# Retrieval and Visualization of tweets from the Twitter API to Android Google Maps

Name: Adam Lieu (100451790)
Supervisor: Ken Pu, Randy Fortier
UOIT
April 18 2016

# Contents

# Introduction

From the beginning, the initial motivation for this project was to use the Android platform and its Maps API. The starting point of this was to begin through the general use and exploration of Android itself in order to familiarise with what it can do and the tools and options available. From there, moving to the use of the Google Maps API that comes included with Android, exploring its uses such as its basic features like zooming and orientation through the compass to more advanced features such as markers and overlays. The Maps API is quite powerful and versatile as it allows for plenty of options for applying visuals to the map and allows for information to be extracted from areas. Using both the features provided by the native Android libraries and the Google Maps API, there was also looking at how to combine certain features like canvas, drawables and Bitmaps with the tools featured in the Google Maps API such as Ground Overlays and Markers. Problems were encountered with the Google Maps API however, as it was shown that there were some drawbacks with features such as Markers. As a result of these drawbacks, certain methods using the native Android API were utilized to make up for the apparent downfalls, such as combining the use of Overlays with Canvas drawings to replace the use of Markers.

The Twitter API is the other important component that makes up this project, as it is used to provide much of the data in order to map those tweets out. The API comes in several variations for different programming languages, but the essential parts of it allows for queries to made through to Twitter that retrieves tweets and the data associated with them based on specified filters. These filters can range from simple text filters and data published to location given in GPS coordinates. The data from a single tweet has a rich amount of information that can be extracted, but in the scope of this project, the most important components are the location and the date it was published. As mentioned before, this project utilizes the location data provided from the Twitter API and its published date in conjunction with the Maps API to visualize where those tweets were located and to show their age in a certain timespan.

# Background

In recent times, mobile phones and social media have exploded in popularity, to the point where it is very commonplace for people to own a smartphone and at least using one type of social media. Android phones especially have increased in popularity as the platform itself has become one of the top leading phone operating systems. For social media, Twitter has become one of the top social networks, which despite its

constraints, allowed it to gather many users and with it, a plethora of data from those users. Not only are both of these widely used, they continue to grow and gather more users. It can be safely said that from younger children to even businesses themselves, such services like Twitter have become very widespread.

This project comes from the motivation and desire to work with Android, particularly in the use of its Maps API in tandem with the Twitter API which allows for the access and mapping of Tweet data that comes from Twitter. Because of the widespread use of Twitter from many people, this has a huge potential with the data as the mapping of data can be used to see aspects of tweets such as what people in certain areas are talking about, when they were talking about it and where exactly they are located. The intention from this project is to be able to show this concept and its implementation and how it can be possible for this to be useful in aspects such as the collection of data for mapping out tweets over a period of time. This type of project can be useful for people who wish to use it to collect data from those tweets, such as finding trends from a location during a specific time. There is a wealth of information that can be extracted from this and given the popularity of Twitter, there is a huge variety of applications from this data depending on how one wishes to use it. For example, an event that is ongoing and extracting data on what people's opinions are or finding out what people are tweeting about during some sort of holiday. Anyone that would be interested in looking at that kind of data would be interested in this project and its potential uses with the visualization aspects of it, such as data collectors for the sake of researching trends or just anyone that wishes to view other people's opinions on a large scale.

# Technology Stack

## Google Maps API

The Maps API allows developers to integrate Google Maps for their own uses, allowing developers to use a plethora of features on top of just the map itself provided by Google. There are a variations of the API on different platforms, such as the Javascript Maps API, but in the scope of this thesis, the Android version of the Google Maps API is used extensively. It was found that the Android version uses version 2 of the Maps API, and this was found to have a significant difference in how certain features are implemented between version 1 and version 2, while other features stayed

primarily the same. These differences will be explored later, as an overview of basic features and more advanced features used in this project needs to be looked at first. To start however, the usage of the API requires an API key from Google itself, which is required for even allowing the application to work and access map information.

One of the most important aspects of the map is interaction, how a user can interact with the map through controls and such. The Maps API provides a number of built-in features in its user interface that allows for basic controls, such as zoom controls, rotation of the map through gestures, panning and getting current location of where the device is at the moment. These very basic features are what a user would come to expect when interacting with a map and it can be noted that it can be disabled by the developer, should it be required. In particular, the API allows developers to pan and zoom the map camera automatically via code, which is quite useful when one wants the map to immediately show a certain location at a specific magnification. To go into more detail about how it works, for something like magnification, the API defines it through zoom levels which starts at 1 for the least zoomed in image, basically showing the world map itself, to a magnification level of 20 which can show individual buildings. For camera position, it is simply defining the center of the view using latitude and longitude coordinates. There are also other aspects that one can define for the map camera but for this project, the position it faces and at what zoom level are the most important parts of it.

Aside from basic interaction, the visualization aspects of this project is another important part that relies heavily on the Maps API. The API provides a number of tools and features that can be used to visualizing the data, but first, how these visuals can be placed on the map is important to know. Since a map is being used, it is evident that GPS coordinates are used to accomplish this, specifically using latitude and longitude coordinates to define a point on the map. A vast majority of the tools and features in the Maps API rely on the use of latitude and longitude, such as Markers or Overlays. Markers especially are very important in this project, as they can be used to draw on the map to a specific coordinate and essentially have a visual that points to it. Markers also have the built-in feature of scaling with the map, as the icon representing them will always stay the same size in respect to the screen no matter on the zoom scale.

```
Marker toronto = mMap.addMarker(new MarkerOptions()
        .position(new LatLng(43.6532, -79.3832))
        .title("Toronto"));
```

Figure 1: Marker Code Snippet

This snippet of code demonstrates what is generally required to place down a single marker on the map, which requires a LatLng object, specifying the latitude and longitude. This can be especially useful in this project since a marker can be used to mark a specific tweet that has location data associated with it. However, while a marker can be useful for indicating a specific point on the map, there is a significant drawback to using them, especially in larger quantities where these markers are all placed in close proximity. A major problem that was encountered was that with enough markers all clustered together, it can be very difficult to see the map underneath all those markers. Furthermore, because markers remain the same size regardless of magnification, markers that are far apart in terms of a city can be clustered when viewing on a magnification of province or a region. As a result of this significant drawback, other methods of visualizing data onto the map is required that can be applied to large amounts of data that are close in proximity. There are other features that are in the Maps API that can accomplish this, such as Ground and Tile Overlays.


Figure 2: Clustering of markers

Ground and Tile Overlays are both image overlays that can use an image to be placed onto the map. The major difference between overlays and markers is that unlike markers, overlays are oriented and placed on the map's surface as opposed to the screen, meaning that any rotations or magnification changes to the camera would not cause the overlay to rotate or scale along with it. Much like markers, overlays rely on latitude and longitude coordinates in order to place them on the map. Between ground and tile overlays however, ground overlays are more suited for single small images placed on the map and while this may appear to be suitable for the task of visualizing

tweets on the map, it has a drawback in which it is very inefficient with a large number of tweets in terms of its overall performance. Thus, tile overlays are more suited for the task of mapping out a large number of tweets. The difference that tile overlays has is that instead of being a single image overlaid, it is actually a collection of images that is displayed on top of the map. This makes it far more suitable for larger numbers of data as well as more spread out data points, allowing tile overlays to cover much more areas of the map. Tile overlays also have another useful feature to it where one can define the tile grids for different zoom levels. For example, at zoom level 1, the tile overlay would have a grid of 1x1 tiles covering the entire map.

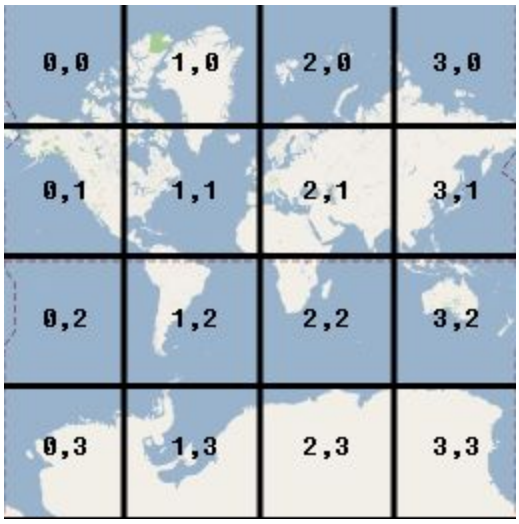


Figure 3: Example of a Tile Overlay and the grid used for a zoom level (Source: https://developers.google.com/maps/documentation/android-api/tileoverlay#coordinate)

Overlays provide a means to place images on the map, but they do not provide the means to produce the images themselves, they require an external image in order to work. This is where polygons come in for that other piece that is required to visualize data. The Maps API offers a built-in feature which allows for easy drawing of shapes like polygons, circles or lines. Much like markers, they simply only require the latitude and longitude coordinates to specify where they should be placed and other specifications such as radius in meters to indicate how big they are for circles or the thickness of the lines for the drawings. The shapes offered by the Maps API gives a simple solution for anyone that wishes to use it for marking out points but it contains a serious drawback similar to ground overlays, they are found to be very inefficient to work with when there are many points on the map that require a shape to represent it. Because of this, it is more beneficial to use the native Android API for drawables like Canvas to visualize the data points on the map. A Canvas is the interface in which the drawables are drawn

upon, through this, the drawings are performed on a Bitmap, a Bitmap being required for a Canvas. Within this project and for the efficiency problems of the Shapes from the Maps API, Canvas is more efficient since shapes can be drawn onto a Bitmap, which that Bitmap can then be placed onto the map itself as a single image. However, this lacks the ease of use that Shapes had where it simply only required the map coordinates to be accurately placed. In order to properly place bitmaps and canvas drawings onto the map, there has to be a way to convert screen coordinates, which bitmaps and canvas uses, to GPS coordinates on the map. The Maps API has a class called Projection that handles the translation between screen location and geographical coordinates. This class can be used to solve the issue of Canvas and Bitmaps only using screen coordinates, such that the screen coordinates can be used to convert those into GPS coordinates so they can be placed in the proper position. However, there was a fundamental issue that was encountered that required another solution that the Projection class was unable to address, this issue will be elaborated on later, but for now, the Projection class was something that sufficed for basic work between screen coordinates and GPS coordinates. Finally, by combining all of these features in the Maps and native Android API, animation with those drawings is possible and can be useful in the visualization. Not much detail is required on explaining animations for this project, but it can be said that within the context of a map, animations is used primarily to represent the changes in data points in each time interval, showing new points on the map with each change in time.

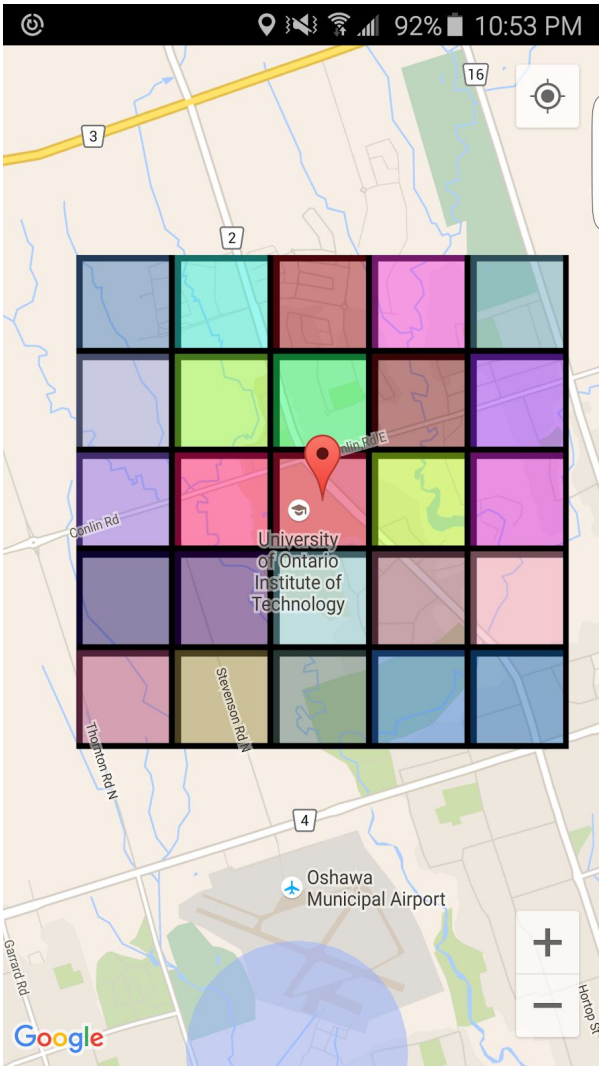


Figure 4: Combining Canvas, Overlays and animations

## Twitter API

The second crucial part of this project comes from the use of Twitter and the vast amount of data, mainly in its tweets that contain location data of where it was posted and when. The Twitter API is very extensive, with many libraries and features to work with depending on what is needed. There are four main objects in the API**:** Tweets, Users, Entities and Places. For this project, Tweets is the most important object to be working with since it contains the data needed. To the users of Twitter, a tweet is simply a sort of message that is similar to a status update that can only contain 140 characters and other associated data such as time posted, the user that posted it and the location.

To developers and anyone using the API however, a single tweet has a large plethora of data that can be accessed that would normally not be visible to the common user. These types of data can range from GPS coordinates to its unique identifier of the Tweet. For this project, the geolocation data and the time of creation are the most important pieces of data from a Tweet, as those can be used to place them visually on the map and in a time animation.

Much like the Maps API, the Twitter API requires authorized access with OAuth before it can be used. This requires the user to have an existing Twitter account and to create a Twitter application that will provide the necessary API keys that will be used for authorization, there are two components for accessing Twitter on an application basis and they consist of a consumer key and a consumer secret. However, Twitter enforces a rate limit in their API that limits on the amount of queries one can make, for tweets, a single query is limited to a maximum of 100 tweets and the API rate limit lists 180 requests in a 15 minute window. This can pose as a significant drawback, as the query limitations means that retrieving those tweets would be very limited to a smaller number, even more so in an area where more people use Twitter actively and more tweets are produced in the same area. Furthermore, queries made in areas that do not have as many people using Twitter would produce not as many tweets and may even appear to be nonexistent at times. Regardless of the amount of tweets that are returned in a query, a query is still used.

As a way around the drawback from the rate limits and sparse data in certain areas, caching query responses would have to be used in order to obtain a decently sized data set. This would have to mean that for the most part, the project would have to work with data sets indirectly, first retrieving them and then storing them for later use. This can be done in a variety of ways, but ideally, an external server running the Twitter API to retrieve tweets automatically would be a good option. The reasoning behind this is due to the large size of the data set as there would be a very large number of data. Aggregating the data in one set would be relatively simple due to how the data is formatted. The Twitter API formats its tweet data in the JSON (Javascript Object Notation) format, good to use as these are native libraries in Android and in other programming languages that can parse JSON data. As a brief summary, JSON uses a format where there are name/value pairs or lists of values. In the context of tweets, there would be many name/value pairs representing the numerous amount of data in a single tweet. Because of this already simple to use format, one would only need to cache data by first retrieving them from the API and then saving it for later use. A crawler that would automatically make queries through the Twitter API and store them in a file with the JSON format would be used to cache the data. This data would then be

used by the project itself by first parsing through the data to retrieve the necessary pieces of data such as coordinates and time of creation, then sorting them in chronological order and then visualizing them in the form of points on the map.


```
{
    "contributors": null,
    "truncated": false,
    "text": "UD: [1 Alm] Fire (Residential) - Grange Crt b/w Grange Avenue / Vanauley Walk, Toronto (7 Trucks)",
    "is_quote_status": false,
    "in_reply_to_status_id": null,
    "id": 673966740674621442,
    "favorite_count": 0,
    "source": "<a href=\"http://ijg.me\" rel=\"nofollow\">ijg</a>",
    "retweeted": false,
    "coordinates": {
        "type": "Point",
        "coordinates": [-79.398491, 43.651645]
    },
    "entities": {
        "symbols": [],
        "user_mentions": [],
        "hashtags": [],
        "urls": []
    },
    "in_reply_to_screen_name": null,
    "in_reply_to_user_id": null,
    "retweet_count": 0,
    "id_str": "673966740674621442",
    "favorited": false,
    "user": {
        "follow_request_sent": null,
        "has_extended_profile": false,
        "profile_use_background_image": false,
        "default_profile_image": false,
        "id": 99788417,
        "profile_background_image_url_https": "https://abs.twimg.com/images/themes/theme1/bg.png",
        "verified": false,
        "profile_text_color": "333333",
        "profile_image_url_https": "https://pbs.twimg.com/
        profile_images/3289419608/102b412f2c69ecf7780dfd3d8b9a70df_normal.jpeg",
        "profile_sidebar_fill_color": "FFF2F2",
        "entities": {
            "url": {
                "urls": [{
```


Figure 5: An example of the JSON format used in a tweet retrieved from the Twitter API

# Process of Development

This project first started with the introduction to the Maps API and the exploration of its libraries and features, starting with very basic functionality such as being able to manipulate controls and cameras views to placing down a marker through long press. Much of the initial work was very simple, with most of the focus being on aspects such

as working with basic interactions between the map and the user, such as handling onclick listeners with different objects, creating a navigational drawer and working with geocoding with respect to a Marker's location. After some experimenting with Markers and observing the drawbacks that would occur, we later moved to and explored the uses of other Maps API features such as Overlays, Canvas drawings and understanding how the map handles the placement of those through the use of latitude and longitude coordinates. Basic work was done using the Map API Overlays in order to understand how they can be applied with datasets, such that we can use those visuals to represent each and every one of the coordinates provided from that set, as well as how those visuals can be customized in order to get the best kind of visualizations to the user. Initial work was done with a small dataset of 85 points-of-interests and Ground Overlays, specifically bike stations in a small area in Toronto. It was found that visibility of the data to anyone viewing would be a crucial aspect, since using something such as an icon of a bicycle can be hard to see when viewing the map from a zoomed out perspective. Therefore, simple shapes such as circles would have to be used in order to properly represent the data. This was also when learning how to work with JSON files came in, since many datasets were in the JSON format and knowing how to parse it properly in order to extract the relevant data would be required later with the Twitter datasets.

As mentioned earlier, the Google Maps API provides built-in functions for drawing shapes and other drawables. This is very useful for our visualizations, as there would be little effort required to place something as simple as a circle to represent a given coordinate. However, it was encountered that the shapes can be a performance issue as using them in large amounts can be quite slow and bottleneck the entire project. They were found to not be very flexible, using them for animations was found to be quite slow and customization for them was quite limited, only with regards to the size of those shapes, their colour, and the stroke width of the drawing. The performance aspect became a major issue, so we moved onto the use of Canvas drawings. Canvas drawings were found to be much more versatile than the Maps API shapes and they do not have the same problems that were encountered from shapes where they had performance issues when scrolling and placing them. We started with basic uses such as combining it with Ground Overlays, since those require a Bitmap which is what a Canvas drawing is drawn onto. We also combined these with animations to see how canvas drawing animations would work, by constructing a grid of colored cells that would randomize every second. Because of how Canvas drawings are placed normally, they did not use GPS coordinates for placement and instead used screen coordinates. This was an issue and obstacle that had to addressed, but we found that Android provides a class in its API called Projection that handles the translation between

geographic coordinates and screen coordinates. Knowing all of this is crucial for applying them to the Twitter API's dataset, as visualization is a very important part of this project.

After much exploration of the Maps API and Android's tools, we moved on to the Twitter API, the other important half of this project. We initially started looking at what you can do with the Twitter API, such as what kinds of queries you can make and how. The most important part of the queries was finding out how to properly filter queries based on their geolocation data, as that would provide us with the coordinates for which that tweet was created and when. The Twitter API offered several libraries for making queries, but all of them required authentication through OAuth, which meant that a developer's account with Twitter had to be created in order to get the API keys needed to make requests through the program. Initially, we used the Twitter library for Java called Twitter4j, which provided classes that made it simpler to filter and make requests for tweets based around Toronto and Oshawa. However, there were a few problems that were encountered when making queries to Twitter. The first was that in the Twitter API, the amount of returned tweets from a query can only go so far back in time, which means in areas such as Oshawa where there may not be that many people using Twitter or using Twitter with geolocation on, there would be very little tweets retrieved. Secondly, the Twitter API enforces a limit on the amount of queries made over a period of time, as mentioned earlier. Thirdly, each query made can only have a maximum of 100 tweets returned. These limits meant that the amount of data that we can show and visualize would be severely limited and not representative of what is being tweeted.

```
double lat = 43.6532;
double lon = -79.3832;
int radius = 10;
String mesUnit = "km";

Query query = new Query("").geoCode(new GeoLocation(lat, lon), radius, mesUnit);

query.count(200);
```

Figure 6: A snippet of code in Twitter4j for making a query

As a workaround for these limitations, caching the tweets was a good option that would allow us to get a large number of continuous tweets over a period of time that we can work with. In order to cache those tweets, a Python script was written using the Python Twitter API to retrieve tweets from both Toronto and Oshawa. This script would act as a crawler that would run on a server over the course of about 3 weeks, retrieving tweets over a time interval.. The crawler would retrieve a tweet and append it to a text file, respective to its location, and the format of each line in this text file is an individual

JSON. The total number of tweets amounted to 52322 tweets from Toronto and 4208 tweets from Oshawa. After the data was retrieved, we parsed each line of text file, retrieving the geolocation coordinates from each tweet and mapped them out. However, a serious bottleneck was found with the previous method of placing down Ground Overlays where we had placed a Ground Overlay for every point in a loop. The performance issue from this method proved to be too slow to be feasible, taking a long time to load just even 10000 tweets, resulting in the program to freeze for several minutes while it draws a circle for each tweet.

```
for(LatLng i : pos) {
    mMap.addGroundOverlay(new GroundOverlayOptions()
                    .image(desc)
                    .position(i, 150)
    );
```

Figure 7: Ground Overlay

Because of the major performance issue from the use of Ground Overlay, different methods were attempted to fix this problem, such as using a single bitmap and placing the circles that represented the points beforehand as a Ground Overlay. However, this had the issue of using too much memory as it would have required a very large bitmap that simply would not work. Furthermore, the use of a class called LatLngBounds was attempted to properly align the bitmap such that the circles would align properly to their geographic locations, but this did not work as the large number of points interfered with it being accurately placed. Finally, a solution was found that came in the form of using Tile Overlays, provided by the Maps API, and other classes that helped with the placement of those tiles. A Tile Overlay is different from a Ground Overlay in that it is a collection of images placed on the map as opposed to just one, this fixes the issue of a maximum bitmap image from Ground Overlays as we can have multiple images that make up a large one. There was still the issue of properly placing our points to the right geographical coordinates, so we had to make a class that handled this, namely a type of map projection called Mercator Projection. Mercator Projection is commonly used in Google Maps and OpenStreetMaps, as well as other applications that used a kind of tiled map that also used Mercator Projection algorithms in order to calculate the placement of those tiles. A class was written to handle the translation between screen coordinates and geographical coordinates. We also had to write another class called newPoint that was basically the same as the Point class but used double values as opposed to integer values since that was important for the Mercator Projection as the regular Point class would result in the loss of data. With these classes implemented, we were able to visualize the points from the tweets without the

bottleneck from before, taking 30 to 40 seconds in total to show the tweets as opposed to the several minutes before that also resulted in the program freezing.

```
public class MercatorProjection {
    final double worldWidth;

    public MercatorProjection(final double worldwidth) {
        worldWidth = worldwidth;
    }

    public newPoint toPoint(final LatLng latlng) {
        double x = latlng.longitude / 360 + 0.5;
        double y = 0.5 * Math.log((1 + Math.sin(Math.toRadians(latlng.latitude)))
                / (1 - Math.sin(Math.toRadians(latlng.latitude)))) / (-2 * Math.PI) + 0.5;
        return new newPoint(x * worldWidth, y * worldWidth);
    }
}
```

Figure 8: Code snippet of Mercator Projection and calculations used for projection

Finally, we implemented animations that would represent the time lapse of tweets through the use of seek bars. Because that was no built-in two way seekbar in Android and due to time constraints, we opted to use two seekbars instead, one representing the lower limit and one representing the higher limit. These seekbars would be used to filter tweets that fall within a time interval, for example, tweets that were created between December 10th to December 20th. These intervals could be adjusted through the use of a navigation drawer that we implemented, where you could set the interval to either be 6 hours, 12 hours, 1 day or 1 week intervals. The seekbars could be manipulated when the program was running such that the time interval can be altered and the tweets that are displayed would change accordingly. A performance issue was discovered when implementing the animation, which was the inefficiency of indexing the parsed coordinates and time intervals inside of a JSON format, resulting in a slow and unresponsive seekbar when manipulating it. This was fixed by switching it to a HashMap, which allowed for very quick and responsive seekbars and change in tweets being shown. There was also another performance issue found with the text files themselves, since each line in the files is an individual JSON, our program would read and parse a JSON at each line. This was found to be quite slow, taking approximately 200 to 300 milliseconds per line, which would add up to about 30 seconds of loading those text files. Lastly, we implemented different colours being displayed when showing the tweets in order to represent the age of the tweet, such that older tweets would be closer to green and newer tweets would be closer to red, this was done so the tweets

can be easier to distinguish when there are large quantities of them that span several different times.

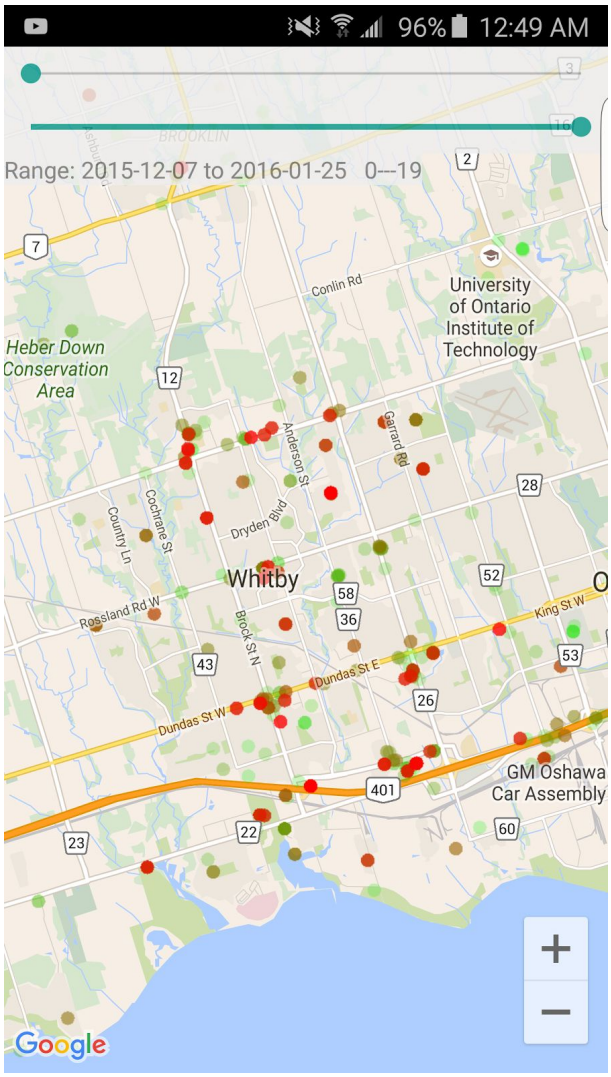


Figure 9: Tweets visualization using different gradient based on time interval

# Conclusion

To summarize, there was a lot learned since starting this project. The original problem was that we had wanted to take a large set of data and their location and visualize them onto the map. As mentioned, this proved to be a challenge due to the issues and drawbacks encountered when attempting to implement it and when working with the possible tools provided, such as the inefficiency of Shapes from the Maps API when using it for large sets of data or the Twitter API's set limit of queries. We had

encountered much of the Maps API with its tools for visualization as well as supplementing that with native Android classes like Canvas, obtaining a solid knowledge and understanding with how to use it properly. Similarly for the Twitter API, a good understanding of it was established through exploring its uses and how to integrate it with the project, despite its set limitation on queries. All of this has led to what we have done, which is a demonstration of all of the obtained knowledge from this project, the result being a visualization of about 40000 tweets set in both Toronto and Oshawa over the course of several weeks on a map.

# References


- Google Developers. Map Objects - Google Maps Android API. February 15 2016. https://developers.google.com/maps/documentation/android-api/map
- Google Developers. Markers - Google Maps Android API. February 15 2016. https://developers.google.com/maps/documentation/android-api/marker
- Google Developers. Ground Overlays - Google Maps Android API. February 3 2016. https://developers.google.com/maps/documentation/android-api/groundoverlay
- Google Developers. Tile Overlays - Google Maps Android API. February 15 2016. https://developers.google.com/maps/documentation/android-api/tileoverlay#remove
- Google Developers. Shapes - Google Maps Android API. February 15 2016. https://developers.google.com/maps/documentation/android-api/shapes
- Android Developers. Canvas and Drawables. http://developer.android.com/guide/topics/graphics/2d-graphics.html
- Android Developers. Canvas Documentation. http://developer.android.com/reference/android/graphics/Canvas.html
- Google Developers. Google APIs for Android - Projection Documentation. October 4 2015. https://developers.google.com/android/reference/com/google/android/gms/maps/Projection
- Google Developers. Google Maps Android API v2 Samples. https://github.com/googlemaps/android-samples
- Android Developers. Seekbar Documentation. http://developer.android.com/reference/android/widget/SeekBar.html
- Google Developers. Getting Last Known Location. http://developer.android.com/training/location/retrieve-current.html

- Android Developers. Creating a Navigational Drawer. http://developer.android.com/training/implementing-navigation/nav-drawer.html
- Android Developers. Geocoder Documentation. http://developer.android.com/reference/android/location/Geocoder.html
- Google Developers. Google API for Android - LatLngBounds Documentation. https://developers.google.com/android/reference/com/google/android/gms/maps/model/LatLngBounds

- Klokan Petr Pndal. Google Maps Tiler: Coordinates, Tile Bounds, Projection. http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/
- Weisstein, Eric W. “Mercator Projection” From Mathworld - A Wolfram Web Resource. http://mathworld.wolfram.com/MercatorProjection.html
- Israel, Robert. Mercator’s Projection. January 20 2003. http://www.math.ubc.ca/~israel/m103/mercator/mercator.html
- OpenStreetMap. Mercator. February 9 2016. http://wiki.openstreetmap.org/wiki/Mercator

- Twitter Developers. API Overview. https://dev.twitter.com/overview/api
- Twitter Developers. OAuth. https://dev.twitter.com/oauth
- Twitter Developers. API Rate Limits. https://dev.twitter.com/rest/public/rate-limiting
- Twitter Developers. GET search/tweets. https://dev.twitter.com/rest/reference/get/search/tweets
- Twitter Developers. API Console Tool. https://dev.twitter.com/rest/tools/console
- Tweepy Documentation. http://docs.tweepy.org/en/v3.5.0/
- Twitter4j Javadoc. http://twitter4j.org/en/javadoc.html