Plan Overview

Names: Aidan, Silas, Josh

Repo Link: https://gitlab.com/mtnp/gametecha4

Video Demo Link: https://youtu.be/9q9QLf7CnOM

Overall Design:

- **Multiple Game States**
    - The goal is to have both a single player mode as well as a new multiplayer mode. The start screen GUI will give players a choice to play in either mode and will set the respective game mode up based on the player's selection.
    - The single player mode is similar to our project in A2 where the player simply tries to collect 10 lives through eating orange power-ups without falling off the map or losing lives from running into cactuses.
    - The multiplayer mode is similar to our project in A3 where the player has the same goal of collecting 10 lives through orange power-ups and avoiding falling off the map/running into cactuses except with the added addition of other nodes trying to eat the oranges too. In A3, we had an AI eater node that tried to eat these oranges. In this project, we will add the support of multiple players and have a similar goal of racing against the others to get the oranges.

- **Lobby System**
    - If the player chooses to enter the multiplayer mode, they will be given more choices in the lobby. The lobby GUI will give a space to provide a username that will be used to identify the various players.
    - After providing a username, the player can choose to either host the game or join a game hosted by another player. Only 1 player can host a game, so other players will only be able to join after a player has already begun to host the game. At the moment, we have a peer to peer system implemented where the server is hosted on the host but our goal is to move this to an external server where a player still hosts the game but the game itself is hosted on an external server rather than on that player's machine.
    - After the host has started the server and all other players have joined the server, the host (and only the host) will be able to start the game. When the host chooses to start the game, all players (host and clients included) will be transported to the game simultaneously.

- **Player Changes**
    - A player will be allowed to disconnect from games as well as reconnect to ongoing games.
    - If a player disconnects from the game, this will simply be reflected on all the other players' screens with the disappearance of the player's character as well as a small note that says "<player username> has disconnected from the game".
    - If a player reconnects or simply connects to an ongoing game, they will be teleported to a spawning area and dropped into the game. This will be reflected on all the other players' screens with the appearance of the player's character as well as a small note that says "<player username> has connected to the game".
    - If a player leaves and then rejoins a game, they will have to restart at zero points.

- **Game End Condition**

        - The game ends when a player gets 10 lives by eating orange power ups. In single player mode, this is when the player gets 10 lives at any points, whereas in multiplayer mode this is when the first player gets 10 lives.
        - In singleplayer, this will give players an end screen where they can restart the game or quit the game.
        - In multiplayer, all players will be returned to the lobby screen where they can restart another game.
- **How are you thinking about the client-server system?**
    - As stated above we currently implement a peer to peer system where the host player creates the server on their machine and all other players join the host player's server as clients. However, our goal is to change this and move the server from the host's machine to an external server. At that point, the host player creates the server on an external server and all other players join the external server as clients.
- **What information will be reliable/unreliable, and when will you send it?**
    - Information that doesn't get resent and provides information that isn't recaptured will be sent reliably. For example, the scene changes must be sent reliably since we want to ensure that each player gets transported into and out of the game reliably, and this information isn't reset. Additionally, the scores of all the players will be sent reliably since that information is sent only when updated and must be kept up to date in order to maintain game state. Overall, the interactions must be sent reliably. Eating fruit to gain lives, running into cactuses to lose lives, and taking lives from other players by jumping on them all don't get resent and must be kept intact to maintain the game's state.
    - Information that gets resent often or rapidly goes out of date can be sent unreliably. For example, the general player positions are resent extremely often since players are almost always moving around. The general positions of the players are not essential to game state and are resent so often that they can be sent unreliably. However, this changes when interactions occur since the positioning matters a lot to maintaining the game state.
    - Information will be sent as soon as they occur to minimize latency and to keep game state maintained for every player. This may be hard due to the nature of the network but we plan to implement dead reckoning in order to mitigate a bit of latency related to movement and positioning.
- **What is the user flow for joining/leaving a game?**
    - We will have a GUI for joining and leaving games.
    - A player in the game may open an in game GUI in which they can exit the game and disconnect.
    - A player disconnected from the game or joining an ongoing game will join the game similarly to how they would join a fresh game. They type their username as before and click join as before. The only difference is that they are instantly transported into the game rather than waiting for the host to start the game.

Software Architecture and Plan: Peer to Peer switch to external server

We plan on removing the ai from the previous project and instead have players essentially fill the role of the ai for each other. Previously the players were racing against the thieves to get to the fruit; now they will race each other. The players will also be able to hurt their opponents by jumping on them, replacing the attacker's function. The game will be a race between the players to be the first to 10 fruits.

We are going to structure our classes such that we have some of the networking taken care of within the GUI scripts and others are taken care of in scene scripts. This is because a lot of the GUI controls are very closely intertwined with certain parts of networking such as creating servers/joining as clients as buttons are clicked (sending signals to set certain things up). We will also have game managing classes and scene managing classes that are specifically geared towards maintaining game state as well as connecting various parts of the game together.

Currently our network system is peer to peer, with one player acting as the host for the game. We plan on switching this to a client server connection. The reasons we would like to make this switch are security (players altering their number of fruit), consistency between players, and so that the game can handle any of the players' connections dropping. Currently the game starts with a menu screen with the options to host or join a game, and a start button. One player will pick host and anyone else selects join. Once everyone has joined, the host can click start to begin the game.

In general we will be using RPC calls to update game state. At the moment, we have the host play a big part in transferring information. A lot of the game state is sent to the host and then sent from the host to all other players. When we move the server from the host machine to an external server, this may change completely or may simply be modified. The host may be taken out of the picture and clients (host included) may send information to and update information with the server directly. The host may also be kept as an intermediate messenger between the server and the clients where information is sent to the server through the host and information is sent to clients from the server through the host.

We have been able to set up the basic networking structure in peer to peer in C#. This took about 2 days. With that experience, we were able to transfer this into GDScript within a single day when we realized that it better fit our needs and would be able to incorporate our previous code in GDExtension better. It will most likely take 1-2 days to set up the external server and another 1-2 days to get game state updated smoothly between all players. At the point, we will spend the rest of the 1-2 days polishing the networking and adding in the engine extension feature that will bring our networking to the next level.

More detailed networking implementations at the software level done already are covered below in the current progress section.

Engine Extension: Dead Reckoning

For our extension, we plan on implementing dead reckoning to predict player positions. All networked games will have some degree of latency just because of the nature of sending and receiving information over a network. The purpose of dead reckoning is to mitigate a part of that latency by making predictions about player positions rather than waiting for the players to reach the position and sending that information afterwards.

The players characters already have position and velocity information, so we will use that to make our predictions. If our predictions are significantly off, this may lead to some odd behavior, but when chasing fruit, we anticipate players moving in a fairly predictable manner. When the players are near each other, we may have to reduce our reliance on dead reckoning, as they may move in unpredictable ways when trying to fight each other. We may need to incorporate some extra calculations and possibly a movement prediction class that can be added to players in order to make those predictions and send them to the rest of the players. In order to keep this scalable, we will make this a generic class that can be incorporated by each player rather than hard coded information for a single player.

Division of Labor:

The goal is to have all of us equally work on setting up the basics of the network since it is very important for each of us to have a good foundation as we branch off for specific features. The goal is for each of us to take turns working on the GUI for the game states and setting up the networking in the lobby system.

From then, Aidan will work on taking care of players disconnecting during games, Josh will take care of players trying to connect to an ongoing game, and Silas will work on setting up the game end conditions where players are transported to the lobby after a game ends in multiplayer mode.

After which, we will again all take turns working on the engine extension since it is a fairly big chunk of work and will be very foreign to each of us. We will finish off by taking turns also working on polishing off the networking and making any improvements to either scalability, efficiency, or simply code structure.

We plan to meet up to code at least once every other day and plan to messages or call with progress reports every day. Our goal is to be able to stay up to date with each others' progress in the project so that work isn't done in a conflicting manner.

At this point, we have set up a general GUI and have set up the basics of the networking system. We implemented a peer to peer networking system in C# at first before realizing that C# have issues with connecting to our existing code in GDExtension. At this point, we have the peer to peer networking system and setting up the foundations of the networked game state in the converted GDScript code. We are working on spawning the players and keeping game state updated for all players. The end condition has been set up but hasn't been fully implemented for multiplayer yet. Although the code base shows Josh and Silas doing the majority of the programming, we all took turns driving and pair programming and giving each other ideas even if not reflected in the codebase.

Expectation for week1:

- Set up main menu GUI
- Set up lobby system
- Set up networking foundations for host and clients
- Update game state for all players

Expectation for week2:

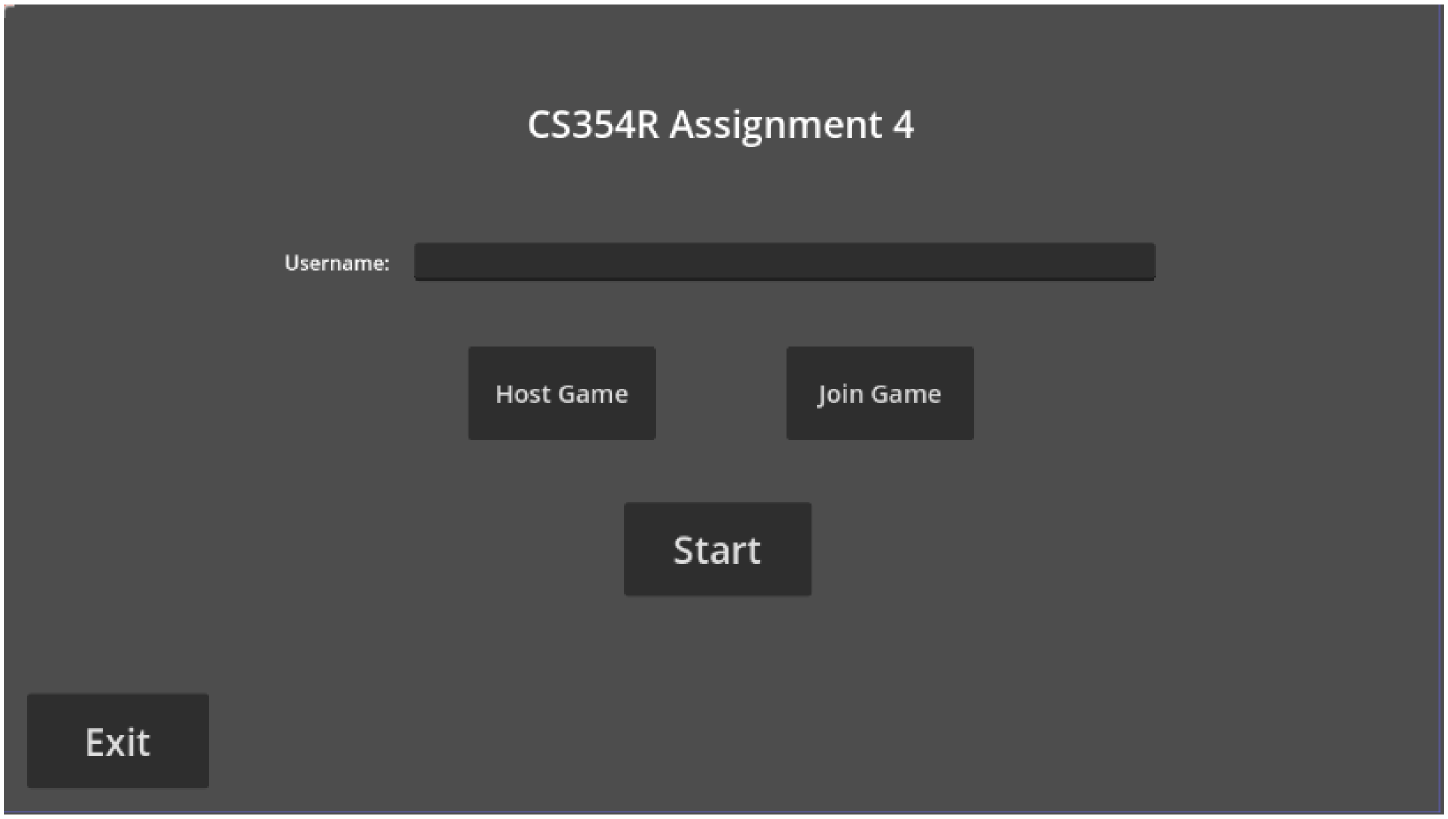
- Take care of player connecting midgame
- Take care of player disconnecting midgame
- Create game end condition for multiplayer
- Engine extension (dead reckoning)

Milestone Demo Code:
The demo code will be included in this repository in the GitLab. The controls are the same as the last assignment at the moment since we haven't implemented the player attack action yet. These controls are displayed on the GUI of the game but are basically WASD for basic movement, space bar for jump, shift and H for ledge interaction, and G for gliding. Rotation mode can be toggled with R and music/sound effects can be toggled with M and the comma (,).

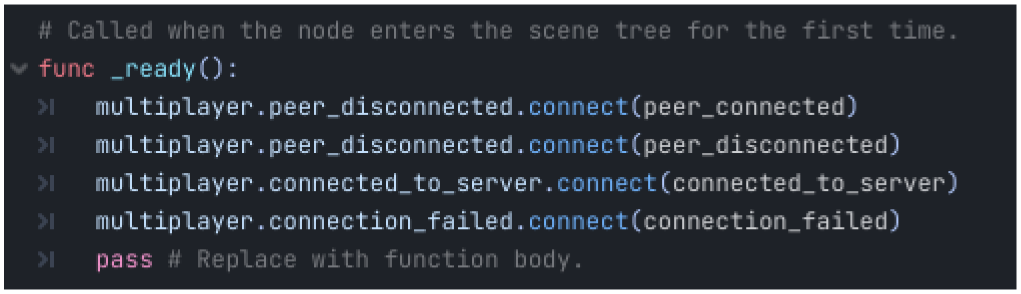
The main new pieces are the basics of networking. We have the main GUI set up where players are able to host and join games in a networked peer to peer system. This allows players to host games and for other players to join those games. We are in the process of keeping game state updated between the players but are still polishing that up. We also have the game end condition set up where they return back to the main screen and lobby area but haven't set up the ability to restart the game yet.

Current Progress:

First, we set up the multiplayer GUI such that players can enter their username before choosing to host as the server or join as a client. Once a player has chosen to host, no other players can host and trying to do so will return an error code. After all players have joined, anyone can start the game, but we are going to try to fix it such that only the host can start the game.

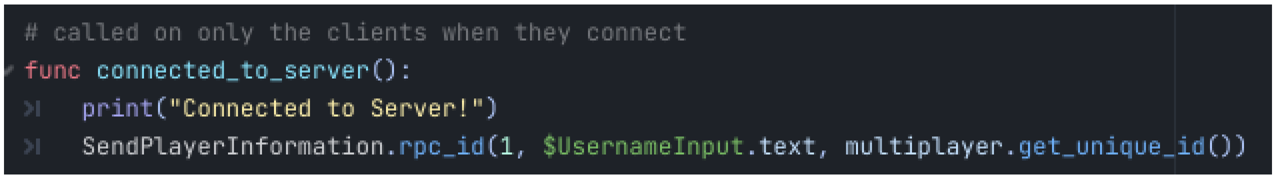


In the initial ready function, we set up 4 signals where players connect and disconnect (both for clients only as well as for clients and the server).

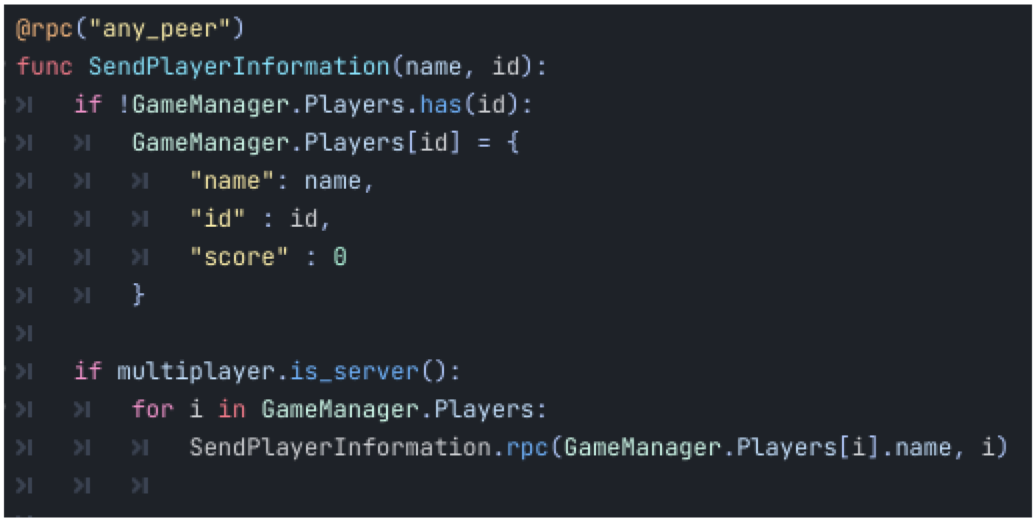


```
# Called when the node enters the scene tree for the first time.
func _ready():
	multiplayer.peer_disconnected.connect(peer_connected)
	multiplayer.peer_disconnected.connect(peer_disconnected)
	multiplayer.connected_to_server.connect(connected_to_server)
	multiplayer.connection_failed.connect(connection_failed)
	pass # Replace with function body.
```

In the method only run by clients when they connect, we send their player information to the server in the SendPlayerInformation function. That function adds the player information to a general storage data structure and then (if the server) sends that information to all other players in the game.



```
# called on only the clients when they connect
func connected_to_server():
	print("Connected to Server!")
	SendPlayerInformation.rpc_id(1, $UsernameInput.text, multiplayer.get_unique_id())
```

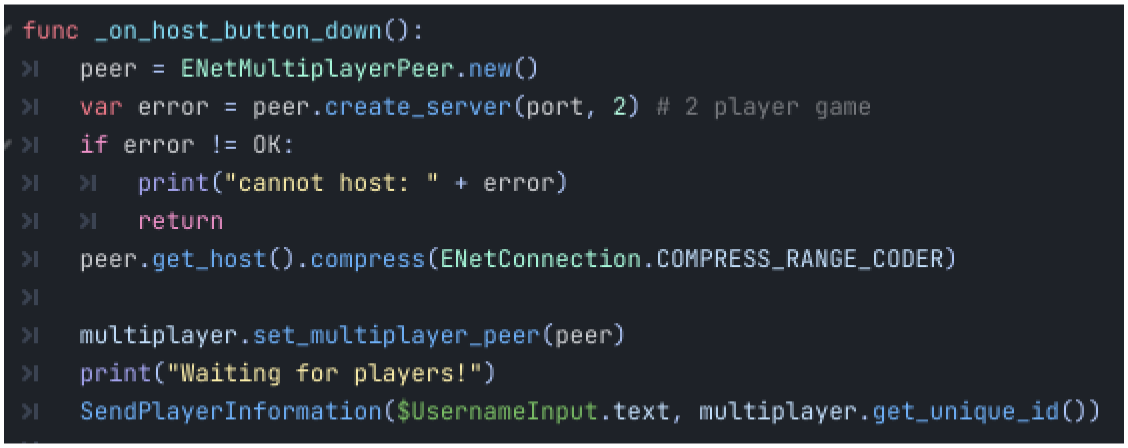


```
@rpc("any_peer")
func SendPlayerInformation(name, id):
	if !GameManager.Players.has(id):
		GameManager.Players[id] = {
			"name": name,
			"id" : id,
			"score" : 0
		}

	if multiplayer.is_server():
		for i in GameManager.Players:
			SendPlayerInformation.rpc(GameManager.Players[i].name, i)

```

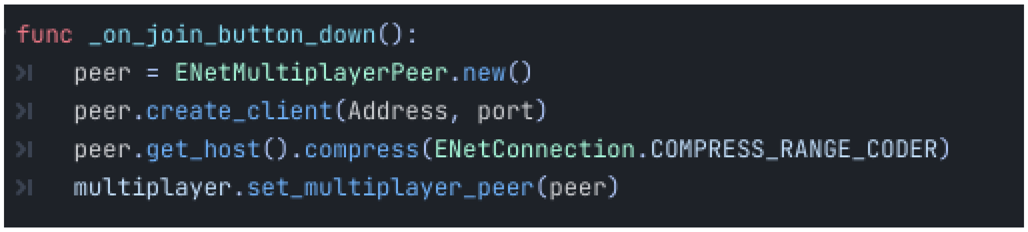
These 2 signals control the networking when a player chooses to host and join a server. When a player chooses to host, it runs the first chunk of code. It creates a new MultiplayerPeer instance and uses it to create a new server (with a maximum of 2 players for our game at the moment). We do a quick error check and some compression configuration before adding this player to our general storage data structure.



```
func _on_host_button_down():
	peer = ENetMultiplayerPeer.new()
	var error = peer.create_server(port, 2) # 2 player game
	if error != OK:
		print("cannot host: " + error)
		return
	peer.get_host().compress(ENetConnection.COMPRESS_RANGE_CODER)

	multiplayer.set_multiplayer_peer(peer)
	print("Waiting for players!")
	SendPlayerInformation($UsernameInput.text, multiplayer.get_unique_id())
```

This second signal controls the networking when a player chooses to join a server. When a player chooses to join, it runs this chunk of code. It creates a new MultiplayerPeer instance and uses it to create a client on the server's port (uses the host's local server at the moment but will be changed to an external server soon). We set up some compression configuration again.



```
func _on_join_button_down():
	peer = ENetMultiplayerPeer.new()
	peer.create_client(Address, port)
	peer.get_host().compress(ENetConnection.COMPRESS_RANGE_CODER)
	multiplayer.set_multiplayer_peer(peer)
```