CMPE 436 TERM PROJECT

PUZZLEBUZZLE

Emirhan Saraç

Abstract

PuzzleBuzzle application is a traditional puzzle game with additional feature of playing against someone else. The game will select two players and bind them to the same game, if both players wants to play same puzzle map. In this game, we will have 16 pieces for the map and distribution is four by four. Each player will have four randomly selected pieces for given map different from each other. After game starts, the goal is to put as mant as pieces to the correct position of the map before your opponent puts a pieces to correct position. If one of the players puts correctly a piece then piece deck of both players will be shuffled. Therefore, other than putting a piece to the correct position before your opponent puts the same piece to the that position, player also need to put anything to confuse opponent. After all map is done, both players will receive some points depending on their performance in the game.

Introduction

Before this project, I have always been interested in games especially multiplayer games. Most of the old classic games always been a single player game so that for this project I have thought to make an old classic game which will be multiplayer based game. I know that GUI for games will put me into hard times while developing games, because I did not have background knowledge about Android and these graphical stuffs. This is why I should have chosen a proper and low level graphical game.

In the beginning, I was not certain about choosing between *sudoku* and *puzzle.* Then I have chosen puzzle game, because it will be more fun to play and it will be more challenging for me to develop. At first, I was planning to make game more general like user can set the maximum number of players that can play this game between two to four. Also I was thinking to make puzzle map 20 by 20. After thinking on those concepts about one hour, I have decided to get rid of those ideas. They would definitely take my considerable amount of time. I had to design GUI so that it can handle all those cases. For other concept, it will be hard to find the correct pieces for 20 by 20 map. Let’s consider demo, just to show what will happen after the game is done, we will probably play at least 5 minutes. All those thoughts are important for me to develop this game. Some was challenging to courage or discourage me from those additional features.

At the end, I have chosen a name for game which is *PuzzleBuzzle* and started to finish this project. I have tried to implement a puzzle game which has four by four grip map. The details and playstyle will be explained in the following sections.

Approach

The project consist of three main parts

- The client side which will be run at Android device
- The server side which will be run at Amazon Web Server(or at private computer)
- The database side which will be run at Amazon Web Server(or at private computer)

3.1 Server Side

I have developed my server side using JAVASE 1.7 at Eclipse IDE environment. Below I will explain each class of server side. I have also used external jar called *"org.json.jar"*. We are going to use that jar for database and messaging purposes. Connection between Android side and Server side will be handled with Java Sockets(TCP).

3.1.1 Semaphore Class

This class will be used to satistf mutual exclusion property. It will also be used to create blocks, too. It has two constructors which will be called depending on purpose. We have two methods that will be used most frequently.

- *Semaphore.request()* → to request lock, if it is granted, process will continue its task. If not, process will wait until one of the other threads releases lock.
- *Semaphore.release()* → to release lock.

3.1.2 PuzzleBuzzle Class

In this class, we have one constructor and method. Contructor will be used for initializing GAME_ID, map, mutex, and pieces array whose size is 16. The other method, “addPlayer”, will be used to initialize one of two players. Let’s explain field variables.

- *Semaphore mutex* → will be used for locking while one of the players tries to insert a puzzle piece so that it will prevent violation of mutual exclusion.
- *JSONObject player1* → keeps the information about player who is in this game.
- *Socket socket_player1, BufferedReader b_p1, BufferedWriter w_p1* → keeps the connections alive so that we can send a message from client to client thround server. Main reason why I am using this, I did not keep the connection of client and server always up. After request from client is done, server disconnects buffers and socket, which does not happen in this case only. (I have the same structure for player2)
- *Int map* → keeps which map type this game has.
- *Boolean[] pieces* → keeps which pieces are inserted in this game. (false means not inserted yet)
- *Int GAME_ID* → used for distinguishing games which has same map type.
- *Static ArrayList<PuzzleBuzzle> games* → keeps all active or waiting games.

The reason why I am using this class is I should have kept the games somewhere at the backend, because one player can be waiting for opponent or prevent same piece insertion to same part of map at the same time from both players. I could not find a easier way to handle this in the client side which will include more message passing. On the otherhand, I have moved some backend methods to client which will be mentioned later in cliend side.

3.1.3 TCP_Server Class

This is the main class of backend side. We are going to initialize port number, initialize database, and listen connections from clients. Server will listen client connection in a server_thread. After connection is settled, server will create another thread to handle connection request. Server will also send the socket of the client to the handler. Before starting the server, it will initialize two Semaphores and port number of server.

We also have an another function *loadDB()*. It checks existence of database in this system. If not creates folder and file. Database will be kept in *"PuzzleDB/users.txt"* file relative to where server runs. Then server initializes the file with "{}" which is a empty JSON object. We have two important Semaphores in this class but it will better to explain there in the below part since they will be used in handler class.

3.1.4 ConnectionHandler Class

This is where all requested are handled and sent respond. After creating this thread in *TCP_Server,* it initializes client socket, connection reader, and connection writer first. Then creates and starts thread for this request. First explain field variables and then go into the deep underlying methodology of this class.

- *Socket connectionSocket* → keeps the socket connection to the client.
- *BufferedReader inFromClient* → reader from client socket connection.
- *BufferedWriter outToCleint* → writer to client socket connection.
- *PuzzleBuzzle GAME* → keeps the game class of this thread, it may be null, if this thread request is in any game at that time.
- *Boolean closeSocket* → it will be used for preventing disconnection after thread is done.

- *Static int PUZZLE*_ID → it will be used to assign unique id to each game so that we can distinguish game from each other even if they have the same game.

After handler thread starts, handler first get the string from client and parse it to the JSONObject. We will always send messages by parsing JSONObject to String and parse back when we receive. Then handler sends this JSONObject to *handler_request* function. In the *handler_request* function, hanler will call other functions depending on the content of JSONObject. All sent data from clients has mapping *"function"* → *"function_name"* in the JSONObject. This mapping decides which functions will be called in order to distunguish different requests. After those functions are done, handler closes sockets and bufferes, if it is not some specific function calls. Let's explain each functions that can be recognized by this backend.

3.1.4.1 log_in → user tries to log in to the system.

Handler locks the *database* to avoid write-write and read-write conflicts and reads the database as String. Immediately after completing read of database, it releases *database* lock. Then parse it to JSONObject. Also request from client has another JSONObject which includes *name* and *password* of the user. If there is such an user and password for this user, this function will send a JSONObject message to the client. This JSONObject has two fields. There two are *"success"* and *"point"*. If there is such an user, then success will have value of "1", and total collected point of user in the database. If there is no such a user, then success will be "0".

3.1.4.2 register → user tries to register to the system.

Handler locks the *database* and reads the database same as in *log_in*. Parse this read string to JSONObject. Sent JSONObject has *name* and *password* fields. Functions checks whether read database has the same username as the sent name from client. If there is, then replies back JSONObject message which has *success* field with value of *"0"*. If there is no such username, this function will update database JSONObject and writes it back to the database file as string. Then reply message will have success field with value of *"1"*. At the end, this function will release *database* lock.

3.1.4.3 waiting → user searches for available game or creates its game and waits for opponent.

This is probably the most complicated function in my project. This function first searches waiting opponents who want the same map. All these search done under *game_search* lock. The reason for using lock, if both player searches for same game map, and there is already a waiting player with same map, there can be violation of mutual exclusion by directing both players to the same game. That will result in 3 players in the two player based game. After search done releases *game_search* lock. If there is no such map, then handler creates game for this user and waits this thread until match is done. This function also checks whether the game is deleted or not. If the user disconnect from server while waiting, the game will still be alive in search. That will result in a player to join game and collect all points without and opponent. If the game is deleted, it also removes game form *games* in the PuzzleBuzzle class. After all these done, handler sends a message to client to start the game.

3.1.4.4 delete → when user cancels waiting.

Handler searches for game which has same ID with the field of received JSONObject by client. While searching, handler requests *game_search* lock again. If there is such a game, removes it from games. Then releases lock. If this function is not called after cancelling waiting, game would be in the list. This can cause to crush and one player to join game without opponent.

3.1.4.5 setConnection → just before when both users starts the game.

This is another crucial part of my project since I am not keeping connections always alive. After each thread operation, my handler closes connections. In this function, Both players will set their connection informations to *PuzzleBuzzle* object in the *game* array list. So that handler can understand what are the players in this game and their connections to the server. Search will done by *id* of received message from clients. Again search will be protected under *game_search* lock. After this task is done, handler sends the opponent name to the client for Android tasks. Another extra thing about this function is after every function handler closes socket and buffers. However, in this function it is not the case since handler keeps them in *PuzzleBuzzle* class.

3.1.4.6 putpiece → when user tries to insert a piece to correct and available part.

Client sends the piece number and id of game to the handler. Handler search for the game with the same id. This search will be protected by *game_search* lock. Then if it finds, handler will call another lock which is *mutex.* This lock prevents inserting the same piece at the same time for both players. That will result in giving point to both players which is not the desired functionality in my game. If it is inserted just before, then sends a message to client not to give a point to himself. If it is inserted by requested client, it sends a message to himself to give a point to himself. Then, it will send a message to opponent so that opponent will be informed about which piece is inserted. Since, we keep player connection informations in the *PuzzleBuzzle* class, we can do that easily. Only struggle in here is that, which player is inserted and that will solved by sending a username of the client.

3.1.4.7 updateScore → when the game ends.

This function has two parts. First is updating the database. Handler will read database and write the new score for the given username. That means after any game, this function will be called twice, from both of the players. This operation will be protected by *database* lock. Second is deletes game from *games* to prevent the unnecessary memory usage. This operation will be protected by *game_search* lock.

3.2 Database Side

My database is a single txt file which will be located in the folder *"PuzzleDB"* and named *"users.txt"*. This file represents the string of JSONObject. Structure of this JSONOBject is like {*username1* → {"*point*" → "..", "*password*" → ".."}, *username2* → {"*point*" → "..", "*password*" → ".."}…}. Therefore, we have only the information of username, password, and score of each player.

3.3 Client Side

Clint side is an Android Application which I will develop in Android Studio. I have used Java to program client side. Client side can run at android which has minimum sdk 23. For GUI design, I have used xml files and drag-and-drop feature of Android Studio. In this project I have seven activities and one class for static variables. I have used Socket programming for communication with server side. In the client side, I have also disabled landscape mode. I will explain each activity and class below.

3.3.1 Helper Class

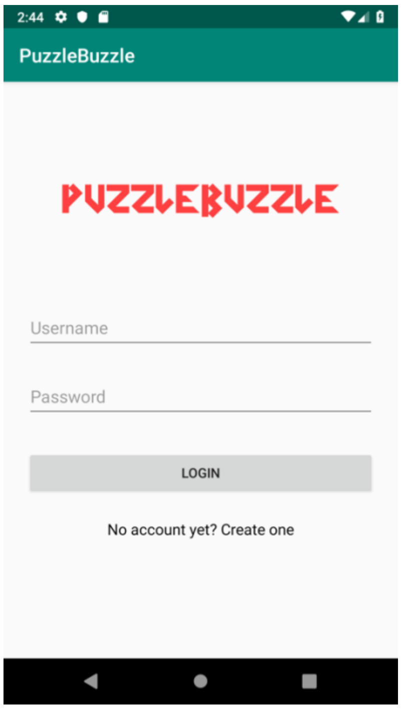
In this file we have only two static variables.

- Static String IP → IP address of server
- Sattic int PORT → Port number of server

3.3.2 LoginActivity

This is the first screen, user sees after opening application. *Figure 1* is an image of what it looks like.

Figure 1



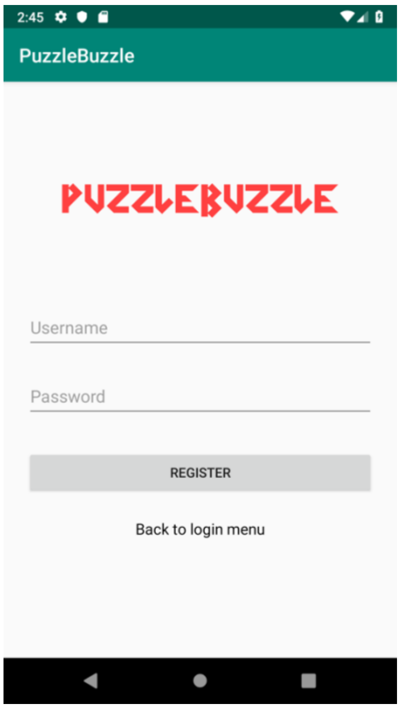
In this screen, we have two editable text, one clickable button ,and one clickable text. There is a logo of the game as well at the center of top. User can write its username and password to the related parts. Then clicking *LOGIN* will send a request to the server. When clicked to the *LOGIN* button, asynchorized task will be called. It will send the JSONObject to the server. This object includes *name* written is Username text and *password* written in Password text. Then depending on received message from server, user will be directed to *MenuActivity* or will stay at the same screen. Either login is success or not, there will be text baloon which has message for user. If login is successful, name and point received from server will be sent to *MenuActivity*.

There is another clickable which is sign up text, which will direct the user to *RegisterActivity*. To prevent multiple clicks at the short time, I have put some codes to all listeners. If I wouldn't put this, one can click *LOGIN* button many times before server respond has come.

3.3.3 RegisterActivity

This activity will be seen when user wants to create an account. *Figure 2* is an image of *RegisterActivity*. In this screen, we have two editable text, one clickable button, and one clickable text. Also, there is a logo of application. User can write name and password for account that he wants to create. After that, if user clicks *REGISTER* button, asynchorized task will be started and client will send a message send to the server. This message will have two

Figure 2

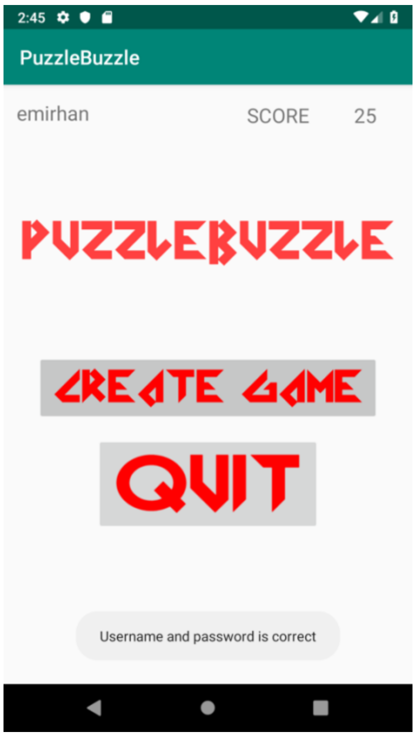


fields which are *name* and *password*. According to respond from server, a message baloon will be shown at the bottom side. If respond is positive, then user will be directed to *LoginActivity* screen. At the client side there is some restriction for username and password. If user writes username and password whose lenghts are less than 6 or more than 12, then a error message will be pop out.

User can also go back to the login screen by clicking *"Back to login menu"* text.

3.3.4 MenuActivity

Figure 3



This is the screen when user successfully logins to the system. First, we set the name and totaly point of the user to the top bar as you can see at *Figure 3*. This information comes from the *LoginActivity*. We have added extras to the intent just before user is directed to here, so we can access those informations and set these informations. There are only two clickable images in this screen. In this screen we don't need any connection with server. User can either quit from the game by clicking *QUIT* image, then user will be directed to *LoginActivity*, or can create a game by clicking *CREATEGAME* image, then user will be directed to *CreateGameActivity*. As in before, we will send *name* and *point* of the user to the next activity.

3.3.5 CreateGameActivity

Figure 4

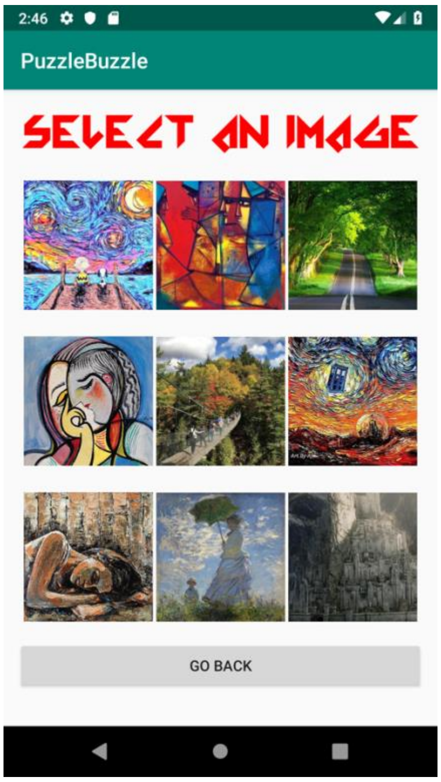
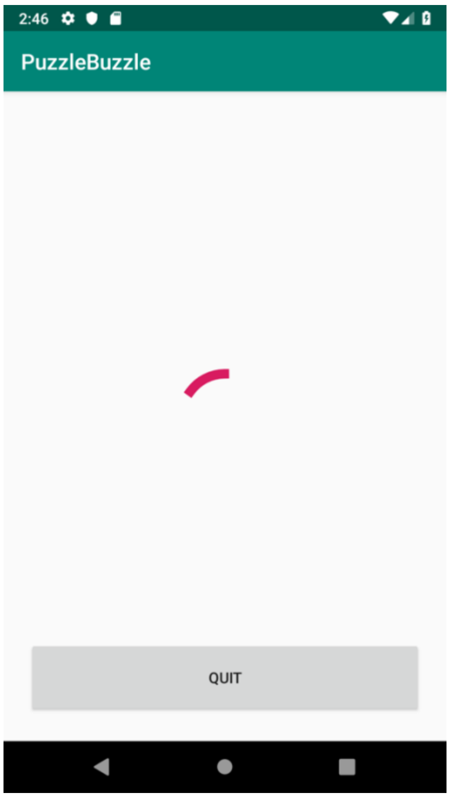


Figure 4 will be seen when user want to play a game. In this screen, there are nine clickable images as you can see and one *GO BACK* button. If user clicks this button, then he will be directed to *MenuActivity*. User will also send *name* and *point* of the user as well. In this screen, now it is time for user to choose which map he wants to play. After choosing one of those, user will be directed to *LobbyActivity*. Client will also send *name*, *point*, and type of *map* to the next activity after clicking one of the images. There is something important in here that I have encountered first in my life. User actually can click the same image several times before directed to the next activity. Therefore, it can cause some problems as will be mentioned later. This is why I have added some code pieces in each listeners that will allow only one click at each three seconds. The problem had been solved like that but theoretically there can still be problems. If my phone freezes more that 3 second, and I click just before freezing and after two times, I expect to see unwanted behaviors. However, for now there is no problem.

3.3.6 LobbyActivity

This screen will be shown when user waiting for a game at *Figure 5*. We have only one clickabe button here. It will be used when user want to quit from waiting. It generally

Figure 5

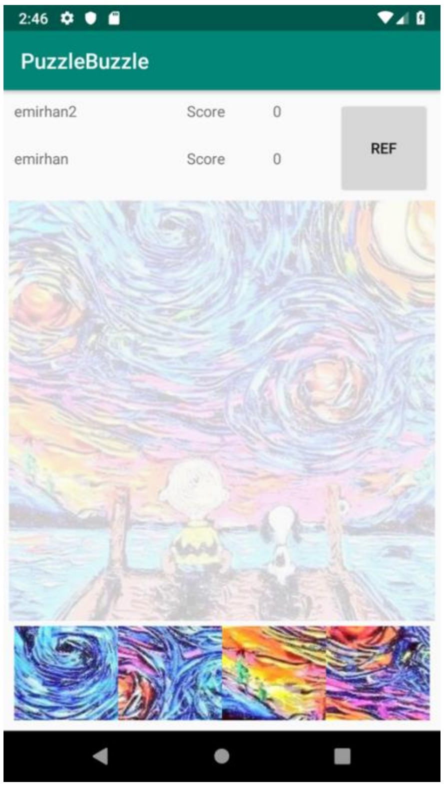


happens when user waits a long time for game or he had to do another stuff. Whenever this screen comes up, a thread will be created and connection will be settled up with server since UI threads cannot use Socket connections in Android. A message which includes *name*, *point*, *function*, and *map* will be sent to the server. Then client will receive message which contains only *id* of game. There is an important thing here to talk about. We have a lock in this activity. We use this lock for blocking. The reason is when user waits for a game, if it quit from the waiting, the game will be already created and waiting for an opponent. To prevent this, we will have a lock with value 0, and will be released only after we get an *id* of game from the server while waiting. We acquire this lock when *QUIT* button is pressed so that when user click this button it will send a message to the server. That message contains *id* and *function* which is "*delete*". This will delete game on the server side then user will be directed to *CreateGameActivity*. While waiting if second respond comes from server, we know that we have found an opponent to start the game and we will go to the *GameOnActivity*. Moreover, we will send *name*, *point*, *map*, and *id* of the game to the next activity.

3.3.7 GameOnActivity

This is the screen of the game at *Figure 6.1*. We have *name* and game *scores* for both players. We will explain how we get the opponent name later. At the top side we have host client name and score and the below part contains opponent informations. We have clickable *REF* button which will refresh piece desk of the client. We have 4 pieces at the bottom side

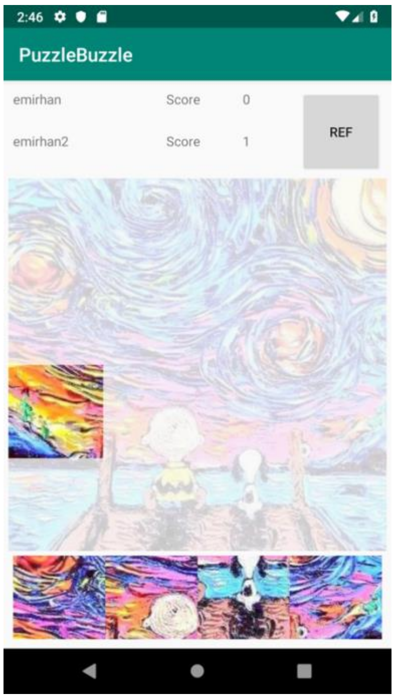
Figure 6.1



for this client. The pieces are not synchorized for both players that means they can have different pieces. We also have transparent image in the middle of screen that shows which map client should solve. This map will be set by client by looking *map* got from *LobbyActivity*. User can set the images by clicking one of the pieces then clicking corrent position of the puzzle. At each correct piece insertion desk will be shuffled like in *REF* button. Also wrong piece insertions will also shuffle the desk. After completing all pieces client will be sent to the *GameOverActivity*. Now let's explain what is happening at the underlying code side.

Whenever UI of screen is done, we will call a function *listen_opponent_moves()*. In this function we create a new thread to connect to the server. We send a message which includes *name*, *id*, and *function*. Then server will respond a message which includes opponent name. Then we will create another thread but this is UI thread in previously created thread to set the name of opponent at the top bar. But why we are creating another thread? The reason is in Android we cannot edit, delete, or create issues about UI in normal threads. It should be UI thread. Whenever we create an UI thread, android will send this thread to the queue and when its time has come, it will do its task. We can also edit non-UI thread but this will destabilize the UI of Android. Then client will receive another message that will only approve connection is completed between clients through server. Then client will listen server always in a while loop. This loop only terminate when remaining pieces are 0 that means all peices are already inserted. If all pieces are inserted user will be directed to the *GameOverActivity*. Client will also send datas to this activity such as *name, score, point, op_name, op_score, id*. Not all will be user in next activity, this has been done for further development.

Figure 6.2



While waiting respond from server, the respond comes only when opponent inserted a piece. How is done? After an repsond comes client will call *pieceInserted* method. The message contains which part of puzzle has been inserted. Client will create another UI thread to handle this task. It will put a piece to the correct position so that user will understand this piece is already inserted and clicking there with a piece will not send a message to the server. Then it will shuffle the desk again even if opponent has inserted. Then it will check how many pieces are left. If it is less than four then some part of the desk will be deactivated and will be seen as dark image which are not clickable. Then opponent score will be increased as one. There is another important and should be taken into account lock here. While waiting an respond from server, if we get responds we will create UI thread. Then we will listen again. However, if UI thread completes its task later coming back to listen server, which is a common case since UI threads are slower than non-UI threads, there may be a problem. Problem is if the last piece is inserted by opponent. Then there will be no piece to insert and end a game. User cannot do anything at this step because we are checking the game end scenario in while condition. So that we have another blocking lock here. We will block next while condition until UI completes its task. We will acquire lock in while statement and release at the middle of UI thread. After opponent piece inserted image will be shown as in *Figure 6.2*.

Now time to talk about what happens when we insert a piece. If we click the correct piece into the correct position *callInsertion* will be called. It will create a thread to connect with server. It sets the connection with server. Then it sends a message which contains *name*, *id* of game, *hint* which is a piece number, and *function*(“putpiece”). If responds is positive, then insertion has been done by this client and *insertPiece* method will be called. If not this means opponent has inserted just before we did. If respond is positive, client increases its score by one and inserts piece into the correct position. Then it checks whether all peices are inserted or not. If all inserted, client will be directed to the GameOverActivity. Client will also send some data to this activity such as *name, score, point, op_name, op_score, id*.

Figure 6.3

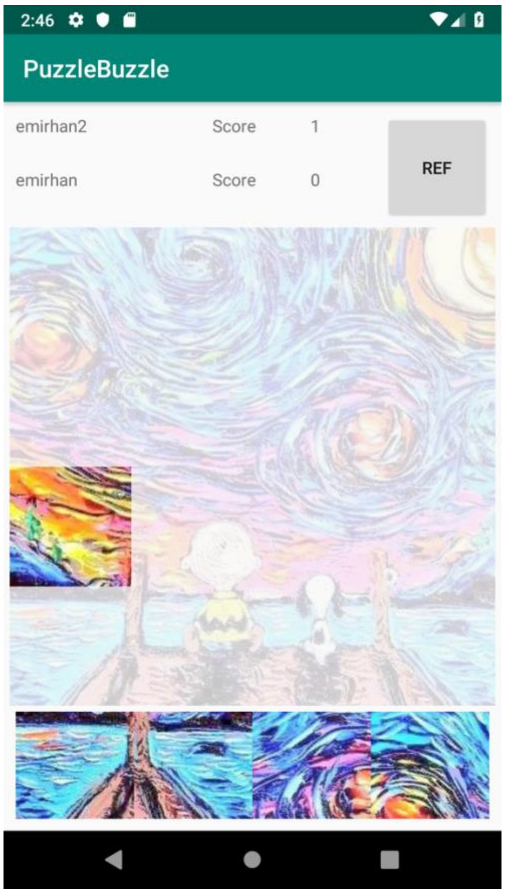


Figure 6.3 shows what will happen on UI after we insert a piece.

3.3.8 GameOverActivity

This is the last screen we are going to mention for my application. In this screen, user will see a one of the 3 types of screen depending of game result which are winner, loser, or draw game. After clicking one of them, asynchorized taks will be started and it will connect with server and send a message which contains *name*, *point*, *id*, and *function.* In here, *point* is total point collected before this game plus collected point from this game. It will update score of user. Then user will be directed to the *MenuActivity*.

Winner screen at *Figure 6.4*, loser screen at *Figure 6.5*, and no winner no loser screen at *Figure 6.6*.

Figure 6.4

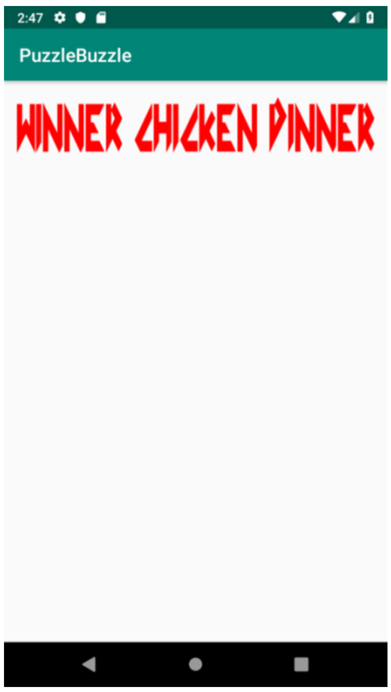


Figure 6.5

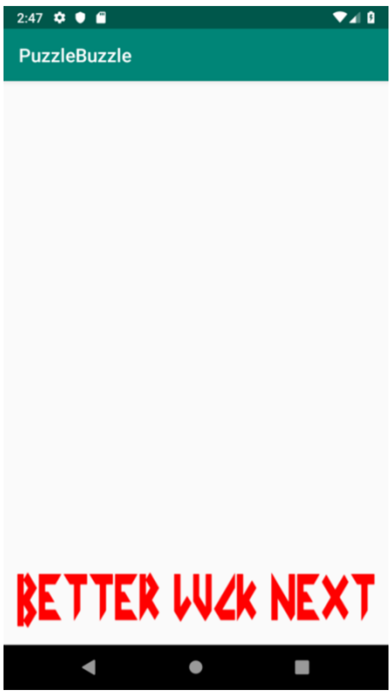
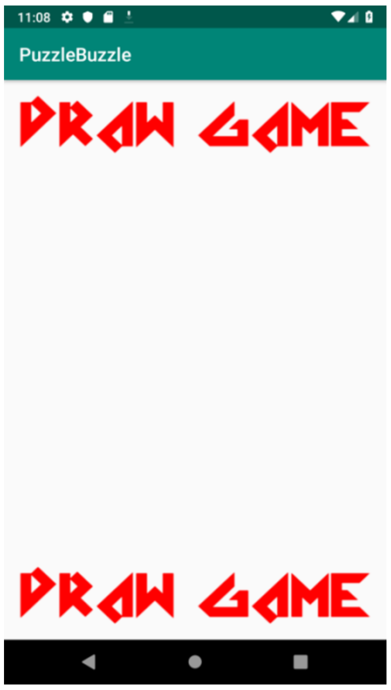


Figure 6.6



3.3.8 Miscellaneous

In my application, I have disabled landscape mode and back button.

Conclusion

This project was so useful for me because I have used low level connections, Sockets, and set connection between backend and Android. I was also my first Android project. In general I am not a fan of coding a graphical stuffs. However, I think I have done well and learned a lot in this project. I have also learned synchorization across different machines, clients, on server side. Sometimes I had to use concurrency for this task as well. This was a mixture of a lot of things we have learned in this course.

There are lot of things I could not finish. I was planning to create map by using camera of phone. However, this will require send a message through Socket. I have also wanted to create games not with only four by four map, maybe some options. Also, I was planning to make games playable with 2,3, and 4 players. It was not easy for me so that I have skipper this part as well.

Overall, it was a great experience and project for my future. Especially, I have learned in this project that it is not easy to make a project that we are thinking. Probably, we will skip some features, but at the end it will be similar a similar project. It will be better to develop and add extra features after completing basic project.