C++ Project: Twittle

A Twitter Application by Loren Segal 5442680

Concordia University
COMP 446 / F08
Dr. Peter Grogono

Table of Contents

1 Introduction

> *"Twitter (http://www.twitter.com) is a new web application that allows users of the site to keep a timeline of their daily, hourly or even minute-by-minute "status" describing what they're doing at that very moment. Friends can use this website to follow their friends' statuses or even have friends follow them. Many well known people and organizations such as NASA and the US Government, as well as many political candidates like Stephen Harper have started to use Twitter as a means of having a more direct communication with their users or fans. This mode of communication has been favoured by certain people and organizations over blogs because of the shorter, to the point messages expected to be written and read by users. Rather than lengthy blog articles which require time and effort to create, Twitter's "tweets" (a message) are limited to 140 characters and are expected to be one or two simple sentences."* – Project Proposal

The goal of this project was to implement a client-side Twitter application to be run on a desktop machine. This goal was actually came from a need for a better Twitter client to suit my personal needs, as no existing client supporting some of the functionality ultimately implemented by Twittle. Most of this extra functionality was in the user interface. Since most existing Twitter programs are inherently cross-platform by design, the user interface does not always translate properly to specific platforms, and this posed a problem for me. My ultimate goal was to end up with a Twitter desktop client that retained the UI features of specific platforms (Windows, OSX). This is unlike existing applications, which attempt to force a UI that is more familiar to the web and less familiar to desktop applications. For this very reason, wxWidgets was chosen as the windowing toolkit for the project because of its work in properly translating UI differences across multiple platforms. This translation will be described in more detail later.

In the end, all features listed in the proposal were implemented except for three (direct messaging, profile editing and drop.io support). However, in their place, a multitude of new features we implemented on the usability end (window transparency, tray icon and tray notifications). A summary of features is as follows (items in *italic* were added in addition to features described in proposal):

- Account Login *(and ability to remember account information)*.
- Following RSS/XML feeds of friend accounts, displayed in the GUI with a profile image and status update beside it.
- Ability to publish status updates.
- Organizing feeds by replies, user accounts or all combined.
- Saving/archiving program options/data locally for next session.
- Ability to shorten URLs in status messages via http://is.gd/
- Ability to upload and link photos in status messages via http://twitpic.com
- *Tray icon support and tray notification bubbles on status updates.*
- *Window transparency.*

2 Design Overview of Twittle

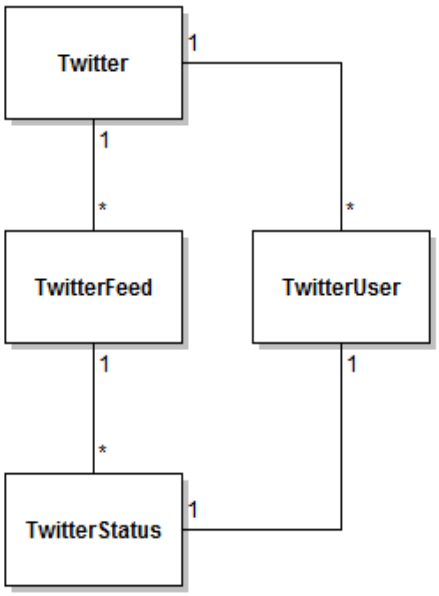
The main design choices in the making of Twittle can be attributed to architecture and the choice of XML serialization. However, it is also important to discuss the implementation details of some of these decisions (how and when generics/templating was used, algorithms, threading), which will be done at the end of the document.

2.1 Model View Architecture

Twittle was designed using a simple form of layered architecture, more specifically, a “Model View” architecture (MVC without the Controller portion). In retrospect, the design would have been more robust if the controller was introduced, which would have allowed abstraction from the wxWidgets UI framework; however for a project this small it would have been more of an overhead than a benefit. Therefore, in this system, the model layer is described by the data used to store Twitter information, while the view layer is mostly the implementation of the UI using wxWidgets components.

2.1.1 The Model Layer

The model layer was basically comprised of the Twitter API and related data structures, namely the TwitterUser, TwitterStatus and TwitterFeed classes. The following domain model describes the relationship of the data structures used to store the data in the application.



The Twitter class is the main controlling data structure implementing the API[1]. It controls a list of feeds which are much like an RSS feed in terms of the data they represent. The feeds themselves hold a set of status items with an associated user, which is stored in a centralized user registry in the main Twitter class. The centralized user registry was done to avoid duplicate user objects and assure that each user reference was synchronized with the latest user data from the server. Serialization (described in detail later) begins from the feed and therefore serialization never occurs for the Twitter object itself. This is done so that feeds can be loaded individually when needed, and not all at once.

One of the major design decisions of the model layer is the use of an observer pattern to notify clients of updates on specific feeds. This is done for efficiency, because now rather than having to poll the object for updates, the clients are immediately notified of any updates to feeds. This observer pattern is implemented via the TwitterUpdateListener interface implemented by the view layer in Twittle, but ideally it could be implemented by any set of components (a component that could e-mail you when a specific feed was updated, for instance).

Except for the TwitterUser class which relies on the application to know where to store profile images on the local disk, this model implementation is completely independent of the view layer and can be reused in any application. With minor refactoring, the connection between the user and application class could easily be cut.

Note also that although there are already existing C++ Twitter API libraries, it made more sense for the purposes of this academic project to implement the API independently. Otherwise the project would have simply been the connecting of multiple existing libraries rather than building something new. And while this implementation of the API is not as complete as others, it serves the purposes of the project perfectly.

2.1.2 The View Layer

The most important design choice in Twittle was the choice of wxWidgets as the UI (or view) layer. While the decision for the data structures was mostly natural, the choice of wxWidgets was made on the basis of cross-platform support. The Wx library is known for its support of multiple platforms but more importantly for its ability to bridge the UI differences of each platform automatically for the developer without having to manually account for such differences. For instance, under Windows, the natural UI expectation is that each application has a “File” menu with an “Exit” menu item at the bottom, yet under OS X, this menu item is titled “Quit” and is placed in a special “application menu” that is titled with the application’s name. Dealing with such issues is necessary to make a seamless user interface experience across all platforms, but doing so manually would likely end up being a lot of work. Fortunately, wxWidgets is able to handle this specific situation automatically for you if you use the proper code conventions. Another example is how the OK/Cancel buttons should be displayed in dialogs. Windows places the OK button to the left of Cancel in standard dialogs, yet this may differ on various platforms (some platforms such as PalmOS don’t even use such buttons as they have dedicated

[1] Twitter API: http://apiwiki.twitter.com/REST+API+Documentation

buttons for such behaviour). wxWidgets is able to auto-populate such dialog buttons for you. These examples, in addition to its general ease of use, were reasons that Wx was decided on as the UI framework.

Implementation of the wxWidgets UI was done using one frame (top level window) and subsequent panels for logging in and viewing Twitter feeds. Each frame was then swapped into the main window. This was done to avoid the overhead of creating many windows and to keep the window frame code consistent.

The feed panel was implemented using the wxHtmlListBox class which uses a "virtual" list box. This abstract class is overridden and the subclass implements the "OnGetItem" method to return the HTML contents for a specific list item. This meant, by design, that all feeds could share the same list box, since "replacing" the list box simply meant returning the contents of another feed in these get item calls. Filtering (described in the implementation details) was done this way.

Wx uses an event loop to pass around events to windows rather than overriding superclass functions (unlike Java Swing). Therefore, most of the UI classes depend heavily on event driven design and use event handlers to receive window messages. Such messages are similar to the Windows API, like the painting of a window of clicking of mouse buttons. This event loop runs in a single thread and is not thread-safe, and so in some cases (where threading was used), custom events needed to be posted to the event queue to have the main thread handle certain events.

2.2 Notable Implementation Details

2.2.1 Portability and Cross-platform Support

One important concern was that of cross-platform support. I personally use many platforms (OS X, Windows for the most part) and needed to be able to develop on either machine. This of course was one reason why wxWidgets was chosen over using the native UI technologies of either platform. However, there are portability issues that extend beyond the use of a GUI (POSIX support, C++ compiler, etc.). To deal with this, the Twittle project was tested with the VS.NET compiler, MingW's GCC (g++) as well as the GCC implementation on OS X using an Eclipse CDT project. The project can be built on both Windows and OS X, though it is only fully supported under Windows (portability is not always as easy as using standardized tools). There are some minor issues under OS X (image uploading and preview is not stable) but the application is able to perform the main task of sending and receiving status updates.

Also note that the tray notification feature is currently implemented under Win32 only. Apparently wxWidgets 2.9 will support the tray notification functionality, but 2.9 is not yet officially released.

2.2.2 STL Usage Versus wxWidgets Classes

wxWidgets is designed as more than a cross-platform abstraction of the GUI layer. It also attempts to abstract certain other potentially platform specific components such as the file system, threads and

sockets. In most cases, Wx provides alternative classes to every STL class. This brought about the choice of using Wx classes versus the STL. The decision was made to use `wxString` instead of std::string for easier integration into the UI (as well as some extra helper functions it provides) and certain file systems were also used instead of fstream for the same reason. However, for all collections (vector, map, multimap), the STL was used over Wx's collection classes. This can be seen in the Twitter and TwitterFeed classes, which rely heavily on maps and vectors from the STL. This one done for two reasons: firstly, to experiment more with some basic C++ features, and secondly because the implementations were familiar and simpler than learning new collection types.

2.2.3 Serialization to XML (Using Generics)

Serialization was done in two ways. For serialization of user settings, a simple wxXmlDocument (XML document object) was used to store contents and serialize directly to XML. The XML backend does not exist for feed items, however, and so serialization to XML for these items needs to go through a serialization transformation step (into XML data structures) before it can be saved. To do this, a Serializer template class was created which allows serialization to any stream type (file system path or some arbitrary stream object). To use this template class it simply must be specialized for the stream and object and then any class can make use of the standardized call to `Serializer<Stream, Object>::Write` or Read. For example, the following could be done for the TwitterUser class (by itself):

```
template<>
class Serializer<wxString, TwitterUser>
{
public:
	static void Write(const wxString& path, const TwitterUser& obj)
	{
		// implement write
	}

	static TwitterUser *Read(const wxString& path)
	{
		// implement read
	}
};
```

This would allow anyone to serialize a TwitterUser object to a file denoted by a wxString path name as:

```
Serializer<wxString, TwitterUser>::Write(_T("C:\\file.txt"), aUserObject);
```

Note that in Twittle this is only done for the TwittleFeed object, though the use of templating allows this design feature to be extended to other data structures. Initially, the Settings object was also going to use this form of serialization; however it ended up being easier to store the setting contents directly in an XML document for the specific object type.

2.2.4 Custom Iterator Class for Feed Filtering

One potentially interesting design item in the UI layer is the filtering of feeds by message type. As mentioned above, the feed panel shared a list box among feeds so that only one list box would need to

be placed in the application. Therefore, filtering messages could easily be implemented by simply iterating with special filtering rules over the list when asked to retrieve an item at a specific index. This would avoid a costly duplication of list contents. To implement this, a class called FilteredIterator was created which acts like a standard STL iterator object, where the unary "++" operator would return the next item in the list that matches the specific filtering rules.

2.2.5 Multi-threading and ThreadCallback

Twittle uses blocking sockets to perform its HTTP operations on the Twitter API servers. Because of this, it is hard to implement a responsive GUI without performing any threading. Therefore, all HTTP operations (including image uploading functionality and URL shortening) make use of threaded calls to keep the UI responsive. Implementing this originally required a lot of overhead because every thread would need its own subclass of the wxThread class.

To simplify the implementation of threading, a ThreadCallback class was created to emulate a "callback" mechanism from a newly created thread. This allows the caller to create a member function on a class as it would normally be written and then create a callback to that thread, essentially calling it in "threaded mode". The code would simply be:

```
class MyClass {
public:
	void SomeMethod() {
		new ThreadCallback<MyClass>(this, &MyClass::ThreadedCallback);
	}

	void ThreadedCallback() {
		// running in a thread
	}
};
```

This would allow SomeMethod() to be called from the main thread, which would call ThreadedCallback() in a separate thread (without blocking). Note that the threads must be allocated on the heap and therefore might leak memory, so ideally they should be pooled and deleted at some point in the future.

The problem with using this code in the wxWidgets library is that wxWidgets runs in a single thread and is not thread safe, so these threaded methods can never talk to the main UI thread. This was an interesting problem to solve, but was ultimately fixed by posting window events from the threaded method to be handled by the main event loop thread.

2.2.6 Inheritance (in the UI)

While wxWidgets is generally event-driven and prefers the handling of events to overriding handler methods, there are certain cases where pure virtual classes had to be implemented or where behaviour needed to be modified. While there is in general very little inheritance in the application (see the doxygen class hierarchy for the complete list), it should be noted that some inheritance-versus-event-handling was by design and not familiarity with inheritance in general. Basically, most problems in

wxWidgets could be solved far easier with event handlers than subclassing, though in a few cases the behaviour of Wx needed changing and so inheritance was only employed for this reason.

3 Individual Experience Report

This section describes my personal experience with the project; things I've learned, and a general opinion of C++ after creating a relatively sizable application on my own.

3.1 Using the wxWidgets Library

One of the first challenges faced was the complexity of compiling and building an application against wxWidgets. This challenge was compounded with the fact that, as mentioned above, Wx needed to be built not once, but three times: once with VS.NET, once with MingW (there is a separate GCC build of wxWidgets) and once more for OS X. One problem with Wx is that it has no binary releases, which would make developers lives far easier than distributing by source. In fact, most of the problems occurred not with using the built wxWidgets libraries, but in attempting to build them on various systems. Surprisingly, Windows was the easiest system to build on although it took a little tinkering with options. On OS X, however, the built libraries caused random glitches and bus errors (the SEGFAULT for the Mac). Fortunately, the OS X Leopard Developer CD (that comes with every machine) ships with a version of wxWidgets 2.8, which was ultimately used to compile a working system. As described in previous sections, there are issues with the OS X build, and these might be due to the version of wxWidgets supplied by the developer CD (the version used on Windows was 2.8.9).

After compilation problems were solved, the use of the wxWidgets classes was more or less seamless. A few issues with the wxHTTP class were noted, namely, it did not properly implement certain methods as one would expect, or in some cases, completely omitted certain important functionality (URL encoding, multipart form data). Also, the base socket code caused problems much of the time, and still does under OS X (again, possibly due to the older version).

Use of actual Window classes was straight forward once the setting up of event handlers was understood. wxWidgets would definitely be considered for future cross-platform UI applications in this regard.

3.2 Multi-platform Support

While wxWidgets is meant to ease the pain of multi-platform development, it cannot fully solve all problems. There were many minor C++ differences between the VS.NET and GCC. For instance, the following code is legal in Visual Studio but not in GCC:

```
wxPostEvent(this, wxCommandEvent());
```

Note that the method signature of wxPostEvent is:

```
inline void wxPostEvent(wxEvtHandler *dest, wxEvent& event);
```

The problem is with the construction of wxCommandEvent as a parameter to the method call. Visual Studio will implicitly cast the object if anonymously constructed in a method parameter, but GCC will not. No form of casting seemed to work. The solution is to declare the object before the method call but the way Visual Studio handles the implicit casting seems much more intuitive to me than GCC's behaviour, though I'm not sure which one is technically correct.

3.3 Threading Issues

Multi-threading is by no means an easy task. Multi-threading code you don't own or didn't write is even more difficult. While most of the multi-threaded code in Twittle was confined to the data structures written by me, a few multithreading issues spilled over into the UI layer (because data was being shared with the feed items which ran in a thread). Some of the threading issues in the UI were fixed by returning object copies rather than references (see *FeedPanel::GetStatusItem*). Cleaning up blocked sockets running in a thread was also not an easy task, and making the application exit cleanly took a bit of time. While there are no remaining noticeable threading issues under Windows, I cannot guarantee the thread-safety of the code I wrote. More specifically, I know of a few potential race conditions which *could* cause issues, though the scenario is rare. They have not yet been addressed because I have no way of sufficiently testing that they are indeed issues.

3.4 Testing

I originally started this project wanting to have unit tests for all of my core functionality which would ensure stability of code. Frankly, this is the only way I could imagine ever maintaining a cross-platform program without changes causing things to go terribly wrong on another platform. Unfortunately, it isn't that easy to do TDD in C++ (in my opinion). The overhead required to manage tests is much more than other languages (Java, Ruby, and Python) and the tool support is rather weak. I found that VS.NET2008 supports Eclipse-like integration with a testing framework, but the framework is a Microsoft-only technology and therefore little use to a multi-platform application.

Ironically, as I now reflect on how I could have tested the code, I have a few more ideas which I could have tried out, but this is likely due to my newfound experience with the language.

3.5 Overall Experience and Thoughts

Ultimately, the motivation was to build something that I would use. Because I truly believe that this project is something usable to me I would say that this project was a success. If other people feel the same way, then that would only further drive the success of the project home. In fact, I currently have one friend beta testing the program and they have found it quite useful. Overall, I am very happy how the project turned out. My only issue is regarding some of the limitations of the wxWidgets UI controls, namely the list box control used to store the feeds; it is not possible to implement such a control where the user can highlight portions of text to be copied. Instead, the only way to copy text would have to be to copy the entire list item which can be done by right clicking the item and clicking "Copy as Text" or

hitting Ctrl+C, though this is not ideal. With more time, the UI can be reworked to implement something more usable.

I've definitely learned a lot about C++. Through the course (and first-hand experience) I've come to realize that I've had many misconceptions about the language. At the beginning of the course I was of the mindset that there was generally no reason to use C++ over C and that iterators were overly complex and not needed. While I still believe templating in the STL can be a little cumbersome, I definitely now see the power of using STL collection classes and iterators. Thinking of all the projects I've done in C, I can see how std::string and std::map could have been greatly beneficial to my programming efficiency. I, however, am still not a big fan of using header files in general (see assignment 4 for a more in depth discussion of this), so C or C++ still will never be on the top of my list of languages, but I believe given the choice between C and C++ in the future, I would have to choose C++.

On a side note, it would have been nice if the course covered or put more of an emphasis on some testing solutions which are used in real world C++ programs. I have yet to see a well tested C++ system and I'm wondering if they do unit testing at all.

4 Twittle User Manual

4.1 Introduction

Twittle is an application to let you use your Twitter account from your desktop. Using *Twittle*, you can follow what your friends are doing and even update them on your latest status yourself, all without ever opening your browser. *Twittle* also allows you to share images and links by integrating with services like TwitPic and is.gd.

4.2 Installing Twittle

Important note for Windows users: *Please make sure you have the Microsoft Visual C++ 2005 SP1 Redistributable Package before installing Twittle. It's a small (2.6mb) free download from Microsoft.*
http://www.microsoft.com/downloads/details.aspx?FamilyID=200B2FD9-AE1A-4A14-984D-389C36F85647

Twittle is easily installed because it is only one file. Simply store the exe file **twittle.exe** somewhere safe and double click the icon to run it. It's as simple as that!

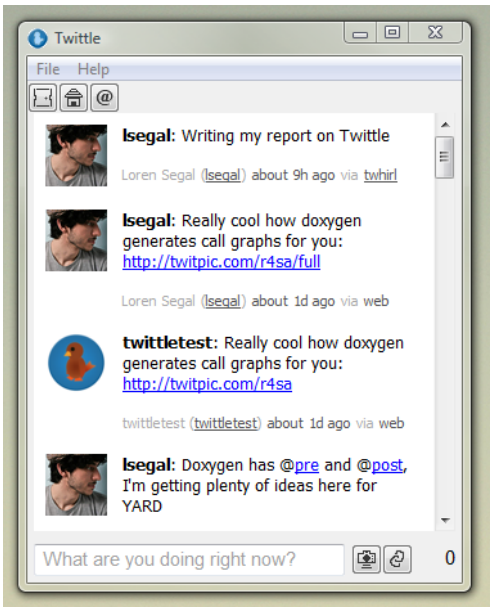
Note: *User account settings are stored on the local disk in your Application Data folder (under Windows XP), AppData (Windows Vista), Application Support (OS X) or '~/.twittle' (UNIX/Linux). See the uninstalling section for more information.*

4.3 Running Twittle for the First Time

Twittle needs to know your Twitter account settings **[*]** before going any further. If you don't have a Twitter account yet, you can visit http://www.twitter.com to register for one. You might also want to add a few people to follow before moving on. That is, of course, the whole point of Twitter! You can find people to add from the Twitter website or simply ask your friends to join.

Once you have setup your account, simply enter the account details in the box as shown below:

If everything worked out right, you will be greeted with your friends' status updates. It should look a little like this:



Now you can update your friends on what you're doing. Simply type a message in the textbox at the bottom of the screen and hit return. Once it shows up with the other statuses, you can be sure your friends who are following you will soon get the message.

You may have noticed on the right there is a number that keeps counting up when you type. This is because Twitter has a length limit on their messages of 140 characters. This is to keep status updates short n' sweet! If the number turns red it means your status will be cut off and your friends might not see all of what you wrote! Try rewriting your update to be more concise. If you're inserting a URL for your friends, consider shortening it with the *Shorten URL* button (see below in *The User Interface* section).

**** Note**: Twittle will never attempt to store your username, password, or any of your local data anywhere other than on your local machine. This notice does not apply to the use of the TwitPic service for image uploading (though they probably don't store your data either). Please see their terms of service for a similar policy.*

4.4 The User Interface

You might notice a few buttons on the top and a few more on the bottom. Let's go over their functionality:

 *View all public tweets*	This button allows you to see what everyone in the Twitter community is saying! Be careful though, things happen very fast in the outside world!
 *View your friends' tweets*	This button allows you to follow your friends' tweets (status updates) only. This is probably what you want to be looking at most of the time.
 *View replies directed to you*	Sometimes your friends will reply to you in a tweet. A "reply" is known in the Twitter world when someone types "*@yourusername*" before the message. You can click this button to conveniently find all the messages where people wrote directly to you.
 *Upload an image*	This button uploads an image on your local machine to http://twitpic.com and puts the URL where you can find that image in your text box so you can quickly and easily share images with your friends!
 *Shorten a URL*	This button takes a URL and shortens it with http://is.gd. Your new URL will look like http://is.gd/something, and it will be inserted into your text box. You can use this when you have a really long URL that you want to share with the world.

You can also do things with your mouse:

Right click on a status item	This will open a popup menu allowing you to copy the item as text or even as HTML.
Drag an image onto the window	This will automatically bring up the image preview so that you can upload the image without clicking any buttons at all.

Now that you know what all the buttons do, let's look at some of the key combinations you can use to make life faster:

Ctrl+L This is equivalent to hitting the *Shorten URL* button. If you already have text selected in your text box it will automatically use this as the URL to shorten.

Ctrl+C If you need to copy a tweet from a friend, you can highlight a status item and hit Ctrl+C, it will be as if you right clicked and pressed "Copy as Text"

Ctrl+O* Brings up the options dialog (we'll see this soon).

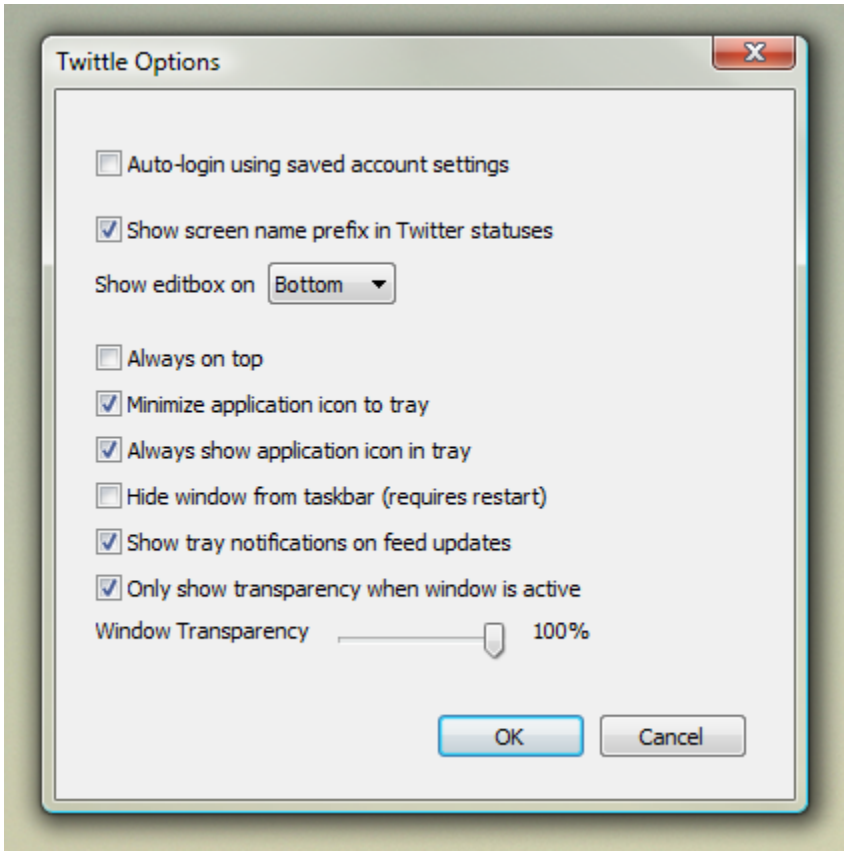
Ctrl+Q* Quits Twittle. Bye Bye!

*** **Note**: Under Mac OS X, the options dialog is located at Command+, and quitting is done with Command+Q.*

That summarizes the user interface. Now let's look at some of the options.

4.5 Twittle Options

To access this menu, click *File -> Options* (*Ctrl+O*), or *Twittle -> Settings* (*Command+,*). You will see this:



Let's go over some of the options:

Auto-login using saved account settings:

> This setting automatically logs you in with your last saved settings (if you clicked "Remember Me" in the login window).

Show screen name prefix in Twitter statuses:

> Usually you will see updates as "**someusername:** I went to the store". If you'd rather not see this prefix, you can disable it with this checkbox.

Show editbox on [Top/Bottom]:

> You might want to customize your user experience by moving the text box on top if you prefer it there.

Always on top:

Shows the application window on top of all your other windows.

Minimize application icon to tray:

When you minimize Twittle you can have it hide from the taskbar and only show in your system tray.

Always show application icon in tray:

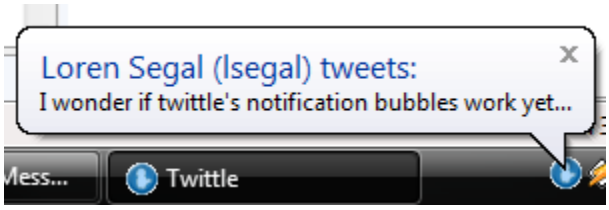
Make sure the application icon is always showing in your system tray.

Hide window from taskbar:

If you are always showing the application in your system tray, you may not want to ever want to see the window in the taskbar. You can remove it with this checkbox. Make sure you are showing the application in your system tray though!

Show tray notifications on feed updates:

Tray notifications are these little things:



If you don't want to see them, you can disable them here.

Only show transparency when window is active:

You can decide to make the window more transparent when inactive. This is useful if you have the window set to "Always on top" and you have other useful applications running under it.

Window transparency:

Makes the window slightly transparent. Try it out!

4.6 Uninstalling Twittle

To uninstall Twittle simply remove the executable from your machine. You may also want to remove any saved settings that twitter stored on your machine. These can be found in the following locations (depending on your operating system):

Windows XP	C:\Documents and Settings\YOURUSERNAME\Application Data\Twittle\
Windows Vista	C:\Users\YOURUSERNAME\AppData\Roaming\Twittle\
Mac OS X	**«** Application Support » in your Home directory
Linux/UNIX	~/.twittle/

Note: *YOURUSERNAME refers to the logged in username in Windows XP/Vista, not your Twitter username.*

4.7 Acknowledgements & Author

Twittle was developed by Loren Segal in 2008 for a C++ course. You can find out more about the author at http://kthx.net. Thanks to Myriam Malca for beta testing the initial release.

4.8 Licensing Information

Twittle 0.5 beta can be downloaded freely from http://kthx.net/twittle.

Twittle will never store your private data on any of its servers.

Twittle may not be sold, resold, rented, leased, or lent without written authorization from the author.

Twittle is provided AS IS without warranties of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. In no event shall the author be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages, even if the author has been advised of the possibility of such damages.