MONASH UNIVERSITY • FACULTY OF INFORMATION TECHNOLOGY

Exploring Twitter Networks with the Leap Motion
Student Project

Course:	FIT2044 – Advanced Project
Course Coordinator:	Ann Nicholson
Supervisor:	Jon McCormack
Team Members:	Nicholas Smith
	Tyson Jones
	Keren Burstein

November 2013

Abstract

This report serves as a documentation of a student group project from Monash University. The project focuses on the exploration of a Twitter network using a 3D graph and the manipulation of that graph with the Leap Motion controller.
This report is mainly intended for an audience with some programming background. However, not all sections are technical and therefore anyone could gain an understanding of the project by reading the report. The report provides an overview of the project, a more in depth explanation about the final application's development and implementation, and a suggested list of further improvements of the current implementation.

Table of Contents

1. Introduction

1.1. Who we are

We are undergraduate IT students from Monash University in Melbourne, Australia. This semester we enrolled in a complementary unit that allows us to complete a project from a selection offered by the Faculty of Information Technology. The three of us chose to participate in a project named “Exploring Twitter Networks with the Leap Motion”, supervised by Jon McCormack.

1.2. The Leap Motion

The Leap Motion controller is a newly-developed experimental device which is basically designed to provide a new form of human-computer interaction, different to the mouse, keyboard or voice control. Leap Motion is a privately funded startup, and as of recently, are collaborating with ASUS and HP to integrate their product into laptop computers.

Figure 1: The Leap Motion controller device

The Leap controller allows the user to control the computer using hand gestures in the air, within a volume of 8 cubic feet. The Leap is equipped with 2 cameras and 3 infrared LEDs to monitor hand movements. It connects to the computer via a USB.

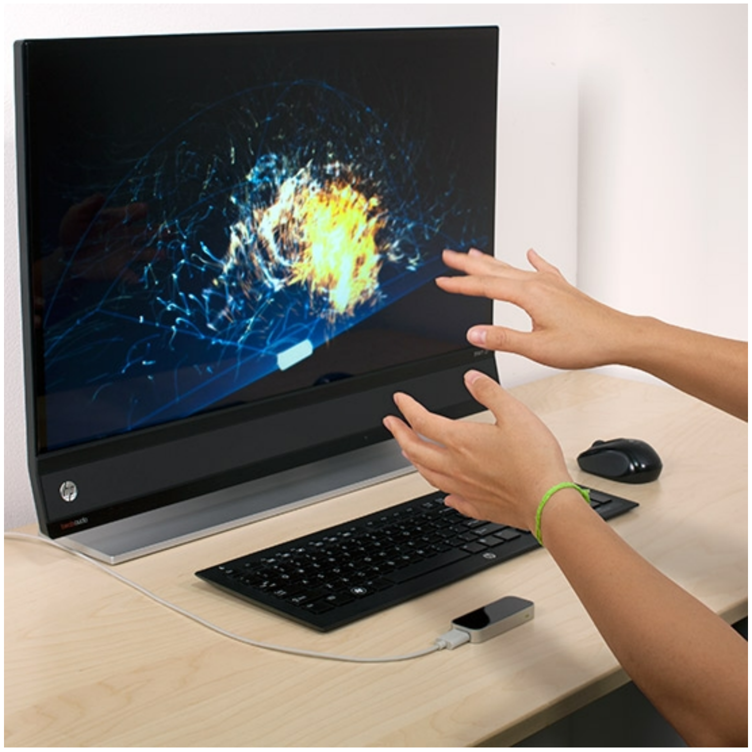


Figure 2: Controlling a flock of fish using the Leap Motion

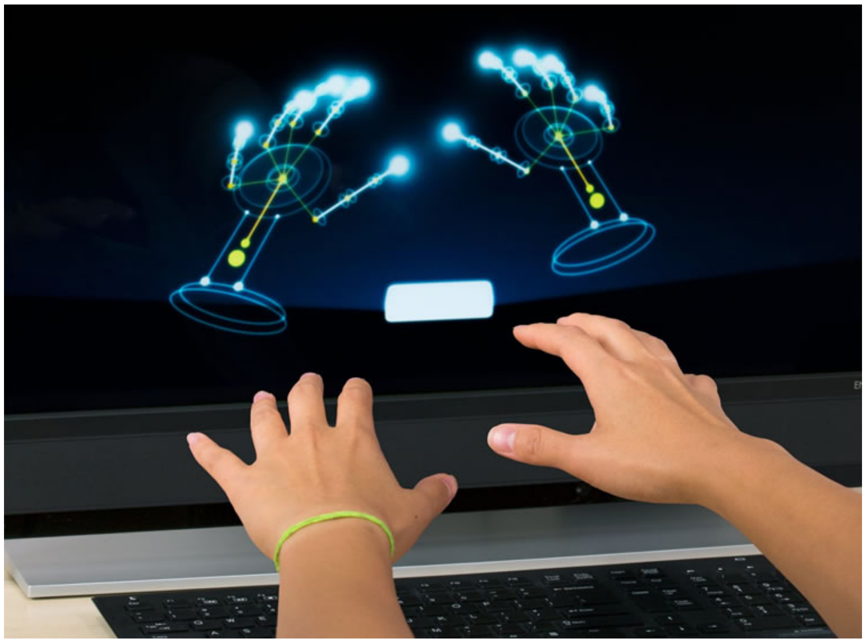


Figure 3: How the Leap sees the user's hands

1.3. Aim of the project

The aim of the project was to explore the abilities of the new Leap Motion controller and the possibilities that come with this new way of interacting. Particularly, the project was focused on how the Leap can enhance the exploration of a 3D graph. Therefore, we have decided to team up and develop an application that would demonstrate whether or not the Leap provides any improvement to the user's experience.

1.4. Project development

The early stages of the project mainly involved exploring which types of functions the Leap improves when being used as an interaction tool, or at least, is expected to improve in its perfect form. After arriving at the conclusion that the Leap is a useful way of simplifying the exploration of a complicated 3D figure, we realized that the best choice would be to expand on graph manipulation. The best way to go about it would be to demonstrate the powers of the Leap when trying to answer questions about the relationships within a social network. This would be done by extracting data from the social network's database and displaying it as a 3D graph. The Leap would then be used for easy manipulation of a graph that is expected to look messy at first glance. The social network we have chosen is Twitter, owing to its limited relationship options with comparison to other publicly available social networks. However, this idea can be done with any network, and adding to that, also with any kind of data that is represented with a graph.

Since a social network can be explored in many different ways and focusing on all possible options would be too time consuming, as the project developed, we narrowed them down to one final "analysis mode". In this mode of the graph, we look at the direct relations between a centralized user and that user's followers or followees with an additional given indication of how much interaction occurred between them recently. Other interesting connections can be spotted while looking at the non-central users' networks which are also optional to show as part of the graph.

2. Gathering the data

Author: Tyson Jones

2.1. The Twitter v1.1 REST API
or: the (rate-limited) tools for the job

Twitter user data is (somewhat publicly) available through the Twitter HTTP 'REST' API, and requires user authentication (using O'Auth - abstracted through our use of the Codebird library) for all relevant API calls, which follow along the principle of…

- **users/lookup -** get a batch (at most, 100) of *User Objects* for a provided batch of *User ids*
  (A *User id* is an unique number representing a specific Twitter account, and in its numerical form, is completely anonymous. A *User Object* contains all information about a Twitter user's account with brief data on their activity)

- **friends/ids -** get a list (at most, 5k) of users following a supplied *User id*, in the form of *User ids*.
- **followers/ids -** get a list (at most, 5k) of users followed by a supplied *User id*, in the form of *User ids*.

Though there are some further API libraries called for collecting tweets, favorites and timeline data, these three methods are the crux of the construction of a twitter network in local memory. (we decided the limits on friends and followers collected - 5k per user - was sufficient, due to our existing limits on graph capacity and performance).

Intuitively, to retrieve a user's surrounding network (to a reasonable depth, lest we pull all of Twitter), we must call these methods and recurse upon their results. The big derailment to performing this task at runtime, client-side (as early prototypes attempted to do), are the *rate limits* imposed on these methods; **users/lookup** may be called 12 times per minute (collecting at most, 1.2k user objects a minute), while **friends/ids** and **followers/ids** may only be called a restrictive once per minute each. More frequent calling causes the API to temporarily block our O'Auth tokens, so no data is received. As a demonstration of this restriction, fetching Obama's network to a depth of 2 (get his friends/followers and their friends/followers) would take at least 75 years. These restrictions render client-side collection entirely impractical; a server-side solution was needed, whilst accepting the hard limit on total data we could receive (given any server-side program will not be given the execution freedom of 75 years).
It was imagined very early - because of the long running time of our scripts, the magnitude of the collected data and the extra libraries needed to achieve server-side concurrency for our recursive algorithm- that a free webhost would be insufficient and eventually we received a red-hat-linux 6 server for our disposal. Development then turned to technical troubleshooting.

2.2. Deciding on an a server-side environment

or: the desperate search for an adequate language

After setting up what was a very bare server (and installing and configuring all of Apache, PHP5.3, MySQL and a variety of other libraries - my first real administrator experience with linux), the actual runtime environment needed consideration. Though PHP was the obvious server-side language, and was also (as well as Javascript) supported by the Codebird library, several problems loomed in its use.
Firstly, PHP is not designed for long-life execution, and though I tweaked the Apache and PHP configurations endlessly to avoid any time-outs, there were intrinsic issues in monitoring the status of a running script.
Secondly, and most importantly, PHP does not support threading or concurrency or asynchronous timers, all of which are foundation necessities in the intuitive data collection algorithm. Considerable effort was invested in trying to install PECL pthreads; a PHP library allowing threading. After installing PECL and PHP-devel (needed for a function 'phpize()' used in pthreads compilation - PHP-devel itself was a world of trouble, belonging to a repository that Yum couldn't find), pthreads could ultimately not be installed, since it required a configuration flag (Zend Thread Safety enabled) to be passed during PHP compilation, which is not possible

through Yum. The administrator explicitly discouraged manually installing the PHP dependencies, since this was known to cause tremendous trouble with Yum during updates.

2.3. Integrating into Node.js

or: the integration from hell

My next idea was to switch to an inherently concurrent (or at least, event-driven) language, such as javascript and explore its use server-side; Node.js appeared as a much needed relief, allowing asynchronous calls in our server-side javascript, which - at first glance - seemed compatible with the Codebird library (for javascript). Assuming compatibility, the launch sequence would involve the User authenticating their Twitter account through PHP (providing a much nicer, more elegant interface than the Javascript paradigm) and PHP passing the authentication tokens to javascript via command line arguments (PHP passing session parameters whenever a redirect was needed), which then initiated the data collection - Node.js could not be the direct user entry point, for it could not establish a server environment in our linux server, for whatever IP related reason..
There were a few emergent problems with this strategy:

- Node.js uses a ‘require’ system for importing modules, requiring a format in the imported code that Codebird did not at all adhere to (not being intended as a Node.js library). This was overcome via a very dodgy workaround, by importing the source code as a string and passing it to *eval*.
- Since the Node.js script is initiated from within PHP (via a dodgy exec call), it’s launch is concurrent to the PHP scripts execution and so all connection to it is completely lost: the output of the javascript could not be monitored, which made error checking and debugging a living nightmare. Server logs were impractical, so all testing was done via launching the Node.js scripts directly from the shell (through Putty) and passing in the authentication tokens, having fetched them separately.
- Codebird.js (since expecting to be used in a client-side, browser environment) uses the XMLHttpRequest object when contacting the Twitter API (or the ActiveXObject in internet explorer), which is supplied by the browser. On the server, there is no supplication of this object (Node.js uses a HTTP object, which a dissimilar interface). This means to use Codebird.js, either the Codebird.js file requires refactoring to use Node's HTTP object, or an emulation of the XMLHttpRequest object would need be imported into Codebird.js. After overcoming some more small obstacles with the importation, the latter option was pursued, using ‘Node-XMLHttpRequest’.

2.4. The data collection algorithm

or: the world’s longest keep-alive

With all pressing problems overcome, the data collection algorithm could be fully developed, keeping in mind that the script would be long life and process a considerable amount of data. Because of the quantity of calls being made to the API and how expensive each is (with the rate limits), a few principles needed upholding:

- data must never be re-fetched: no API call should ever be repeated, no matter where the original data or call is in the program cycle.
- the script must never halt until recursion to the current depth has been finished (since execution can not be restarted to a given point) - when all collection activity has stopped.

Requests for data about a user's network (ie; a request for their friends or followers) is associated with a *depth* number which indicates how deep the recursion on the fetched data should continue (a depth fetch of 0 is disregarded).
To explain the main algorithm, we can explore the *transit* that a *request* for data can make through the program. A new (ie, the first) *request* for a network around an *id* to a certain *depth*, would have the *id* appended to a *network queue* and the *id* vs *depth* added to a *transit dictionary*. Then, when the *network queue* is periodically popped, the *request* is sent to Twitter asynchronously. When the data is returned from Twitter, it is pushed to the database and the *depth* is fetched from the *transit dictionary* (and is removed), and the algorithm recurses on the data based on the *depth.*
A new *request* is not just added straight to the *network queue* however, it goes through several conditions.

- Firstly, we check if the *requested id* is already in the *transit dictionary*. If it is, we don't want to add it to the *network queue*, since we know it must already be in there (or just left it), and since it's in the *transit dictionary*, the data has not yet been recursed upon. Therefore, if the *depth* at which we were intending to request is larger (deeper) than that in the *transit dictionary*, we merely update the *transit dictionary* item to our new *depth*, otherwise we discard our request.
  Therefore, no *requests* can be concurrently repeated, and an existing *request depth* can be increased.
- If not in the *transit dictionary*, we check the database, for the event we had a similar request in the past. If the *depth* indicated by the database equals or exceeds ours, we discard our *request*, since we already have all (and possibly, more) the data our *request* could fetch.
  If our *depth* is larger than that in the database, we need to *remake* the request, so we add it to the *network queue* (since *friends* and *followers* are managed separately, we can not recurse on the existing database contents, since we are not guaranteed it has yet arrived. However, because the root user around which we expand has the largest depth, it is rare that a new *request* exceeds the database depth, so this is not an expensive sacrifice).
- If neither in the database of *transit dictionary*, then we have a completely new request and so add it to the *network queue*.

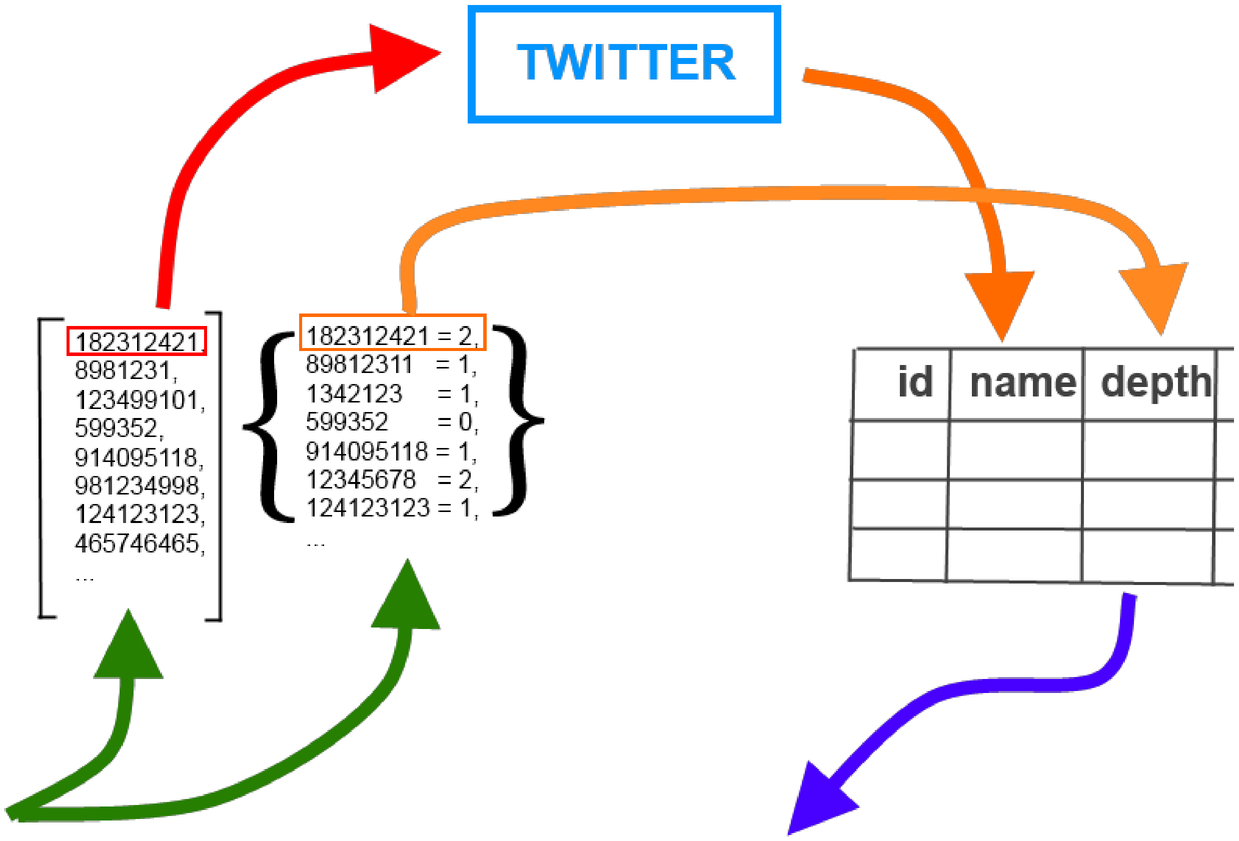


Figure 4: an overview of the parallel queue and dictionary model

With this model, no matter when a request is made, whether an existing request has happened in the past, is waiting in the *network queue* or is on its way to or back from Twitter, no requests will be repeated. This maintains our first script principle.
This model (of queue and dictionary) is independently repeated 3 times, for each of the base API methods, whose rate limits do not interfere. On top of this are some less complicated algorithms for asynchronous database fetching, timeline and favorites fetching and image collection (in response to the cross-origin policy in our main graphing program).

In accordance with the second principle, a timer periodically checks the activity of the entire program (whether *requests* or data or images are in transit or awaiting transit, etc) and if data collection has finished, stops the script, otherwise, execution continues indeterminably.

The program collects data and stores it in a MySQL database split across two tables, *users* (within which each user has 1 row with columns dedicated to all user data such as id, username, depth and a JSON encoding of the user's Twitter object) and *network* (where every row corresponds to ids in a follower -> followed relationship).

2.5. Maximising data integrity

or: please don't let the last 6 hours be wasted

Since this script requires long, unattended execution, several precautions were taken to maximise its robustness, including:

- all database interaction is done through a single, global database handle (since thousands of database interactions can happen concurrently, and thousands of connections leads quickly to a crash), which is periodically destroyed and recreated, to avoid timeout (destruction processes connections before ending).
- if a Twitter HTTP request times out or returns an error code, the *request* is added back to the *network queue* (the bottom), for all request types and images.
- all scripts declare a unicode encoding, as do all explicit and default database connections and all string database columns are UTF8 (to accommodate unusual char-sets in user profiles).
- a few database columns (in the *users* table) dedicated to flagging whether the user's friends or followers have yet reached the database, so that the integrity of the data may be checked, post-collection.

2.6. Accessing the data at runtime

or: how I learned to stop worrying and love the bomb

With all the data we're at liberty to wait for in the server (which is usually my own network to a depth of 2), the next step is receiving this data in the graphing program at runtime. This is easily performed through a few PHP scripts (multiple, since the actual paradigm to be implemented in the main script for fetching was undecided). These include:

- 'fetch-user-only.php', which accepts an 'id' or a 'screen_name' for a user and if they're in the database, returns the user object and friends/followers list for that user.
- 'fetch-user-neighbours.php' returns the user objects and friend/follower list for every friend and follower of a passed id or screen_name; the specified user itself is not returned. The order of this return (an array) is equivalent to the order of ids in the friend/follower lists (index consistency), and 'FOLLOWERS' then 'FRIENDS' are returned in that order. This script accepts two arguments; a start and end index for both of followers and friends so that batches of user objects can be fetched at a time.
- Finally, 'fetch-user-network.php' returns every user object somehow connected to the specified user id / screen_name in the database (unlimited depth).

With this entire system, we ultimately collect the Twitter data of our targeted *root user* from Twitter*,* offline and server-side (a few hours or days before we intend to proceed), and deliver it locally to the graphing software at runtime, whilst tip-toeing around the overzealous API rate limits.

3. Network visualisation

3.1. Representation

Author: Nicholas Smith

We chose to implement the visualisation in JavaScript, rendering in WebGL via three.js. Using JavaScript made interfacing with the server easy, since we could receive data from the server via regular HTTP requests. Using three.js for graphics saved time which would be otherwise spent away from the core goals of the project. It served our needs well, but for a larger project of this kind plain WebGL would most likely be a better choice to improve rendering flexibility and performance.

Graph layout is a complicated research topic, and for this project we simply needed a fast solution that worked reasonably well. We chose to implement a force-directed model to guide the layout of the graph.

In our implementation, we apply repulsive forces between each pair of nodes using the same principles as electromagnetic repulsion. This works to spread the graph out over a large area. To pull closely related parts of the graph together, we apply a force between every pair of *connected* nodes. This force models that of a spring, with a resting length and stiffness pre-defined by us. To discourage constant movement of nodes, we apply drag forces and deceleration forces such that a node will only begin to move when it experiences a large net force.

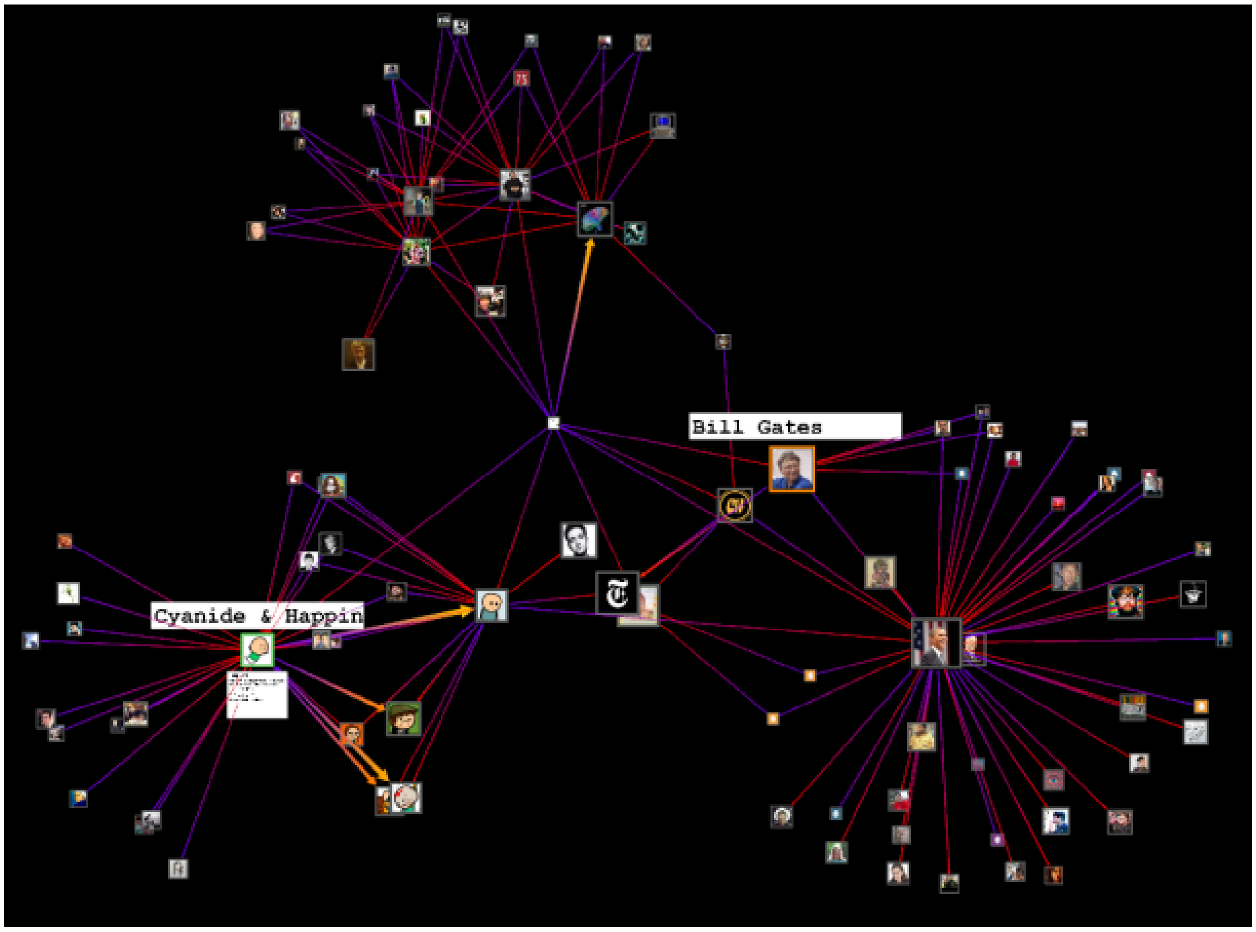
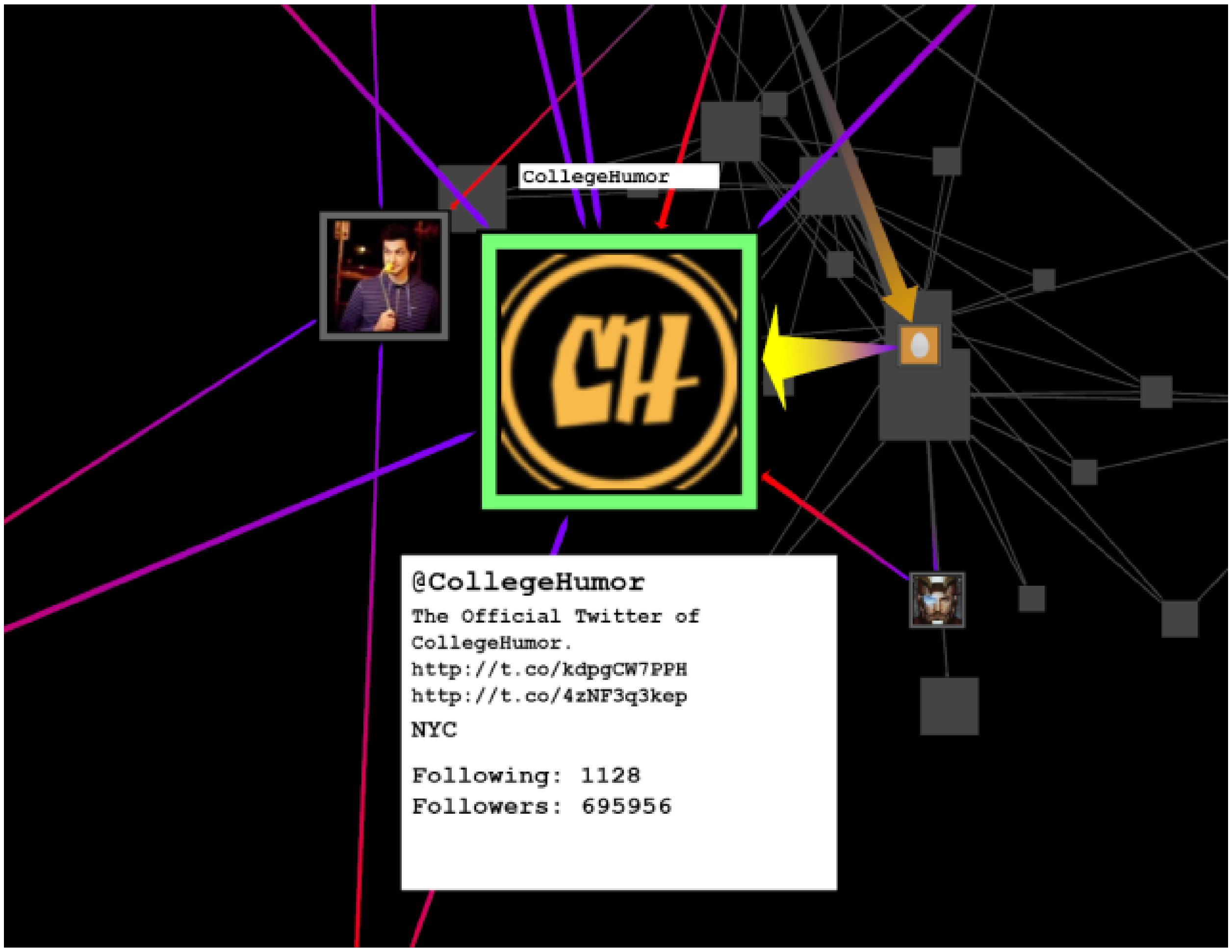


Figure 5: An example of a typical graph layout using a force-directed model

In the visualisation, a Twitter user is represented as a node in the graph. A node displays a user's profile picture and some basic information about them. We chose to make the size of a node proportional to the number of followers the user has, and as such it is easy to identify popular users as you explore the graph.

The relationships between users are shown as edges in the graph. An edge will be red/orange at the end connected to a user that is being followed, and purple otherwise. The red/orange ends will also have an arrow head. This allows the viewer to quickly identify the direction of relationships. Whether the edge is a shade of orange depends on the level of activity between the edge's users, as will be explained later.

When a user in the graph is selected, we display a summary of their profile in an information pane below their picture. This contains the user's username, profile description, stated location, the number of people they are following, and the number of people that are following them. Due to time constraints, we do not display a user's tweets, but this information is readily available behind the scenes.



@CollegeHumor
The Official Twitter of
CollegeHumor.
http://t.co/kdpgCW7PPH
http://t.co/4zNF3q3kep
NYC
Following: 1128
Followers: 695956

Figure 6: A user's profile summary

3.2. Leap integration

Author: Nicholas Smith

Integrating the Leap Motion with the visualisation was relatively simple. We managed to implement a set of basic gestures to allow full exploration of the graph using the device.

A dot appears on the screen at the position we detect you to be pointing at. By pointing at a node, you can display their name. If you wish to select a node you are pointing at, you can make a grabbing gesture with your other hand. The user will then move to the centre of the screen and you will be able to read their profile summary.

You can rotate the graph around the selected user by placing one open hand in the Leap's field of view with the palm facing downwards, and then panning the hand left, right, up or down.

Panning your hand towards or away from the screen will allow you to zoom in and out.

When you have a user selected, you can view their neighbours by first placing your hands close together and then quickly moving them outwards and apart. To hide their neighbours, you quickly bring your hands together as if you were performing a clap.

We discovered that such maneuvers need to be done fairly precisely in order to work well with the Leap Motion. Sometimes gestures need to be repeated multiple times. We found out what works largely through trial and error, and by analysing existing Twitter applications. Due to the limitations of what positions and actions the Leap can recognise, we discovered several rules which we had to abide by in order to get decent results:

- If a gesture relies on the recognition of multiple fingers of a hand, it must make do with the palm of the hand facing down or up and with fingers spread apart.
- If a gesture requires two hands, these hands can’t come into contact with one another or obscure one another when viewed from below.
- The Leap can’t be used effectively outdoors or next to a window due to interference from infrared radiation from the Sun.

3.3. Relationship analysis

Author: Keren Burshtein

3.3.1. What it is

At its current state for our project, the graph gives only a single analysis function that is calculated behind the scenes and presented neatly graphically, on top of the obvious relationships and total counts of followers. This function, which we have failed to properly name, determines how actively a user, say user2, is talking about one of the users its followees, say user1 (i.e. user2 is following user1). This only looks at any activities done recently, or to be more specific, activities within the scope of our database’s updated contents. The measure of “influence” of user1 on user2 is given as a number between 0 and 1 that is assigned to that relationship edge. This influence number is then scaled logarithmically and used as the size and color indication for the arrow representing the edge:

- The color would be red and in its smallest form when user2 has been silent about user1 recently.
- The color would be yellow and in its largest form when 50% or more of user2’s recent activity has been directed at user1.
- The color would take shades of orange and different sizes for the rest of the cases, as a function of the calculated influence figure.

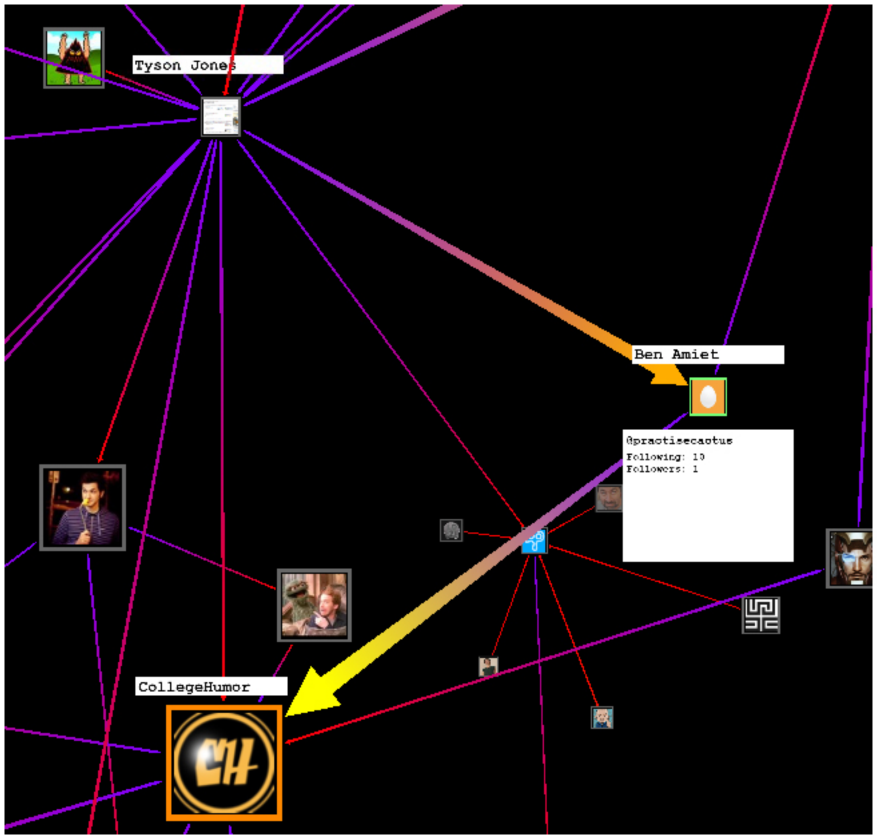


Figure 7: Screenshot of the graph showing that Ben Amiet is currently talking more about CollegeHumor than Tyson Jones about Ben Amiet

3.3.2. How it is done

The assigned number is calculated by the following steps:

1. Counting the total amount of user2's:

- Retweets of user1's tweets
- Replies to user1's tweets
- Mentions of user1 in user2's tweets

This is done by going through all of user2's tweets available in the database and for each checking

- if it is `in_reply_to_screen_name` of user1 (which then indicates it is a reply) or
- if `tweetsUser2List.entities.user_mentions` contains user1's screen_name as part of the usernames mentions in that particular tweet, which then indicates this tweet is either
    - a retweet of user1's tweet,
    - a mention of user1 (@user1) or
    - a mentions of user1 in a retweet of user1's tweet.

If the check returns true, we increment the count.

This is not very accurate since Twitter does not give a straightforward indication of how much a user talked about a specific user in a specific time, and for that reason this information must be recursively calculated by the program. An inaccurate count can happen because Twitter does not guarantee the inclusion of user1's details in all of user2's retweets, since users are not obliged to have user1 featured as a "mention" in that retweet (@user1). However, this is a close enough indication to the real number and for bigger data, which this program is initially intended for, will do just fine. Avoiding the margin error is time consuming and requires more data retrieval and storage in the database. To do so we would need to call GET statuses/retweets_of_me for user1 (which must be authenticated) and for each of the returned tweets then call GET statuses/retweeters/ids. We would loop through each of the tweet's lists and search for user2's screen_name.

2. Counting the total amount of user2's favorites of user1's tweets. This is done by going through all of user2's favorites available in the database and for each checking if the tweet belongs to user1.

3. Calculating the raw activity measure that would then be scaled. Percentages are assigned to each of the total numbers counted above:

```
activityPercentage = tweetPercentage*0.75+favoritePercentage*0.25;
```

where `tweetPercentage` and `favoritePercentage` is how much the counted number of mentions and retweets or favorites constitutes of the total number of user2's tweets or favorites, respectively.

These steps would be repeated for every new edge created, where there is a distinction between a follower edge and a followee edge.

4. Conclusion and further work

Many aspects of this project could be improved upon if we invested more time in its development. The following is a list of suggestions that anyone who is interested in the expansion of our project can draw ideas from:

1. The interface for initiating data collection is largely underdeveloped and is near unusable to a typical user. An unfinished system of monitoring collection status, pausing or cancelling collection or predicting the length of the collection could be further developed and the entire algorithm optimised for complete robustness through error reporting and state preservation.
2. The visualisation is unable to display many more than 100 nodes at once due the quadratic growth rate of calculations in the physics simulation with respect to the number of visible nodes. This could be improved with the use of space-partitioning data structures such that only nodes in close proximity apply repulsive forces to one another.
3. We don't currently have a method for determining which neighbours of a user are worth showing. At the moment, we simply show at most the first 40 neighbours (20 followers, 20 following) of every user under some arbitrary ordering. With more time, we could explore methods of analysing user importance and intelligently selecting who should be shown first.
4. The current algorithm for showing and hiding users has flaws that mean when a circle of connected users each request for their neighbours to be visible, that circle is unable to be

hidden again. A much more sophisticated algorithm is necessary to resolve this issue. There are most likely published solutions to this problem.

5. The activity measure calculation could have an improved precision, since Twitter releases more information about authenticated (logged in) users than non-authenticated users. Therefore an addition can be made where:
    - if user2 is an authenticated user:
      ```
      activityPercentage = tweetPercentage*0.6375+favoritePercentage*0.2125;
      ```
      The last 0.15 is added only if user2 has the SMS notifications option available for user1 tweets.
    - If user2 is not an authenticated user (which is the majority of cases), the percentage is scaled to the percentages in the original code.

In conclusion, much more work could be done with the Leap Motion in this project. With our current implementation, we would not say that the device necessarily improves the user experience. However, we have only scratched the surface of Leap interaction, and are not in a position to make conclusions about its potential to improve the user experience in general.

5. Appendix:
Additional ideas for network exploration with graphs

Other analytical functions can be integrated in the graph's implementation. As discussed in **section 1.4.**, during the development of the project, many ideas were brainstormed about how the graph's analytical features could have more depth and interest. We have divided these ideas into possible "modes" the graph could be in when the application is run. And as discussed in **section 3.3.**, we have chosen to implement the *"influence/activity mode"* for our project. This mode was the only feasible mode to implement in the given timeframe, hence its selection.
The following is a general overview of each of the discussed modes:

I. <u>"location mode"</u>

This mode requires the graph or, rather, the graphs to be visually fitted upon a globe. The idea was to have the globe first divided by continents, allowing the visual comparison between the most interesting users within and between the continents. Only those selected users will be displayed in possibly disconnected graphs (unless they happen to be all connected), while giving some kind of general representation to the numerical facts about each continent (either by a special node or maybe textually on the continents plane on the globe). Interesting users could be:

- The authenticated (logged in) user's private network
- The users with the most followers in that continent
- The users that are most followed in that continent
- The users that are most followed by strictly users from that continent
- The users with the most number of tweets in that continent

The "location" factor could then be refocused on a smaller section of the earth by selecting continents and then countries within them, and so on. The above idea would then be recursively subdivided. An option would be to keep the previous interesting users and add new ones or totally remove the globe visually from the screen and only have the focused location on display.

II. <u>"theme exploration – in a community mode"</u>

This mode focuses on the tweets' contents rather than the connections between users, hence introducing a whole accept to the study of social relations within Twitter. This would have the same visual aspects as our current graph. The addition would be the power to see analytical information displayed in a clear matter about topics, or "themes", of tweets within the network. This would require adding a way to type in the desired theme and see by some kind of visual indication (color shading or saturation, sizing, distance, etc) which clusters or specific users have been recently talking about the selected theme. This mode could also have a default option, where before a theme is selected, the analysis will be done with respect to the most actively talked about themes at that time.

III. <u>"theme exploration – in a location mode"</u>

This, as its name suggests, is the idea of combining the two modes introduced above into one. In other words, we would have a graph or graphs displayed on a globe and the interesting users would then change to users that are talking about a selected theme rather than users with a lot of connections. This mode could also have a default option, just as suggested above.

IV. <u>"tweet exploration – in a community mode"</u>

In this mode we would have the same graph implementation as of our current implementation. A possibility will be given to select a tweet out of the users' tweets (which would be displayed in a reverse chronological list), type in a desired tweet or find it on Twitter and see the lifespan of this tweet. This could be visually represented by a glowing light that goes from the creator, through the retweeters, up to its last destination at the moment. This is a particularly interesting functionality, since decent discovered can be made when studying the flow of tweets.

V. *<u>"tweet exploration – in a location mode"</u>*

This would be the same idea as mentioned in the mode above and as mentioned in the first mode. In this case the interesting users would be the ones that were part of the tweet's lifecycle. The two "tweet exploration modes" could be also accessible through the "theme exploration modes" by selecting specific tweets from the theme.

VI. *<u>"clusters mode"</u>*

This was the other mode that was nominated to be the mode for our current graph implementation, but was then rejected since is requires displaying more nodes than our graph can handle. For this idea, we would need to display networks within networks, i.e. the formation of clusters. This can be done by a variety of algorithms that can be found on the internet using the degree of centrality of users (and some that cannot be found, like Google's "page rank" algorithm).