Game Description

The game has the same rules as Deny and Conquer. Players will use their mouse and try to fill as many squares as possible on the shared gameboard. To fill a square, the player must draw and color in the majority of the square, to which the server will then lock in that player's color for that square. The player with the most squares in their color wins once all squares are filled in.

Game Mechanics

If multiple players contest for the same square, only one player will get access (first come first serve) and all other players will be locked out. To attempt to gain access to a square, the player must left click anywhere in it. Once exclusive access is granted, the player must continue holding the left click to fill it in. Upon releasing the left click, the player forfeits access. Squares are permanently filled in (and hence awarded to the player) when the player has scribbled in the majority of the square.

Players will need to be fast in order to contest and fill in as many squares. However they must balance speed with also filling in an adequate amount in a square. If a player does not fill in the square enough and releases access, the square resets and becomes open for anyone to try and color it to their own.

Application Dependencies

A dependencies file is included with external modules required to successfully run the game application. Prior to application use, users are required to execute the following command in their command line terminal:

pip install -r requirements.txt

Organization:

The application is organized into three base classes: Box, Board and Player, a client file, and a server file. The base classes are used to implement the gameboard and game play. The client and server files are responsible for opening and maintaining a connection between the client and server and sending and receiving updates about game progress. The client and server files import the three base classes, and the three base classes interact with each other to emulate the game environment and the rules of the game.

Base Classes:

Box:

The Box class imports the external Pillow module containing classes Image and ImageDraw for frontend implementation. It also imports the standard Python threading module to allow concurrency for Box objects.

Purpose:

This class is used to represent each individual cell of the gameboard grid. Modularizing the grid is necessary to accommodate multiple users accessing, interacting with, and making modifications to a single gameboard simultaneously.

Important Attributes:

The attributes of the Box class allow for safe sharing of Box objects amongst several clients and are used to enforce the rules of the game.

The “lock” attribute allows for safe concurrent gameplay. It initializes an object of type Lock from the imported threading module. The Box class constructor instantiates the “lock” attribute and the Player class handles appropriate acquisition and release of the lock to ensure that only one player may access a Box object at any instant in time. The remaining attributes: “is_taken”, “owner”, and “color” are responsible for recording information pertinent to keeping track of score, game progress and enforcing game rules. The attributes are initially set to False, None and None respectively to represent a blank, unconquered white cell on the gameboard. The values are set by functions in class Player when the server receives data that a cell has been filled past the 50% threshold required to conquer a cell. Boolean attribute “is_taken” prevents other players from attempting to conquer a cell that has already been conquered. Attributes “owner” and “color” are used for accounting purposes to maintain a record of which players have conquered which boxes, and to fill the color of those boxes accordingly.

Important Functions:

The most important function in the Box class is scribble. The function takes the player (of class Player) and the coordinates of the mouse click as input parameters. It utilizes the external Pillow module classes Image and ImageDraw to allow a player to draw on the gameboard and fills the box on the gameboard with the color associated with the Player input parameter.

Board:

The Board class imports the Box class, as a Board object comprises of many Box objects, and the external Pygame module for UI interaction.

Purpose:

This class is used to create and represent the gameboard itself. Its main purpose us to act as a container for an indexed grid of Box objects and to provide an organized way for the application to access specific Box objects within the grid at a given set of coordinates.

Important Attributes:

The Board class features the attribute “boxes”, a 2D array of Box elements used to represent a modularized gameboard. The 2D array organizes the grid into columns, or the outer array, and rows, or the inner array. The row and column values also serve as an XY axis and allows for coordinates on the gameboard to be mapped to index values.

Important Functions:

The functions most essential to gameplay in this class are as follows: get_current_box, is_game_over, string_to_board, and board_to_string. The get_current_box function takes the coordinate of a player’s mouse click as input parameters and uses the size of a box to determine the equivalent index of the Box element being accessed within the 2D array of Box elements by dividing the given coordinates by the size of a box. The is_game_over function iterates over the 2D array of Box elements checking each element’s “is_taken” value to determine whether or not each box on the gameboard has been successfully conquered. This function is called in a loop in the server file to continuously check whether or not the game should be concluded by the server. The string_to_board and board_to_string functions are used to map a Board object to an equivalent String representation. These methods are important for improving communication quality between the client and server. The server is responsible for broadcasting updated copies of the gameboard to all connected clients. While it is possible to deserialize and serialize the complete Board object via pickling and unpickling the data prior to and following transmission, it is not necessary and accurate translation of the object data is not guaranteed. To sidestep this approach, the board_to_string method creates a String equal to the total number of Box elements in the 2D array. The function iterates over the array and sets each index of the String to one of the following characters: ‘R’, ‘G’, ‘B’, ‘Y’, or ‘0’ representing red, green, blue, yellow or blank cells on the gameboard, respectively. The string_to_board function iterates over the indices of the String and applies the changes to the associated Box element in the local 2D array. These functions allow for frequent updates of the real-time status of the gameboard while limiting the size of the data transfer required.

Player:

Purpose:

This class represents a client as a player of the game executes the actions or moves performed by players to progress towards the end goal of conquering boxes to complete the game.

Important Attributes:

The attributes of this class are used to represent the player themselves, as well as to represent the player's current state or current action. The attributes "color_key" and "taken_boxes" represent the player. They're used to record the color associated with the player and their current score. The attributes "current_box", "drawing_flag", and "threshold_reached" are used to represent the player's current action. The "current_box" attribute indicates which Box of the 2D Board array is currently being accessed by the user. This attribute is an alias and points to the same memory address as the Box in the client or servers local 2D Board array. The "drawing_flag" is set to True when a player is actively drawing in the previously mentioned "current_box" and the "threshold_reached" attribute is set to True when the player successfully fills more than 50% of the "current_box". Together, the value of these attributes can be used to assess which box on the gameboard should be blocked from being accessed and whether or not a box should be filled with a specific color.

Important Functions:

The most important functions of this class are start_drawing, stop_drawing, and stop_drawing_server_colored. They are responsible for acquisition and release of the "lock" attributes of the Box elements of the local 2D Board array. The stop_drawing function attempts to acquire the lock of its own "current_box" attribute. If it finds that the lock is not available, it returns a String reading "box taken". This String is used for communication between the client and server to relay that the box the client is trying to access is in use. Next, it checks if the Box attribute "is_taken" is True, if so, it is communicated between the client and server that the box is not available. Otherwise, the function allows the user to draw in the box and attempt to conquer it. The stop_drawing function releases the lock so that the box can once again be accessed by other players (as long as it hasn't been conquered). This function also records the number of pixels of the box that were filled during the player's move and uses the recorded value to determine the percentage of the box that was filled. If more than 50% of the box was filled, the "threshold_reached" attribute is set to True and this information is communicated to the server. The stop_drawing_server_colored function is only called if the "threshold_reached" attribute value is True. This function changes the parameters of the Box object to reflect that it was successfully conquered and by whom it was conquered. These functions are important for controlling access to Box objects on the gameboard by acquiring and releasing the lock following the appropriate events logged from the client and also for progressing the game towards completion.

Class Interactions:

Box and Board:

A Board object possesses an attribute of a 2D array of Box elements. The Board class interacts with the Box class in its board_to_string, string_to_board, and draw_boxes functions, where it accesses a specific Box element within its 2D array and accesses the given Box's attributes using the dot operator.

Box and Player:

A Player object possesses the attribute "current_box", which is an object of class Box. The Player class interacts with the Box class by using the Box object to call the scribble function in its start_drawing function implementation. The Player class implementation also involves accessing and modifying Box attributes in its start_drawing, stop_drawing, stop_drawing_server_colored, and continue_drawing functions. The Player class is used to acquire and release a Box's "lock" attribute, and to modify other attributes upon successfully conquering the Box object pointed to by the "current_box" attribute.

Board Interactions:

Neither Box nor Player classes directly interacts with the Board class. Instead, a single Board object is instantiated in the client file and another in the server file. Both the client and the server code use the Board object as a means to have access to all Box objects on the gameboard.

Client and Server Files:

In terms of clients and servers, any player can be the host and start the game. All players possess the same files, which include a server and client file. Whoever wants to host the game server runs their server program before also running their client program. All other players can join by updating their client file to the host's IP address.

The client and server files are responsible for creating, setting up, and initializing sockets and establishing the connection between client and server sockets for communication. The client and server files import the Board and Player classes, the external Pygame module required for the UI interaction, as well as the following modules and classes from the standard Python library for backend implementation: socket, pickle, time, sys and threading.

The client and server communication and implementation relies on a token-based system, where specific actions from the client trigger events via interactions with the UI using the pygame module. The unique events that a client can perform during gameplay each produce different Strings that represent the

action taken by the player. The String is then passed as data from the client to the server. The server compares the String to its list of expected tokens and branches to different blocks of code to perform the action necessary in response to the client's actions.

Overall Application Workflow:

The server socket is set up, initialized, and bound to its address. The server socket begins to listen for incoming data transmissions.

The client socket is set up, initialized, and connected to the server via the server's IP address and established port number.

Both the client and server initialize their respective Board objects to maintain separate records of the ongoing game.

The server creates a new thread called "client_broadcaster" dedicated to broadcasting frequent updates to all connected clients regarding the current state of the gameboard, so that all clients see changes to the gameboard as they occur in real-time.

This thread calls a function called "handle_broadcast". This function implements a while loop that uses the server's Board object to call the board_to_string function to convert the Board to a smaller piece of data for transfer. It iterates through the list of Player elements, sending the updated state of the board to each client. Each iteration of the loop checks if the game has concluded by calling the is_game_over Board function. Once this condition is met, the while loop exits and closes all client connections.

The server initializes an array of threads and a counter for the number of players. It begins to listen for client connections.

The client chooses the color they will play as from the UI and sends a "new_player" token to the server with their color information passed as an argument.

The server accepts the connection from the client and retrieves the return address of the client. It creates a new thread and assigns the thread to handle this client, appends the thread to the thread array and increments the number of total players' counter.

The thread calls a “handle_client” function, which takes input arguments of the client’s socket and the client’s return address. The function runs a continuously while loop until one of two conditions is met: the client stops sending data or the game is over. Until either of these conditions is met, the function continuously checks if the incoming data matches any of the established tokens for the application: “new_player”, “start_drawing”, “stop_drawing”, and “stop_drawing_threshold”. The “new_player” token creates a new Player object, assigns it the color which was selected by the client and adds the new player to an array of Player elements representing all clients who have joined the game server. The remaining tokens index into the array of Player elements to access the correct client using the associated client socket (key) information and call the functions start_drawing, stop_drawing, and stop_drawing_server_colored as outlined in the Player class description. On each iteration of the while loop, the thread uses the global Board object to call the is_game_over function to check if the game has concluded. Once this condition is met, the thread returns from its execution and waits to be joined in the main thread.

The client creates a new thread called “client_listener”. This thread is given the client’s Board object as an input argument, and it calls the “listener” function. This thread is dedicated to listening for incoming data from the server. The function executes a while loop to receive data. The server only sends the client two types of tokens: “winner_color_key” and “box_locked”. The first causes the while loop to exit, the latter prints an error message informing the client that the server has denied access to the box they were attempting to conquer. The function assumes all other incoming data is a broadcast of the current board state from the server and uses the clients Board object to execute the string_to_board function to apply all necessary updates to the client’s local board.

In the client’s main thread, a while loop executes for as long as the game has not been determined to have concluded. The loop detects events in the form of user interactions with the UI using the external pygame module. The detectable events are as follows: MOUSEBUTTONDOWN, MOUSEBUTTONUP, and MOUSEBUTTON MOTION. The MOUSEBUTTONDOWN and MOUSEBUTTONUP events trigger the client to send the server the “start_drawing” and “stop_drawing” tokens to the server, which executes the appropriate action on its side as outlined previously.

Once the game concludes, all threads return to the main thread for both the client and server code. The threads are joined, and connections closed.

Design Considerations and Limitations:

Failed Implementation 1:

Initial Approach: Utilizing Non-Blocking Sockets

In the context of real-time applications over a TCP connection, the use of non-blocking client sockets has several advantages, including heightened responsiveness, reduced latency, and optimal resource utilization. However, it's important to note that the behavior of functions interacting with non-blocking sockets deviates from expected behavior. Therefore, proper implementation and utilization of non-blocking sockets requires additional attention to event and error handling.

Nature of Issue: Early Return of Socket Functions Before Full Data is Received

When a non-blocking socket is engaged in a read operation, and no data is immediately available for reading, the non-blocking read function returns immediately. It will not block or wait for the remaining data to arrive.

Suspected Cause of Failure: Multithreaded Client Handling on Server Side

Failure of this design can be attributed to the utilization of a multithreaded design approach on the server-side of the application. It is possible that the thread of execution was switching mid-transfer, and the temporary cessation of transfer was enough for the socket to abort its read execution. Additionally, delays in data reception from the server stemming from the high volume of data traffic or network congestion, could result in a non-blocking socket returning prematurely as well.

Revised Approach: Removal of Non-Blocking Setting

The inclusion of the non-blocking setting during the initialization of the client socket was omitted. Instead, a dedicated listener thread is now established within the client script with a listening function. By doing so, even if the client socket enters a blocking state or awaits data, it can persistently remain in this state in its separate thread until complete data reception, while allowing the main program's execution to proceed without interruption.

Failed Implementation 2:

Initial Approach: Use of UDP Sockets for Client-Server Communication

The initial networking approach used UDP sockets to establish connections between the client and the server. Leveraging the speed advantages of UDP, particularly beyond the initial setup phase, seemed promising for implementing an application that required real-time responsiveness. However, it's important to recognize that UDP's speed comes at the cost of reduced reliability. UDP operates on a "best-effort" basis, making no guarantees regarding the successful delivery or the order of packet transmission.

Nature of Issue: Premature Closing of Client and Server Sockets Due to Idle Periods

During testing, both the client and server sockets displayed an unexpected tendency to prematurely close when short idle periods were encountered. The closure of the client sockets caused game disruption, but closure of the server side caused the game to crash completely.

Suspected Source of the Issue: Absence of a 3-Way Handshake in UDP

TCP employs a rigorous 3-way handshake mechanism to establish and sustain connections, but UDP lacks this safeguard. Consequently, it is possible that the absence of a handshaking process in UDP contributed to undesired and untimely socket closures during even short periods of inactivity.

Revised Approach: Use of TCP Sockets for Client-Server Communication

The application's networking design was reconfigured to utilize TCP sockets for establishing connections between the client and the server. Adopting TCP sockets led to more reliable data transfer within the game. While TCP may not match the speed of UDP, the benefits of data integrity and orderly transmission offsets the sacrificed speed.

Failed Implementation Alternative 3:

Initial Approach: Utilizing Pickling and Unpickling for Board State Transfer

The initial networking approach used the process of pickling and unpickling to exchange game update information between the client and the server. Pickling and unpickling refer to converting a Python object into a byte stream and vice versa. In the initial implementation, the Board object was pickled, along with all its associated attributes. This serialization transformed the Board object into a format suitable for transmission via the server socket. Upon reaching its destination at the client socket, the Board was unpickled to restore the transmitted data to its original object state. While the pickling and unpickling process is generally efficient for simpler objects, there seem to be inconsistencies in data when dealing with more complex objects, such as a Board object that contains nested object attributes.

Nature of Issue: Client Not Consistently Receiving Board Updates

The client was not receiving broadcasted updates from the server regarding the gameboard's current state. Additionally, the incoming data failed to trigger the error handling sequence intended to capture instances of "no data" or "incomplete data".

Suspected Cause of Failure: Unsuccessful Pickling and Unpickling of a Complex Object

The Board object contains a 2D array, and each element within this array was, in itself, an object. This intricate nested structure may have been too complex to be repeatedly pickled and unpickled successfully. It is possible that the pickling and unpickling transformations led to data inconsistencies, rendering the received data on the recipient's end unusable by the application.

Strategic Pivot: Manual Serialization and Streamlined Data Transfer

The current implementation involves manual implementation of serialization and deserialization mechanisms, sidestepping the challenges associated with complex object pickling. The program now implements functions written to accept a Board object as an input parameter and produce an equivalent String representation of the board. This String, of significantly smaller size and complexity, is used for data transfer between the client and server instead.