CS Android Programming: Homework Project Hex

You have chosen to do the homework project instead of designing your own project. This homework will be on the difficulty level of Reddit, so somewhat large. But you will not have to design and code your own project. The maximum grade you can receive is a B+. Though if you do something impressive above and beyond the specifications of the homework, you can get a higher grade.

You must do this assignment on your own. NO TEAMS.

Submission. All submissions should be done via git. Refer to the git setup and submission documents for the correct procedure. Please fill in the README file that you will find inside your Android Studio project directory (at the top level).

There is NO code collaboration for this homework project. Each student must do their own coding and they must do all of their own coding. You can talk to other students about the problem, you can talk to the instructor or TA. If you discuss the homework deeply with someone, please note that in your README.

Overview. Let's start with motivation. My son showed me this video and it won me over.
"Hexagons are the Bestagons."
`https://www.youtube.com/watch?v=thOifuHs6eY`

Ok, that was fun.

We are building a strategy game called Hex. Here is how I learned about it, from the fount of all knowledge, the wikipedia. Basically, blue tries to connect its borders while blocking red and vice versa.
`https://en.wikipedia.org/wiki/Hex_(board_game)`

There are many online resources for Hex, but best not to get too distracted as our focus is on Android programming. But here is a link to the Hex wiki.
`https://www.hexwiki.net/index.php/Main_Page`

The hexagons in the board are called counters in Hex literature. I call them counters (in a few places), hexagons, or cells.

The big button on top in the middle I call the turn indicator. It shows whose turn it is by its color, but it also shows the move number and whether one side won.

Board layout. This is an advanced topic that you can come back to.

The sources describe Hex as being played on a rhombus shaped board. I do not like rhombuses. They don't look good on a rectangular screen in my opinion. Unfortunately, I didn't see the diamond version of the board, because that one might have worked.

What I did do was crush the rhombus into a square. The red borders are on top and bottom and the blue borders are on the left and right. That means red is always trying to connect vertically, and blue horizontally.

Displaying the hexagons in a square means that the neighbors of a hexagon depend on whether it is in an even or odd row. To be able to think about hexagon neighbors, you can look at the display with the interior hexagon labeled (the "Label interior hexagons" switch in the dashboard). I show this display on the next page.

We use letters for columns and numbers for rows in the display. (In the code we just use two integers.) So a1 is the first playable hexagon. Look at the labeled display of hexagons and find the hexagon labeled b1. It has a1 as a left neighbor and c1 and a right neighbor. That is true for all hexagons in any row, that (column-1, row) and (column+1, row) are neighbors.

b1 also has b0 as an above neighbor and b2 as a below neighbor, which are located at (column, row-1) and (column, row+1). Every hexagon has these four "square" neighbors. I call them square because they are the neighbors we expect from standard rectilinear Cartesian coordinates.

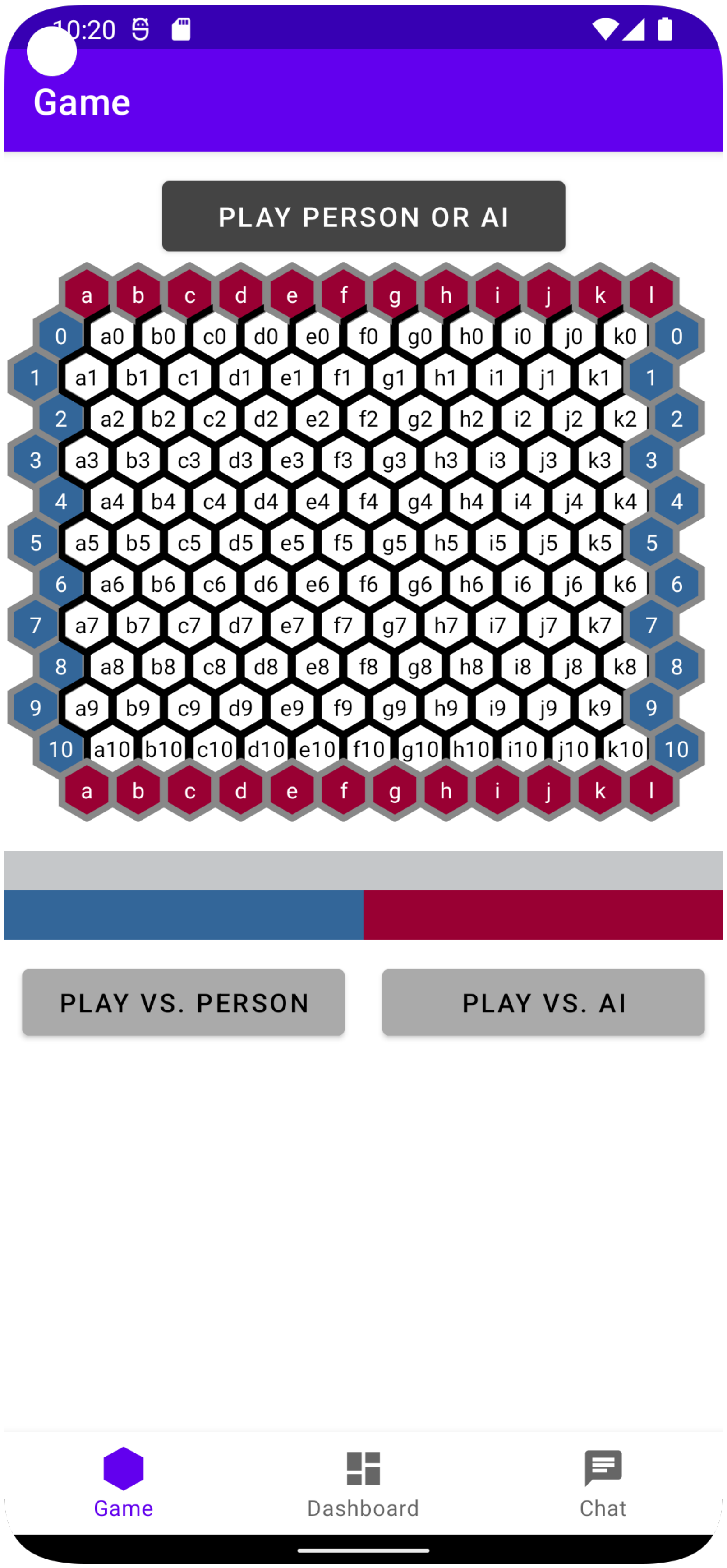
But b1 also has a0 as an upper neighbor and a2 as a lower neighbor. Those crazy hexagons (when crushed into a square).

Now look at b2. It has left: a2, right: c2, upper: b1, lower: b3, which are the "square" neighbors. But it also has c1 as an upper neighbor and c3 as a lower neighbor. It has column+1 neighbors, while b1 has column-1 neighbors.

I will try to keep you shielded from having to implement this logic, but you should be aware of it.

I also want mention all of the border hexagon types. There is the red border on top/bottom and the blue border on the sides. These hexagons have a grey outline instead of the black outline of the interior hexagons. Also, there are two "orphaned" hexagons at (0,0), and (0, boardDim-1), where boardDim is the dimension of the square board. These are the NeutralBorder hexagons that are transparent. They are excluded from play.

Navigation. There is a bottom navigation view that we set up for you, like we did in the demos. There also is a navigation transition that you have to write, described below (from the dashboard to the game).


10:20
Game
PLAY PERSON OR AI
a b c d e f g h i j k l
0 a0 b0 c0 d0 e0 f0 g0 h0 i0 j0 k0 0
1 a1 b1 c1 d1 e1 f1 g1 h1 i1 j1 k1 1
2 a2 b2 c2 d2 e2 f2 g2 h2 i2 j2 k2 2
3 a3 b3 c3 d3 e3 f3 g3 h3 i3 j3 k3 3
4 a4 b4 c4 d4 e4 f4 g4 h4 i4 j4 k4 4
5 a5 b5 c5 d5 e5 f5 g5 h5 i5 j5 k5 5
6 a6 b6 c6 d6 e6 f6 g6 h6 i6 j6 k6 6
7 a7 b7 c7 d7 e7 f7 g7 h7 i7 j7 k7 7
8 a8 b8 c8 d8 e8 f8 g8 h8 i8 j8 k8 8
9 a9 b9 c9 d9 e9 f9 g9 h9 i9 j9 k9 9
10 a10 b10 c10 d10 e10 f10 g10 h10 i10 j10 k10 10
a b c d e f g h i j k l
PLAY VS. PERSON
PLAY VS. AI
Game
Dashboard
Chat

Game dynamics.

- **NotPlaying**. We start in the GameState.NotPlaying state, which is shown in the picture above. There is no text in the gray stripe and there are no blue or red player names and nothing is in the chat area. The turn indicator urges the user to play.

- **Play vs. Person**. Button clears the board (more for testing than anything else) and Snackbars a message about this not being implemented. It doesn't affect the turn indicator. Just use what we give you.

- **Play vs. AI**. This starts a new game by randomly generating a Boolean to determine whether red or blue goes first. Please use the same random object (in ViewModel) for all random number generation. Also, the person and the AI are assigned blue/red randomly. The turn indicator shows what number turn it is and by its background color whose turn it is. Each move is a turn, so after blue goes, that is a turn, and then red is another turn.

  If the AI is chosen to go first, it moves.

  During play you can switch to the chat pane and send a message. The last 4 messages are shown, bottom up, in the chat display area at the bottom of the screen. The details of the layout are explained in GameFragment, below. The layout is given, you just have to use it.

  The user presses an unclaimed hexagon to make a move. If the user clicks the border or a claimed hexagon, the background should flash red.

  When one player wins, their winning path is "lit" by yellow circles and the turn indicator says "Blue won (xx)" or "Red won (xx)" where xx is the number of moves made. I don't have an outcome for a tie because I read that ties are not possible in Hex.

  As you play against an AI, the AI will send messages at random intervals.

- **Replay game**. When the user selects a game from the dashboard, it transfers back to the game fragment, but now we are in "replay mode." Certain view elements are cleared, like the chat history. There are additional visible controls on top. From left to right: go to the start of the replay game, go back one move, turn indicator (not a control), go forward one move, go to the end of the game.

  Also in replay mode, the gray strip above the blue/red players has the date of the game that is being replayed.

  You need to write a lot of the logic for a replay game. This mostly consists of traversing the move list in the replay game and filling in or clearing the affected hexagon. While in replay mode, we never write to firebase. We don't create or update games and we don't write into the chat.

  When the user presses Play vs. Person or Play vs. AI, the replay game mode is exited. The controls disappear, the replay game's date disappears and the chat disappears.

Dashboard. This is mostly described below in DashboardFragment. It contains information about the current user, some game configuration, and a recycler view of previous games.

Chat. This is a pretty straightforward chat view that shows my messages and the other player's messages in a format and coloring resembling the text messages on an iphone (sorry Android).

Tour of Kotlin files.

- **model/ChatRow**. Do not change. This is the model (data) for a row in the chat. Most things in here should make sense. I include the moveNumber, even though I don't yet make use of it.

- **model/FirestoreGame**. Do not change. We have a HexGame and a FirestoreGame. Both represent a Hex game, but a FirestoreGame is the serialized form of a Hex game. That means it is the form we read and write to storage. It contains "just the facts" like the list of moves.

  The HexGame is more focused on "the current state of play" for a given game. As such it contains additional information like a member variable for the current move number. There is no need for the current move number in the saved game (the FirestoreGame format). But we display the move number during the game, so the HexGame keeps track of it (and more).

  We translate from HexGame to FirestoreGame to create the game, and we translate between the HexGame move list and the FirestoreGame move list when we update our stored game. We provide this code.

  When we replay a game, we keep it in FirestoreGame format, but we also "play" it using a HexGame, which allows us to move forward (and backward) by one move and have the display update properly.

- **ui/glide/AppGlideModule**. Do not change. Currently unused. I had plans to have images in the chat, but we don't have them yet.

- **ui/ChatAdapter**. Do not change. The chat layout is probably more complicated than it needs to be, but you can just use it as is.

- **ui/ChatFragment**. I do like how it scrolls to the bottom of the chat when the chat becomes active. You have to write `initComposeSendIB`, which sets an on click listener on `composeSendIB` and "sends" a chat message by creating and storing a chatRow object.

  Snackbar a warning if the user tries to send an empty message or if they try to chat during a replay game.

  If the user chats while the game state is NotPlaying, just don't crash and don't save the message. You don't have to Snackbar.

- **ui/DashboardFragment**. You have to write most of the functionality here.

  - **Sign out button**. Signs you out and brings you to sign in page.
  - **Display info about current user**.
  - **Border label switch**. Turns on and off border labels.
  - **Interior label switch** Turns on and off interior labels.
  - **List of past games** Get the list of past games, display them in the proper format, and if any are clicked (anywhere), then navigate back to the game screen and pass "true" for replay.

- **ui/FirestoreGameAdapter**. This adapter takes a callback that is passed the FirestoreGame that the user picks. You have seen this pattern several times in this course. It means you are going to pass a lambda to FirestoreGameAdapter when you create it that uses the firestoreGame.

  You need to write onBindviewHolder, which should be a pretty familiar pattern for you at this point in the course. Please use the dimGray as the background color. Formatting dates is a little tricky (but use our dateFormat object). The red player is player at index 0, and the blue player is at index 1.

- **ui/GameFragment**. This file controls the game view. I give you some support functions, but ask you to hook up all of the pieces. These mostly interact with the view model. I give you the implementation for pressing the play another person button, because that is incomplete. But it is a good hint for what you must write.

  Here is the list of what you must do.

  - **replayGameView, interactiveGameView**. These are for the controls and the replayDate. The chat is handled differently.
  - I give you showChatRow, but you will have to initialize chatViewMe and chatViewThem in onViewCreated.
  - **redTurn, blueTurn**. These are mostly about the turn indicator, but when we aren't in a replay game, you need to call the view model.
  - **startReplayGame**. Clear the board, establish the replay game view and start the replay game.
  - **chatList**. Notice this local variable. You will have to pass it to FirestoreDB and observe it for both replay games and regular games.
  - **Make the game's view**. The view model holds the game, and the game can construct a view, but it needs to be called from the fragment that has the bindings to the play area. Here we are!
  - **For replay games, establish the replay game view**.
  - **Respond to changes in game state**.
  - **Observe and fill in player names**.
  - **Observe bad presses and flash the background if you see any.**
  - **Play AI button.**
  - **Controls for replay game.** Go to game start, to go end, previous move, next move.
  - **Display last 4 lines of chat.** This is probably the trickiest task, so you might want to leave this until later. Observe chatList. Think about the order of the lists you are manipulating.
    You only need to observe the chatList once. The way we are using firestore, we get automatic updates. Yay. (Though we will see that we have to put some effort into updates).
    My chat messages appear offset to the right, with a `lightGray` background, while my opponent's are left justified, but end before the end of the window (and have a transparent background). You will need to study the game layout a bit to see what is going on, but we give you showChatRow.

The code in `onResume` is a bit unfortunate. This code is called on navigation events. We want to know when we are called with the "replay" argument set to true, but oddly, if we use navArgs, once the argument is set to true it stays true. That is not the behavior we want, so we play this trick to clear our arguments after we check them.

- **ui/HexagonDisplay**. This provides the functionality for hexagons, like their on click listeners. It also handles which hexagons are neighbors of which others.

  The key method here is **newState**. If you want the Hexagon view to update, you should call this function. Take a look at it to see what is going on to control the look of a given hexagon cell.

  In `newState` you will implement labels for the borders and the interior hexagons. Use 12sp font size and white text for borders, black text for interiors.

- **ui/HexagonMaskView**. Do not change. This draws the hexagons.

- **AuthInit**. Do not change. This is the same functionality as we used earlier in the course. `setDisplayName` is not called at all in my code.

- **FirestoreAuthLiveData**. Do not change. This manages the FirestoreAuth state.

- **FirestoreDB**. This object (singleton) interacts with the firebase database, and you have to write at least part of every single function. You should find the data model video useful for the data model. This object is mostly about reading and writing data, it isn't about game dynamics.

  You can start testing other parts of your code without this functionality, you just won't be able to save or replay games.

  The write routines are straightforward, like createGame, updateMoves, and saveChatRow. These do their business and set an on success/failure listeners.

  But the read routines will use real time updates (addSnapshotListener). Listening for the list of games is straightforward and should be done whenever updateGameList is called (this code is given). We use live data to communicate any changes to the game list. (I use the term read, query and listen pretty much interchangeably here).

  The list of messages in a chat needs to be redone when the live data changes or when currentGameID changes. We remember the chatList live data in a nullable class variable and use it if currentGameID changes.

  I just give you this code because it is tricky, but you need to understand how to use it or your chat state in the game display won't update properly.

  The game list should be in descending date order. Chat should be in ascending (the default) timestamp order.

  You will have to build an index!. One important detail is that we fetch all of the games for a given user, sorted with the most recent games first. When you write this code, even if you write it correctly, it will not work when you first execute it. That is because in order for the query to work, firestore has to build an index. If you look at the output of your program (in logcat or run) you will see a message saying that firestore needs to build an index for this query to succeed and it gives you a URL to build the query. Just click on the URL and wait for the index to be built.

If you don't get this right the first time, it is ok. You might end up building several indexes (though you only need one). Even if you do, you should be ok. But you can delete the indexes in the firestore console if it bothers you.

- **GraphUtil**. Do not change. We determine if a player won by compute the shortest path from one goal border to the other using Dijkstra's algorithm. To do this, we convert the hex board to a graph and run Dijkstra. See, your data structures and algorithms class really does help in real life.

- **HexBoard**. There are a lot of hard-won, tricky details in this file about managing board state and converting the board to a graph.

  You should familiarize yourself with HexState. You should know what every value of this enum means.

  You need to implement some pretty straightforward functionality that will get you familiar with some of these data structures.

  - **clearMarker**. State should become unclaimed and the view updated.
  - **clearBoard**. Call `clearMarker` on the play area of the board.
  - **lightPath**. This one is a little tricky. circlefy the state of each element in the path, with a 100ms delay between each state change. This involves launching a coroutine, but it is a simple case.

- **HexGame**. Understand each GameState.

  - **startGame**. Mostly we initialize our class variables here, but we need to create the game in FirestoreDB if we are not playing a replay game.
  - **makeMove**. If something goes wrong, set `badPress.value` to cause the view to flash the board background. Check that the move is legal and the game state is valid. If it is, update the board state. I give you the end of the function, which calls completeMove.
  - **Replay games**. You need to implement the replay game functionality here, though I give you some function structure that I used (the private functions `nextReplayMove`, `prevReplayMove`), and some actual helpers, like `startReplayGame` and `whoseReplayTurn`. Here is my rank order in terms of difficulty for the externally visible functions. You might want to implement them in this order: `doReplayGameBegin`, `doReplayMoveNext`, `doReplayGameEnd`, `doReplayMovePrev`.
  - **legalMove**. It should be someone's turn and the location they pick should be unclaimed.

- **HexPlayer**. Do not change. Self explanatory.

- **HexStrategy**. Do not change. Unless you want to design an AI strategy that can beat random choice.

- **MainActivity**. Do not change. It sets stuff up.

- **MainViewModel**. Most of what is here is just fine. You need to complete `startAIGame` and `playReplayGame`.

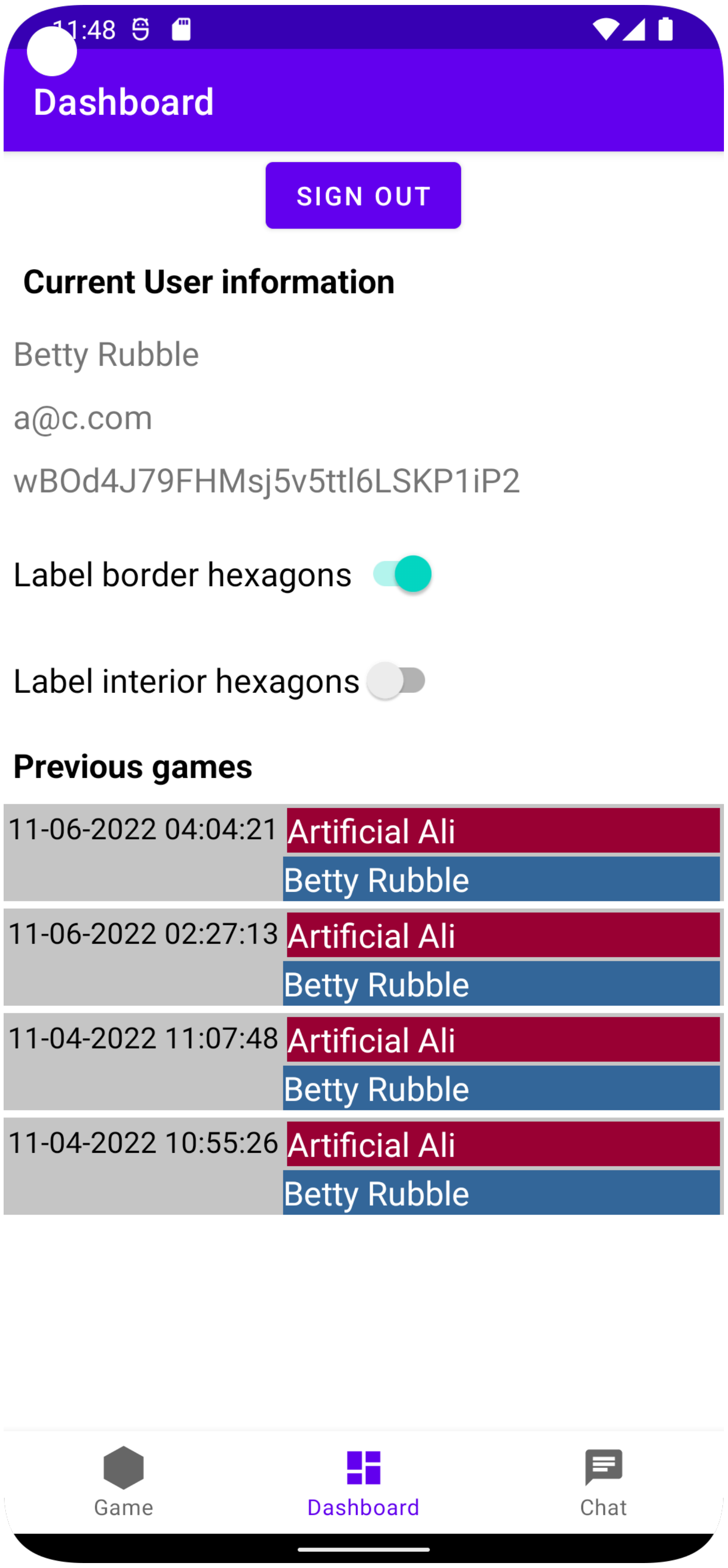
- **Storage**. Do not change. Not currently used.

Tour of resource files.

- **activity_main.xml**. Do not change. Basic bottom navigation.
- **fragment_chat.xml**. Do not change. The chat layout is a recycler view and then some controls on the bottom to compose and send a message.
- **fragment_dashboard.xml**. You need to implement this. A full-sized picture appears on the next page. Your layout should look like that. I don't care about a few dp of padding or margin here or there, but it should look good.

  The sign out button signs out using the routine in MainActivity.

  All my text sizes are 18sp.
- **fragment_game.xml**. Do not change. It is complicated. The play area is sized dynamically. It might not look quite right on your phone because there is some fine tuning in here. It should look fine on the class specified Pixel 5.
- **row_chat.xml**. Do not change. This is surely more complicated than it needs to be. But it works.
- **row_game.xml**. This is used in the fragment layout. Note the way the date appears on the left and then both players appear on the right. Combined, they go edge-to-edge. Clicking on them anywhere brings you to the replay saved game state.


11:48

Dashboard

SIGN OUT

Current User information

Betty Rubble

a@c.com

wBOd4J79FHMsj5v5ttl6LSKP1iP2

Label border hexagons ●

Label interior hexagons ○

Previous games

11-06-2022 04:04:21	Artificial Ali
	Betty Rubble
11-06-2022 02:27:13	Artificial Ali
	Betty Rubble
11-04-2022 11:07:48	Artificial Ali
	Betty Rubble
11-04-2022 10:55:26	Artificial Ali
	Betty Rubble

README file. Just modify the one we give you. It should be in plain text and named README.txt. It will be in the root directory of your submitted files. It includes these items.

1. Name:
2. EID:
3. Email:
4. How many hours you worked on this assignment:
5. Slip days:
6. Collaborators:
7. Comments:
8. Video demo: None

Extra credit ideas.

- I changed the layout of the Hex board, which created complications for layout and computing the neighbors of a given hexagon. But did it change the strategy or create an advantage for a given player? If you can prove that I did or did not change the strategy, I'd like to know. Heck, if you can provide some convincing quantitative evidence like many simulated games, I'd also be interested.

- The functionality to allow two human players to play each other shouldn't be too bad.

- Replay the chat move by move. Moving forward and backward should update the chat properly.

- Button to send screenshot of play area to the chat. Right now there are no images in the chat at all, though there is some code to support writing to Firebase storage.