ECE297 Milestone 2
Drawing Maps

"The real voyage of discovery consists not
in seeking new landscapes, but in having new eyes."

–Marcel Proust

Assigned on Wednesday, Jan. 31

Due on Tuesday Feb 26 at 7 pm

In-lab demo = 13/15

Code style and project management = 2/15

Autotester = -1 penalty if errors

Total marks = 15/100

1 Objectives

In the previous milestone you learned how to query libstreetsdatabase to get useful information about a map, and you started building your project in libstreetmap by implementing a number of functions. In this milestone you will use a graphics library called EzGL to visualize the map, and you will add functions to find elements in the map and display detailed information about them.

After completing this milestone you should be able to:

1 | Use the EZGL graphics library.

2 | Visualize a graph of a map using computer graphics.

3 | Respond to basic user queries about the map.

2 Problem Statement

In this milestone you will extend your project to implement the function in "m2.h" by adding one or more files to implement the graphics functions for your application. You will try to implement all the features that you proposed in your **"Graphics Proposal"** document, starting with the most important ones in case you run out of time. You can adjust the plan you laid out in your graphics proposal without penalty, however, as in this course we recommend iterative development which frequently results in plan changes.

3 Specification

There are several **required features** that your program must support.

- Your program must be able to load and visualize any map *without recompilation* (i.e. you must get the map name from user input in some way).
- You must be able to visualize streets, intersections, points of interest and features (lakes, buildings, etc.) – essentially all the items provided by the layer 2 `StreetsDatabaseAPI.h`. Use the equirectangular map projection described in milestone 1.
- You must be able to show street names and point of interest names.
- The user must be able to tell major roads from minor roads, and tell which roads are one-way and in which direction.
- The user must be able to click on an intersection; the intersection should be highlighted and information about it should be either displayed in the graphics or printed to the command (terminal) window.
- The user must be able to click on a `Find` button in the user interface, type in the names of two streets (in the GUI or at a command prompt), and have all intersections between those street names be highlighted in the graphics and information about the intersections be shown in the graphics or printed to the terminal window.
- The find feature above should also work with partial street names so long as the text typed so far uniquely identifies the start of a street name.
- The map must be fast and interactive to use, and while it displays all the information above it should do so in ways that do not overwhelm the user with unnecessary detail and clutter.

To receive full marks, you must not only implement the required features above well, but also devise and implement some interesting and useful **extra feature** functionality. Some ideas to get you started are listed below, but you can and should come up with other ideas on your own. Implementing multiple extra features will make your program stand out (if they are done well) and lead to higher marks. *Possible* extra features include:

- Extract and display additional information from the more detailed "layer 1" API we have provided to OSM, which is described in Section 4.2.
- Implement additional highlighting functions when a user clicks the mouse, such as highlighting a street, building, park, etc.
- Upgrade the `Find` button so that it can auto-complete or suggest street names, or find an intersection even if there are minor spelling mistakes in the street names. This feature would also be very useful in milestone 3 to improve the usability of your program for travel directions.
- Implement additional `Find` button features that allow the user to find streets, points of interest, buildings, etc. by name.

- Make your user interface more usable by incorporating elements like dialog boxes to enter data for find or other commands, rather than requiring users to enter text at the command line.
- Extract useful live data such as traffic accidents from a web site and display this data on your map.
- There are many more features you could implement. Be creative and come up with your own, focusing on features that are truly useful!

4 Walkthrough

~30mins

Refer to "ECE297 Quickstart Guide: Graphics with EZGL" and the associated example graphics project to get started with EZGL.

After learning how to use the EZGL graphics library, you will update your NetBeans project to add the EZGL .h/.cpp files. Only one team-member needs to execute the command below (others will need to perform a subsequent `git pull -r`):

```
> ece297init 2
```

Next, implement the function in **m2.h** (Listing 1). The function `draw_map` should draw the map and allow the user to interact with it. Your main program should (and our unit tests will) call `load_map` on the appropriate map before calling `draw_map`. See the **"WD1: Graphics Proposal"** communication deliverable on the course website for a discussion of important aspects of making your graphics interactive, user-friendly, and capable of displaying information in a way that enables understanding by the user.

```
/*
 * Copyright 2019 University of Toronto
 *
 * Permission is hereby granted, to use this software and associated
 * documentation files (the "Software") in course work at the University
 * of Toronto, or for personal use. Other uses are prohibited, in
 * particular the distribution of the Software either publicly or to third
 * parties.
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
#pragma once

//Draws the map which has been loaded with load_map().
//Your main() program should do the same.
void draw_map();
```

Listing 1: m2.h.

Note that unlike milestone 1, you must make a `main()` function for this milestone, and typically you will write it in `mapper/main/src/main.cpp`. You willdemo your mapper program, which starts execution in your `main()` function, to your TA when you are marked. You can also use the `ece297exercise` command to test your `draw_map` function on various cities, but your demo will be of your mapper program, not these unit tests.

Your graphics should support any map in our ".bin" format. You will find a few maps to test out under the **`/cad2/ece297s/public/maps`**; the smallest of these maps is "saint_helena" which is useful for testing for memory errors or leaks with Valgrind.

Use a small map, and run your function with Valgrind to make sure that your code has no memory leaks.

4.1 Features

You already know how to use most of the functions of *StreetsDatabaseAPI.h* from milestone 1. However, you haven't used the **features** functions in Listing 2 yet.

```
unsigned getNumFeatures();
typedef int FeatureIndex;

//------------------------------------------------
// Natural features
// Can query any feature with index 0 to
// getNumberofFeatures() - 1 with the functtions below
const std::string& getFeatureName(FeatureIndex featureIdx);
FeatureType        getFeatureType(FeatureIndex featureIdx);
TypedOSMID         getFeatureOSMID(FeatureIndex featureIdx);
int            getFeaturePointCount(FeatureIndex featureIdx);
LatLon             getFeaturePoint(int idx, FeatureIndex featureIdx);
```

Listing 2: Features functions from StreetsDatabaseAPI.h.

Features on a map include parks, ponds, rivers, lakes and beaches; the various types are given in Listing 3.

```
enum FeatureType {
    Unknown = 0,
    Park,
    Beach,
    Lake,
    River,
    Island,
    Building,
    Greenspace,
    Golfcourse,
    Stream
};
```

Listing 3: Types of features available in the StreetsDatabaseAPI, from Feature.h.

Each feature is a polygon or multi-segment line defined by a number of Lat/Lon points. To find the number of points in feature polygon, use the `getFeaturePointCount` function, and to find the Lat/Lon of each of these points use the `getFeaturePoint` function. Most features are closed polygons which allow you to draw them as a closed shape, but a few are just a series of points that form a *multi-segment line*. An example of a closed polygon is a `Lake` or `Building`, while an example of a *line* feature is a `River` as shown in Figure 1.

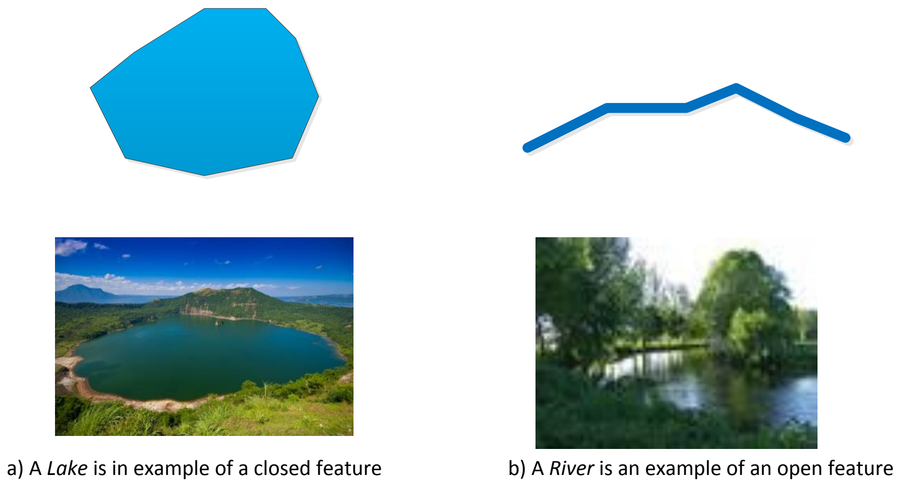
a) A *Lake* is in example of a closed feature
b) A *River* is an example of an open feature

Figure 1: Some features are closed polygons like lakes, while others are a series of points like a river.

To test whether a feature is open or closed, check the first and last Lat/Lon points – if they are the same position, then it's a closed feature, while if they are not the same, they are an open feature.

A closed feature has the same first and last Lat/Lon position.

You can also ask for the name of a feature using `getFeatureName(id)`. Some features have names like "High Park" while others do not have a name and will return a string like "<noname>".

4.2 More Data: OSM layer 1 API

Thus far in milestone 1 and in the earlier part of this document we have used only the "layer 2" StreetsDatabaseAPI; this API provides OpenStreetMap (OSM) data in a simpler and more organized format. In this milestone you can **optionally** also use the "layer 1" StreetsDatabaseAPI to access additional OSM data, as shown in Figure 2. This data is generally less structured so it will require more investigation to determine what data you want and how to use it, but it can allow you to add extra features to your map that will make it stand out. One such feature would be showing the subways in a city, which we will use as an example below. Listing 4 lists the functions available in this API.

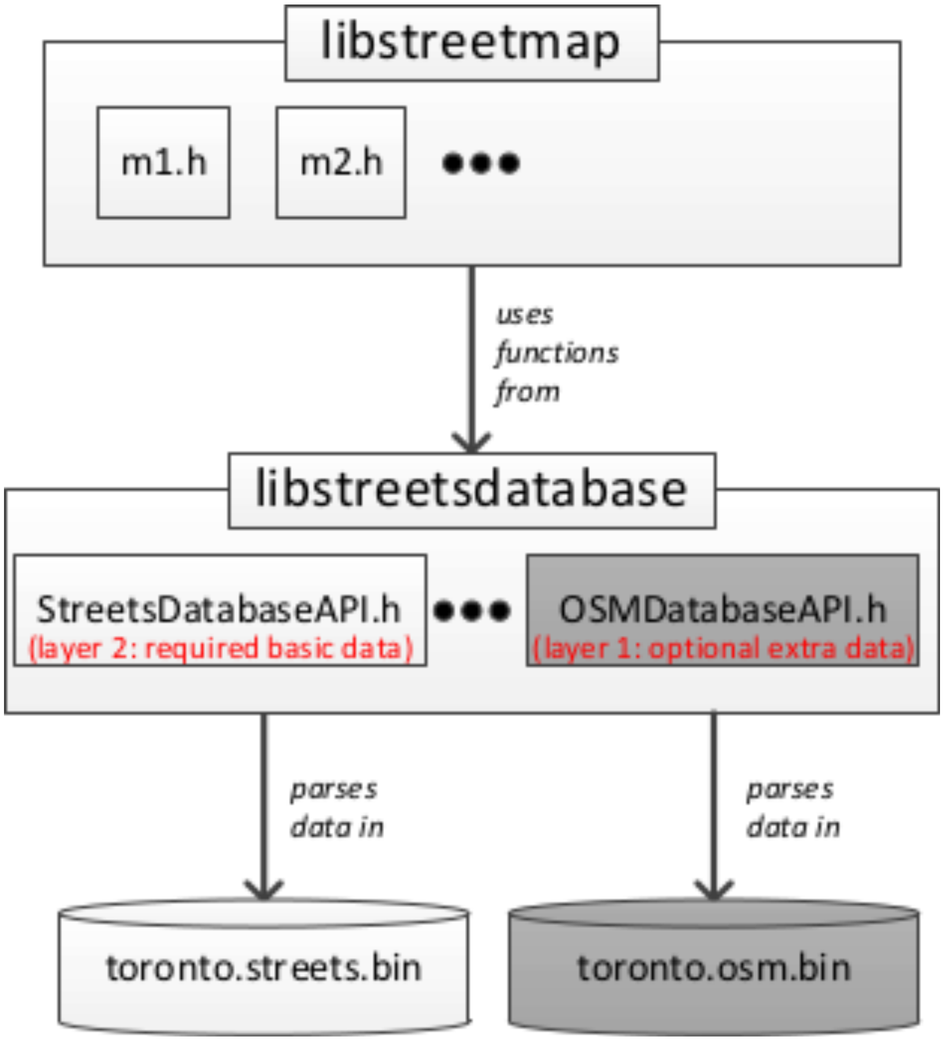

Figure 2: The two layers of the libstreetsdatabase API.

```
/*
 * Copyright 2019 University of Toronto
 *
 * Permission is hereby granted, to use this software and associated
 * documentation files (the "Software") in course work at the University
 * of Toronto, or for personal use. Other uses are prohibited, in
 * particular the distribution of the Software either publicly or to third
 * parties.
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 */
#ifndef OSMDATABASEAPI_H
#define OSMDATABASEAPI_H
#include "OSMEntity.h"

#include "OSMNode.h"
#include "OSMWay.h"
#include "OSMRelation.h"

/*******************************************************************************
 * LAYER-1 API INTRODUCTION
 *
 * ** This is a more complex API than layer 2 which is built on top of it;
 * please start with layer 2. The information accessible here
 * is not _required_ for any milestone, though it may help you provide
 * additional features. **
 *
 * The Layer-1 API handles parsing the OSM XML data, and removes some features
 * which are not of interest (eg. timestamps, the user names who made changes)
 * to conserve time and space. It also loads/saves a binary format that is
 * pre-parsed and so more compact and faster to load.
 *
 *
 * The object model of this library and API follows the OSM model closely. It
 * is a lean, flexible model with only three distinct entity types, each of
 * which carries all of its non-spatial data as attributes.
 *
 * There are three types of OSM Entities:
 *    Node      A point with lat/lon coordinates and zero
 *    Way       A collection of nodes, either a path (eg. street, bike path) or
 *              closed polygon (eg. pond, building)
 *    Relation  A collection of nodes, ways, and/or relations that share some
 *              common meaning (eg. large lakes/rivers)
 *
 * Each entity may have associated zero or more attributes of the form key=value,
 * eg. name="CN Tower" or type="tourist trap".
 *
 * After processing by osm2bin, files are stored in {mapname}.osm.bin ready to
 * use in this API.
 */
```

```

// Load the optional layer-1 OSM database; call this before calling any other
// layer-1 API function. Returns true if successful, false otherwise.
bool loadOSMDatabaseBIN(const std::string&);

// Close the layer-1 OSM database and release memory. You must close one map
// before loading another.
void closeOSMDatabase();


/*******************************************************************************
 * Entity access
 *
 * Provides functions to iterate over all nodes, ways, relations in the database.
 *
 * NOTE: The indices here have no relation at all to the indices used in the
 * layer-2 API, or to the OSM IDs.
 *
 * Once you have the OSMNode/OSMWay/OSMRelation pointer, you can use it to
 * access methods of those types or the tag interface described below.
 */

unsigned getNumberOfNodes();
unsigned getNumberOfWays();
unsigned getNumberOfRelations();

const OSMNode*    getNodeByIndex    (unsigned idx);
const OSMWay*     getWayByIndex   (unsigned idx);
const OSMRelation*  getRelationByIndex  (unsigned idx);


/*******************************************************************************
 * Entity tag access
 *
 * OSMNode, OSMWay, and OSMRelation are all objects derived from OSMEntity,
 * which carries attribute tags. The functions below allow you to iterate
 * through the tags on a given entity acquired above, for example:
 *
 * for(unsigned i=0;i<getTagCount(e); ++i)
 * {
 *    std::string key,value;
 *    std::tie(key,value) = getTagPair(e,i);
 *    // ... do useful stuff ...
 * }
 */

unsigned getTagCount(const OSMEntity* e);
std::pair<std::string, std::string> getTagPair(const OSMEntity* e, unsigned idx);
#endif
```

Listing 4: Layer 1 libstreetsdatabase API, from OSMDatabaseAPI.h.

To use this API, `#include OSMDatabaseAPI.h` in your code, and call `loadOSMDatabaseBin(mapname)` where mapname could be (for example) `/cad2/ece297s/public/maps/toronto_canada.osm.bin`.

You can then make calls (e.g. `getNumberOfNodes()` to get the number of OSM nodes, ways and relations in this map. OSM uses nodes, ways and relations as its fundamental data types.

- A *node* is a (latitude,longitude) point.
- A *way* is an ordered list of nodes that makes up a contour such as a portion of a road (StreetSegment) or a park boundary (Feature).
- A *relation* is a group of nodes, ways or other relations that form some more complex logical concept. For example a subway line would be a relation that grouped the nodes that represent the subway stations.
- Nodes, ways and relations are collectively called *OSM entities*. They can each have *tags* that are (key, value) pairs of strings that store additional information; for example a key of "name" would have a value of "High Park" for the OSM way that gives the boundary of High Park in Toronto.

For more detail on OSM's data representation, see `http://wiki.openstreetmap.org/wiki/Node`.

You can determine the location of an OSMNode by calling `OSMNode::coords()` which will return the `LatLon` of the calling object. To obtain other information about an OSMNode, you query `getNodeByIndex (index)`, where index can be any value from 0 to `getNumberOfNodes() - 1`. This function returns a pointer to an OSMNode object, which inherits from OSMEntity. Using this pointer, you can query other functions (`getTagCount(OSMEntity*)` and `getTagPair(OSMEntity*, index)` to obtain the tags – (key, value) pairs – that are relevant to this OSM object. For example, subway stations will have a tag of "railway"="station" (where railway is the key and station is the value). Hence by looping through all OSM nodes and checking which have this tag, we can find all the subway stations in a city. You obtain extra information (tags) about ways and relations in a very similar way.

For more details on the values of different attributes, go to `http://wiki.openstreetmap.org/wiki/Map_Features`.

To draw the proper links between subway stations, we would have to search through all the OSM relations to find the one that represented the subway line, and then we could use `OSMRelation::members()` to get the (ordered) vector of OSMNodes that form the subway line.

To determine the data you are looking for, such as what tags might be relevant, you can search the relevant *map.xml* file in `/cad2/ece297s/public/maps/`. These files are large, so either use a text editor that can handle very large files or Unix commands like `less` or `grep`.

Finally, there are API calls in `StreetsDatabaseAPI.h` (i.e. the layer 2 API) that return the OSM ID and entity type for intersections, points of interest, street segments and features. This data can be used to look up the OSM tag information associated with each element using the layer 1 API calls described above.

5 Additional Resources

Two possible extra features you could implement are

- accessing data from the web and displaying it on your map
- tab-completing commands and names.

See the libcurl Quick Start guide for information on how to access data on the web from a C++ program.

See the gnu readline Quick Start guide for information on how to use this library to implement tab completion of commands and names.

6 Grading

This milestone is open-ended so you are encouraged to be creative in your implementation. You will demo your map to your TA in the week this lab is due, and your grade will be determined by your TA based on the quality of your implementation compared to an average quality implementation in this class.

One part of your mark will be determined by **basic functionality**. This mark is mostly based on how well you implented the required features of Section 3. This mark will also include automatic testing of your programs ability to load, draw and close maps cleanly through `valgrind` via `ece297exercise`.

Another large part of your grade will be based on **usability and aesthetics** – how user-friendly your map is and how good it looks. Is it is easy to find information using your map, and does it respond quickly to user input? Can it show all the relevant data about map items to users who want detailed data, without overwhelming users looking for the big picture when they are examining large parts of the map? Is the most important and appropriate information easily visible at every zoom level?

Other marks will reflect the quality and ambition of your **extra features**.

Finally, there are marks for **project management and code style**. Project management will include how well you divide up tasks and track them on the wiki.

- **5 marks:** Required functionality marks.
- **4 marks:** Usability, responsiveness and aesthetics marks.
- **4 marks:** Extra features marks. Bonus marks possible for outstanding implementations.
- **2 mark:** Project management and code style