Rocket Stats

Senior Project Final Report

December 2016

Kyle Cornelison

Overview

Rocket Stats is the first and only mobile stats application for the popular and rapidly growing online game “Rocket League”. Rocket League is a physics based game which pits players against each other in a soccer match, however instead of controlling people the players control rocket powered battle cars. Rocket League was released on July 7, 2015 and quickly made it’s way to the ESports scene within a few days of its launch. Since its release, Rocket League has only been growing in popularity. By June 2016, Psyonix reported more than 5 million sales of the game across all platforms with more than $110 million in revenue and over 15 million unique players.

The physics based and emergent nature of Rocket League tends to create a very competitive atmosphere - and that is where Rocket Stats comes in. Players devote large portions of their life to developing their skills in Rocket League and are constantly looking for ways to help them improve or show off their accomplishments to their friends. Rocket Stats is the first application to satisfy both of these needs in a mobile setting. It provides players with global rankings, current stats, stat progression, and stat comparison. With Rocket Stats, players will no longer be “all talk” to their friends or constantly thinking about ways to improve - they will have the information right at their fingertips.

Technologies Used

- XML 1.0
- Java 8
- Android Studio 2.1
- JSoup
- Android 6.0 Marshmallow

Becoming an Android Developer

For the first few weeks of this quarter, I had to familiarize myself with the ins and outs of android developing. It was a new experience for me and I would like to highlight some of the basics.

The features core to android developing are Java, XML, and Gradle. Java is used for the backend (controller) while XML is used for the front end (view) and Gradle is the glue that holds them both together.
Java programming in Android Studio actually uses a specific subset of the language to carry out its tasks. The developers of Android created many API’s in the form of java

classes that can simply be imported and used together to create a fluid and working Android application.

The main concept behind these API's is known as an "activity". An activity consists of a Java class controller and its corresponding XML view. One of these activities is defined as your primary page that will be ran as soon as the app is started. From there, you can switch to other new activities via an Intent switch which I will get into later. In addition, once inside an activity, you can run methods from other java classes either inside the Android API or out. Although, one strange thing about activities is that there is no main method. Instead, there are a set of constructor and callback methods that the program will run if in a given state. These methods are commonly referred to as the Android Activity Lifecycle. An activity begins on the bottom left with the android character. Here we see the set of initializers and callback methods that an activity holds. Let's look at their functions.

- **onCreate()**
    - This method is called when the activity is first created and before any view is shown. This is where you should do standard initialization and retrieve any variables from a previous activity or a previously frozen state. onStart() is always called right after onCreate().

- **onStart()**
    - Called when the activity is becoming visible to the user. onResume() is called right after if the activity is in the foreground, onStart() is called otherwise.
- **onResume()**
    - Called when the activity starts interacting with the user. Here you are guaranteed that the activity is in the foreground and user input is going to it. onResume() is followed by onPause().
- **onPause()**
    - onPause() is called when the activity is going into the background but has not yet been killed. In other words, if you switch to another activity, the primary activity will be paused.
- **onStop()**
    - Called when the activity is no longer visible to the user such as the application is minimized. After this method, onRestart() or onDestroy() may be called depending on the user.
- **onRestart()**
    - Called after the activity has been stopped and is restarting again. It is always followed by onStart().
- **onDestroy()**

- Called when the activity has been terminated. Either the application has been closed, the system is closing the activity to save memory, or someone called finish() on the activity.

All of these methods' definitions are left completely up to the user with the exception of a call to super() and setContentView() in the onCreate() method as it is basically the constructor for the activity. Through these seven methods activities can be used and manipulated even better than a standard main-run program can.

The only thing about control-flow that is not explained by these methods is the starting of a brand new activity from an existing activity. To do this, we have to use the Intent class. We simply create a new Intent with the current context and the .class path of the desired activity controller. If you would like to pass information from one activity to the next, use the .putExtra() function which takes an id to reference the data and the data itself. Once satisfied with the intent, call startActivity() on it. This will call the onCreate() method of the desired activity and will kick off the chain of calls in order to start the activity.

Once you understand these Android concepts, the rest is up to Java programming and learning the vast number of Android APIs available to you.

The next side of Android developing is XML. XML is fairly basic and Android Studio XML still uses those basic concepts. However, Android also provides several APIs that give schema for XML documents. With these, you can create many different layout formats and visible attributes such as buttons, switches, checkboxes, and many more. The cool thing is that all of these schema correspond to a Java class, so the UI can be dynamically created in Java as well as pre-coded in the XML files.

Gradle is the last Android development basic. Gradle provides a way to build the project from the XML and Java files combined. It puts everything together along with the Android OS to make a working application. Most of what Gradle does is not seen by the programmer and we simply let it do its job and take it as a luxury. However, you can add scripts to the Gradle files to do non-standard things at build time.

Outline of Current Functionality

My initial proposed design plan can be found here. Currently, I have almost all of the functionality that I initially planned for, but a few additional things that I had not planned for. After the first quarter, I developed future plans in my progress report. My functionality matches more closely to those plans detailed after the first quarter rather than the initial plans.

The Login screen consists of two buttons, a space to enter a username, and a platform switch. The platform switch located directly under the “Leaderboards” button that will change from Steam compatibility mode to PS4 and vice versa. The user will have to change this to their desired platform before inputting their name to find stats.

The user can click the “Leaderboards” button (as seen in the title page) to bring up a tabbed view of the top 50 players globally. The page has 4 different tabs - one for each game mode in Rocket League. The names are split into different cells which allows the user to click on any of the top names in order to view their stats page.

After inputting their username in the field at the bottom and pressing “Get Stats!”, the stat page will come up with respective stats for the unique Steam or PS4 ID. The basic stats currently displayed include wins, goals, saves, mvps, assists, mvp%, shots, and shot accuracy. In addition, if the user scrolls down there is a display of their current rankings in all four game modes along with a visual representation of their rank.

Two new buttons have also been added to the stats page. The first of these is the “My Playstyle!” button. This button will open up a chart that shows a ratio of goals to saves to assists. Based on the ratios and global averages, the player will be assigned a “playstyle” in the title of the chart.

The second button is the “Stat Progression!” button. This page was the most difficult to implement as it required many things running in the background. This page shows a graph of stat progression over time. RocketStats begins tracking a user’s stats when a user is searched and their stats are retrieved for the first time. After that, RocketStats will update stats daily and only record changes.

Each stat is detailed by a different line. As you can see, I haven’t been playing much lately as my stat progressions are almost straight lines.

There have been a few UI design elements added. The home page has a custom Rocket League design and is visibly appealing to the user. I have also added tabs and scroll panes to the leaderboards to enhance user experience. A visual upgrade has been made to the stats page, leaderboards page, and a visual representation of ranks are available. Custom charts are now available in the stats page as well.

Advanced Features

There were a few things added in the last quarter of this project that require some advanced knowledge in android programming that I think are worth mentioning.

First, in order to set up a system that updated user's stats every X amount of time, I had to make use of various background processes and get some help from the Android OS.

The chain starts off when the android system detects that the device has been turned on or if the app has been run. It will then pass its value, "android.permission.RECEIVE_BOOT_COMPLETED" to the receiver registered in the AndroidManifest.xml file, the StatsReceiver. Upon receiving the signal, the receiver creates an AlarmManager that calls setInexactRepeating(). This method allows a repeat alarm to be created, but gives the android operating system power over exactly when it runs the code in order to maintain efficiency. Every time the alarm goes off, a signal will be sent to the second receiver which simply starts running the service. A service is a type of thread that can be run in the background, even when the app itself is not running. This service connects to the database, rocketleaguestats.com, and retrieves stats for all the users it is currently tracking. If these stats have changed from the previous retrieval, then they are recorded and stored in memory. My activities can then access these files and use them to populate the views with stats.

Second, passing data between activities was a bit interesting. Each activity in android programming has its own memory space. This means that you cannot directly pass a reference to an object because a pointer means nothing if it's in the incorrect space. With that said, the android API has a solution to this problem - Parcel. A parcel is basically an array of elements given its own pointer in memory that is outside of any activity space. This way, you can pass these "global" pointers to any activity and they can be received without problem. However, in order to pass custom classes in a Parcel, they must first implement the Parcelable class and implement a series of rather confusing methods. This proved to be a challenge and, ironically, after I was complete I found parcelabler.com - which automatically converts any java class to implement Parcelable. These parcels are passed along with the Intent object that globally changes between activities.

Storing objects to memory also required some serious thinking. Storing objects in memory in an android application is very similar to a regular java application. Simply implement the serializable class, and you can output the class to any file! However in android programming, there is a bit more to consider. Since so many threads are running in the background and you are never sure exactly when things are going to finish, you can run into many problems trying to open a file that is already being used. In order to communicate between these threads, you must use a Handler. This class can set up a channel to pass messages between two separate running spaces. This way, you can wait to run a thread until it receives a signal from the thread that is required to run before it, all the while keeping the main UI thread running at full speed.

I used a HashMap<Player, HashMap<String, PlayerStats>> to represent stat data for multiple players over different time periods. Because this data structure used a custom class as its key, I had to override the Player's equals() and hashCode() methods so that my map would work properly when serialized. After writing and reading from the serialized file, I no longer had access to the previous objects that were put into the map so I could not get anything out of the hashmap without these overrides.

Lastly, I used the package achartengine to create the charts for my stats. Achartengine consists of series and renderers that work together to form a chart.

Every series object is one set of data, or what would be represented as a single line on a line chart. You can then bring together all the series in an XYMultipleSeriesDataset which compiles all of the datasets together to fit on the same chart.

The renderers work in a similar way, every renderer represents the rendering for a single dataset in the chart. Here you can change things such as size of the line, color of the line, and whether or not the actual values are shown at the data points. Once again, you compile all the individual renderers together in an XYMultipleSeriesRenderer that will fit on one chart. With this object, you can set more global style parameters such as the background color and axes names.

The ChartFactory object then takes the XYMultipleSeriesRenderer and the XYMultipleSeriesDataSet and combines them to create a visible chart.

Difficulties

I did encounter a few difficulties while going through the introductory process to Android programming and creating the foundation for my application.

First of all, I began this project without knowledge of an existing Rocket League API that could be used to retrieve stats. I assumed that there would be one for a game with such popularity but I never actually looked for it until I had already begun the project. I was surprised to find out that there was no current official API and one had supposedly been in the works for several months now by Psyonix (the developers of Rocket League) but had not been finished yet. With a heavy heart, I began a search for an unofficial API. I found a couple that looked promising but both required a personal email to receive an API key. After several emails and weeks of waiting, I had not received an API key. So, I had to resort to web scraping. There is a popular Rocket League stats website (rocketleaguestats.com) that holds a large amount of information. I found a library called JSoup which assists with web scraping and things went pretty smoothly from there.

However, because I have to use web scraping, the loading time is a bit longer than it should be - especially for the leaderboards. It takes about 4-5 seconds to load a leaderboard currently and I would like to bring that time down if possible.

As a result of using JSoup's networking operations to web scrape, I ran into some strange networking issue with Android Studio that I didn't really know how to handle. After some careful consideration and research, I found that in Android development networking operations are not allowed on the main thread. This means that every networking operation has to be run by a separate process that is running simultaneously to your main application. I discovered the AsyncTask class in java which creates a separate thread and allows you to run networking operations. I then had an issue where I could not retrieve data from the AsyncTask because it was a separate thread and you would never know when it was going to finish. To solve this, I had to create a separate class to give the AsyncTask that would hold the web scraping url and the Document object returned by the JSoup operation so I would have a reference to it in my main thread. In addition, I had to poll for the AsyncTask to finish or else the reference would always be null.

I also had some trouble understanding the Fragment hierarchy and getting tabs to work properly. First of all, there are many deprecated methods of doing this that I found examples of online. I went through the process with two of them only to discover at the end that part of it was deprecated and I should use something else. Finally, I found Fragments, the Fragment Activity, and the FragmentTabHost. Understanding the hierarchy of these classes and how to specifically use them to create tabs and a view for each tab was rather difficult. Each activity has a view and each fragment has a view as well.

Every fragment has its own view that has to be set as the activity's view separately and presented separately to the user. Once I understood this concept, I had to understand how to translate it to tabs. This just added another layer to the hierarchy.
The FragmentActivity holds a FragmentTabHost which is set to be the FragmentActivity's view and holds Fragments which each have a view that will be set depending on the current tab selection.

Another major difficulty I had was my app skipping frames. When I started making this app, I had no idea that using threads was so crucial to the speed of it. Every other application I've made before had plenty of space, memory, and speed to get the job done on one thread. However, in android applications the main thread (UI thread) has an incredible amount of work to do in simply coordinating the app to work with the operating system and sending the views to the screen. So virtually every job other than UI placement has to be done in its own separate thread. This presents many other

difficulties such as timing when threads are complete and communicating with other threads.

In fact, I still have not completely removed all the frame skipping problems. It presents a rather large problem because I cannot give the user any feedback when the program is loading information. During the load time, my application skips frames so the loading screen is never showed. This also applies to my leaderboard not giving feedback to the user when they click on a player. This is a rather high priority issue that requires immediate attention if I am going to continue work on this app in the future.

Another difficulty I had was setting padding on each of the activity views. There is a global padding setting and it turns out that sometimes the global padding is applied, but other times the local padding is applied. It seemed to be rather random and I could not get a specific answer of when each padding was applied from anywhere - not even stackoverflow.

Speaking of stackoverflow, they seemed to be lacking answers when it came to SocketTimeoutExceptions as well. Often, my app will attempt to connect to the internet and the socket will timeout before it can. I could not find an answer as to why these timeout exceptions are happening or how to effectively remove them from the app. The solution for now is to simply retry when I receive one of these exceptions.

Setting the format of the Leaderboards properly was very difficult. The view consists of many textviews that each have their own padding, margins, and text size. Getting these to fit together properly and look nice was quite tedious and challenging.

Probably the biggest difficulty I had was completely restructuring my data retrieval in the middle of my program. As mentioned earlier, I had to resort to web-scraping my data since there was no official API available. However, halfway through the second quarter the website I was scraping the data from decided to do a complete restructure. This left my app completely broken. I had to redo much of the code I had already done and it was quite time consuming. However, while re-coding my app, I found many design flaws that I had made as a novice android programmer and was able to make my application much more efficient.

Future Plans

- UI improvements
    - Update leaderboards lists to give feedback when a player is clicked on
    - Consider different backgrounds (possibly animated)
    - Conform to ADA standards

    - Complete UI revamp of Stats page
    - Find a better way to present text on the switch at login
    - Display loading screens and better error screens
- Individual global rank
    - Find a way to get the global rank of each player in the stats page. This statistic is currently not on rocketleaguestats.com where I am getting my information from.
- Stat progress tracking/Graphs
    - Store data in my own server to reduce memory and speed complexity
    - Find a more efficient way to convey changes in stats at a very high level. (once a player has very high stats, daily changes are too insignificant to notice)
    - Add a rank progression chart as well, this one is a bit more difficult as all the values are categorical.
- XBOX ONE compatibility
    - Currently, this app only works for Steam and PS4 platform players. I would like to add support for XBOX ONE players also.
- Create a custom image for the app Icon
- A feature to compare yourself against another player (this feature was in the initial design but I decided to go a different direction instead)
- Links to rocket league training applications based on