PHELMA
Grenoble INP
Minatec – 3 Parvis Louis Néel
CS 50257 – 38016 Grenoble Cedex 1

University of Nevada, Las Vegas
Transportation ,Research Center
4505 Maryland Parkway, PO Box 454007
Las Vegas, NV 89154-4007
United States of America

Internship Report

Developement of a trafic and driving simulator based on Open Street Map

Major: Software development, C++, Object-oriented Programing
Code and diagrams are available on Git-Hub : https://github.com/Dimitri1/glosm

Dimitri Gerin
SICOM 2nd year

Supervisor
Dr. Alexander Paz, Ph.D., P.E.
Transportation Research Center,
University of Nevada-Las Vegas

Technical leader
Romesh Khaddar, Graduate Assistant
Transportation Research Center,
University of Nevada-Las Vegas

07/10/2013

Acknowledgement

First, I would like to thank Dr. Alexander Paz for accepting me in the UNLV's internship program, and giving to me the opportunity to accomplish a part of my study abroad an in addition in an English speaking country. I also would like to thank him for assigning me an interesting project, through which I learnt a new language and new tools.

I want to thank Tamar Vered and Penny Tan, from the UNLV's Office of International Students and Scholars for giving me much of their time for all the administrative formalities, essentially the visa procedure.

I particularly want to thank Romesh Khaddar, my technical leader for assisting me during the development and giving me interesting advices and methods. Thanks to him, I discovered new tools and systems, which will be useful for me in the future.
I also want to thank him for the confidence he gave me about technical choices.

Finally, I want to thank the Rhone Alpes district for giving me a financial grant and Margaux Ravix from Grenoble INP, for the administrative preparation.

Summary

Introduction

For my 2nd year in PHELMA, I chose to attend a ten weeks internship in the Transportation Research Center of the University of Nevada, Las Vegas.

During this internship, I took part on a project of development traffic and driving simulator. The driving simulator part is a matter of 3D rendering of the complete real time traffic simulation, for which the player takes control of one car, in a first person viewing mode. The specificity of this simulator is that it has to be based on Open Street Map for the map generation.

The final goal of the traffic simulator project is to provide a way to analyze the influence of human driving behavior on the whole traffic simulation. This could be used to gather information in order to improve security or to analyze the indirect repercussions of human factors on the traffic. For that, the final software will have to be implemented on the TRC Driving Simulator platform (*annex 4)*.

Regarding the traffic simulation (without visualization), many models of Cars Behavior already exist. All these models govern the interactions between cars, and are based on statistical models. Even if we haven't reached the step of Car behavior model implementation during this internship, we present one of them, which may be used in a further development. A first extremely basic rendering framework has already been implemented with Blender, a free and open-source 3D graphics software. This rendering framework has been merged with a micro-simulation core model, but it was not satisfying for the needs of TRC. Actually, this first project was not based on open street map data, and in addition, they got compatibility problems with Blender when they tried to implement it the code on the simulator computer.

My technical leader requested me to use the OpenGL Library for the graphical part, because he absolutely didn't want our traffic-simulator based on any 3D graphics *SDK*. OpenGL is a low level library to generate 3D geometries, and many other more advanced graphics like lighting and shadowing effects. As we didn't wanted to start from scratch for the Graphical part, the first main part of my internship was to prospect about existing open source projects of 3D OSM map rendering based OpenGL library. Then, the second main task of my internship was to build our traffic and driving simulation on the chosen project, as an extension feature. In this report, I will first do a quick overview of the University and the Transportation Research Center, then I will describe the used tools. I will continue by presenting the glosm project, which has been chosen to build our simulation system. And, in a short part to introduce my real contribution to this project, we will present some traffic-simulation global idea, and give some concepts and definitions about traffic-simulation existing chosen model. Finally, in the last part, we will detail as concrete as possible, the traffic-simulation implementation choices and issues.

1. Used Software [4],[5],[6]

Cmake is a Cross-platform making software, and I had to use it for compiling my projects in C++. It makes the compilation possible for all platform (Linux, Window, and Mac OS), with the same Compilation description file. This Compilation description file is named CMakeList.txt, and lists the sources, the headers, and all the useful libraries, which takes parts in the projects. When launched, Cmake scans the CMakeLists.txt and produces all the compilation files depending on the OS. So, for Linux based OS, Cmake produces a Makefile and with Visual studio it produces the Visual Studio projects files.

Git is a *distributed version control* and source code management system. It is widely used by many software development communities around the world. It provides many useful functionalities for project management like complete history and full version tracking capabilities. For each project, the principle is the same. All the project sources are referenced in a local git repository, present on the local host. Every modification of the tracked sources are detected and referenced by git and saved in the Git repository file system. The user must commit every modification at each step of development and in addition, pushing (uploading) the local repository on the git hub server. One of the fundamental feature of Git is to create branches, whishes are separate way of implementation, exactly like clones. For instance, when the user needs to implement a new functionality, he just has to create a new branch in his repository, on work on this branch. All the new commitments (changing validation) will exist on this new branch and the previous code (before branching) will never be affected by the changes. By this way, if the user wants to regress and come back to the previous state of development, he just has to go back on the previous branch (often the main branch named *master*). If the source code tracked by this new branch is satisfying, the user can merge the new branch to the original branch. At every step of development, the user can save his files on the git-hub, as a copy of the local Git repository (present on his computer).

Doxygen is software, which scans the tracked source files of a project and produces documentation in much format type like html, PDF and Latex. It is very practical to use because Doxygen simply scans the code and extract automatically all the instances of a source code like class, attributes and methods if case of an oriented object language like C++. In addition, it extracts the comments for each class, attribute and method and finally builds a rich documentation. I used it to generate a website, which contains all the detailed information about my code.

2. Brief presentation of the UNLV [2]

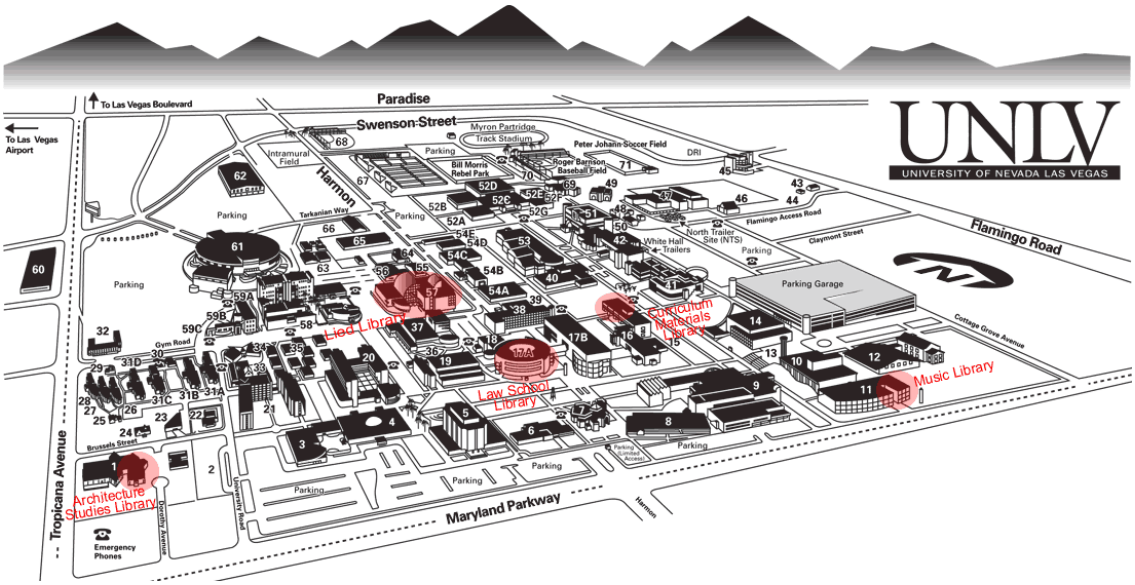


Figure 1: UNLV Campus map

UNLV (University of Nevada, Las Vegas)s is a premier metropolitan research university. Its 332-acres (134-ha) main campus, located on the Southern tip of Nevada in a desert valley surrounded by mountains, is home to more than 220 undergraduate, master's, and doctoral degree programs, all accredited by the Northwest Commission on Colleges and Universities. The UNLV is a doctoral-degree-granting institution with more than 28,000 students, more than 7,000 of who are graduate/professional students. Nearly 120 graduate degree programs are offered, including 36 doctoral and professional degrees. The university is ranked in the category of "high research activity" by the Carnegie Foundation for the Advancement of Teaching. UNLV offers a broad range of respected academic programs.

Figure 2 : Science and Engineering Building (SEB), UNLV

The TRC is physically located in the new Science and Engineering Building (SEB). It is the main research center of the Nevada University Transport Center (NUTC). The main mission of the NUTC is to perform research to advance expertise and application of technology in traffic operations and management in rapidly growing urban areas, and to provide extension and outreach services to public and private sector organizations.

3. Presentation of Open Street Map [1]

Open Street Map Wikipedia definition: http://en.wikipedia.org/wiki/OpenStreetMap

OpenStreetMap (**OSM**) is a collaborative project to create a free editable map of the world. Created by Steve Coast in the UK in 2004, it was inspired by the success of Wikipedia and preponderance of proprietary map data in the UK and elsewhere.

The specificity of our driving simulator is that the 3D map has to be generated from Open Street Map data and not from an arbitrarily built map, like a city in a video game for example. The Glosm project (described in the next part of the report) takes his information from OSM. So for my internship, I had to understand how the OSM data are organized.

Data management in OSM

The OSM data's (.osm) are written in XML format.

XML Wikipedia definition: http://en.wikipedia.org/wiki/XML

Extensible Markup Language (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

This language is basically composed of tags, elements and attributes, and could describe all elements in a hierarchical way, and could accept many attributes types, like strings and float. The tag are used to instantiate an object. Each tag represents an element, and each element contains attributes.

Example: <img src="madonna.jpg" alt='Foligno Madonna, by Raphael'/>
This is a declaration of an *img* element, which contains two elements *src* and *alt*, which takes the values of "madonna.jpg" and 'Foligno Madonna, by Raphael' respectively. In OSM there are two different types of element, the way and the node. The *node* are the remarkable geometric point (or vertex). The way is an abstract element and can represent many sub types like buildings, highway, and tree for example. The *way* declaration is fully dependent of the *node*, because *ways* are built on node. For example, for the declaration of a square shape building, we must first declare 4 node, with their respective position's attributes (latitude and longitude in OSM), and then declare a way of sub type building, and mention that this building is associated with the 4 node described above.

```
<node id='-23563' action='modify' visible='true' lat='53.893335223876946' lon='-29.529670738241325' />
  <node id='-23562' action='modify' visible='true' lat='53.89336616257267' lon='-29.529906995604907' />
                                …
                                …
<way id='-23464' action='modify' timestamp='2011-02-20T17:58:02Z' visible='true'>
   <nd ref='-23144' />
   <nd ref='-23138' />
   <nd ref='-23140' />
   <tag k='highway' v='service' />
  </way>
  <way id='-23462' action='modify' timestamp='2011-02-20T17:58:02Z' visible='true'>
```

In this example, we can see a declaration of two *node* and a declaration of a 'highway' way type.

Exportation of OSM data
It is possible to export any place of OSM map , directly from the website, but the size is limited in terms of node number. However, there are many Internet sites, which make large map available. It is possible to find .osm files for each major city. It is even possible to find the complete file which reference the whole earth planet in planet.osm.org, but the size is 320 GB uncompressed.

4. Presentation of Glosm Project [4]

Glosm Wiki definition : http://wiki.openstreetmap.org/wiki/Glosm

Glosm is hardware-accelerated OpenGL-based OpenStreetMap renderer by User:AMDmi3. Both tile rendering and realtime first-person viewer are supported.

The Glosm project is exactly what I needed as a starting point, because it already provides a stable 3D generation of OSM data, and a real-time first-person viewer, as cited above. The author of this project is Dmitry Marakasof, an active Open Source Software Programmer.

a. Glosm Principle

The principle of the glosm project is to generate 3D object from Open Street Map data, which is a 2D data type.

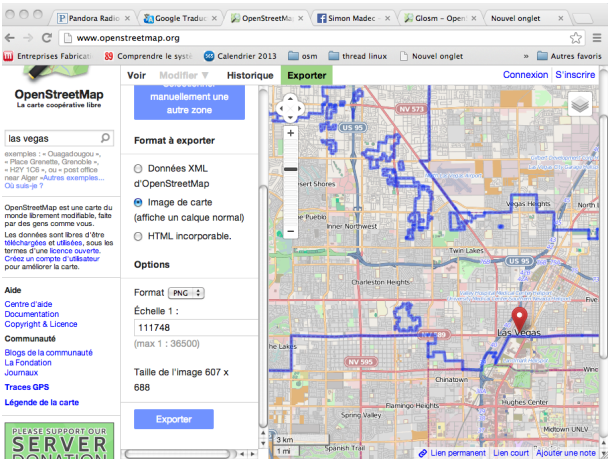


Figure 3 : OSM Screen capture

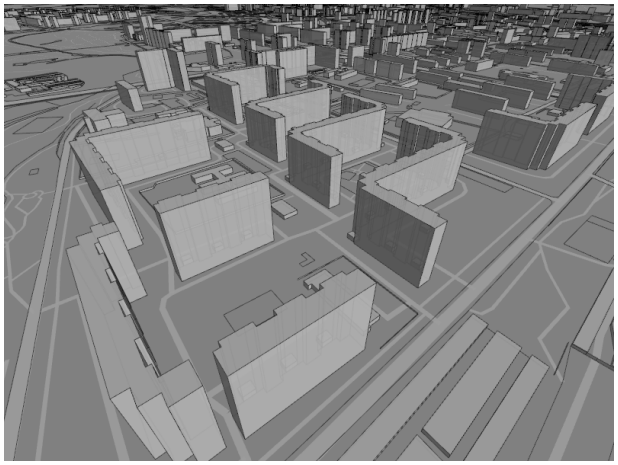
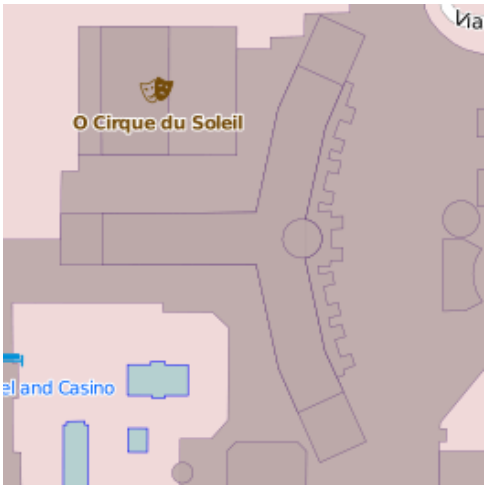


Figure 4 : Glosm Screen capture

Generation of the Buildings

At now, glosm manage the 3D shapes only for the buildings. It means that the ground elevation is not represented, so the glosm world is flat. For the buildings, the OSM database contains some 3D information, even if they are inutile for the OSM visualizer, which is a 2D visualiser like Google Map. So, glosm extract 3Ds data from the *.osm* file, and build 3D shapes from those. But , those information are very basic, like the height of the building, and the roof shape. It means that the shape of the generated buildings is totally arbitrary for the programmer. In the glosm case are square, and only the roof shape could change. So, the buildings are generated by putting together many square exactly like a LEGO game, were the 2D size (length and width) of each square correspond of the foot print of the building.

Example of a glosm building generation (Bellagio Hotel, Las Vegas) :



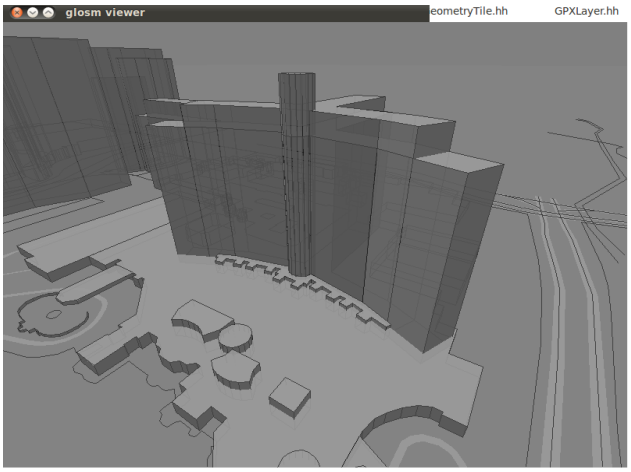


Figure 5 : Bellagio hotel OSM rendering

Figure 6 : Bellagio hotel glosm rendering

Generation of the roads

The roads is an important part for us, because it is the place where the car should progress, during the simulation. In the OSM file system, the roads are declared as a junction between many nodes.

```
<way id="116614243" user="robgeb" uid="336460" visible="true" version="1" cha
ngeset="8361612"timestamp="2011-06-06T16:05:45Z">
<nd ref="1314230193"/>
<nd ref="1314230247"/>
<tag k="highway" v="footway"/>
</way>
```

This example shows a road of instantiation in the .osm file. What is remarkable is that we could see that the tag type is highway tag k="highway", and it links two node : nd ref="1314230193" , nd ref="1314230247". All of the roads are instantiated like that, and it is even the same for all OSM entities. Way and road have both the same signification in English but we do not must make any confusion. Actually, the way tag is use to declare all entities in OSM, so the way tag *<way* is used to declare the buildings, the roads, and all other types. This tag doesn't provide any information about the instance type. In our example, the type declared in the *line <tag k="highway" v="footway"/>,* so the character chain to represent the road type name is "highway". This type of instantiation implies that the roads cannot be curved. They are drawn as portions of segment.

b. glosm architecture [5]

Glosm works with different levels of graphics generation of OSM data. First, the .osm file is scanned by an XML parser and all the primitives (Node and Way) are extracted. Then, the GeometryData source processor transforms the basics primitives into 3D triangles. Finally, the 3D triangles are associated in the GeometryLayer processor. The second layer is the GPX Layer, which represents the GPX tracks. This two layers are finally gathered by the Render. Finally, the Viewer allows the user to observe the Render Data's with a FirstPersonViewer mode.

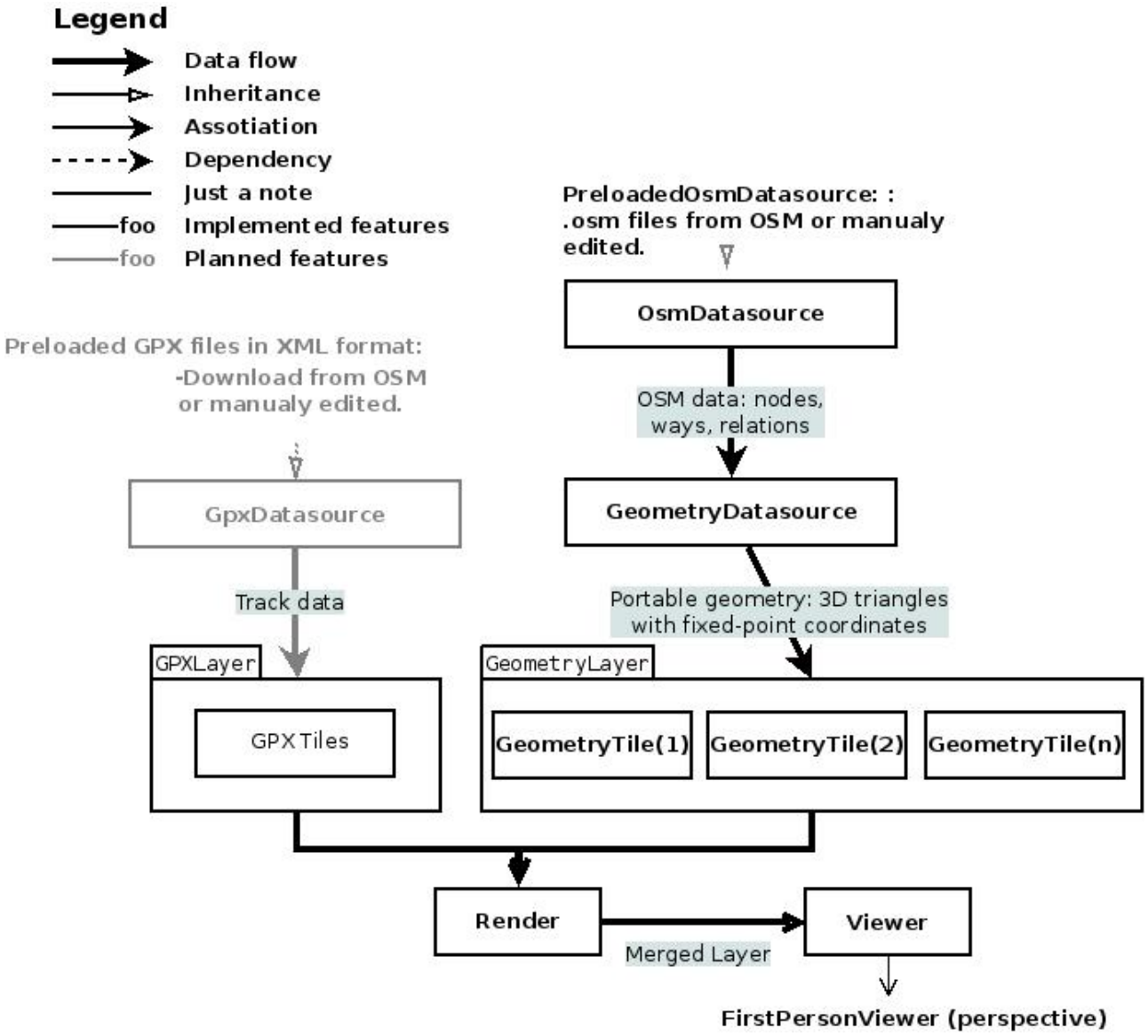


Figure 7: Simplified glosm architecture
(The complete glosm architecture is available in *annex 1.*)

c. Implementation

Glosm is implemented in 3 main libraries, libglosm-geomgen, libglosm-server, and libglosm-client.

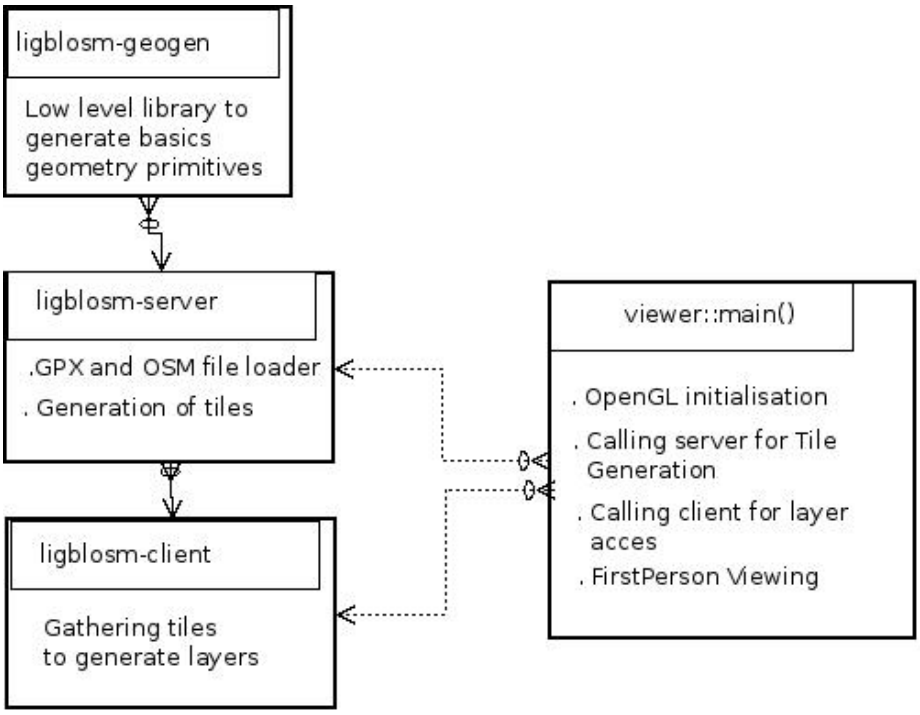


Figure 8 : Glosm Code implementation

Compilation system

As, glosm has been programmed to be buildable in all platform, and due to the quite important size of the project, Cmake is used to compile glosm.

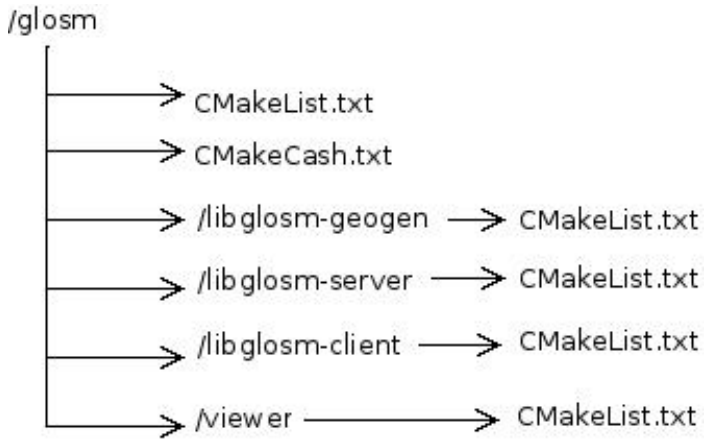


Figure 9 : glosm Compilation system

For each directory, which contains sources files, a CMakeLists must exist to list them. A CMakeList is also present under /glom dir. This is the main CMakeLists, which links the low level CMakeLists, by listing all the concerned sub directory. Then, in our case, on Linux, it produces Makefiles in glosm directory and also for each sub directory.

5. Traffic and driving simulation: Models and definition

The final goal of this project is to obtain a both traffic and driving simulator. So, our choice was to consider the driving simulation feature as an extended possibility of the traffic simulation. Instead of following the traffic simulation car model, the player's car is simply controlled by the player and provides a first person viewing. The first work was to define a way to build a both efficient an accurate traffic simulation framework. Many of them are already existing, and for all, the idea is quite the same: Separate the simulation in two layers, meso-simulation zone and micro-simulation zone, to simply reduce the computing time when implementing the system on a machine.

Micro simulation model

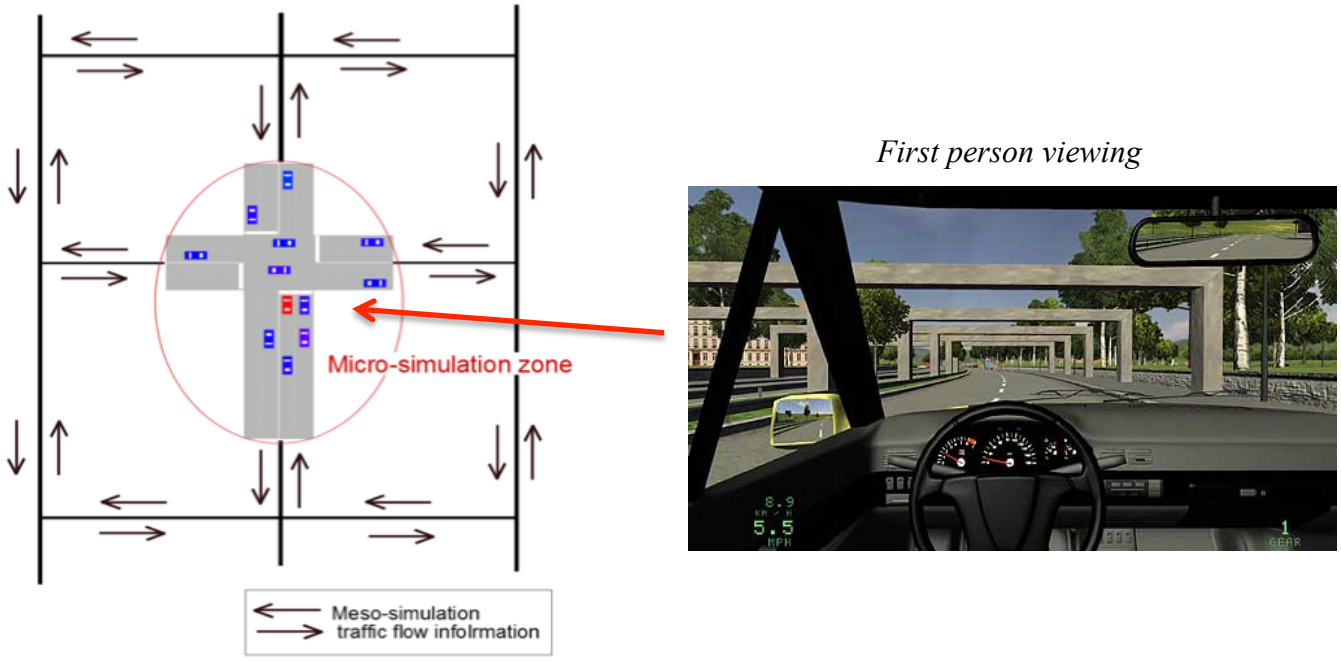


Figure 10 : Micro simulation model

The meso-simulation layer provides global traffic flow information from the whole simulation area, whereas the micro-simulation deals with information about car presents in the user proximity area. The meso-simulation has not yet been treated during my internship and we chose to start developing the micro simulation. The micro simulation model has not been implemented properly yet, but only the basic traffic simulation architecture necessary to implement the model in the future.

The Car Following models [10],[11],[12],[13]

The project was not enough to reach the Car Following model implementation step,
but I had to make researches about it at the same time.
A common car following model is the MIT's one, composed of 3 different regimes.

1. Car Following Model
2. Emergency Regime
3. Free flow regime

This model proposes one equation for each regime, which is the image of the acceleration in time.

1 . Car following regime

When the speed of a car is limited by another slower car ahead, the car following regime is active. The main equation of the car following is :

$$a_{n+1}(t+T)=\frac{C(V_{n+1}(t+T))^m[V_n(t)-V_{n+1}(t)]}{[X_n(t)-X_{n+1}(t)-l_n]^L}$$

a_{n+1} V_{n+1} : Acceleration and speed of the following vehicle.

$X_n(t)-X_{n+1}(t)-l_n$: Gap between the two vehicles

C, m, L and n **:** Constants to calibrate, and must be different for acceleration and deceleration.

2 . Free flow regime

When there is no car ahead or if the gap is important enough according to a time interval or/and a distance criteria. This is the only regime taking the driver's wanted speed into account; So if a car desired speed is lower than the leading car's actual speed, or if the car is above its desired speed, we might also switch to this regime. Here is its equation:

$$a_n=\begin{cases} a_n^+ \text{ if } v_n<v_n^{max} \\ 0 \text{ if } v_n=v_n^{max} \\ a_n^- \text{ if } v_n>v_n^{max} \end{cases}$$

a_n : Current Acceleration

v_n : Current Speed

a_n^+ v_n^{max} : Maximal acceleration and desired speed

3 . Emergency regime

When two cars are too close to each other, according to a distance and/or a time interval criteria.

$$a_{n+1}=\begin{cases} min\left\{a_n-\frac{(v_{n+1}-v_n)^2}{2g_{n+1}}, a_{n+1}^-\right\} & \text{if } v_{n+1}>v_n \\ min\{a_n, a_{n+1}^-\} & \text{if } v_{n+1}\le v_n \end{cases}$$

Other models to implement in a further development

To build a complete basic driving simulator, we also need to determine two other main models under the micro simulation: The *changing lane model* and the *gap acceptance* model (the car following model don't treat the gap between car, but only the acceleration).

6. Traffic and driving simulation: Implementation on glosm

At this point, I had all the sources present on my computer and all the compilation required library.
I spent an important time to understand many important details of the glosm code before being able to build on it. The final goal of this project is to obtain both traffic and driving simulator, so, our choice was to consider the driving simulation feature as an extended possibility of the traffic simulation. Instead of following the traffic simulation car behavior model, the car is simply controlled by the player and provides a first person viewing.

Note about Development organization and objectives
We chose to adopt an incremental development. The first increment objective was to build the architecture of basics classes required for the traffic-simulation, especially Car, and Car-container classes. The second increment was to provide to the Car class, all the functionality required for position and motion management in the OSM map type. We call it *navigation handler* in this report. The driving simulation feature has not been treated properly during the internship. I ended my internship during the second increment, but the UML diagram of increment 3 has already been defined, and we present it at the end of this report, in the *further development* part.

Global principle of traffic-simulation implementation
As we implement our code under glosm, we are fully dependent of the glosm framework. An important consideration we have to keep in mind is that glosm used the OpenGL library for 3D graphics rendering. OpenGL functions are used through the *glut* API. The way of programming and using glut is problematic for us especially because it runs under a timer frequency to update the graphics, and for many others technical reasons whishes are not detailed in this report. So, we needed to build a code in the most possible glosm independent way, to be independent of OpenGL. The best solution to be independent of glosm is to use detached thread [14] to run our simulation in separate process. Our thread function is the function, which is executed when calling a new thread. This function must implement the whole Car Behavior handling, by using the *Car Navigation Handler* Class resources (increment 2). The output of one thread functions is the trajectory of the Car (one thread for one Car). More precisely, this Car thread function produces a vector container with contains all the points coordinate of the whole trajectory during the traffic-simulation. At the same time, the goal of the increment 1 was to implement in itself a testing method to validate the advancement at each step. So, to validate the trajectory of each Car (produced by each thread), we chose to use a functionality already implemented in glosm : The *GPX tacks* rendering. Glosm allows a GPX tracks rendering in the viewer mode (first person viewing). We simply chose to use it to directly verify the trajectory of each car (increment 2), under the viewer mode. To sum up, we implemented a Car Container class, which contains Car objects (CarGlosm class), in which there is a dethatched thread method which handles the Car behavior. In the future, all the new Car Behavior models have to be handled from this thread method.

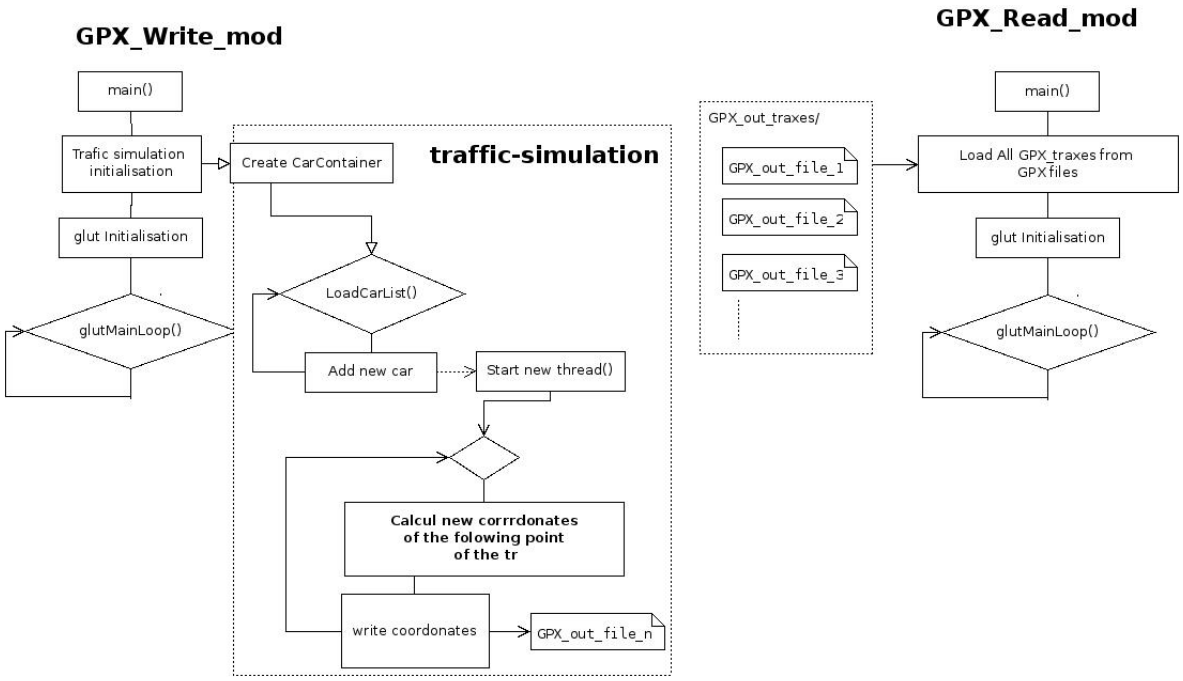


Figure 11 : The two modes for developing and testing the traffic simulation under glosm

This diagram represents the system we obtained on the increment 1. It gives an idea of the development philosophy, by describing the algorithm of the two modes. The simulation is launched in the GPX_Write_mod, while the GPX_Read_mod is just the original glosm viewing mode, which allows us to observe directly on the map, the GPX tracks resulting of the simulation.

a. Increment 1 : Basic classes

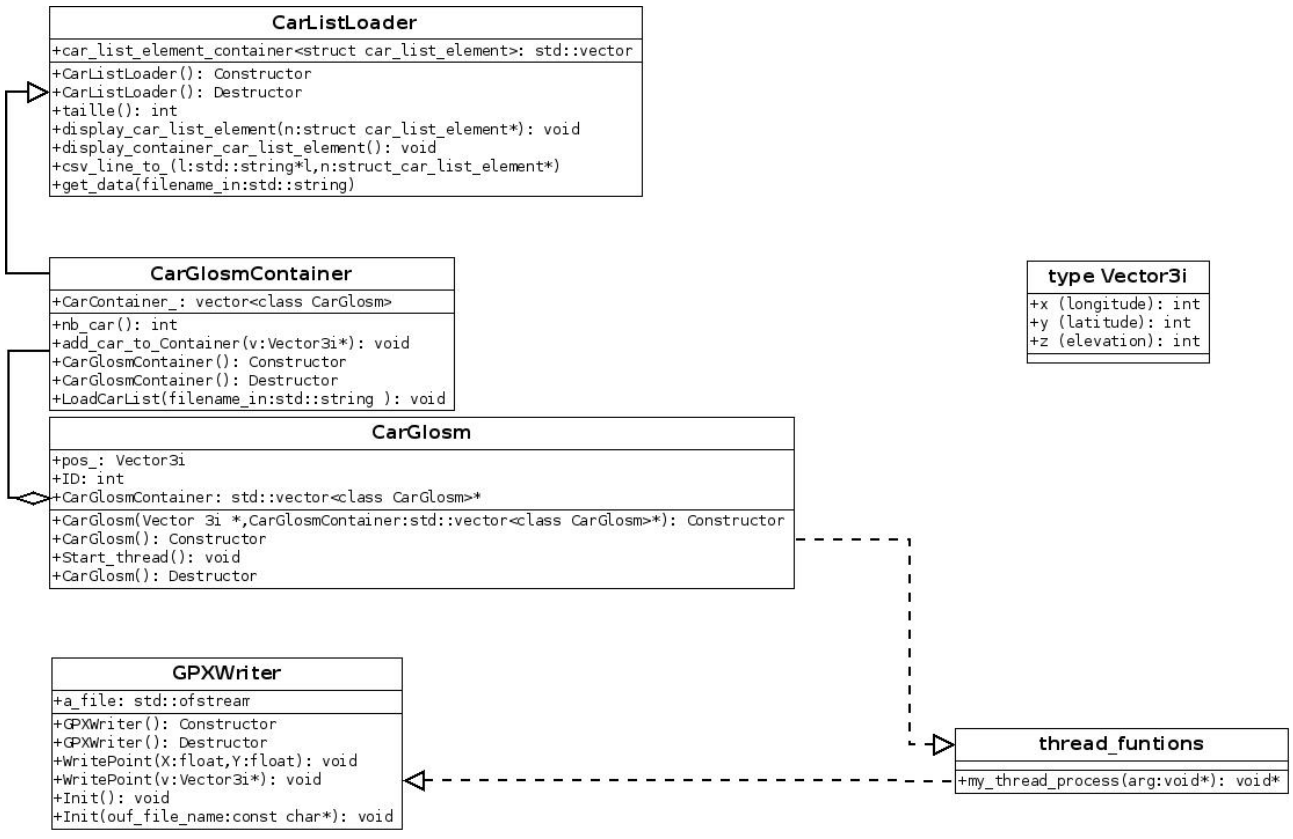


Figure 12 : UML diagram : increment 1

Each car is represented by the *CarGlosm* class, and an now there are two main attributes, one to represent the 2D position (pos_) and one for identification (ID). The *CarGlosmContainter* class is used to store the car and inherits *CarListsLoader*. *CarListsLoader* is a class for parsing an input CSV format file, which lists each car of the simulation. Each listed car must be written like this :

ID,"car","latitude","longitude" ***example :*** *1,"car","-29.53850173950195","53.89432144165039"*

All the thread functions are called by each *CarGlosm* object on the *CarGlosmContainer* and have access to all the attributes of the *CarGlosm* object. *CarGlosm* class also has a *CarGlosmContainer* pointer as attribute. This pointer provides a full *CarContainer* access to the *CarGlosm* class. This access is absolutely required because each car must know the position of all other cars during the simulation. The *GPXWriter* class is used to create a XML file, which references the coordinates of the car's trajectory during the simulation. As we have seen previously, this class is used only in the simulation mode (*GPX_Write_mod)*.

b. Increment 2 : Car Navigation handler

Point-on-a-way check

This part is the most difficult one, mainly because it deals with the OSM data, which is not adapted to our problem. We need a system, which allows the extraction of information about the configuration of the near road network for a given position. This system that we called point-on-à-way, will be useful to compute trajectories and also to check if a point lie on a way. As it is explained in the OSM presentation part, OSM data's are organized by couples of node and way. This level of description is very low, and it implies an important amount of data and an important computing time to deal with. There are different ways to implement a *point-on-a-way check*. The most obvious one is to iterate all node, then for each node, iterate all node again, then for pair of node find a way which goes between them and finally check if our point lane on this way. This method implies an horrific complexity: o(N.N.W.n), where N is number of node, W is number of way and n, the average number of node by way. A better solution is to directly iterate through all way and for each way segment, check whether the point in question lies on it. The complexity will be o(W.n) because getting a node by it's ID is constant, as *hash maps* are used for these in glosm. But glosm already implements a very interesting tool for us, the *Boundary Box (BBox)*, which allows a filtering of way for a determined square shaped area. As the *BBox* returns all the way intersected by the square shaped area (important to know a way represent each object in OSM and not just roads), we must make a second filtering to extract the type *highway,* which represents the roads. The complexity of one filtering operation with a BBox is constant, and so, if we takes our second filtering in consideration, we get a complexity of o(W), where W is now the number of way extracted from the BBox. The example below illustrates the BBox detection of two way around a point.

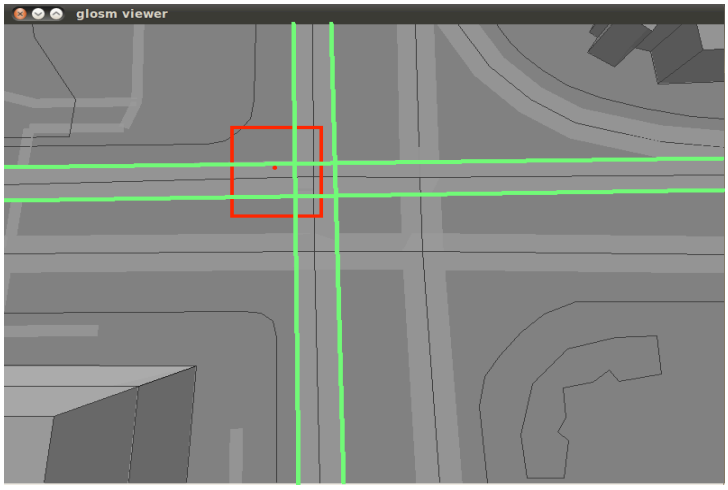


Figure 13 :
Highways intersected by a BBox

Note: You can find an example of BBox using in ***annex 3***.

Trajectory builder

Once our point-on-a-way check principle was validated. We had to implement a way to guide the car while navigating. The idea was to obtain a method, which provides us a complete trajectory of many points, by passing the origin and the destination points coordinates. In fact, the OSM way (roads) are represented by portions of many segments, because it is impossible to build a curve in OSM. So, by combining the Point-on-a-way check and the *Trajectory builder*, it is possible to create a complete highway trajectory, by re-building a new trajectory at each new segment of the highway. The point point-on-a-way check will be used to obtain the two vertex points of each highway segment. Then, the trajectory builder provides the trajectory points between them, with a determined step.

Figure 14 : Illustration of the trajectory builder principle

 *: Position for building a new trajectory with a BBox*

In the figure above, you can see an illustrated example of a complete trajectory (in red), composed of several small trajectories of each segment of the curved highway detected by the BBox. Note that the BBox and trajectory builder calling occurs at each segment endings.

Increment 2 : Issues

Regarding the point-on-a-way check, only the basic code has been implemented but we encountered many issues with the BBox way extraction, whishes have not been fixed. About the trajectory builder, the code has been completely implemented but we encountered an issue of precision. This problem concerns the *GetPointsTrajectoryFromTo* method, and occurs in the case of we compute a trajectory vector for a less than 1 meter step (distance between points).

Increment 2 : UML diagram with CarNavigationHandler

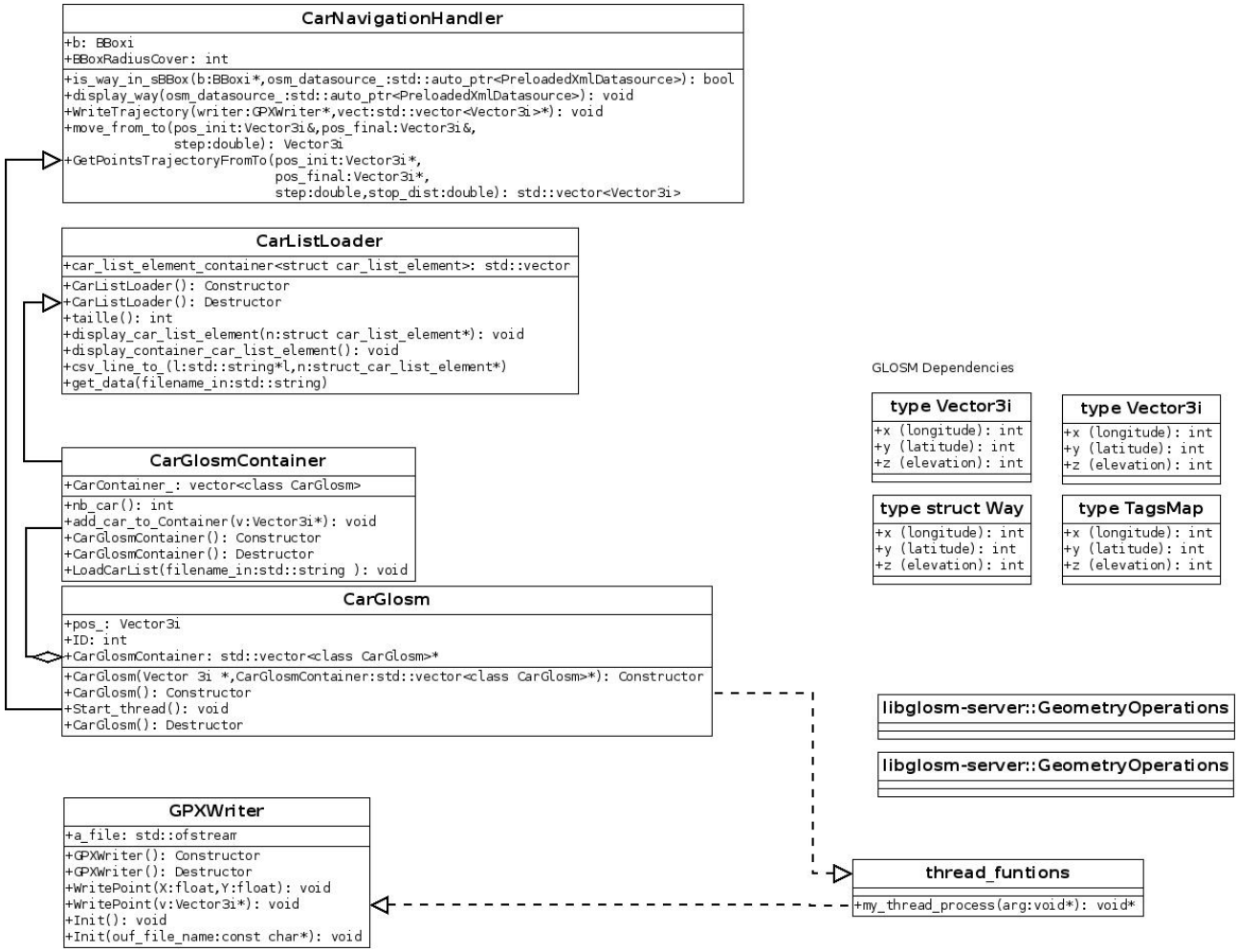


Figure 15 : UML diagram : Increment 2

The *point-on-a-way-check* and the *trajectory builder* are implemented in the *CarNavigationHandler* class. To access these functions, *CarGlosm* inheritates *CarNavigationHandler.*
It is important to know that the trajectory builder uses the Vector3i types, which is a structure containing 3 32bit fixed point variable type, to represent each component (x,y,z).
At now, there are two main functions: *GetPointsTrajectoryFromTo* returns a vector of Vector3i type, which contains all the point composing the trajectory between the two point passed in parameters, with a determined step passed in parameter to. *WriteTrajectory* is used to write all points of the vector returned by *GetPointsTrajectoryFromTo* in the GPX file. For that his function uses the *GPXWriter* class features. There is an example of a built trajectory in *annex 2.*

Issue about the used variable type [5]
I have first started by using float type for Navigation operations, but we rapidly encountered a problem of precision. In fact in OSM, the conversion between *int* and *float* to represent a position are:
float float_coord = int_coord / 10000000.0f ;
*int int_coord = (int)(float_coord * 10000000.0f) ;*
so, 12.34 would be 123400000 in osmint_t (signed int), and one meter is equal to 0,0000152 in float representation.
It implies that it is risked to make geometric operations in a meter range with a float, due to the limited size of 23 bit for mantissa. It is for this reason that we chose to use the Vector3i type. In Vector3i, one meter is equal to 152, so there is no more problem of precision with operations in meter range. An other problem of precision has been found if the step passed to

GetPointsTrajectoryFromTo is less than one meter, but this problem has not been fixed yet. But,a step of one meter should be widely sufficient to observe a trajectory with precision.

c. Further development: Advanced UML diagram : Adding Player Hander

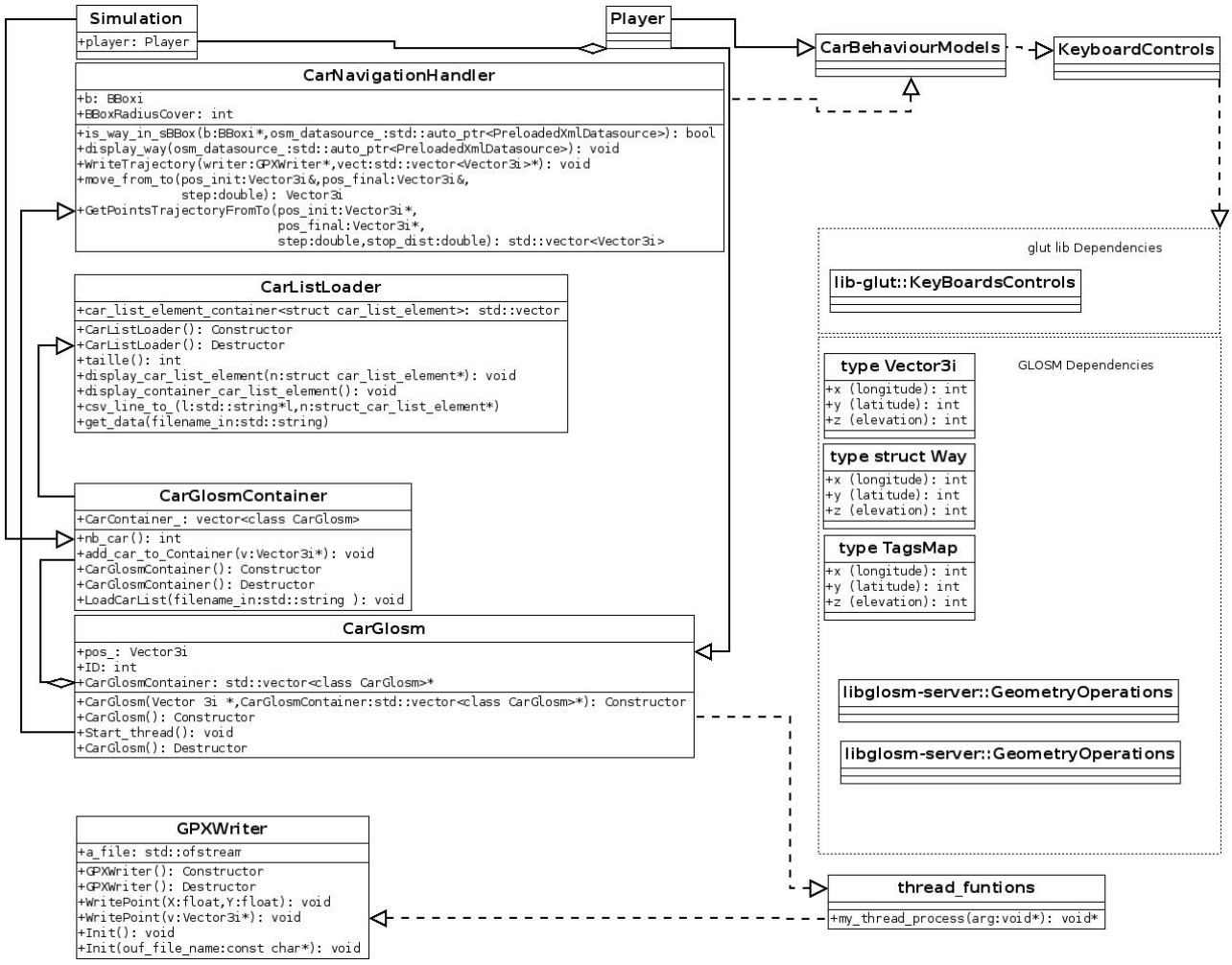


Figure 16 : UML diagram, further development

This UML diagram has not been implemented during my internship, but it will be useful in a further development and it was a part of my work to define it. It just provides a way of implementation for 3 new classes, the Simulation class, which should be the top-level class of the traffic-simulation, the player class, which inherits of the *CarGlosm* and *KeyboardControls* classes. The *KeyboardControls* class will be the first fully OpenGL dependent class. In fact, the glut OpenGL's API already provides a full keyboard control interface, which is already used in glosm to handle the *FirstPersonViewer* motion. So, the idea is to re use this glut-*KeyBoardControls* to implement our car player *KeyBoardControls* in our driving simulator. Another class, called *CarBehaviourModel* inherits *KeyBoardsContols* class and is implemented by the *CarNavigationHandler* class. The idea is that *CarNavigationHandler* class woks like a filer with Keyboards signals in input and motions information in outputs. These motions information will be finally interpreted by the *CarNavigationHandler* to build a trajectory. Note that this motion information in input should be only an acceleration variable, and not a direction. The direction should be calculated directly by the *CarNavigationHandler* because the directions determination implies a knowledge's of the highway's segment orientation on the map, and this is implemented only *CarNaviGationHandler*.

7. Conclusion

In conclusion, I am satisfied of this internship. An appreciated point for me is that I was not limited for my work, and I had to make many choices for myself like choosing the starting point project and define the starting point principle for the implementation. But, this quite important liberty of development was a difficulty in the same time. In fact, I tried to do my best to define logic and clear way of development, in order to don't fall into a technical impasse. A other difficulty was to learn languages and tools in the same time than developing. I spend an important time to find and solve simple language syntax errors, which has reduced my coding efficacy. An interesting component of this internship was the fact of working in the conditions of a real C++ Open Source Project like glosm, without any *SDK*, and with all files manually edited on Linux.

8. Bibliography and useful sources

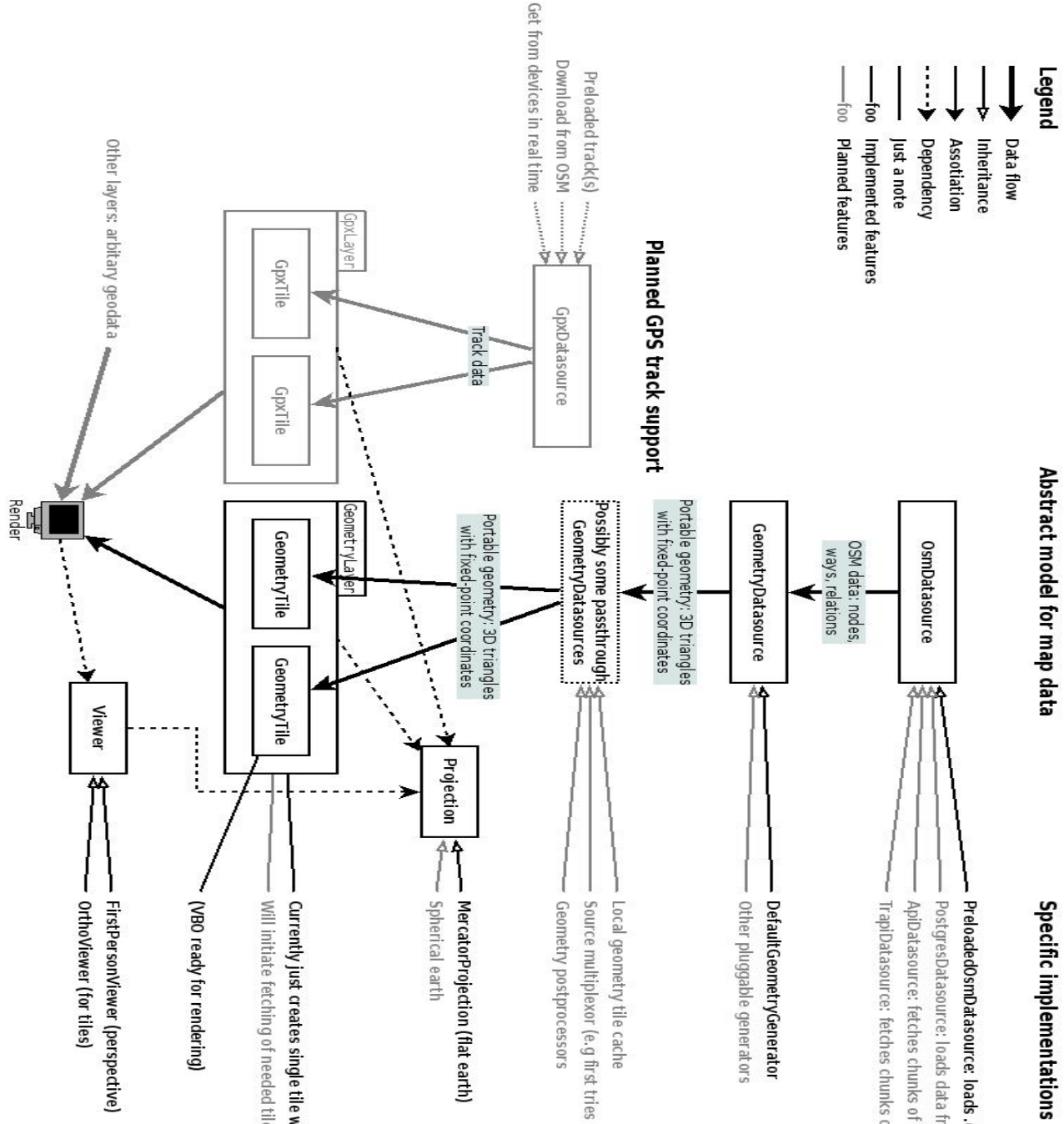
[1] www.wiki.openstreetmap.org
[2] www.nutc.unlv.edu
[3] www.cplusplus.com
[4] www.wiki.openstreetmap.org/wiki/Glosm
[5] www.github.com/AMDmi3/glosm
[6] www.git-scm.com
[7] www.doxygen.org
[8] www.cmake.org
[9] www.opengl.org
[10] Mark Brackstone, Mike McDonald, "car-following: a historical review"
[11] Johan Olstam, "Simulation of Surrounding Vehicles in Driving Simulators"
[12] Hamid Al-Jameel "Examining and improving the limitations of Gazis-Herman-Rothery car following model"
[13] John Olstam, "Traffic Simulation Models: Simulation of Surrounding Vehicles in Driving Simulators »
[14] Pierre Ficheux Website , Linux Multi Threading : www.pficheux.free.fr
[15] Nicolas Roussel, LRI, INRIA : Introduction à OpenGl et Glut
[16] Nate Robins : OpenGL turorial : user.xmission.com/~nate/tutors.html

9. Glossary

Distributed version control : Type of software used for computing project management.
GPX tracks : Name used to describe a simple point on a map, like a GPS position capture.
GPX file : Source file which contains the GPS position capture.
GPXWriter : Class used to build GPX source files.
Meso-simulation : High level of simulation. Deals with global traffic information.
Micro-simulation : Low level of simulation. Deals with discrete objects (cars in our case).
OSM : Open street Map.
OSM Datasource : Source file which contains Open Street Map data.
CSV : Coma separated value format.
XML : Extensible Markup Language (used for GPX tracks file in glosm).
XML parser : Program for extracting data from an XML file.
KeyboardControls : Controls from Keyboard.
Way : In OSM , way is a tag to represent each object like highway, building , tree…
BBox: (Boundary Box) A common tool in OSM and glosm, which provides many features to manage map's data, from a selected square shaped area.
FirstPersonViewer : A viewing mode to simulate the user point of view, like in reality.
SDK : Software Development kit.
API : Application programing interface.
CarGlosm : Class to represent a vehicle under glosm.
CarGlosmContainter : Class used to store many CarGlosm objects.
CarNavigationHandler : Class used to handle the car navigation under glosm.
GetPointsTrajectoryFromTo : Method used for the trajectories building under glosm.
int : Entire variable type.
float : Decimal Floating point variable type.

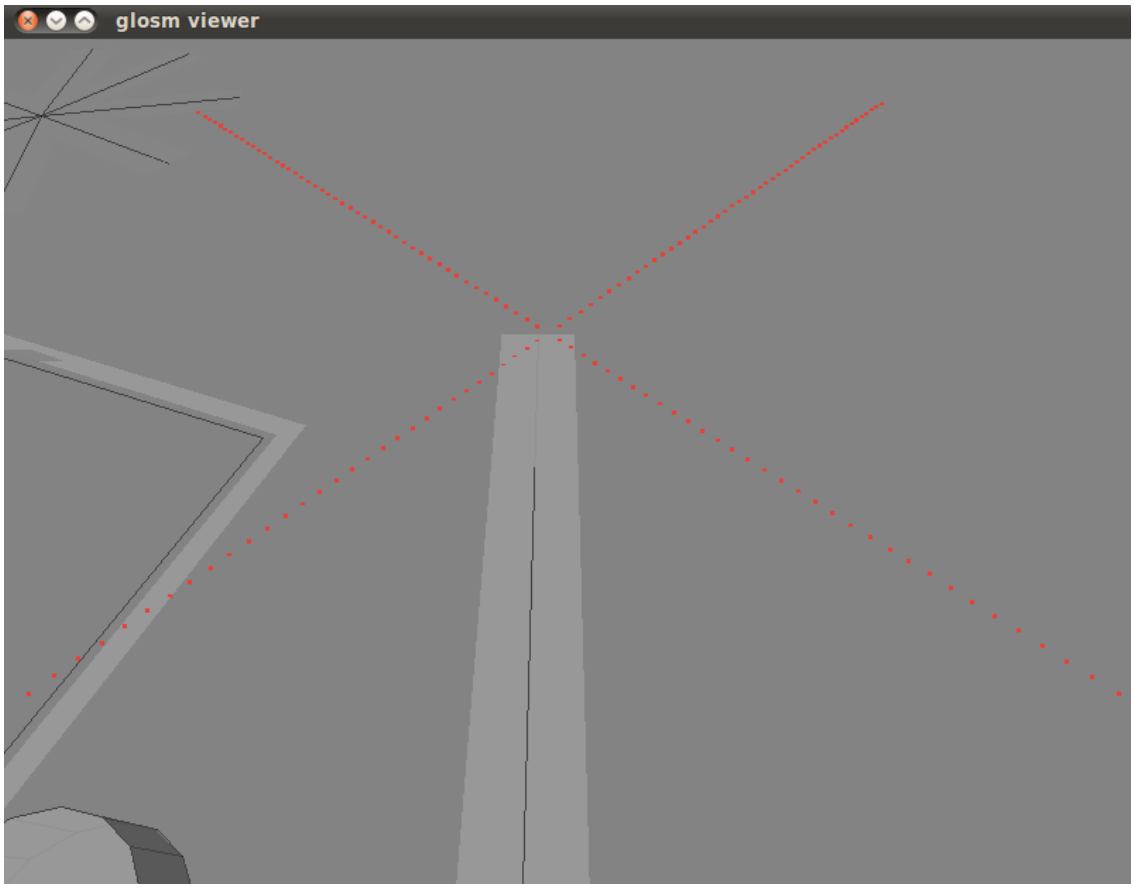
Appendix

Annex 1 : *Complete glosm architecture by* Dmitry Marakasof [5] :



Annex 2 : *Example of a trajectory builded by GetPointsTrajectoryFromTo (increment 2) :*

```
136      Vector3i c0((int)(-29.53850173950195*10000000),(int)(53.89432144165039*10000000) );
137      Vector3i c1((int)(-29.53850173950195*10000000) + METTRE_VECT_3I_x*coef ,(int)(53.89432144165039*10000000) + METTRE_VECT_3I_y*coef   );
138      Vector3i c2((int)(-29.53850173950195*10000000) + METTRE_VECT_3I_x*coef ,(int)(53.89432144165039*10000000) - METTRE_VECT_3I_y*coef   );
139      Vector3i c3((int)(-29.53850173950195*10000000) - METTRE_VECT_3I_x*coef ,(int)(53.89432144165039*10000000) + METTRE_VECT_3I_y*coef   );
140      Vector3i c4((int)(-29.53850173950195*10000000) - METTRE_VECT_3I_x*coef ,(int)(53.89432144165039*10000000) - METTRE_VECT_3I_y*coef   );
141
142
143
144 //Testing many trajectories
145     std::vector<Vector3i> test   = car->GetPointsTrajectoryFromTo(&c0,&c1,3.,5. ) ;
146     car->WriteTrajectory(&writer,&test)  ;
147     test   = car->GetPointsTrajectoryFromTo(&c0,&c2,3.,5. ) ;
148     car->WriteTrajectory(&writer,&test)  ;
149     test   = car->GetPointsTrajectoryFromTo(&c0,&c3,3.,5. ) ;
150     car->WriteTrajectory(&writer,&test)  ;
151     test   = car->GetPointsTrajectoryFromTo(&c0,&c4,3.,5. ) ;
152     car->WriteTrajectory(&writer,&test)  ;
153
```



In this example, we first define 5 points c0,c1…,c4, and we succevely computes and write in GPX file the four trajectory : c0 to c1, c0 to c2 ,…, c0 to c4. Note that for each trajectory, we choose a step of 3 meter (distance between each points) and a stop distance of 5 metter.

Annex 3 : BBox : *example of way extracting and displaying (increment 2) :*

```
112   osm_datasource_.reset(new PreloadedXmlDatasource);
113   osm_datasource_->Load( car->OSMfileName );
114   std::vector<OsmDatasource::Way> ways ;
115   std::vector<OsmDatasource::Way>::iterator it ;
```

The initial step consist of declaring an *OSM Datasource* object an load the OSM data from the input file. We also have to declare a Way vector (and iterator) to store our way resulting of the next extraction.

```
159     Vector3i c0((int)(-29.53850173950195*10000000),(int)(53.89432144165039*10000000) );
160     car->pos_ = c0  ;
161
162     BBoxi b(  car->pos_.x - METTRE_VECT_3I_x*coef   , car->pos_.y + METTRE_VECT_3I_y*coef
163               , car->pos_.x + METTRE_VECT_3I_x*coef , car->pos_.y - METTRE_VECT_3I_y*coef )  ;
164     osm_datasource_->GetWays(ways,b) ;
165     fprintf(stderr,"Nb way : %d \n", ways.size() ) ;
166     std::map<std::string, std::string>::iterator it_TagsMap;
167
168         for(it = ways.begin(); it != ways.end(); it++)
169         { fprintf(stderr," Nb Tags  : %d \n",(it->Tags).size()) ;
170
171            for(it_TagsMap = it->Tags.begin(); it_TagsMap != it->Tags.end(); it_TagsMap++)
172             { fprintf(stderr,"  Tags : (%s), (%s)\n",(it_TagsMap->first).c_str() , (it_TagsMap->second).c_str() ) ; }
173         }
```

Then, we first declare a Vector3i to store our position and a BBoxi (BBox declared by integer type).
After that, we extract all the way present in the BBox , thanks to the GetWays method.
A double for loop is finally used to display the results of the detection.

```
Nb way : 1
 Nb Tags  : 1
        Tags : (border), (glosmtown)
```

In this example, there is one way which contains one tag of type *(border),(glosmtown)*.

Annex 4 : *TRC Driving simulator platform*

Abstract

During this internship at the University of Nevada in Las Vegas, I took part in an oriented object programming project in C++ language, in which the goal was to develop traffic and driving simulator based on Open Street Map for the 3 dimensions map generation. The principal issue of this project was that it must not be based on any private 3D graphics software. In order to not start from scratch, my first task was to look for an existing project, which fill constraints previously cited. The chosen project was glosm, a 3D Open Street Map render, developed in 2007 by Dmitry Marakasov. My second main task was to define a working method and a task assignment. After that, the project was divided in 2 fully independent parts. The first one could be qualified of Graphical. Its aim was to implement a 3D car model into glosm (no evocated in this report). For the second part, which is the Driving and traffic simulator (without graphics), the aim was to implement all the required functions to the representation and the motion handling into glosm. I started, in the first increment by defining and implementing a way of testing, necessary to all the future development of the navigation handler. Then, I defined and implemented all the basic classes to represent and store the car type. In a second increment, I defined and implemented the first elements of the navigation handler for the Car type. The second increment was the principal difficulty of this internship. Its goal was to be able to know the configuration of the Open Street Map type road network from a given position, in order to compute the navigation trajectories. In fact the data system of Open Street Map is strongly unsuitable to our problem, because the roads are represented in the same way than the other object. My internship ended during the second increment. A more advanced UML diagram has been proposed in order to ensure the continuity of this project in good conditions.

Durant ce stage à l'université du Nevada à Las Vegas, j'ai participe à un projet informatique de type programmation orientée objet en langage C++ visant à développer un simulateur de trafic et de conduite basé sur les données Open Street Map pour la génération de la carte 3 dimensions. La principale contrainte de ce projet fut le fait que le code devait être entièrement basé sur la librairie graphique OpenGL et non pas sur un moteur graphique propriétaire. Afin de ne pas partir de rien, ma première tache fut donc de trouver un projet déjà existant, remplissant les contraintes citées précédemment. Le projet choisit fut g*losm,* un générateur de rendu 3 dimension de carte Open Street Map basé sur OpenGL, développé en 2007 par Dmitry Marakasov. Ensuite, ma seconde tache fut de définir une méthode de travail, et une répartition des taches. Suite à cela, le projet a été divisé en 2 parties totalement indépendantes. La première partie que l'on peut qualifier de purement graphique vise à implémenter un modèle 3D de voiture dans glosm et n'a pas été traité pendant ce stage. La seconde partie appelée *Simulateur de trafique et de conduite*, auquel j'ai été affecté, concerne toutes les fonctions nécessaires à la représentation et à la gestion des déplacement des véhicules dans l'environnement glosm. J'ai donc commencé, dans un premier incrément, par définir et implémenter un moyen de test nécessaire au développement futur du gestionnaire de navigation. J'ai ensuite définit et implémenté les classes de bases permettant de représenter un objet de type véhicule et son *conteneur*. Ensuite, dans un second incrément, j'ai définit et implémenté les premières briques du gestionnaire de navigation. L'incrément 2 fut la principale difficulté de ce stage. Son but fut en effet d'être capable de connaître la configuration d'un réseau routier du type Open Street Map, à partir d'une position donnée, afin de déterminer des trajectoires de navigation. Open Street Map possède en effet un système de données non adapté à notre problème, ou les routes sont représentées de la même façon que tout autre objet. Mon stage s'est terminé lors de l'incrément 2. Un diagramme UML plus avancé à néanmoins été proposé afin d'assurer la continuité du projet dans de bonnes conditions.