

The Color Exercises

Project Writeup for CS 395T Android Programming Fall 2020 Final Project

12.06.2020

Qin Yang

Email: qyangaa@gmail.com

UID: qy2368

Glendon Ng

Email: glendonx@gmail.com

UID: gn4568

Overview

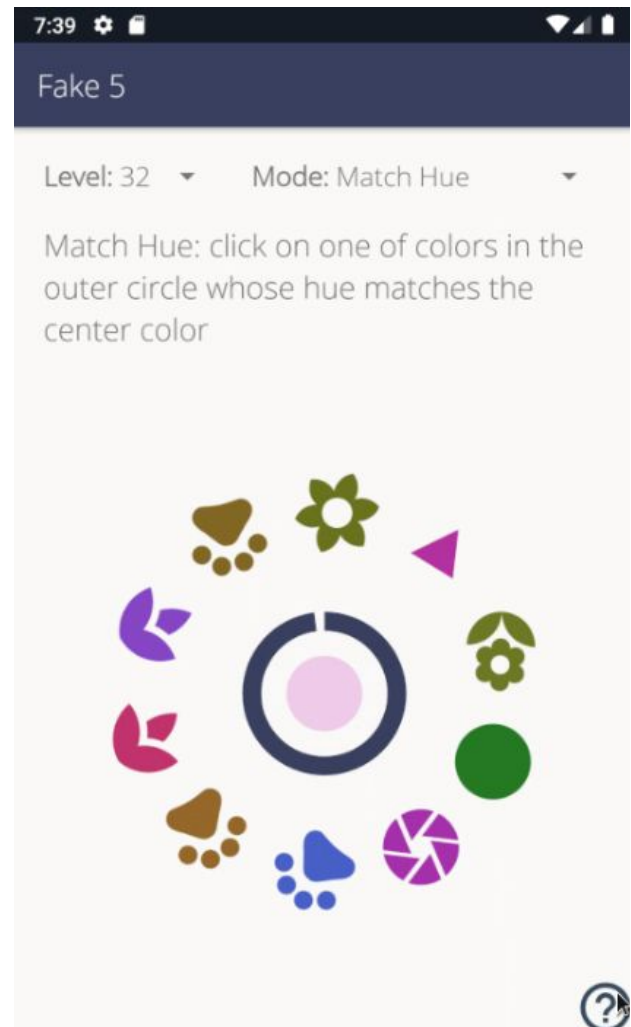
This project is an educational application with color matching exercises to train users to identify relationships between colors. Users can apply their training by constructing and sharing color palettes within the app.

Key Functionality

1. Color matching game

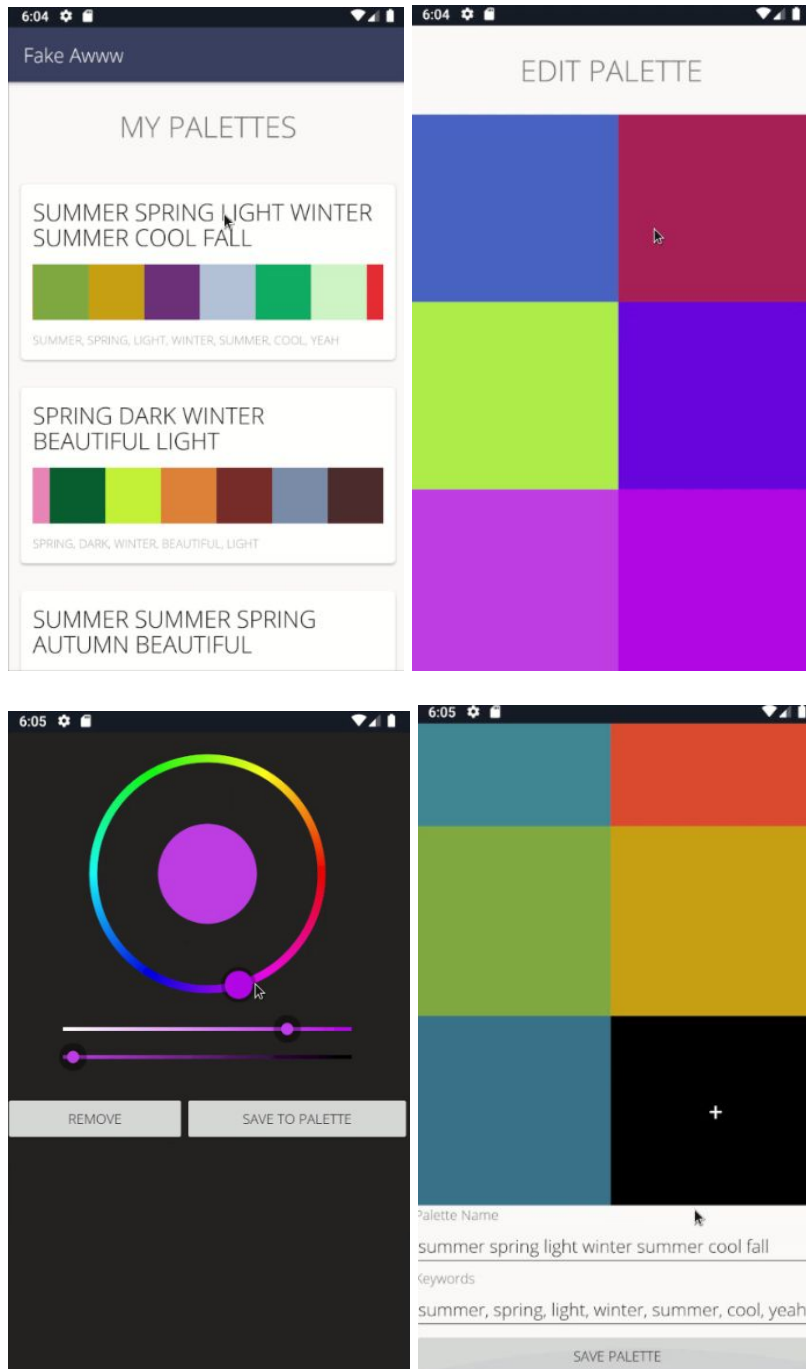
A game with multiple levels of difficulty and modes that train users to identify different relationships between colors (by hue, luminance, or concepts in color theory) . This game has the following features:

1. Automatic and random generation of exercises based on current difficulty level and mode. No exercise repository is used, and an infinite number of exercises can be generated procedurally.
2. Users can choose between exercise modes, which include matching value, hue, complementary and triadic relationships.
3. Unlock and play new levels, or revisit levels that have already been completed, if the user is logged in.
4. A circular progress bar around the center circle that indicates current progress to unlocking the next level.
5. A help button which directs users to external websites providing key information regarding the exercise at hand.
6. Rewarding animation with each new exercise set and when a new level is unlocked.



2. My Palettes

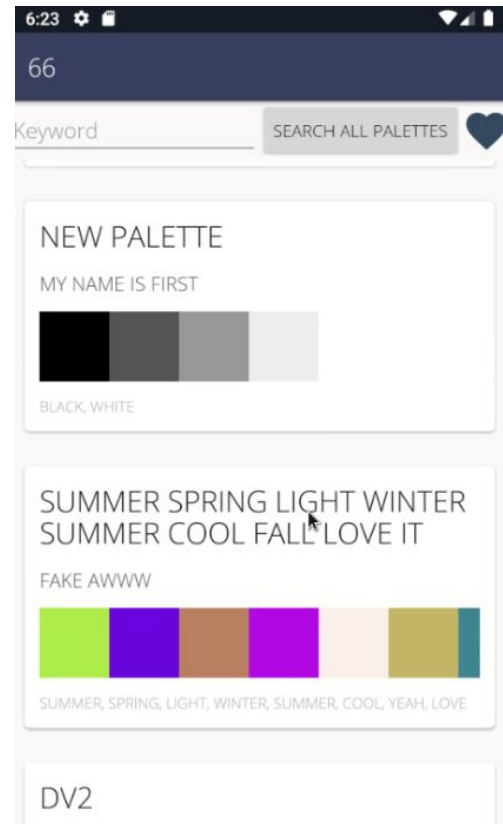
A repository of palettes constructed by the logged in user. This is the screen where new palettes can be added or edited.



3. Browse Palettes

This screen allows users to browse all palettes that are created by The Color Exercises community. Here are its features:

1. Palette access
 - a. If the user is not the owner of the palette, he can add the palette to a favorites list.
 - b. If the user is the owner, the palette can be edited.
2. Search all palettes by keywords.
3. The logged in user can see all favorited palettes.



APIs

Android Features Used Extensively

The following features are used extensively throughout the project. Their implementation details are outlined in the Noteworthy code section.

1. Android animation
2. Spinner view

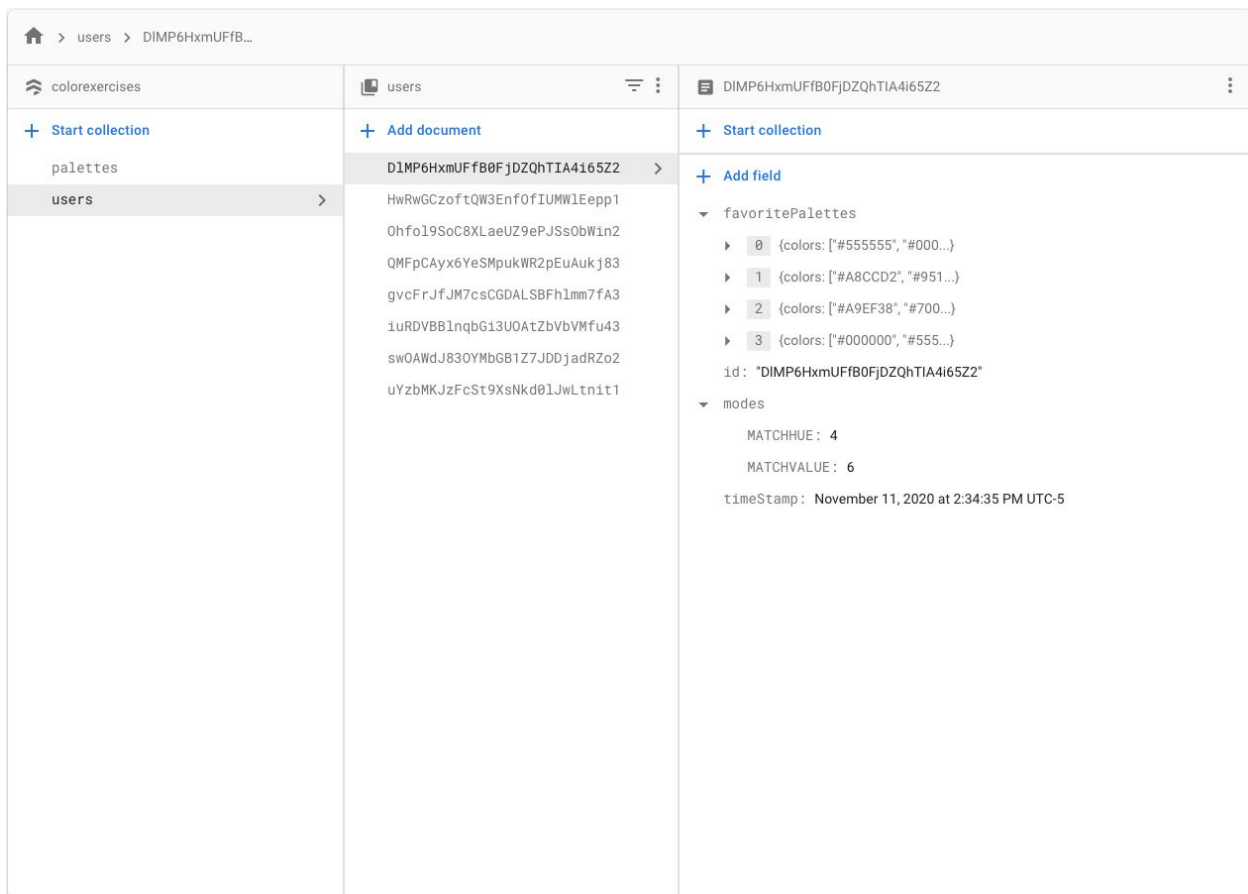
Firebase Schema

Palettes: all fields in palette objects are stored as string

<div> <div>home > palettes > 02xDWt5aSLps...</div> <div> <div>colorexercises</div> <div>palettes</div> <div>02xDWt5aSLpsLiDfpmv0</div> </div> </div>		
<div>+ Start collection</div> <div>palettes ></div> <div>users</div>	<div>+ Add document</div> <div>02xDWt5aSLpsLiDfpmv0 ></div> <div> 2HiYnjVIpS5M8ZEe27F9 21Set7kH04XwFV3Z8T7p 39ZkR87B2eRGKyFK8j0u 3a4xonGrIqjRaCfy5c6S 4GeorT3gaCAG6q10LQ6D 6zJUThq6c4BBLnU5aRZ6 86EmYCN3o9Ej13tPewo4 CfoShWJlkrFq1dHKQss6 DeMBHk1XYgexAF0xurFY DrUFEnm35nvny4D0py3q EKkbU7FDIPpYuZqr3Idv IPcwsKoe0adZUcRLSGe7 JELFtuuac3jbieu2R0rS LOiN69hMF1Qnp5Q80we0 McUu7c3QCnY7sbSaR7hb NNaZ5Xk09eZzpZxPqwM1 Ppz0x17Xw7upkQ5k0sAs QnqEbakTEa1zsealYzQQ RCLaYC7DwPQVTq8LVY5u SFnoMfDNg7Q0YkLgqxCC SVwf0r10bpmDHyKuw7nz VGjclZptyViMbJUIKoQ4 Y45E+W0GslX1BzNGnus2 </div>	<div>+ Start collection</div> <div>+ Add field</div> <div> <div>colors</div> <div> 0 "#812071" 1 "#621B48" 2 "#97130C" 3 "#6A382D" 4 "#BB5A84" 5 "#274DA8" 6 "#6D0929" </div> <div> <div>favoritedUsersList</div> <div>0 ""</div> <div>id: "02xDWt5aSLpsLiDfpmv0"</div> <div> <div>keywords</div> <div> 0 "beautiful" 1 "dark" 2 "summer" 3 "spring" 4 "light" </div> <div> name: "beautiful dark summer spring light" ownerUserID: "uYzbMKJzFcSt9XsNkd0UjwLtnit1" ownerUserName: "Fake Smith" timeStamp: November 17, 2020 at 8:12:21 PM UTC-5 </div> </div> </div> </div>

Users:

- favorite palettes are stored as List<List<String>>, with colors encoded by hex strings.
- modes are stored as dictionary of (String mode: int available levels) pair
- User id is stored as string
- User authentication information are stored in FirebaseAuth service rather than here to enhance security



The screenshot displays the Firebase console interface. The breadcrumb navigation shows the path: `users > DIMP6HxmUFFb0FjDZQhTIA4i65Z2`. The left sidebar shows the 'users' collection selected. The main panel shows the details of the document `DIMP6HxmUFFb0FjDZQhTIA4i65Z2`. The document structure is as follows:

```

{
  favoritePalettes: [
    { colors: ["#555555", "#000..."] },
    { colors: ["#A8CCD2", "#951..."] },
    { colors: ["#A9EF38", "#700..."] },
    { colors: ["#000000", "#555..."] }
  ],
  id: "DIMP6HxmUFFb0FjDZQhTIA4i65Z2",
  modes: {
    MATCHHUE: 4,
    MATCHVALUE: 6
  },
  timeStamp: "November 11, 2020 at 2:34:35 PM UTC-5"
}

```

Third Party Libraries/ Services

[com.ogaclejapan.arclayout](#)

Arclayout is used in the ExercisesFragment to achieve a circular arrangement of color buttons. This library offers easy XML implementation of clean UI, as well as straightforward programmatic initialization. However, we encountered several challenges while implementing arclayout in the context of constantly changing contents:

1. Arclayout does not support adapter integration as recycler view/ card view does. Therefore, we have to write custom binding functions to allow content updates as the exercise progresses.
2. Arclayout does not support automatic appending or deleting of items as recycler view/ card view does. We worked around this issue by changing the visibility of items according to the current size of the exercise set, while leaving a long list of items in the .xml file.

[pl.droidsonroids.gif](#)

This library enables playing animated .gif files as soon as they are added to the layout file. It allows us to handle .gif images the same way as static images in implementation, and is easy to set up with a few gradle file changes. We used this library to display a level-up animation in the matching games.

[com.larswerkman.holocolorpicker](#)

We used this color picker to let users choose colors for their palettes. This specific color picker was used because it supported color selection through hue, saturation, and luminance. Since the matching game was implemented with hue and luminance modes, it made sense for the color picker to include these parameters.

The color picker provides a nice GUI for the user to select any color. This tool was well made, easy to customize, and it integrated with our app pretty well.

Noteworthy code/ logic/ challenges/ debugging stories

Back end/ processing logic

Color and color-set generation

With color as the major theme of our application, we developed a fully self-contained, programmatically implemented system to generate colors and color sets with different requirements. Generating colors around constraints was one of the major challenges in the development of the matching game. We learned that none of the color representation systems used in android (HSL, HSV, RGB, etc.) allowed direct generation of colors given hue and value (value is the term used in art and design disciplines with the same meaning as luminance). Moreover, lightness in HSL and value in HSV do not have a linear relationship with luminance. Although we can calculate luminance directly from RGB values, generation of RGB values from luminance and hue can easily result in a non-displayable color.


After many failed attempts to generate colors analytically, we ended up using a numerical search technique to find a lightness value in the HSL encoding. In this method, we use binary search to find a lightness value, which is verified by the luminance value calculated from the resulting HSL. Binary search allows for fast convergence, and therefore did not slow down the application. Because binary search always results in quantized results given a certain tolerance factor, we have to add a random scaling factor to the results of each step to allow for more flexibility.

With a color generation scheme based on numerical search used as a foundation, we implemented a color generator class with full functionality ranging from finding a color with random hue/ luminance/ saturation to creating a set of colors with a specific relationship to each other. We also used the ideas about color encoding we learned here to create a color judge class to calculate the accuracy the user achieves within the exercises.

With these classes, our application allows for fully automatic generation of exercises with an infinite number of combinations, without the need to retrieve exercises from the database.

Coordinating Multiple Scenarios of Palettes Viewing and Editing

We have two screens for palette selection: one consisting of palettes made by the logged-in user ("My Palettes"), and the other screen, which includes all palettes across users ("Browse Palettes"). Both screens consist of a list of palettes, but the items in the Browse Palettes screen had less information on each palette compared to My Palettes. We



decided to use sub-typing on an adapter for displaying the two lists. By subtyping, we were able to pass through only the relevant parameters to our activity for viewing/editing a given palette. Additionally, we were able to specify different .xml files for formatting the palettes between the two screens. Alternatively, we could have just had one adapter, and we could have passed in a value for every field, leaving it up to our activity to decide which values to use. This alternative approach would've been simpler. We wanted to gain flexibility here, and having a dedicated .xml file to lay out palettes in each of the two screens was appealing. As the app is now, there's only one small difference in how we lay out the palettes between the two screens. We have the flexibility that we wanted, but what we ended up doing to maintain the two layouts when they were modified was to copy the layout over and remove one view--Browse palettes shows the username of the palette owner, and My Palettes doesn't.

For the palette viewing/editing screen, we had some access control logic regarding which users are able to edit a palette. Access control wasn't much of an issue, as we just checked if a palette's author was the logged-in user. If so, we displayed the appropriate UI for editing a palette; otherwise, the user can only view a palette. If a user is logged in, and he's viewing a palette that he didn't create, he can add it to his favorites list.

To have the palette functionality tie together with the color matching game, we allow users to add colors from the game to their palettes. By long-clicking on a color from the matching game, the user can add the selected color to any of the palettes in the user's library of palettes. The challenge here was passing the selected color to the adapter, and then to the activity for editing a palette. There is also a case that the user has no palettes in his/her library, and a color is selected from the matching game to add to a palette. Here, we send the user straight to the screen for creating a new palette, with the selected color already added to the list of default colors that we start all palettes with. The challenge for this case was to check if the user had any palettes before sending him/her to the appropriate screen for choosing an existing palette or creating a new one from scratch. We ran into some issues with asynchronously checking for whether the logged-in user has any palettes stored in firebase. Our issues were typical asynchronous stuff, where the order of things happening was unexpected.

UI/UX display

Coordinated spinner views for level and mode selection in Matching Game

We chose spinner views for the user to choose a difficulty level and mode while doing the matching exercises, for the view's clean and simple look. However, it was challenging to coordinate the display of these two spinners with possible data updates from multiple sources: user selection of different level/ modes, available level-mode combinations from the cloud database, and unlocking new levels as the user progresses

through the game. We used separate adapters for each spinner, and let the two adapters communicate through multiple layers of data structures:

1. Object level integer variables 'level' and 'mode': used to keep track of current level and exercise mode the user is on.
2. levelsArray: a mutable list of integers containing unlocked levels for the current mode. This array serves as the data source for levelsSpinner.
3. levelsMap: a mutable map that provides levelsArray given mode as a key. This map handles all updates mentioned above.
4. modesList: a list of modes and their information fetched from the exercise modes repository.

These data structures allow sufficient decoupling of functionalities and we were able to achieve smooth communication between the user states and the two spinner views.

Animation in Matching Game

We implemented two types of animations in the matching game using android's animation library. First is an animation effect adapted from the original Arclayout demo code. This animation allowed us to have the color circles/buttons to spring outward with a little twisting action. Translation was calculated by measuring the distance of the current button from the center button. One challenge we encountered when implementing this animation was that the views were not laid out yet when the distance is calculated. Therefore, we moved the response code into an `OnGlobalLayoutListener` to ensure completion of layout before extracting any location information from buttons. `OvershootInterpolator()` provides a smooth physical spring effect.

Second is the level-up animation shown in `ExerciseResultFragment` when the user unlocks a new level. We coordinated the motion between a gif and a textview to achieve a reveal - fade-out sequence.

What we learned

A lot of what we learned is in the details of various things, mostly. We struggled through color algorithms, UI layouts, asynchronous issues, NoSQL database problems, re-doing some things here that we did in class, structuring our code, etc. The team dynamic went smoothly, with a little miscommunication leading to us saving redundant values in the database. We also had a small coordination issue involving a branch left unmerged in git, and then enough code had been changed later that we had merge conflicts, so we manually copied in the code to our master branch.

We were on the same page throughout most of the project. We had UI mockups done before proceeding with our implementation. We also started planning our repository using a relational database, because we had just completed the SQLite Flipped Classroom, and we thought that was the way to go. We ended up using Cloud Firestore, as discussed earlier.

Project Build

The project can be built with gradle files included without additional requirements. We use Firebase as our database service, and the set up process is similar to that outlined in lectures.


Here are the indexes we've added to the db:

Composite				Single field	
Collection ID	Fields indexed			Query scope	Status
palettes	keywords	Arrays	timeStamp Descending	Collection	Enabled
palettes	ownerUserID	Ascending	timeStamp Descending	Collection	Enabled
palettes	favoritedUsersList	Arrays	timeStamp Descending	Collection	Enabled

Statistics

Line Count

	Kotlin	xml	Java
Total Lines	1926	1280	26
Authored Code	1747	870	0



We counted total line number with cloc application, and subtracted lines that are not original manually, those lines include:

1. Codes copied from the course, external libraries and online resources
2. Codes auto generated by Android Studio

Detailed calculation is included in the following sheet:

<https://docs.google.com/spreadsheets/d/1H3fhTUKdpnJCoRKfdMkG-Ddzph12UGH4LU6CfhG1CdE/edit?usp=sharing>