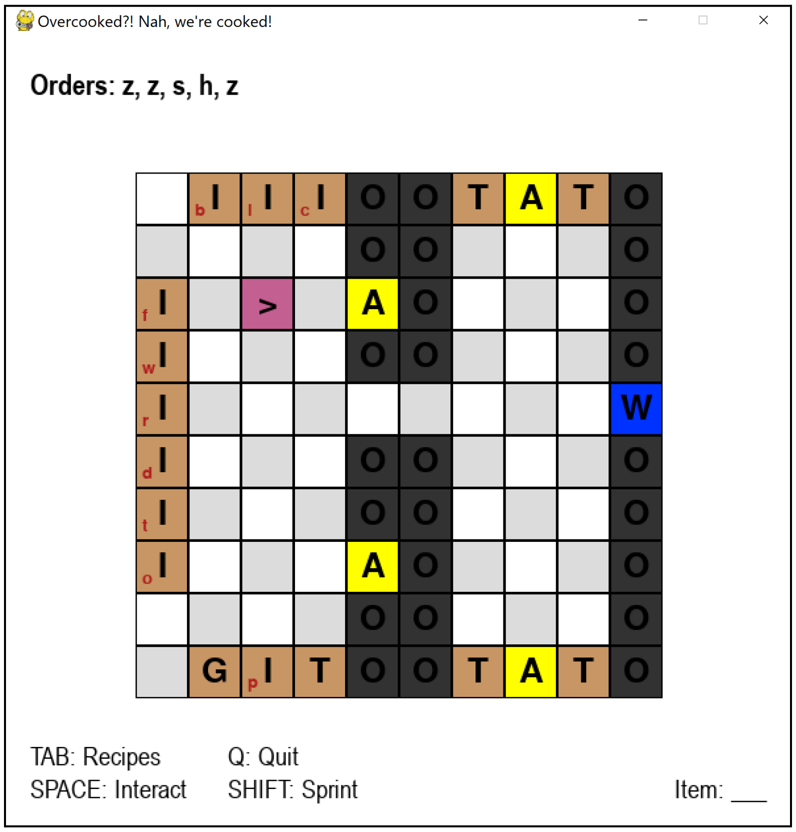
Project Report: Overcooked? Nah We're Cooked!

Authors: Ali Devjiani, Fraser Wong, Jin Yang, Ariel Lin

1. Project Overview



“Overcooked? Nah We’re Cooked!” is a multiplayer co-op cooking game inspired by the popular video game series, *Overcooked!* Players work together in a virtual kitchen to prepare and serve food. Players are assigned a list of food orders that must be fulfilled by gathering the appropriate ingredients and assembling them in the correct sequence. Once the ingredients have been prepared, the completed dishes need to be brought to the serving window to fulfill the orders. The game concludes once all assigned orders have been completed. A working demo of the video can be accessed here:

Demo link: https://www.youtube.com/watch?v=9wXEmCheVOE

Core Game Mechanics:

- **Multiplayer support**: Up to 4 players can connect simultaneously to a server
- **Food Preparation**: Players need to collect ingredients and assemble them at stations to create dishes
- **Order Management**: A list of orders is randomly generated at the start of each game
- **Real-time Collaboration**: Players need to work together in a kitchen grid and complete the task list

Supported Recipes:

- Hamburger (h): Bun (b) + patty (p) + cheese (c) + lettuce (l) + bun (b)
- Sushi (s): Seaweed (w) + rice (r) + fish (f)
- Pizza (z): dough (d) + tomato (t) + cheese (c) + pepperoni (o)

---

2. Game Design

2.1 Planning

The first step of designing this game included discussing what we want our game to entail. We discussed the domain of actions a player could take, as well as how to implement it. This included movement and ingredient interaction. We also discussed the end goal for the game and ultimately decided that completing a list of orders that were randomly generated would be great. Following that, we discussed the specifics of what items the orders could consist of, as well as the required ingredients and recipes. After deciding on the specifics of the game itself, we moved on to discuss the client and server designs as well as multiplayer support, which are discussed in more detail in the next section. We also discussed what our shared object would be, we ended up with a few shared objects, that include ingredients, and grid positions. We explain how we deal with this in more detail when handling race conditions.

2.2 Architecture

The game uses a centralized server architecture where:

- **Server**: Handles the game state, resources, player interactions, and shared task list
- **Client**: Handles user input, game grid rendering, and showing the game state.

The game world is represented by a 10 x 10 grid containing:

- Ingredient Bins (I): Source of ingredients
- Tables (T): Storage for Items
- Assembly Stations (A): Where dishes are prepared and created
- Garbage Bins (G): Waste disposal
- Service Window (W): Where food is served.

---

3. Client Server Interaction

3.1 Design Overview

The game uses a client-server architecture where the server is responsible for placing each client-controlled player on a grid. Player movement is initiated on the client side through keyboard inputs, which are sent to the server for processing. After handling the input, the server updates the game state and returns the modified grid to the client for display. This allows for a simple communication system that minimizes redundant information since all the computation is done by the server. The game is implemented in Python, with the Pygame library used to develop the graphical user interface on the client side. Communication between the server and clients is handled via a TCP connection, supplemented by a heartbeat mechanism to ensure that clients continue to receive periodic updates from the server even in the absence of player input to ensure that information the player sees is up to date.

3.2 Application-Layer Messaging Scheme

Client-to-Server Communication: The client can send player movement and player interactions to the server through a key queue. These messages are encoded as UTF-8 strings and sent to the server. The supported messages include

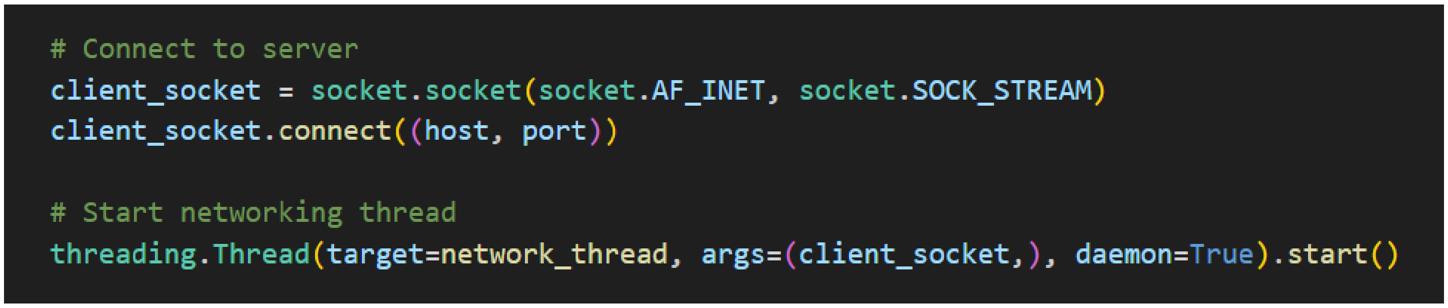
- up: Moves the player upward
- down: Moves the player downward
- left: Moves the player left
- right: Moves the player right
- interact: Interact with a station, depending on the situation pick up or drop off item
- heartbeat: Requests status update from the server
- quit: Closes the client connection

Server: Once the server has processed the client request, it will send an encoded JSON object containing

- grid state: Current layout of the game grid (includes players)
- player inventory: Current item the player is holding
- task list: List of food orders
- task list completion flag: Boolean flag indicating the tasklist completion

3.3 Client Side Design

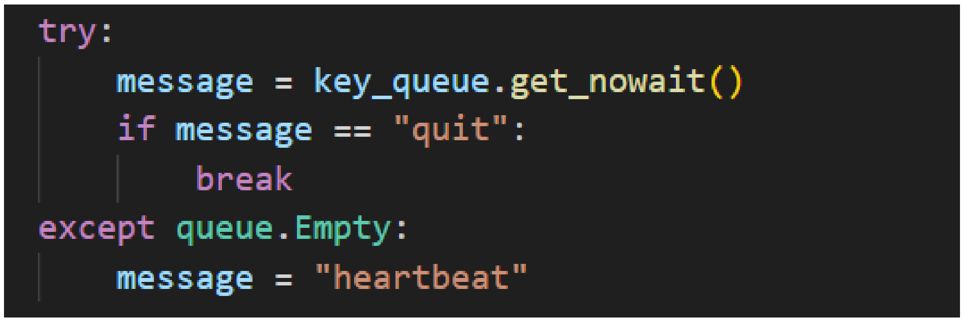
We wanted to ensure that our game had a good playing experience, this meant that we needed to ensure that the gui was separated from the client server interaction to reduce delays. We used threading to run the communication, which ensured that the player inputs could be handled at the same time. This is shown below in Figure 1.



```python
# Connect to server
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((host, port))

# Start networking thread
threading.Thread(target=network_thread, args=(client_socket,), daemon=True).start()
```

Figure 1

Additionally, we used a queue to store messages that needed to be sent to the server, this ensured reliability while also allowing us to send a heartbeat message that prompts the server to send updated information constantly. This is shown below in Figure 2.

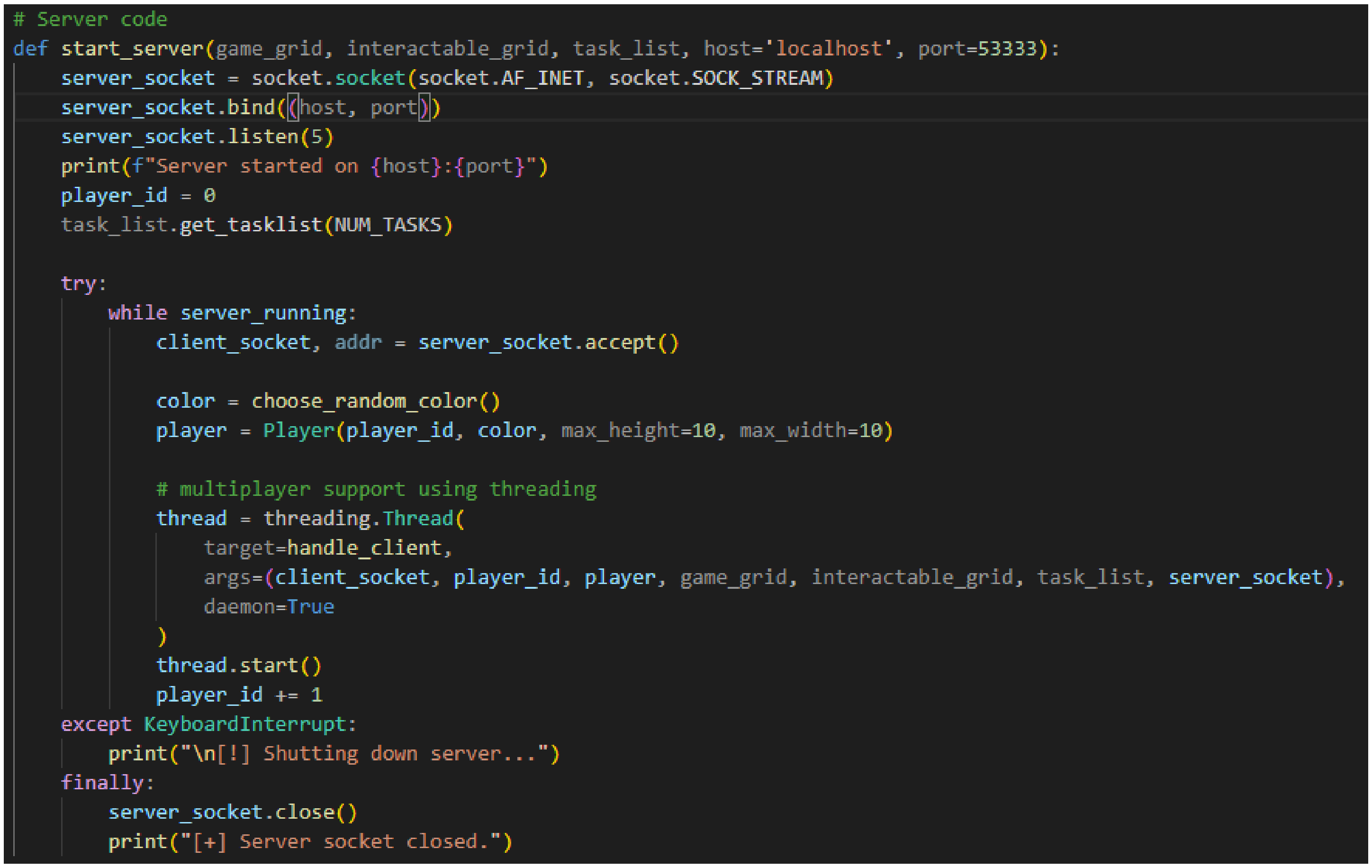


```python
try:
    message = key_queue.get_nowait()
    if message == "quit":
        break
except queue.Empty:
    message = "heartbeat"
```

Figure 2

3.4 Server Side Design

Our server design required a few key features. Firstly, we had to ensure multiplayer support. This was done by handling each connected client on a different thread. The creation of the server socket, as well as multi client handling is shown in Figure 3.



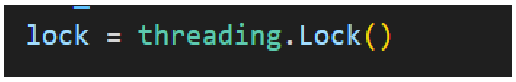
```python
# Server code
def start_server(game_grid, interactable_grid, task_list, host='localhost', port=53333):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(5)
    print(f"Server started on {host}:{port}")
    player_id = 0
    task_list.get_tasklist(NUM_TASKS)

    try:
        while server_running:
            client_socket, addr = server_socket.accept()

            color = choose_random_color()
            player = Player(player_id, color, max_height=10, max_width=10)

            # multiplayer support using threading
            thread = threading.Thread(
                target=handle_client,
                args=(client_socket, player_id, player, game_grid, interactable_grid, task_list, server_socket),
                daemon=True
            )
            thread.start()
            player_id += 1
    except KeyboardInterrupt:
        print("\n[!] Shutting down server...")
    finally:
        server_socket.close()
        print("[+] Server socket closed.")
```

Figure 3

The handle client function, which was the target for each of these threads, handled player movement, interaction, as well as packaging and sending responses to the client.

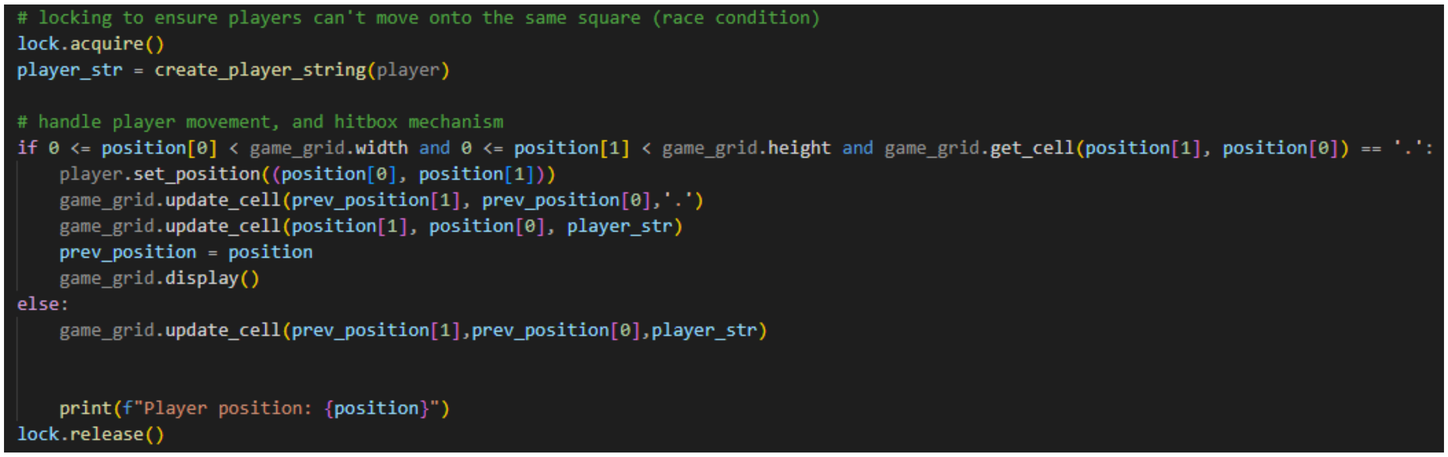
Race Conditions:

The addition of multi client support also included the addition of some race conditions. Since we wanted only 1 player to be able to occupy a grid position at any time we needed to handle the case of 2 clients checking if a grid position is empty and updating the grid at the same time. To do this, we used a lock to ensure that only 1 thread could check if a grid position is empty, and update the grid with the respective player's new position at any point in time. This is shown in Figures 4 and 5.



```python
lock = threading.Lock()
```

Figure 4


```
# locking to ensure players can't move onto the same square (race condition)
lock.acquire()
player_str = create_player_string(player)

# handle player movement, and hitbox mechanism
if 0 <= position[0] < game_grid.width and 0 <= position[1] < game_grid.height and game_grid.get_cell(position[1], position[0]) == '.':
    player.set_position((position[0], position[1]))
    game_grid.update_cell(prev_position[1], prev_position[0],'.')
    game_grid.update_cell(position[1], position[0], player_str)
    prev_position = position
    game_grid.display()
else:
    game_grid.update_cell(prev_position[1],prev_position[0],player_str)


    print(f"Player position: {position}")
lock.release()
```
Figure 5

In addition to that race condition, we also needed to handle a similar case where 2 players try to pick up an item at the same time. This was handled using clever map design such that only 1 player can interact with an interactable object that would require locking at any time.

4. Challenges

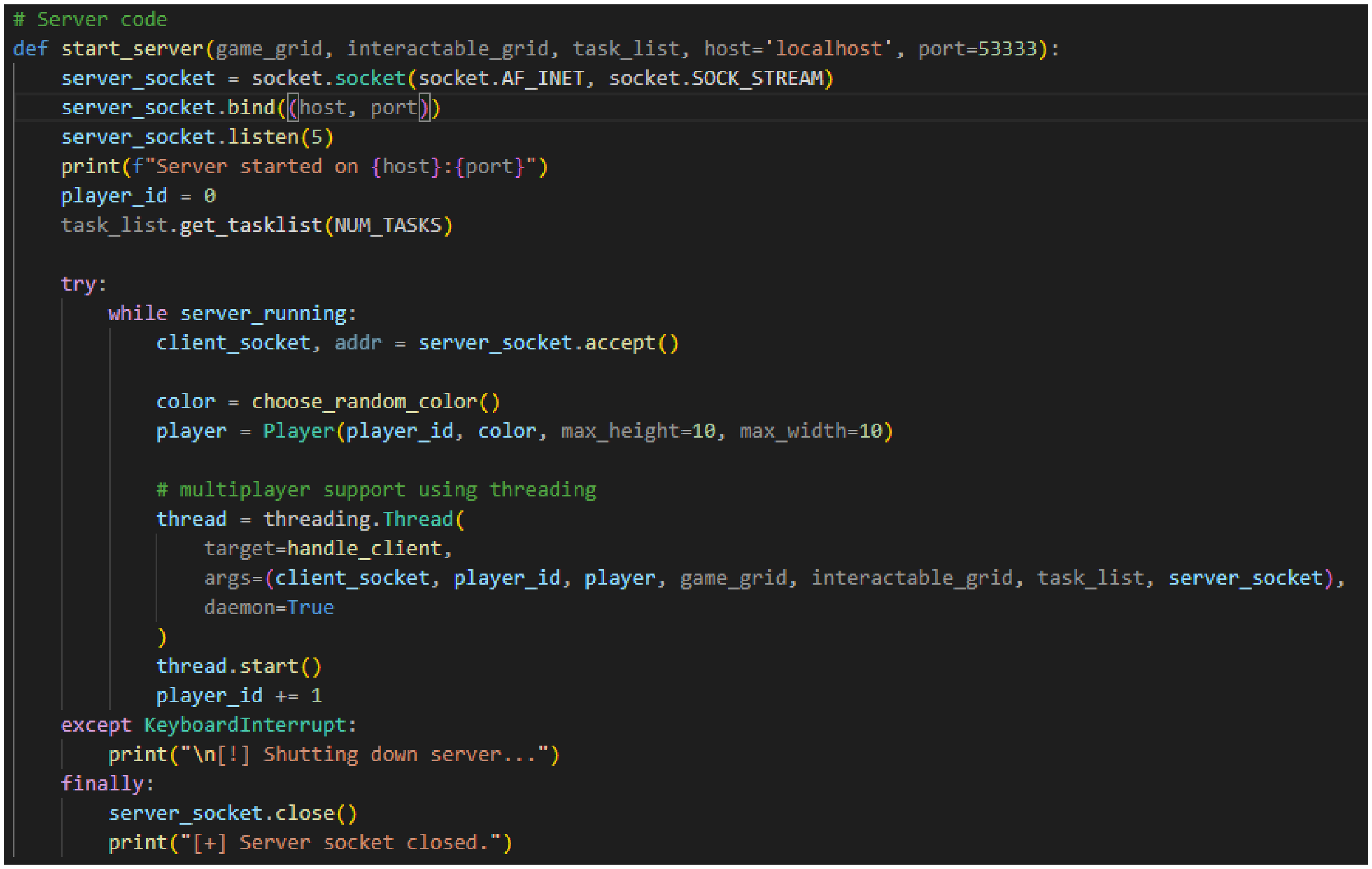
We dealt with a few challenges throughout the course of this project. The first big challenge we dealt with was regarding the client server interaction. We initially tried 2 different approaches, one was to have the server send the whole grid and having the client just display the gui in the terminal. This approach handled everything on the server side. The second approach was to have computation on both the client and server side, while using pygame to display the user interface. We ended up deciding to merge these 2 approaches by keeping the client server interaction simple, where the server handles any computation, and the client side unpacks the response information and uses pygame to make a graphical user interface.

We also had a few issues initially, with handling how information was sent to the client since the client needed more than just the grid for the game to function. We eventually discovered that the method we used to send the grid actually allowed us to append additional information and package the response to be sent to the client, which it could then unpack and use as necessary. This allowed us to implement the task list functionality, item HUD, and also make every player a distinct color.

5. Code Snippets

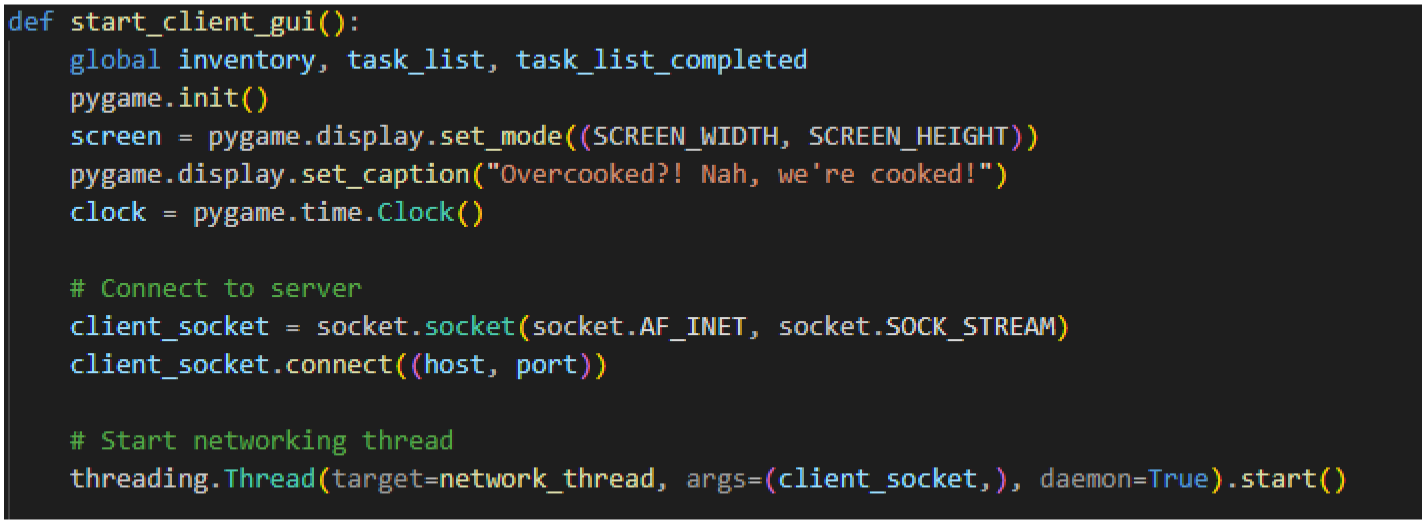
5.1 Opening sockets

Server



```python
# Server code
def start_server(game_grid, interactable_grid, task_list, host='localhost', port=53333):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(5)
    print(f"Server started on {host}:{port}")
    player_id = 0
    task_list.get_tasklist(NUM_TASKS)

    try:
        while server_running:
            client_socket, addr = server_socket.accept()

            color = choose_random_color()
            player = Player(player_id, color, max_height=10, max_width=10)

            # multiplayer support using threading
            thread = threading.Thread(
                target=handle_client,
                args=(client_socket, player_id, player, game_grid, interactable_grid, task_list, server_socket),
                daemon=True
            )
            thread.start()
            player_id += 1
    except KeyboardInterrupt:
        print("\n[!] Shutting down server...")
    finally:
        server_socket.close()
        print("[+] Server socket closed.")
```

Client

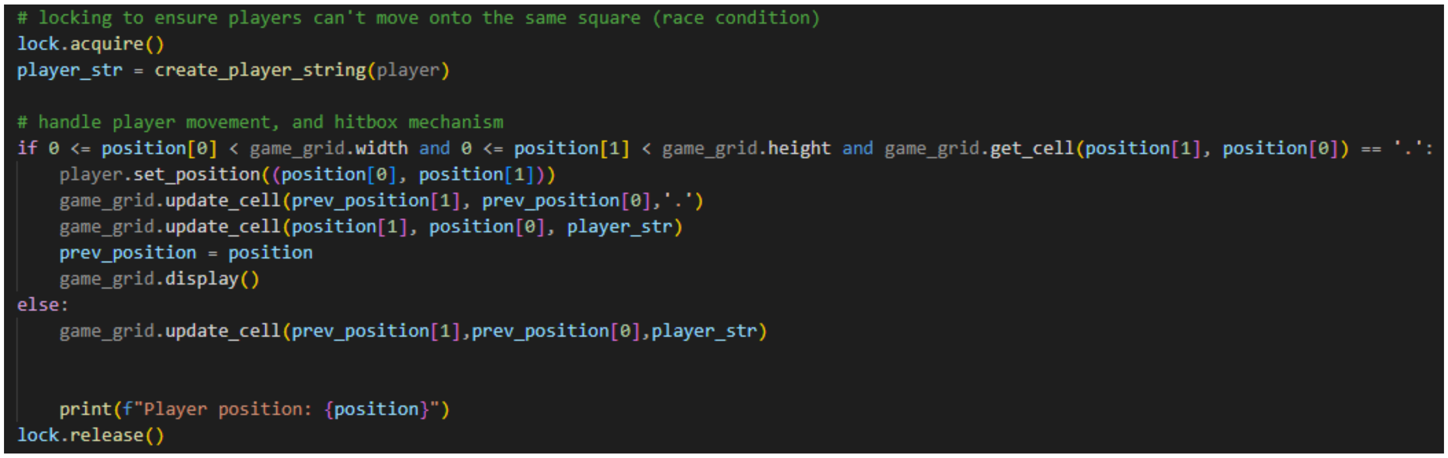


```python
def start_client_gui():
    global inventory, task_list, task_list_completed
    pygame.init()
    screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
    pygame.display.set_caption("Overcooked?! Nah, we're cooked!")
    clock = pygame.time.Clock()

    # Connect to server
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((host, port))

    # Start networking thread
    threading.Thread(target=network_thread, args=(client_socket,), daemon=True).start()
```

5.2 Handling Shared Objects

```
lock = threading.Lock()
```

…



```
# locking to ensure players can't move onto the same square (race condition)
lock.acquire()
player_str = create_player_string(player)

# handle player movement, and hitbox mechanism
if 0 <= position[0] < game_grid.width and 0 <= position[1] < game_grid.height and game_grid.get_cell(position[1], position[0]) == '.':
    player.set_position((position[0], position[1]))
    game_grid.update_cell(prev_position[1], prev_position[0],'.')
    game_grid.update_cell(position[1], position[0], player_str)
    prev_position = position
    game_grid.display()
else:
    game_grid.update_cell(prev_position[1],prev_position[0],player_str)


    print(f"Player position: {position}")
lock.release()
```

---

6. Group Members and Contributions

Ali Devjiani: 25%
Ariel Lin: 25%
Fraser Wong: 25%
Jin Yang: 25%

---

7. Source Code Repository

The complete source code is available at: FraserW123/Overcooked-Nah-We-re-Cooked-cmpt-371-project

---

8. References

Overcooked 2 | Overcooked 2 PS4 | Team17
Pygame Front Page — pygame v2.6.0 documentation
Shervin's Lectures