Kaitlyn Bush
rtb799

Title
OctoSquid Invaders!

Description
“OctoSquid Invaders!” is a arcade style shooting game where the player can control a ship with their finger to shoot invading octo-squids. This app maintains a leader board of the top 100 scores and the high score and preferences for each user.

APIs used
There were two parts to Firebase that I used in my app, the Firebase authentication and Firestore.

The Firebase authentication used email and password for users, but I abstracted the email so that the user would only have to input an username which would then be converted to the proper email format. This was so that I could maintain a list of unique usernames without having to write a separate service to keep track of already taken usernames. The username would then also be put into the database as the display name.

I used Firestore for maintaining the leader-board, user high scores, and user settings. In maintaining the leader-board I created a collection for the top 100 high scores with the names and associated values. I used a naming convention of ‘nx’ & “vx’ where ‘x’ was an integer from 0 – 100, ‘n’ represented name, and ‘v’ represented value. This was I would only have to make one database call to obtain/set all the leader-board high scores, which in turn greatly sped up the process. In maintaining the user scores and settings, I created a collection of “users” which contained documents for each username. Each document held the users’ high score and preferences, which then could be accessed when necessary.

Third party libraries
None.

Third party services
Firebase.

UI/UX/display code information
The most noteworthy UI/UX aspect is the background animations and space ship mechanics. What I did was create a custom view which would draw various vector images given to it. To simulate movement I set up a timer using a Runable & Handler, each time it was called it would update the position for each image, reset the objects for the view, and invalidate the view.
Using object oriented practices, I created classes for PhysicalObjects and Projectiles. Physical objects have an X and Y position and some assigned image. Projectiles inherit the Physical object properties and also have an X and Y velocity. This velocity would modify how much their location is displaced whenever the update function was called. Having these classes allowed for me to create a verity of interesting objects and animations.

Another important UI feature which I added was the “tap to fire/move” feature for the spaceship. Originally I planned on having it constantly fire and only have the user move the spaceship, but during development I realized that adding more user interactivity would make the game more dynamic. To implement these features I added an “onTouchEvent” listener which activated whenever the user hit the

screen. If the touch was in a given radius (500 pixels) of the spaceship if would fire a projectile. Otherwise it would set the location for the spaceship to move towards.

Back end / processing logic information

The most important back end code written would be the main game. What I did was maintain a list of physical objects including enemies, the player, bullets, and rocks. During each game tick I would iterate through the objects to update them. I would also check if the object had collided with another in which case I would remove the object. The general idea isn't too complex, but the details of getting everything centered properly and behaving as it should was quite difficult. One major challenge was getting the space ship to be properly centered on the players touch location, which involved a whole lot of math and troubleshooting to fix. Almost all of the logic behind the game was done in the GameBackend class, which utilized all of the other classes for physical objects, the spaceship, and the weapon for firing projectiles.

The second most import back end code written would be the high score leader board. I spent a decent amount of time initially planning the best way to setup the leader board to keep track of the top 100 scores. I eventually settled on using FireStore to maintain a high score document which contained 100 name/value pairs. The challenge in coding this was to reduce the number of database calls as much as possible. My approach involved first pulling the scores from the database and caching them locally. Then when the user had finished the game I could reference them to see if the endgame score was higher, in which case I would make a call to the database to update said score. So anytime the endgame score is a new high score, it will make a database call.

What I learned

I learned the importance of having strict project feature requirements for finished app. During the process of creating this app I found that there were constantly more features which could be added or improved. I believe the term is "Feature creep", and it really does sneak up on you. Originally I wasn't planning on saving user settings in the database or even having a settings page, but as I designed the app it seemed like something that would overall be useful to have. So I spent 4 or so hours implementing it and it works just like I envisioned it! But it wasn't necessary. The same was also true for having the octo-squids and rocks explode with particles when hit. I think it looks great but it wasn't necessary for the end app. While these features do add to the overall app I think it's important to plan out the design process with features and stick with that plan, otherwise you may get sidetracked. In other words, the plan that I had written with the project proposal should have been followed more closely.

The Most Difficult Challenge

In creating this app I never thought this would be a issue but I spent a decent amount of time trying to figure out how to get sounds working. My initial approach was to use the "MediaPlayer" class as I was familiar with it from the previous homework. I worked great for the background song, but there were issues in playing the SFX. What would happen is that the SFX would work great for about 5 – 10 uses but then would stop playing all together. I figured it was an issue with how the MediaPlayer initialized, so I decided to switch to using a different class. After some searching I found a class called "SoundPool" which seemed to worked okay for playing SFX for the first few moments, but quickly stopped playing the sounds every time. I found out that I needed to increase the max quantity of sounds from 10 to 100 and that solved that issue. Unfortunately using the "SoundPool" did not work at all for playing the background song. Rather than track down the root issue I figured I could try using the "MediaPlayer" class again. It kind of worked, but if more than one were created than the SFX would only play around half the time. So my solution was to only have one media player. Doing some more

testing I found it worked great! After all this trial and error I had finally found a combination of libraries that consistently played the sounds.

Code Count

Language	files	blank	comment	code
Java	15	215	125	1311
XML	35	64	2	937
SUM:	50	279	127	2248

To account for the potential lines used from stack overflow & other resources the total lines authored by me alone are probably around 2000. I did reference code from the homework and flipped classrooms but I canalized it so much it has essentially became my code.

For the repository chart, I only started using github on the last day. I didn't know we were supposed to use it since the beginning, so sorry for the terrible data. To give you an idea of the work I did, I spent the first half of Thanksgiving break completing the main game. Then I spent the last few weeks setting up the database and polishing up the app.