Oculus, so I needed to ensure that any updates to the main camera also happen to the XRRig. This includes ensuring that the XRRig has the same position and location as the player. I also made sure that the functions to move the player only if isLocalPlayer is true. In other words, I made sure when a player moves on their controller, the opponent isn't moved. The PlayerScoring.cs script tracks each player's score. Every time a player collides with a collectible object, the script detects collisions with the item, and updates the score accordingly based on how many points the item is worth. More details regarding the implementation are provided in a later section. The ResetCanvas.cs script sets the scoreboard to be at a world location in the game that is visible to both players. When using screen space to display the score, the scoreboard is visible when in the Unity editor but not the Oculus, so using world space solved this problem. The NetworkManager.cs script is responsible for operations between the server and clients in order to maintain consistency amongst the players. This will be discussed in more detail in the following sections. The PickUp.cs script is attached to the PickUp prefab. This script rotates the collectible items so that they can be easier to spot for the players. This script also turns the item's color to red if there is a player within a certain radius from the item. This is so that when a player successfully collects an item, the item doesn't just disappear; there is a visual effect. A sweeping sound from https://mixkit.co/free-sound-effects/ is also played on a successful collection.

Ensuring Consistency in the State of the Game for Both Players:
In order to ensure that the state of the game is consistent for both players, I utilized the Mirror Unity package. The source code for this package can be found at https://github.com/vis2k/Mirror and the documentation can be found at https://mirror-networking.gitbook.io/docs/. To simulate a multiplayer setting, I utilized the ParrelSync editor extension, which allows having multiple editors of the same project open. The source code for this extension can be found at https://github.com/VeriorPies/ParrelSync. The components of the game where the state needs to be consistent among both players include the location of the players, the location of each collectible item, whether each item is active, and the current score.

Players:
I followed this tutorial https://www.youtube.com/watch?v=8VVgIjWBXks to ensure consistency in the players' locations. This consisted of creating a NetworkManager GameObject with the NetworkManager.cs and NetworkManagerHUD.cs scripts attached. These scripts come with the

Mirror package. In the inspector under the NetworkManager.cs script, I selected the player prefab that I created for the “Player Prefab” field. In order to do this, my player prefab needs to have the Network Identity and Network Transform components.

Items:
In order to ensure consistency in the items, I also added the Network Identity and Network Transform Components to the prefabs. In the OnServerAddPlayer() function of the NetworkManager.cs script, I would make sure that the items spawn only when both players have joined the game, using the numPlayers field from the Mirror package for this. After instantiating the items, I made sure to call NetworkServer.Spawn() on the instantiated GameObject so that it would be spawned on all clients. Since the items are spawned from the server instead of the client, this ensures that their states are consistent.

Score:
In order to ensure consistency in the players’ scores, I added a SyncVar field to keep track of each player’s score. For the function to update this field, I made it a Command, which is a function that is called from the client and runs on the server. This ensures that the server always has the most updated state of each player’s score.

Playing the Game on an Oculus device:
In order to play the game on an Oculus Device in the lab, the Oculus device must be connected to the computer. The Oculus related packages in Unity must be installed in the Package Manager. From the Oculus, when pressing play in the Unity Editor, the game should start in 3D. Moving the left controller left, right, up, and down moves the player left, right, forward, and backward respectively. Moving the right controller up and down will move the player up and down respectively while moving the right controller left and right will rotate the player as well as the camera left and right respectively.

Playing the Game on the Quadcopter System:
As of right now, the PlayerMovement.cs script is responsible for detecting player input from the keyboard and the BasicMovement.cs script is responsible for detecting input from the Oculus devices. When running this game on the quadcopter system, these inputs will be taken from the quadcopter interface instead.