
Department of Computer Science

Henrik Andersson and Peter Oreland

# City Mobility Model with Google Earth Visualization

C-dissertation (10p)
Software Engineering

| | |
|---|---|
| Date: | 07-06-05 |
| Supervisor: | Katarina Asplund |
| Examiner: | Martin Blom |
| Serial Number: | C2007:03 |

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

# City Mobility Model with Google Earth Visualization

**Henrik Andersson and Peter Oreland**

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Henrik Andersson and Peter Oreland

Approved, 07-06-05

Advisor: Katarina Asplund

Examiner: Martin Blom

# Abstract

Mobile Ad Hoc Networks are flexible, self configuring networks that do not need a fixed infrastructure. When these nets are simulated, mobility models can be used to specify node movements. The work in this thesis focuses on designing an extension of the random trip mobility model on a city section from EPFL (Swiss federal institute of technology). Road data is extracted from the census TIGER database, displayed in Google Earth and used as input for the model. This model produces output that can be used in the open source network simulator ns-2.

We created utilities that take output from a database of US counties, the TIGER database, and convert it to KML. KML is an XML based format used by Google Earth to store geographical data, so that it can be viewed in Google Earth. This data will then be used as input to the modified mobility model and finally run through the ns-2 simulator. We present some NAM traces, a network animator that will show node movements over time.

We managed to complete most of the goals we set out, apart from being able to modify node positions in Google Earth. This was skipped because the model we modified had an initialization phase that made node positions random regardless of initial position. We were also asked to add the ability to set stationary nodes in Google Earth; this was not added due to time constraints.

# Contents

# List of Figures

# 1 Introduction

What later became known as ad hoc networks started out in the 70's as "packet radio" networks, sponsored by DARPA[1]. Since then, considerable research has been done in the field. Wireless Ad hoc networks today are flexible, self configuring networks that do not need a fixed infrastructure. Recently, they have appeared in the news with the $100 laptop, a cheap laptop intended for distribution in developing countries, which use wireless ad hoc networks to establish a network "out of the box".

The work in this thesis focuses on simulations of mobile ad hoc networks. We will create a set of utilities to enable visualization of a simulation of a city section. The project work consists of several smaller parts. First, we will have to get road coordinates from a database (census TIGER) and convert it to a format Google Earth can display, KML, an XML based format used to store geographical data. Then, in the second part, we will convert the Google Earth data to a mobility model, which outputs all the movements on the city section for use in the simulator. Mobility models are used to model node movements, which in our case will be limited to a city section, a network of roads.

The project work will be aimed at helping researchers visualize their simulations. By using Google Earth, it is also possible to put markers on the map to symbolize nodes that can read into the model. It also makes it easier to add improvements and additional functionality later on.

The finished solution included the major goals of the project, apart from a few limitations imposed by time constraints and by the mobility model that it was built upon. We did not implement functionality to set starting positions of nodes in Google Earth. This was skipped due to the fact that the mobility model randomizes starting positions in the initialization phase. Secondly, we were asked to have stationary nodes on the city section. We were never able to add this functionality due to time constraints. It would have required what we deemed to be rather extensive modifications to the model.

[1] Defense Advanced Research Projects Agency, an agency of the US department of defence.

This thesis is structured as follows. First, in chapter 2, we go through some basics that will be needed when reading through the rest of the paper. Then follow the chapters for the different steps, or parts, of the entire application. In chapter 3 we describe the process of converting and linking road data from the United States Census Bureau TIGER database into Google's KML format. The next major section is the modification of the mobility model. Here we take a look at what modifications where made to the existing model and also how the KML data was enabled as input. The final major part deals with the simulations using the ns-2 simulator. We will describe the processes of setting up the simulator and presenting some of the results, for example showing a finished simulation visualized in a network animator (NAM).

Appendix A contains relevant code for the mobility model, including the modifications made to it.

# 2 Background

In this chapter we provide background information that will be useful when reading the rest of this thesis. We start off by looking at the basics of wireless networks and more specifically ad hoc networks, the kind of networks that our model deals with. Then, in section 2.3 and 2.4, follows information on KML, Google's language for storing three-dimensional geographic data, and XML, which it is based on. These sections are intended to give an overview of what XML is and the syntax of the KML files, which we used to store data from the TIGER database.

The KML files are opened in Google Earth, which is discussed in the following section. In this section we briefly have a look at what it is and how it is used. Next comes a section on the TIGER database, the database we used to get information on road networks and, finally, section 2.7 and 2.8 discusses ns-2 and its related software. ns-2, or *Network Simulator 2*, is the simulator that will be running the simulations that our mobility model will output.

## 2.1 Wireless Networks

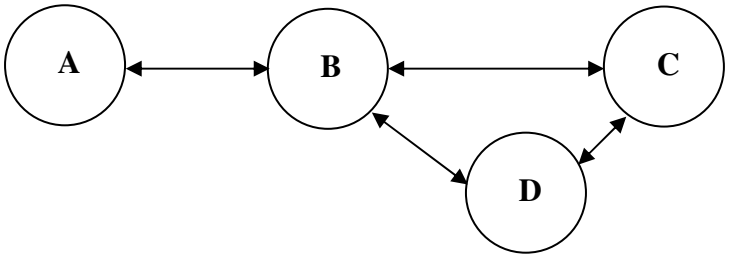
The most common wireless networks [3] today are the ones where you have one access point, i.e. a router. All computers that want to log on to the network have to check in through the router to gain access to the network and all traffic is sent through the router as well.

The most common wireless network standard is IEEE 802.11, which includes a few different standards within itself that are divided by alphabetic characters such as a, b and g. These standards use an unlicensed radio spectrum to provide wireless ethernet.

## 2.2 Ad hoc Networks

The origin of ad hoc networking can be traced back very long ago when you needed to send messages in a hurry. The most common way was to have people on tall structures or heights shouting from one position to the other until the message reached its destination.

An ad hoc network works in a similar way. There are no access point, instead nodes discover other nodes within its range and forms a network together. In this way, you can reach nodes out of range by going through nodes within range, which will forward your messages, so called multi-hop relaying. Without base stations, the nodes does not only work as end-systems, but also as routers forwarding packets.




*The nodes make up the network themselves. C and D cannot reach A directly so B needs to route and forward the traffic.*

*Figure 1: Example of ad hoc net structure*

The ad hoc network was developed because developers saw an opportunity to make a network without a fixed infrastructure. Self-configuring and maintenance properties are built into the network, which makes the ad hoc network both quick and cost-effective when deployed. Application domains include battlefields, search and rescue operations and collaborate computing. The network can be setup as a completely standalone network or it could just be connected to the internet.

In mobile ad hoc networks (MANets)[4] the nodes are free to move about and organize themselves into a network. This works with the main idea that when a new node wants to be a part of the network the node announces its presence and listens to broadcast announcements from its neighbors. The node learns about new nodes in range and ways to reach them, and may announce that it can also reach those nodes. As time goes on, each node knows about all other nodes and one or more ways to reach them.

Because of the mobility in manets, path breaks, packet collisions and transient loops often occurs and the network needs a good routing protocol to resolve these matters as well as having constant updates about which neighbors you have.

## 2.3 XML

XML (*Extensible Markup Language*) [1, 2] is a meta-markup language and a W3C[2] standard. In contrast to other markup languages, such as HTML, one can make up own tags according to what is needed as long as they are correctly organized. XML provides a structured and organized way of sharing data. For example, if you are working with personnel files you could write something like the code below:

```
<employee>
    <name>Daniel Myer</name>
    <position>Programmer</position>
    <salary = "2200" />
    …
</employee>
```

*Figure 2: XML code example*

XML's simplicity makes it easy for programmers of any level and even for those without programming experience to understand and write. At the same time XML is easy to parse, since it doesn't include any formatting instructions, but only describes data. As such, it is easy for virtually any program to process the data.

XML files are mostly created with text editors, which can range in complexity, from a simple editor such as *vi* to a fully WYSIWYG[3] editor. The final document can then be processed by a parser that makes sure that the data is well formatted (if it is syntactically correct). The document can finally be read by an application. For example, it can be opened in a browser, which formats the data and displays it to the user.

A syntactically correct XML document should have one root element, it should be properly nested and it should have values within quotes. Finally, one must also keep in mind that it is case sensitive. If a document follows these rules it is considered well formatted.

## 2.4 KML

KML (*Keyhole Markup Language*) [3, 4] is an XML-based language used to display three-dimensional geographical data in Google Earth and Google Maps. It can store information

[2] World Wide Web Consortium. International consortium that work to develop Web standards.

such as paths, placemarks, ground overlays (images "draped" over existing textures), 3d models etc. Each object is always given a longitude and a latitude. Furthermore, KML-files can contain descriptive data, sometimes formatted with HTML, special icons, a camera view etc.

As an example, a placemark with HTML formatted description can look like the one below (the CDATA tag is used to make sure that HTML is parsed correctly):

```
<Placemark>
   <name>CDATA example</name>
   <description>
     <![CDATA[
       <h1>Some title</h1>
       <p><font color="red">This is a <i>placemark</i></font></p>
     ]]>
   </description>
   <Point>
     <coordinates>102.595626,14.996729</coordinates>
   </Point>
</Placemark>
```

*Figure 3: KML placemark example*

KML files can be created with the Google Earth interface or written directly in a text editor and then compressed together with any images using the ZIP-format. Such ZIP-archives are called KMZ and is the most common way to distribute KML-files.

We mainly used the KML-files to store roads (paths) and nodes (placemarks) that were used as input for our mobility model. We had to get the road-information from another source than Google since it is impossible to retrieve road-layer information from Google Earth.

## 2.5 Google Earth

Google Earth [5], formerly Earth Viewer, is a 3d globe program developed by Keyhole, inc. and later acquired by Google in 2004. A year later its name was changed to Google Earth.

[3] What You See Is What You Get. What you see when editing is very close to or exactly what you get in the final product.

Images are obtained from satellite and aerial photography and thus resolution vary depending on points of interest. In large cities for example, cars and people can be discerned while smaller towns and other places have considerably lower resolution.

Geographical information, for example roads and terrain, are stored in layers that can be turned on or off. Apart from geographical layers there are also layers with other information such as *Geographic Web*. It displays points of interest that link to information from Wikipedia[4] or Panoramio[5]. Such layer information can be stored in KML-files that Google Earth can read and display on the virtual globe. It is also possible to save some information to KML-files.

Google Earth is currently available for Windows, Linux, Mac OS X and FreeBSD in three versions. Besides the free version there are two commercial versions (plus and a pro) which provide further functionality and customer support for a price.

## 2.6 TIGER

TIGER (*Topologically Integrated Geographic Encoding and Referencing*) [6] is a database of geographic and cartographic information by the United States Census Bureau. The database covers all US counties and a few surrounding islands.

The database is organized after states that contain one compressed file (ZIP format) for each county. Each county contain a set of record types describing different aspects of the graphical map. As an example, record type one (RT1) contain address information along with start and end coordinates of the roads. Using the ID-number in RT1, additional coordinates are retrieved from record type two, coordinates that makes up the curves of the roads. To get complete information about a geographical feature it is often necessary to combine information from several different record types in this way.

TIGER-records contain all kinds of geographical data; roads, water, buildings, borders etc. and are spread out across a range of different record types. Apart from purely geographical

[4] Free online encyclopedia, maintained by its users. http://en.wikipedia.org/
[5] A site that host images linked to where they were taken.

data, such as buildings, there are also information in the records that specify address ranges, zip codes an such information. There are also record types providing information about the database itself, for example record type H. This file stores a history of every complete chain (record type 1), when they were split or merged. TIGER files can be displayed with most GIS[6] applications.

The current edition of the database is the second edition from 2006 (released March 6 2007) and will be the last files the Census bureau intends to release in the current format. Future TIGER spatial data will be released in shapefile format.

## 2.7 ns-2

ns (*network simulator*) [8, 9, 10] is a discrete event simulator, meaning it operates on an chronological sequence of events. ns was programmed in C++ with a simulation interface written in OTcl, an object oriented extension of the scripting language Tcl. It provides simulation for several types of protocols over both wired and wireless networks.

ns started out as a variant of the REAL network simulator in 1989 and had six years later, in 1995, gained wider support from DARPA, among others. Today it is developed by different institutes and researchers and has reached its second generation (ns-2). Work on ns-3 was started July 1, 2006 and is planned to be completed sometime in 2010.

## 2.8 NAM

NAM (Network Animator) is a TCL/TK (Tool Command Language / TK GUI Toolkit) based animation tool for viewing network simulations. When a simulation has been run in ns-2 it outputs a trace file that can be opened in NAM and it's then possible to watch the nodes move and to see the simulation take place. With NAM you can watch the whole simulation both forward and backwards, you can yourself change the speed of the simulation and zoom in and out on nodes etc. See Figure 4.

[6] Geographic Information System. Computer system used to handle geographic information.

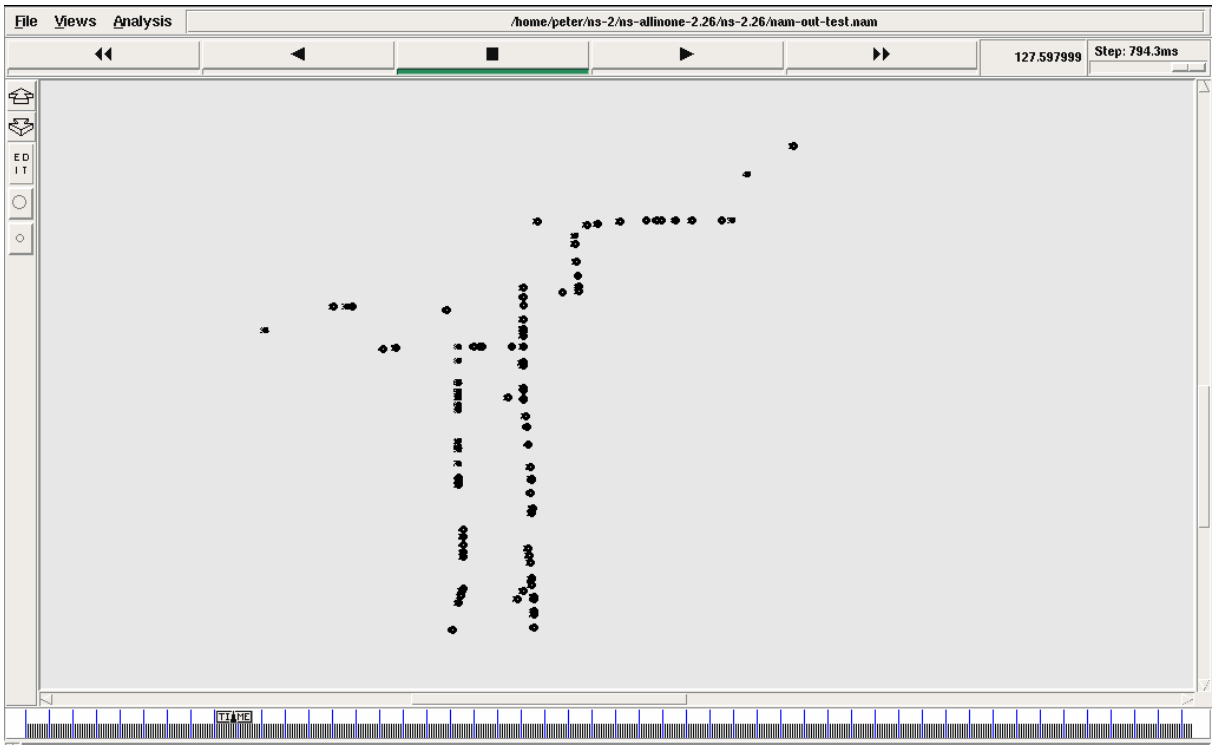



*Figure 4: NAM running a simulation*

# 3 From TIGER to Google Earth

This chapter describes the part of the project that is responsible for extracting geographical data from the census TIGER database and converting it to Google's XML-based KML format. The output at the end of this stage is readable by Google Earth and will display all the roads in a specified area of a county.

First, in section 3.1, we will take a look at the motivation behind using the TIGER database. Then follows section 3.2, where the conversion of TIGER data will be described in detail and also what is needed from the TIGER database and how the data is processed. Next, in section 3.3, we will describe the development process, what problems were encountered and how they were solved. In the final section, 3.4, there are instructions detailing how the converter is used.

## 3.1 Why the census TIGER database

Before going into details about the implementation and development of the conversion utility, we explain the motivation behind using the census TIGER database. Google Earth already has a road layer that contain most of the roads across the globe. Furthermore, this layer can be displayed separately, so why not use it? This was our intention at first, but however, it turned out not to be that simple. It is impossible to save road layer information from Google Earth, because it is streamed from Google's servers and we were not allowed to store it. Therefore, it became necessary to obtain this data from a second source, which we wanted to display in Google Earth.

The U.S. Census Bureau maintains a large database of geographic and cartographic information on every county in the United States, the TIGER database, which is available free of charge to anyone. Using the TIGER database does of course introduce a limitation to the project; we will be restricted to only using roads in the U.S. On the other hand, it provides us with all the data that we need. Determined to be the best option available, it was chosen.

The TIGER database does not store its data in a way that Google Earth can interpret, therefore conversion became necessary.

## 3.2 How it works

The conversion part of the project was responsible for extracting and linking together information from the TIGER database and storing it in a way Google Earth could interpret. This conversion was done by our utility that we call *TIGERconv*.

To extract necessary information from the TIGER database it is sometimes necessary to link together several file types using various id numbers. Before examining this content closer it is worth mentioning something about the terminology. Census TIGER uses the definitions from the *Spatial Data Transfer Standard* (SDTS) [11]. Interesting for us is the definition of a *complete chain*:

> "A chain [a sequence of non-intersecting line segments] that explicitly references left and right polygons and start and end nodes."

The file types that contain information on these complete chains that we needed were primarily record types one and two (see 2.6 about TIGER structure). Type one is called complete chain basic record and stores the start and end points for the complete chains that we needed. Apart from these coordinates, it also contains other data that we needed to filter out, such as street name, address range and different flags.

Since it is not possible to know how many fields each line have before going through it, we parsed the lines to see if it contained a feature code (*Census Feature Class Code*) that indicated that it was a road. Road feature codes are of class A, which means it starts with an A. As an example, a line in the record type of class A25 lets us know that it is a "Primary road without limited access, US highways, separated".

After having established that the line is actually a road we could start taking out the elements we needed. Obviously, we needed the coordinates but also the TIGER/Line ID in order to be able to find additional data in other record types. Thus, now that we knew how the line looked, we stored the first field (the id) and the last four (the coordinates).

The next step in the conversion program was to obtain additional data from record type 2, called *Complete Chain Shape Coordinates*. To be precise, certain checks were performed before this point to see if it was necessary to proceed with the linking, for example to make sure that it was within the coordinate-box specified by the user. If the chain passes the check, the record will be searched for the TIGER/Line ID and if a match is found (not every line in record type one have a matching line in record type 2) the coordinates will be extracted.

The coordinates that are stored in a long sequence, sometimes spanning more than one line, are points between the end and start of the chain. The coordinates comprises the shape of the road, giving it its bends. Coordinates are read until the first "0-field" is found, signalling that there are no more coordinates in the chain.

The final step in the program was to save the data to a KML file, Google Earths XML based format. Headers and various tags are added to produce a final result such as the KML file depicted in Figure 5:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.1">
<Document>
 <Placemark>
  <LineString>
   <extrude>1</extrude>
    <coordinates>
      -86.767923, +36.158745, 0
      -86.768027, +36.158624, 0
      -86.769684, +36.157902, 0
    </coordinates>
  </LineString>
 </Placemark>
…
</Document>
```

*Figure 5: a section of KML output from the conversion*

The section of the KML file, shown in Figure 5, shows one road segment. The first tag is the XML header, which will always be first in each KML file. The header is followed by the KML namespace declaration and will also always be present as line two of each file. Further down is the first *placemark* element, the most commonly used feature in Google Earth, which can contain anything from points to polygons and models. In this case it contains our road segment. Next, we have a *LineString* which creates a path (road) with *extrude* set to 1, specifying that the line should be extended to the ground. Finally, we give the coordinates, in this case we have three points: the start, a middle point and the end.
Coordinates are given in the form:

*-longitude, +latitude, height*

A generated KML file can contain countless road segments declared in the way shown in Figure 5. As an example, a segment of Jersey City (shown in Figure 8, later in the chapter) can produce an output of over 4000 lines. When viewed in Google Earth, each road segment can be viewed individually by selecting them from a long list in the interface, as can be seen in Figure 6.


*Figure 6: output.kml expanded to show all road segments*

## 3.3 Development, problems and solutions

The first step in the development was to make sure that it worked with only the start and end points of the chains, i.e. to only use record type 1. After going through the material available about the database we realised that we could not simply go through the file and select, for example the fourth field in every line. The issue was that since all fields are not always included, no easy separation between them is available and, furthermore, sometimes two fields are not separated at all. The solution was to tokenize as much as possible and then we would get the most important fields (ID and feature code).

Having extracted the feature code we came upon the first problem in the development; the same *atoi* (ASCII to Integer) conversion gave different outputs on different systems. After a lot of gruelling debugging we started to realise that the problem was not in the code but seemed to be linked to the fact that we got different results on different versions of our Windows IDE, Visual Studio. Soon after, the problem was localized to the project settings and resolved.

At this point we had our first working output that could be examined in Google Earth, as shown in figure 7.

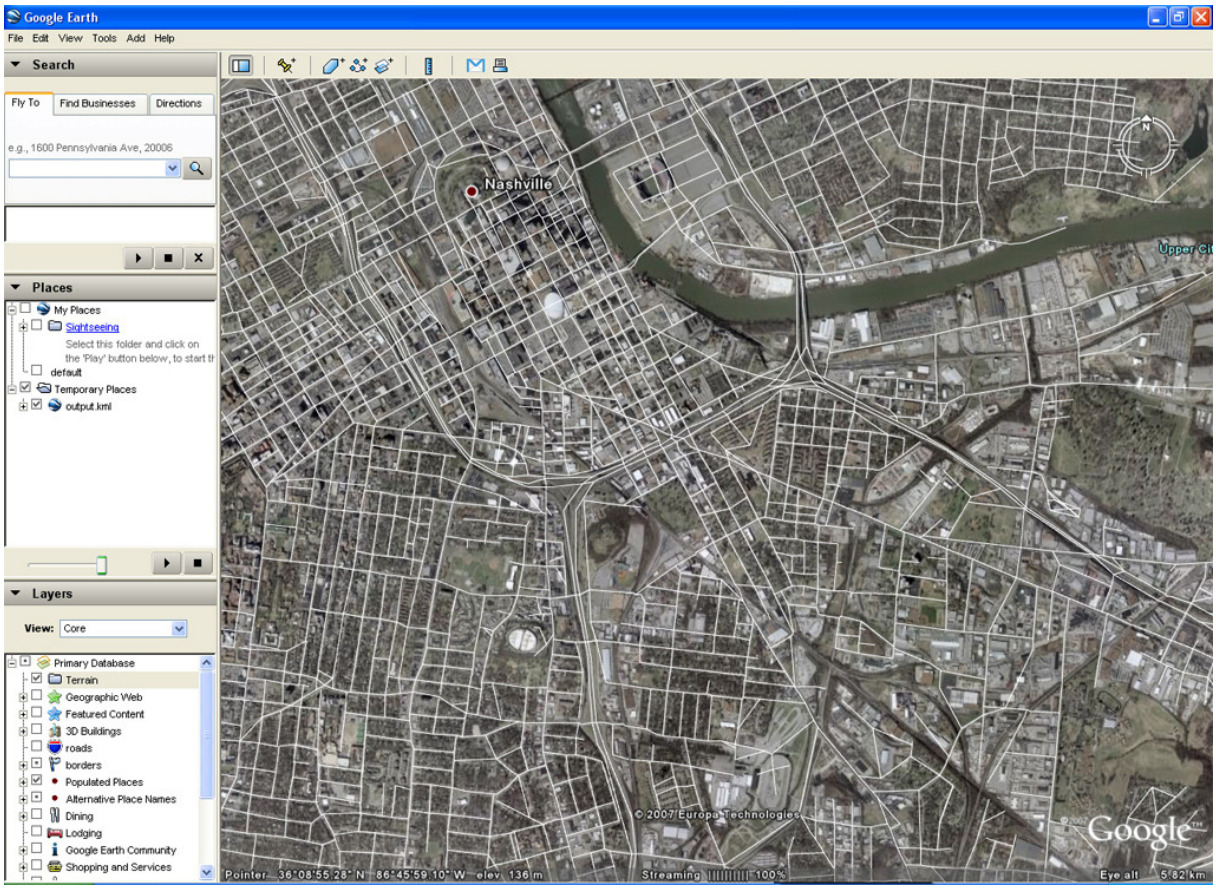



*Figure 7: Google Earth showing output from TIGER Record Type 1 on the city Nashville*

The next step was to introduce limits to what area should be extracted, since it would otherwise take too much time to link together the record types and it would produce a very large output. When we had an algorithm designed and were ready to test it we were faced with another problem. The coordinates that were given in the Google Earth interface didn't match those in the KML and TIGER files. After some research it was concluded that the coordinates given in Google Earth were shown as minutes and seconds instead of degrees, which we wanted. Later on we discovered that it was possible to change from the standard in the settings.

After getting coordinates directly from the KML file we tuned the algorithm and were able to proceed with the linking of record type 2. Working in data from type 2 didn't take much time to finish since it was basically much the same tokenizing and selection that we had previously done, with some structural differences. There was, however, one strange error left; some roads seemed to pass through structures and deviate quite a bit from the underlying image.

After further testing we were puzzled since the coordinates were not altered in the transition from TIGER to KML. As it turned out, the "problem" was actually very simple. When we checked the documentation it turned out that some of the feature codes of the roads were actually tunnels. We decided to continue to use all roads, even though they were actually tunnels.

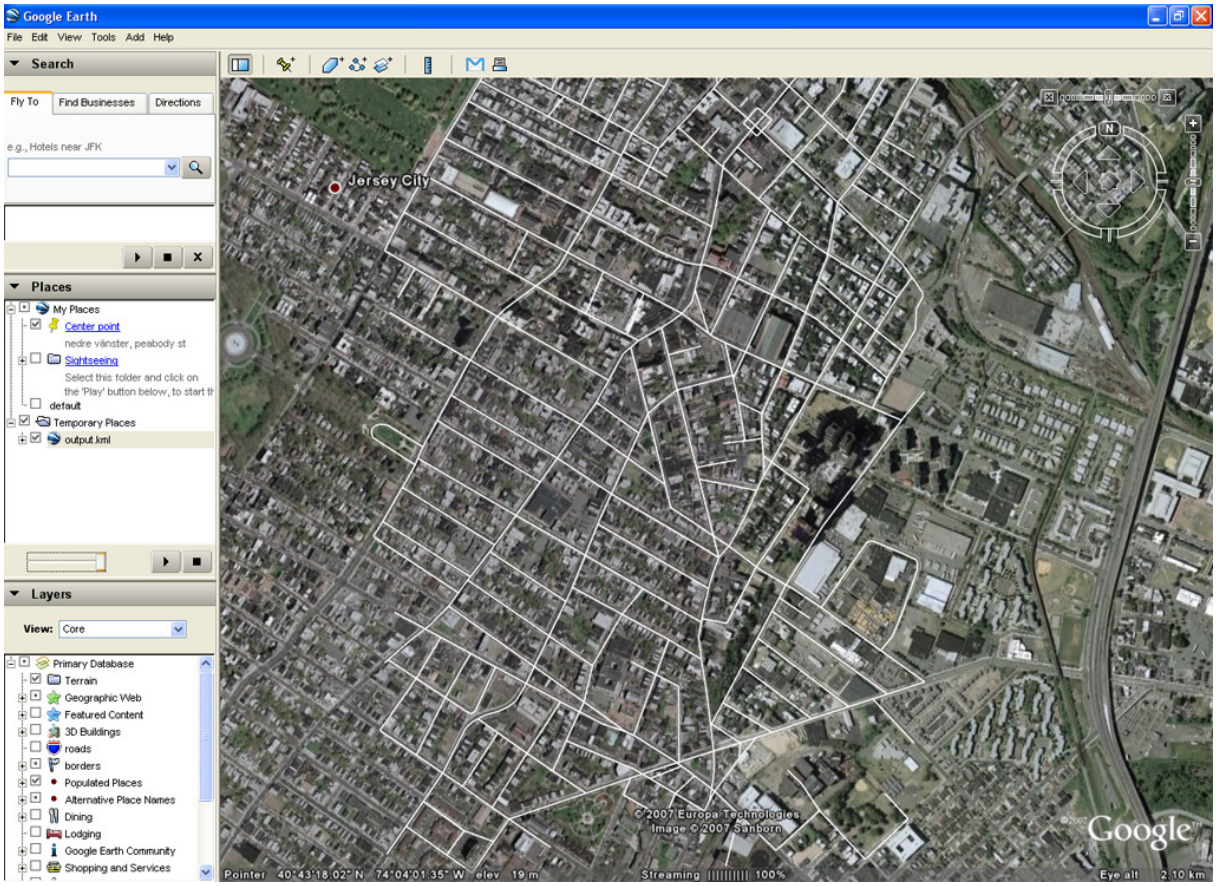



*Figure 8: Final version showing a section of Jersey City in the New York metropolitan area*

Another problem that was present during several stages of the project was various rounding errors. The first and easiest to find was a calculation done when writing to the KML file. The calculation was rounded off since it was not explicitly cast to a double. The second rounding error that was a bit trickier to find was that the floating point precision was wrong, but after setting the right number of decimals the roads lined up well with the photograph.

A final issue that we needed to consider was the change in TIGER file structure presented in the new edition of the database. This second edition was presented after we had completed the

entire conversion program and the first edition files that we were using would be removed shortly after. Thus after going through the new documentation and some notes on the difference in structure we concluded that no alterations to the code would be necessary.

## 3.4 Usage

The conversion utility is a console application that takes a filename as input, it should be the record to be converted (e.g. TGR47037). The following two coordinates are the coordinates of the box and the next couple of coordinates specifies how long the sides should be. This will output all the roads within the box as a KML file. Usage should thus look like this:

*TGRConv filename coordinateLongitude coordinateLatitiude devLongitude devLatitude*

# 4 Modifying the mobility model

In this chapter we will primarily focus on the mobility model and our modifications to it. First the mobility model is described, how it works and where we made our modifications. Then we will see how the KML files are read, processed and integrated into the mobility model. Specifically, we will take a closer look at the class that deals with the KML files, *kmlReader*, responsible for reading and converting KML-data for use in the model.

## 4.1 The Mobility Model

The mobility model [12, 13] we extended is one in a collection of models by Professor Jean-Yves Le Boudec at EPFL, Switzerland and Milan Vojnovic at Microsoft Research, Cambridge. It consists of three parts. The first part contains random waypoint and random walk with wrapping and reflection, the second part contain restricted random waypoint on a city section and, finally, the last part contain random waypoint on a generalized domain. In its simplest form a random waypoint model picks a path in a set of paths, according to some algorithm, and upon reaching its end chooses a new one.

The models we were using are more realistic than this basic example, and as a result, not quite as simple. For example, the models have perfect sampling, which means that they start in a steady state, avoiding any initial deviations. We will, however, not examine the complex mathematical background for the models, but instead focus on the implementation. The implementation we examine is that of the model we will be using and extending, random waypoint on a city section. See Figure 9 for a class diagram over relevant classes.

In its unmodified form the random waypoint on a city section model basically operates on a set of paths, the city section, which is passed to it as a file. The city section is then populated by nodes whose movements are simulated by the model and a file is outputted to be used in the ns-2 simulator. Other data of interest passed to the model is the length of the simulation (length in seconds), the number of nodes, vehicles, pause times (how long nodes will be stationary) and an output filename (were the output is written).

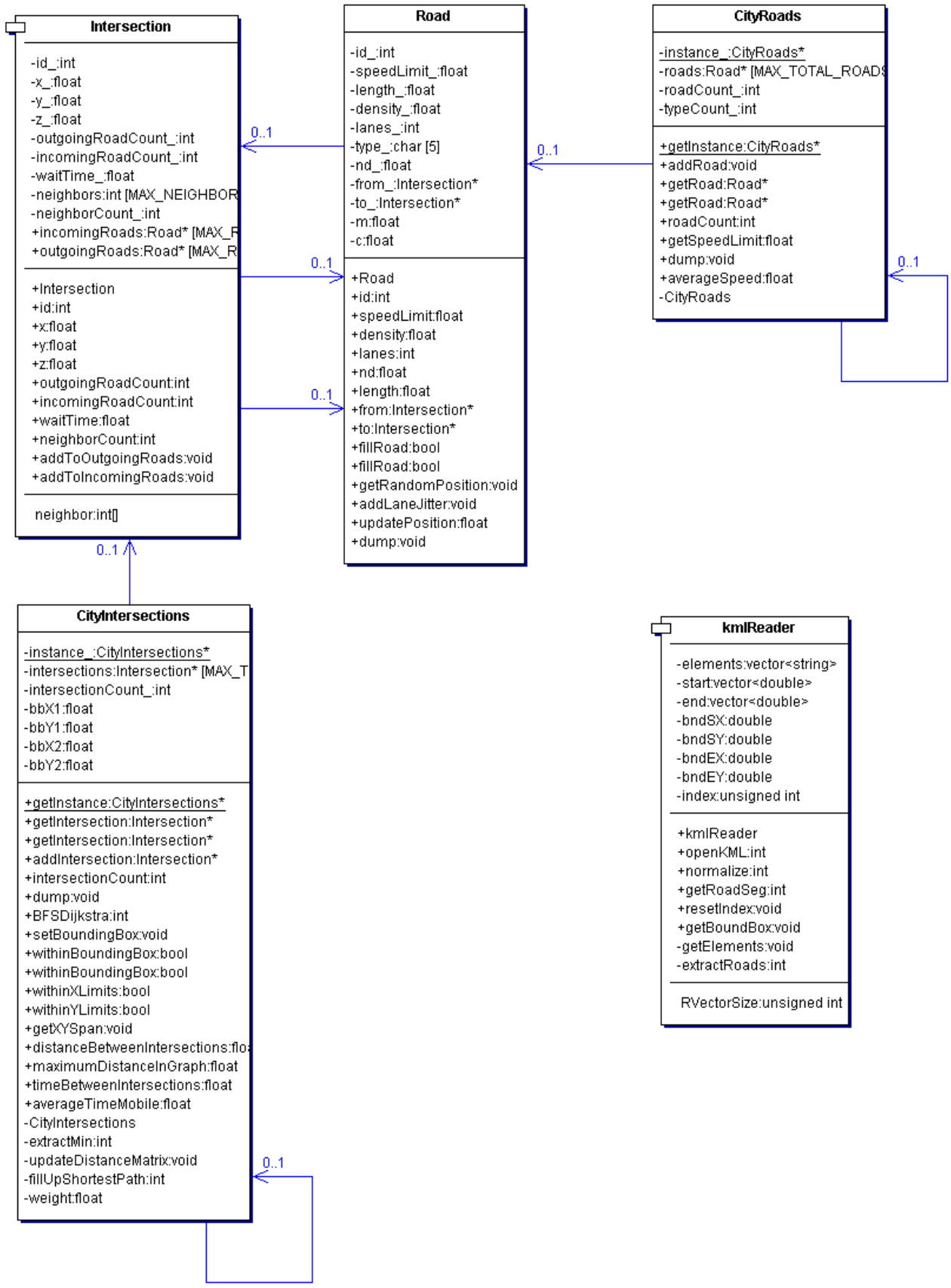



*Figure 9 Class diagram for relevant parts of the mobility model*

The first part we altered was the reading of the paths by creating a new class, KmlReader, used to obtain data from KML files, described in detail in the next section (See Figure 9 for an overview of the class). The city section is set up by passing a file pointer to the *addRoad* method in the singleton[7] class *CityRoads* class, responsible for storing the paths. The method will then read one path from the file and store it. Thus, what we initially intended to do was to override the addRoad-function to take parameters from KmlReader. This did however seem to be problematic (see section 4.4) so we had to rethink our approach. The solution that we finally decided upon was to use a temporary file in order to maintain the precision of our data.

Therefore, what we had to do was to modify the initialization function to take other arguments passed to the file. We introduced a new flag, *-kml*, which indicates that the file passed to the model is a KML file. Then it was opened with KmlReader and stored in our temporary file, which in turn was passed to the addRoad method. The method assumes that all roads are bidirectional, storing one road in each direction (of course depending upon if they are within the bounding box) in an array of Road objects. The Road class in turn is, in comparison, simple. It stores information on roads such as length, speed limit etc. while also containing pointers to intersections (of class Intersection, not described here) showing where the road starts and ends.

At this point another consideration that had to be made was how to handle speed limits. Since we do not get any information on speed limits from the KML files we had to use a fixed speed limit. The speed limit used throughout the example files for the model was 5.0 so we decided to continue using it.

The file structure which the mobility model use looks like this:

*junk limit fromX fromY toX toY*

We are not entirely certain what the “junk” values are, but we have a theory. The mobility model had extracted some roads from TIGER records using some simple script (although they probably didn’t link in other road data from other record types) for use in the examples. The junk data seem very similar to the road id’s that is used to link together data from the different

[7] A design pattern that is used to limit the instantiation of the class to only one object.

record types. The other fields are fairly straightforward, limit is the speed limit and the others are coordinates.

Finally, as an example, a line in the file looks like the one below (our junk value is a value borrowed from the examples, but could be anything since it is discarded by the model):

*96062880 5.0 907.100 1006.000 907.600 1006.000*

Now we were ready to produce the first output from our KML-files. It looked fine, but it turned out that the nodes were not moving as they should, instead the nodes were alternating between two intersections. The cause was, quite simply, that all roads were not read. This had to be modified further by altering the reading process to count the number of lines that were written to the temp-file instead of have the number of lines as a parameter to the model.

After the initialization phase came the next section that we discovered needed changes, the *generateScenario* function. The generateScenario, as the name implies, is responsible for actually generating the scenario. For example, it is responsible for placing vehicles on the map. In the function there were some values on maximum distances and other parameters that were hardcoded for the map, *westUnivPlace*, that the model was using. Therefore, this generation needed to be done automatically somehow. It turned out that there were functions in one of the classes that were intended to generate these values. It increases the time it takes to generate the output significantly, but it is needed to be able to use other maps.

## 4.2 KmlReader

KmlReader is the class we wrote to retrieve and format KML data for use in the mobility model. It takes a KML-file as input and outputs the data in a format usable in the mobility model. It is used for several tasks in our model, for example to establish a correct bounding box (a box that contains every coordinate of the roads) or to get the next road start- and endpoints.

The first step was to parse the information in the KML file, i.e. to extract all the tags and data,

which is done when the file is loaded. All the elements of the KML file is split and stored in a vector so that it can be easily accessed when we need to search it for specific data. Thus, the vector can now be processed for the wanted information, for example roads.

In the case of roads, the parsing is done in two steps, where the last step is possible to skip. First, the road segments are extracted from the vector and the start- and endpoints are ordered and stored. Every segment is padded with the value -1 since the length of segments varies. To clarify, a segment might look something like this:

| | | | |
|---|---|---|---|
| -86,766319 | -86,766314 | -1 | |
| +36, 161582 | +36,161582 | -1 | |

*Figure 10: Section of the road vector*

This segment contains only one road, from (-86,766319, +36,161582) to (-86,766314, +36,161582) and will be read until one segment contains -1. At this point it is possible to start retrieving coordinates. They are, however, still in Google Earth's format which the mobility model is not designed to handle. Furthermore, the bounding box is not yet determined. Therefore, we want to convert the data once more.

All calculations on the coordinates were now performed. First off, we did not want negative values so 180 (degrees) was added to all coordinates. Then the smallest and largest values were retrieved by traversing the road vectors. With the data now gathered we could transform the coordinates and determine the bounding box. The coordinate values are adjusted by the following formula:

finalCoordinate = (oldCoordinate – smallestValue) * multiplier + 1

The formula is fairly straightforward, but the multiplier might need some explanation. It is a multiple of 10 used to get the largest value to somewhere between 100 – 2500 in order to avoid working on large floating point values. Thus, what is done is that the smallest value is subtracted from our coordinate (making the smallest value 0) and then multiplied by the *multiplier*. Every field containing -1 is skipped since it shouldn't be altered. 1 is added in the end to avoid having fields with 0. We wanted values ranging from 1 – 2500 (2500 being the currently specified maximum value).

After the calculations were completed we had all coordinates ranging from 1 to 2500 which also conveniently provided us with the bounding box values. Bounding box values are retrieved by taking the largest and smallest values of X and Y, giving us a box which contains all coordinates. The data was now ready for use in the mobility model.

## 4.3 Problems and Solutions

Once again we had problems with the precision of our calculations, this time in our KML reader. It turned out that several different coordinates, for example -86.766319 and -86.766314, would eventually be converted to the same number, in this case 899. The difference, albeit very important, was only 0.000005, which initially led us to believe that the error came from the numbers being rounded off at some point.

It did, however, turn out that the lack of precision came earlier; when the roads were extracted the last digit was lost. The problem turned out to be that the last digit was cut from the coordinate when they where extracted from the KML vector.

Another precision problem came when we were about to integrate the KmlReader class with the mobility model. Throughout the model we handled data as doubles, to maintain precision, and not floats as was used in the mobility model. We encountered an error when we tried to cast our values to floats using *static_cast*. It turned out that the value was strangely rounded off. For example, a double value such as 170.0000 would become 169.9999 when casted to float. We tried a few alternate solutions but we just didn't have the time to get stuck. The solution, as outlined in section 4.2, was to dump our data to a file and let the original function handle it.

The last problem, not related to precision, came after we realised that the output the model was producing didn't actually do anything useful. The problem seemed to be that all 100 vehicles jumped between two coordinates. What had happened was that it only read as many lines as specified by the parameters passed to the function. Since the temp file was generated while the model was running, it had to be done automatically. As outlined in section 4.2, we

modified the function to make it read all lines from the temp file. It was simply fixed by adding a counter as the KML reader wrote lines to the file and then letting the addRoad function read the same amount of roads.

# 5 NS-2 simulator

In this section we explain what happened after we got the input from the mobility model, running it through the ns-2 simulator. We also give examples of simulation as viewed in NAM as well as how the ns-2 input is structured. Finally, we take up the problem we had with these different parts.

## 5.1 Simulations

After the output from the mobility model had been made a simulation in ns-2 was possible. From the simulation output you can get all sorts of information about the simulation of this network. We concentrated on showing the simulation in NAM. When the simulation is complete you have, among other things, an output file that is a NAM file, in our case the filename was *nam-out.nam*. When you put the file into NAM you can watch the simulation that was done.

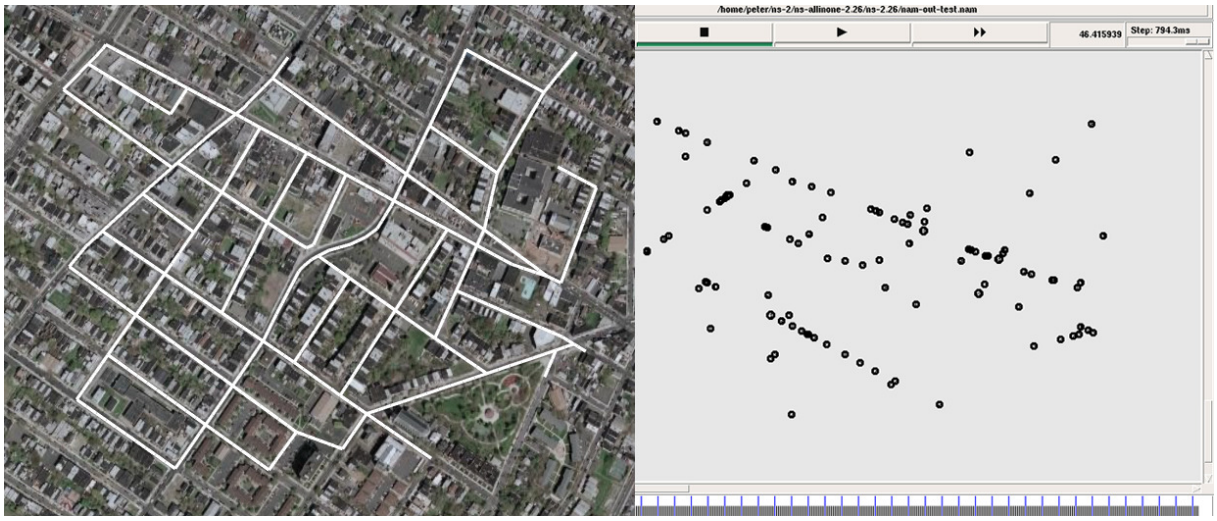



*Figure 11: City section in Google Earth with NAM visualization on the right*

From the mobility model we got node movements as output. This data was used as input for the simulation. The files start out by setting initial coordinates for all nodes, which can look like what is seen in Figure 12.

```
$node_(0) set X_ 492.239441
$node_(0) set Y_ 368.665100
$node_(0) set Z_ 0.000000
$node_(1) set X_ 225.169586
$node_(1) set Y_ 252.758728
$node_(1) set Z_ 0.000000
$node_(2) set X_ 420.013031
$node_(2) set Y_ 151.794113
$node_(2) set Z_ 0.000000
$node_(3) set X_ 77.541893
…
```

*Figure 12: initial part of mobility model output*

After the initialization the node movements are listed, ordered chronologically, starting with 0. The length of the simulation is specified by the parameter passed to the mobility model. The movement-part of a mobility output can look like this:

```
$ns_ at 0.000000 "$node_(0) setdest 517.700012 410.299988 6.312355"
$ns_ at 0.000000 "$node_(1) setdest 221.162842 254.835297 0.451288"
$ns_ at 0.000000 "$node_(2) setdest 420.851898 151.359970 0.094456"
$ns_ at 0.000000 "$node_(3) setdest 78.500000 590.099976 2.035775"
$ns_ at 0.000000 "$node_(4) setdest 761.414429 339.027527 8.195236"
$ns_ at 0.000000 "$node_(5) setdest 703.400024 334.299988 5.626622"
$ns_ at 0.000000 "$node_(6) setdest 488.524384 362.589996 5.590453"
$ns_ at 0.000000 "$node_(7) setdest 511.500000 29.900000 1.522174"
$ns_ at 0.000000 "$node_(8) setdest 713.538818 330.173553 0.024292"
```

*Figure 13: movement-part of mobility model output*

Using a Tcl script the simulation was run using the node movements specified by the model. After the simulation was performed we got the NAM file that was relevant for our work. The result can be seen in Figure 11. In NAM we can now examine all the nodes and see them move around.

## 5.2 Problems

From the first instructions we got it was said that we wouldn't need to use the ns-2 simulator but when we wanted to check that our output was correct in NAM. We noticed that we couldn't use NAM without running it through the ns-2 simulator. Therefore, first we downloaded ns-2 simulator 2.26. We tried to unpack the simulator on the school computers but got the message that the disk quota was exceeded, we had 50MB and needed around 300MB, or so we thought at the time. We asked for more capacity and got some but it wasn't

enough. Because neither of us has a computer with Linux on we got an account on an OpenBSD[8] server and used SSH[9], but with this server we got several errors when we tried to run the makefile. We tried to switch the make to gmake[10] but still got many errors that we weren't able to sort out. We tried a different version of ns-2, one that would be easier to compile and run. However, still we faced a couple of errors that we didn't know how to solve.

After these trials we felt that the easiest way to go forth was to take a laptop and change the operating system to Linux so we installed Ubuntu[11] on a laptop. First we tried with the latest edition, Ubuntu 7.1, but the computer didn't boot the CD. After a few hours of trying we decided to change to an older version and installed Ubuntu 5.04 and then tried to run the ns-2 simulator again but the problems remained. Then we downloaded a newer version of ns-2 from sourceforge which had a configure file in it that could say what was missing to make the simulator work.

When we finally did get ns-2 to compile a new problem arose, we couldn't find out how to make a complete simulation. First we tried to use our own file from the mobility model as input but only got error messages. Then we tried to understand the ns-2 documentation but we didn't find any example of how to start a simulation. At last we did send a file that we had created to Dr. Andreas Kassler, whom had given us the project, and he made a simulation with our file. We then got a script to the ns-2 simulator that would work with our output files from the mobility model.

When we tried the script we noticed that we needed to change back to the first version of ns-2 that we had got, because the script was made to that version. After that we were able to run our simulations.

[8] A Unix-like operating system based on the *Berkley Software Distribution*.
[9] *Secure Shell*, a network protocol for secure communication.
[10] GNU Make. A Unix make utility, used to automatically build projects.
[11] A Linux distribution based on Debian.

# 6 Conclusions and future work

The project was completed with most of the goals fulfilled. We created a visualization of the city section in Google Earth, converted the data for use in the mobility model, which in turn outputted meaningful data to the simulator. We encountered several problems along the way, notably several problems with rounding errors, limitations of the mobility model and problems working with the ns-2 simulator.

There were, however, some features that was initially intended to be included that didn't make it into the final utility, the most important being that users cannot place nodes on the actual map in Google Earth. The reason we didn't implement it was mainly because of the mobility model. In its unmodified form the model takes nodes as a parameter. The number of nodes, or vehicles, populating the city section are passed as a parameter and then randomly placed on the roads. There were also some form of initialization where the nodes where moved around for a while to remove any initial deviations in the simulations. Therefore, we decided to skip the node placement on the maps since they would be moved around randomly in the initialization anyway. We were also asked to add stationary nodes, but there just wasn't enough time to alter the model to handle stationary objects.

The previously mentioned problems with rounding errors came at the stages where data needed to be converted between the various formats. At points, they were casted, read from file and processed in other ways. This often left us with smaller deviations that would leave the roads disconnected, not forming perfect edges for example.

Future additions that would really improve the user friendliness of the road layer construction is some kind of improvement to the way the roads are extracted from the TIGER database. As for now, the user needs to manually specify the coordinates of the "box" from which all roads are extracted. This could be simplified by perhaps having set markers in Google Earth that can be read or have some simple GUI to handle it.

Other parts that could be improved are the previously mentioned node placement. It might require considerable work to modify the model, but it would definetely add to the usefulness

of the application to be able to see where nodes are placed, particularly eventual stationary nodes.

Finally, a nice addition could be to visualize the output from the simulator in Google Earth in some way. For example, to have each node's movement as a separate post, similarly to how road segments are now visible (see Figure 6). You would then be able to in turn set every node to "on", showing the path taken throughout the simulation.

# 7 References


[1]. Elliotte Rusty Harold, *XML Bible 1.1*, 3rd Edition, Wiley 2004, ISBN: 0-7645-4986-3

[2]. Harvey M. Deitel, Paul J. Deitel, Tem Nieto, Ted Lin, Praveen Sadhu, *XML: How to program*, Prentice Hall 2001, ISBN: 9780130284174

[3]. James F. Kurose, Keith W. Ross, *Computer Networking A Top Down Approach Featuring the Internet*, 3rd edition, Addison Wesley 2005, ISBN: 0-321-17644-8

[4]. C. Siva Ram Murthy, B.S. Manoj, *Ad Hoc Wireless Network Architectures and Protocols,* ISBN: 9780131470231

[5]. Google KML documentation, http://earth.google.com/kml/

[6]. English Wikipedia, http://en.wikipedia.org/wiki/Keyhole_Markup_Language, 2007-05-20

[7]. English Wikipedia, http://en.wikipedia.org/wiki/Google_Earth, 2007-05-20

[8]. U.S. Census Bureau TIGER technical documentation, http://www.census.gov/geo/www/tiger/tiger2006fe/TGR06FE.pdf

[9]. BonnMotion documentation, http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion/README

[10]. ns-2 official webpage, http://www.isi.edu/nsnam/ns/

[11]. English Wikipedia, http://en.wikipedia.org/wiki/Ns-2, 2007-05-20

[12]. NAM official webpage, http://www.isi.edu/nsnam/nam

[13]. Spatial Data Transfer Standard (see 2.3.2.5.1 Complete chain), http://mcmcweb.er.usgs.gov/sdts/SDTS_standard_nov97/part1toc.html

[14]. Mobility model source code and brief description of parts, http://www.cs.rice.edu/~santa/research/mobility/

[15]. Power Point presentation on the mobility model, http://ica1www.epfl.ch/perfeval/slides/leb-perf05.ppt

# A Appendix

## A.1 Mobility Model

### A.1.1 main.cpp

```
/*****************************************************************************
Santa @ EPFL - Summer 2004
[Code based on vehicular mobility model. Amit Saha(amsaha@rice.edu)
VANET'04]

The graph mobility model : 6 Aug
- Pause and movement alternates
- steady speed on a road around the speed limit for that road
- steady state initilization
*****************************************************************************/

#include "road.h"
#include "intersection.h"
#include "vehicle.h"
#include "rng.h"
#include "statistics.h"
#include "common.h"

//kmlReader is used to read KML-data
#include "kmlReader.h"
#include <cstring>

void generateScenario(int vehicles);
void initialize(int argc, char * argv[]);
double steadyStatePauseProbability(float);
double residualTime(double mean, double delta);

RNG *rng = 0;
FILE *scenFptr = 0;
FILE *speedPtr = 0;
FILE *posPtr = 0;
float maxSimulationTime = 0.0;
float pauseMean, pauseDelta ;

int main(int argc, char *argv[])
{
          initialize(argc, argv);
          generateScenario(atoi(argv[5])); // change for bounding box
default
          return 0;
}

void initialize(int argc, char * argv[])
{

          if(argc != 8) {
                    if (argc != 9){       // -kml
```

```
                                        fprintf(stderr, "argc: %d, ",argc);
                                        fprintf(stderr, "Usage: %s",argv[0]);
                                        fprintf(stderr, " <input file>");
                                        fprintf(stderr, " <number of lines in
input file>");
//                                      fprintf(stderr, " <Bounding box lower x
coordinate>"); // argv[3]
//                                      fprintf(stderr, " <Bounding box lower y
coordinate>"); // argv[4]
//                                      fprintf(stderr, " <Bounding box upper x
coordinate>"); // argv[5]
//                                      fprintf(stderr, " <Bounding box upper y
coordinate>"); // argv[6]
                                        fprintf(stderr, " <pause time mean> <pause
time delta>");
                                        fprintf(stderr, " <number of nodes>"); //
argv[5]
                                        fprintf(stderr, " <simulation time
(sec)>"); // argv[6]
                                        fprintf(stderr, " <output file>\n"); //
argv[7]
                                        fprintf(stderr, " set kml mode with
optional -kml\n"); // argv[8]
                                        fprintf(stderr,"Exiting...\n");
                                        exit(1);
                        }
                }

                bool KMLmode;
                if (argc == 9){
                        if (strcmp("-kml", argv[8]) == 0)
                                KMLmode = true;
                }
                else
                        KMLmode = false;

                kmlReader kml;                  //only used if KMLmode == true
                FILE *fptr = fopen(argv[1],"r"); //only used if KMLmode == false
                if (KMLmode){
                        if (kml.openKML(argv[1]) == -1){
                                fprintf(stderr, "could not open KML\n");
                                exit(1);
                        }
                        else
                                kml.normalize();
                }
                else{
                        if(!fptr) {
                                fprintf(stderr,"Input file %s does not
exist !!!\n", argv[1]);
                                fprintf(stderr,"Exiting...\n");
                                exit(1);
                        }
                }

                /* Set the bounding box for the map */
                // exact needed only for statistics. We initialize to default
//              CityIntersections::getInstance()->setBoundingBox(atoi(argv[3]),
atoi(argv[4]), atoi(argv[5]), atoi(argv[6]));
                //CityIntersections::getInstance()->setBoundingBox(0, 0, 100,
100);
```

```
            if (KMLmode){
                        double bx1, by1, bx2, by2;
                        float  fx1, fy1, fx2, fy2;
                        kml.getBoundBox(bx1, by1, bx2, by2);

                        fx1 = static_cast<float>(bx1); fy1 =
static_cast<float>(by1);
                        fx2 = static_cast<float>(bx2); fy2 =
static_cast<float>(by2);
                        //CityIntersections::getInstance()-
>setBoundingBox(fx1, fy1, fx2, fy2);
                        CityIntersections::getInstance()->setBoundingBox(0,
0, 1200, 1200); //ONLY for testing, fix
            }
            else
                        CityIntersections::getInstance()->setBoundingBox(0,
0, 2499.99, 2499.99);

            // read in the pause time distribution
            pauseMean  = atof(argv[3]);
    pauseDelta = atof(argv[4]);

            if(pauseMean<0)
            {
                        fprintf(stderr,"Error: pause mean must be greater
than or equal to 0\n");
                        exit(1);
            }
            if((pauseDelta>pauseMean)||(pauseDelta<0))
            {
                        fprintf(stderr,"Error: pause delta must be greater
than or equal to 0 and less than or equal to pause mean\n");
                        exit(1);
            }

            if (KMLmode){
                        /* Read in the roads for the map, from KML file */
                        double startX, startY, endX, endY;
                        float fromX, fromY, toX, toY;
                        int temp;
                        FILE *tempFptr = fopen("temp","w");
                        if (tempFptr == NULL){
                                    printf("Could not create temporary file,
exiting...\n");
                                    exit(0);
                        }

                        int count = 0;
                        while (kml.getRoadSeg(startX, startY, endX, endY) !=
-1){
                                    /*
                                     * It is not an optimal solution to write
to a temp file, but under the time
                                     * constraints it was the only way we
could eliminate the rounding errors int time.
                                     * See C-diss for more information
                                     */
                                    fprintf(tempFptr,"96062880 5.0 %f %f %f
%f\n", startX, startY, endX, endY);
                                    count++;
                        }
```

```
                        fclose(tempFptr);
                        tempFptr = fopen("temp", "r");
                        if (tempFptr == NULL){
                                printf("could not open temp file for
reading, exiting\n");
                                exit(0);
                        }
                        while(count > 0){
                                CityRoads::getInstance()-
>addRoad(tempFptr);
                                count--;
                        }

            }
            else{
                        /* Read in the roads for the map */
                        int cnt = atoi(argv[2]);
                        while(cnt > 0) {
                                CityRoads::getInstance()->addRoad(fptr);
                                cnt--;
                        }
                        fclose(fptr);
            }

            /* RNG class imported from ns-2.27 distribution in 'stand_alone'
mode i.e.
             * use -Dstand_alone when compiling
             */
            rng = new RNG;
            rng->set_seed(RNG::HEURISTIC_SEED_SOURCE);

            // Get the simulation time
            maxSimulationTime = atof(argv[6]); // change for bounding box
default

            // Open the output file
            scenFptr = fopen(argv[7],"w"); // change for bounding box
default
#if STATISTICS
            //Open some speed/position stat file
            speedPtr = fopen("nodeSpeed.dat","w");
            posPtr = fopen("nodePosition.dat","w");
#endif
#if TESTING2
            fprintf(stdout,"\n\nRoads...\n\n");
            CityRoads::getInstance()->dump(stdout);
            fflush(stdout);

            //fprintf(stdout,"\n\nIntersections...\n\n");
            CityIntersections::getInstance()->dump(stdout);
            fprintf(stdout,"\n");
            fflush(stdout);
#endif
}

/* Generate random src destination pairs for vehicles. Basically a vehicle
 * starts at a src and goes towards the destination. Dijkstra's BFS is used
to
 * find out the route from the src to the dest and then the scenario for a
 * vehicle is generated.
```

```
 */

void generateScenario(int vehicles)
{

#if PRINT_NS2_SCENARIO
           fprintf(scenFptr,"#\n");
           fprintf(scenFptr,"# Number of nodes: %d\n", vehicles);
           fprintf(scenFptr,"# Pause Time, Mean & Delta: %f %f\n",
pauseMean, pauseDelta);
           fprintf(scenFptr,"# Number of roads: %d\n",
CityRoads::getInstance()->roadCount());
           fprintf(scenFptr,"# Number of intersections: %d\n",
CityIntersections::getInstance()->intersectionCount());
           fprintf(scenFptr,"#\n");
           fflush(scenFptr);
#endif
           CityIntersections* instance = CityIntersections::getInstance();

// Find the steady state
           //float avgSpeed = CityRoads::getInstance()->averageSpeed();
           //double q0 = steadyStatePauseProbability(avgSpeed);
           // For roads with varying speed
#if STEADY_STATE
           // IMP: The following calc takes time, and is fixed for a
specific
           // graph and road speeds.....Hence reuse.

           float avgMobile = instance->averageTimeMobile();
           // For 1200x1200 westUniv. speed 0.01-9.99. avg-5
           //float avgMobile = 161.3248205;

           double q0 = pauseMean / (pauseMean + avgMobile);
           printf("AverageTimeMobile %f\n",avgMobile);
           printf("SteadyStateProb %f\n",q0);
           fflush(stdout);

           float maxGraphDistance, distBwPoints;
           // calculate the MAX delta
           // IMP: again, following takes long time and is fixed for
graph.
           // Hence: reuse

           maxGraphDistance = instance->maximumDistanceInGraph();
           //exit(1);
           // for 1200x1200 westUniv
           //maxGraphDistance = 2192.9275;

           printf("Maximum Distance in Graph: %f\n",maxGraphDistance);
#endif

           bool mobile;
           float pauseTime, unirand;
           int src, dst;


           for(int i=0;i<vehicles;i++)
           {
#if STEADY_STATE
                    // Steady state initialize here:
```

```
                        // Pause or move, // src & dst, // position // speed
or residual pause time
                        // Create a new vehicle. src and dest will be later
steady stated
                        while(1){
                                    if(rng->uniform() < q0){
                                                mobile = false;
                                                break;
                                    }
                                    else{
                                                src = rng->uniform(instance-
>intersectionCount()) ;
                                                dst = rng->uniform(instance-
>intersectionCount()) ;
                                                /* We dont want src and dst to
be the same !!! */
                                                while(dst == src)
                                                  dst = rng->uniform(instance-
>intersectionCount()) ;
                                                unirand = rng-
>uniform((double)maxGraphDistance);

                                                // find dist between src and dst
                                                distBwPoints = instance-
>distanceBetweenIntersections(src,dst);
                                                if(unirand < distBwPoints){
                                                            mobile = true;
                                                            break;
                                                }
                                    }
                        }
// TODO: testing. start all nodes mobile
                        if(!mobile){ // node starts paused
                                    pauseTime = residualTime(pauseMean,
pauseDelta);
                                    src = rng->uniform(instance-
>intersectionCount()) ;
                                    CityVehicles::getInstance()-
>addVehicle(src,pauseTime, 0.0);
                        }
                        else{ // node starts mobile
                                    // STEADY STATE SPEED WITHIN addVehicle
                                    CityVehicles::getInstance()-
>addVehicle(src,dst, 0.0);
                        }

#else   // FOR NON STEADY STATE CASE
                        // NODE STARTS MOBILE
                        /* Generate a pair of random numbers and take the
modulo of the number
                         * of intersections.  */

                        src = rng->uniform(instance->intersectionCount()) ;
                        dst = rng->uniform(instance->intersectionCount()) ;
                        // We dont want src and dst to be the same !!!
                        while(dst == src)
                                    dst = rng->uniform(instance-
>intersectionCount()) ;
                        // Create a new vehicle with this random src and
dest.
```

```
                        CityVehicles::getInstance()->addVehicle(src,dst,
0.0);

/*                      // NODE STARTS STATIC
                        src = rng->uniform(instance->intersectionCount()) ;
                        float pt = 0.1;
                        // make it paused for a fraction time before
moving....
                        CityVehicles::getInstance()->addVehicle(src,pt, 0.0);
*/
#endif
            }

//
*****************************************************************************
// FROM NOW, STEADY AND NON STEADY STATE ARE EXACTLY EQUAL ..............
//
*****************************************************************************
// Now that we have all the vehicles we will start moving these vehicles.
// Our time will proceed at timesteps of DEFAULT_TIME_STEP
            bool reachedDestination;
            Vehicle* currentV;

            int intTime = 0;

            for(float time = 0.0; time < maxSimulationTime; time +=
DEFAULT_TIME_STEP)
            {
            printf(" ********* Current time = %f ******************\n",
time);
                        //fprintf(speedPtr,"TIME = %d\n",intTime);
                        //fprintf(posPtr,"TIME = %d\n",intTime);

                        for(int i = 0;i<CityVehicles::getInstance()-
>vehicleCount();i++)
                        {
                                    currentV = CityVehicles::getInstance()-
>getVehicle(i);
                        // PRINT STATS about SPEED and COORDINATE EVERY 10
seconds
                                    intTime = (int) time;
#if PRINT_STATISTICS
                                    //printf("intTime=%d
intTime%10=%d\n",intTime,intTime%10);
                                    //if(intTime%10 == 0) {
                                                fprintf(speedPtr,"#%d %d
%f\n",intTime, i, currentV->speed());
                                    //          fflush(speedPtr);
                                    //}
                                    //if(intTime%10 == 0) {
                                                fprintf(posPtr,"#%d %d %f
%f\n",intTime, i, currentV->x(),currentV->y());
                                    //          fflush(posPtr);
                                    //}
#endif
                                    if(!currentV->mobile()){// node is paused
                                                currentV->pauseTime() -=
DEFAULT_TIME_STEP;
#if TESTING

            fprintf(stderr,"pause=%f",currentV->pauseTime());
```

```
#endif
                                                if(currentV->pauseTime() < 0){

            //printf("pause=%f",currentV->pauseTime());
                                                        currentV->pauseTime()
= 0.0;
                                                        currentV->mobile() =
true;
                                                        currentV-
>reset(time+DEFAULT_TIME_STEP, true);
                                                }
                                    }
                                    else {// node is mobile
                                                reachedDestination = currentV-
>update(time, DEFAULT_TIME_STEP);
                                      if(reachedDestination)
                                      {
                                                // Choose another random
destination with the current
                                                // destination as source and
then start moving towards it.
                                                float pauseTime = pauseMean +
rng->uniform(-pauseDelta, pauseDelta);
                                                currentV->reset(pauseTime,
false);
                                      }
                                    }
                        }
#if PRINT_STATISTICS
                        fprintf(posPtr,"\n");
                        fprintf(speedPtr,"\n");
#endif

#if TESTING2
                        // Calculate all the metrics for this mobility
scenario at this time instant.
                        CityVehicles::getInstance()->dump(stderr);
#endif
#if PRINT_STATISTICS
                        Statistics::getInstance()->calculateStatistics(time);
#endif
            }
//#if PRINT_STATISTICS
//          Statistics::getInstance()->printGrid();
//#endif
            return;
}

// The steady state pause probability:
// check calculation for roads with variable speed
double steadyStatePauseProbability(float avgSpeed)
{
        double alpha1, q, delta1;
            float pauseHigh = pauseMean + pauseDelta;
            float pauseLow = pauseMean - pauseDelta;
            float speedHigh = avgSpeed +
CONVERSION_FACTOR*SPEED_LIMIT_LEEWAY;
            float speedLow = avgSpeed -
CONVERSION_FACTOR*SPEED_LIMIT_LEEWAY;
```

```
            alpha1 = ((pauseHigh+pauseLow)*(speedHigh-
speedLow))/(2*log(speedHigh/speedLow));
            // find the bound on max distance

        //delta1 = sqrt((maxX*maxX) +(maxY*maxY));
        q = alpha1/(alpha1+delta1);
        return q;
}

// The residual time left for uniform distribution
double residualTime(double mean, double delta)
{
        double residual;
        double t1 = mean - delta;
        double t2 = mean + delta;
        double u = rng->uniform();
        if(delta!=0.0)
        {
                if(u < (2*t1/(t1+t2)) )
                {
                  residual=u*(t1+t2)/2;
                //fprintf(stdout, "# Case 1 u: %f ", u);
                }
                else
                {
                 residual=t2-sqrt((1-u)*(t2*t2 - t1*t1));
                 //fprintf(stdout, "# Case 2 u: %f ", u);
                }
        }
        else
                        residual=u*mean;
#if TESTING
        printf("# Initial residual Time: %f\n",residual);
#endif
        return residual;
}
```

### A.1.2 Road.h

```
#ifndef _ROAD_H_
#define _ROAD_H_

#include "common.h"


/* Forward declaration of class Intersection
 */
class Intersection;

/* This class denotes a Road between two nodes
 */
class Road
{

            /* The unique identifier for this road */
            int id_;

            /* The speed limit on this road in m/s (compatible with ns2)
             * We have to be politically correct you know.
             */
```

```
            float speedLimit_;

            /* The length of the road
             */
            float length_;

            /* Average density of nodes on this road
             */
            float density_;

            /* Width of the road in lanes */
            int lanes_;

            /* Type of road */
            char type_[5];

            // STEADY STATE Nominal Distance
            float nd_;

            /* This road is a road from node 'from' to node 'to'.
             * This means that roads are directional and hence for a two way
             * road the road has to be
             */
            Intersection *from_, *to_;

            /* variables for describing the line of the road (y = mx+c form)
             */
            float m;
            float c;

public:

            Road(int id);

            /* Accessor functions */
            int id();
            float speedLimit();
            float density();
            int lanes();
            float nd();
            float length();
            Intersection* from();
            Intersection* to();
#if 0
            // int vehicleCount();

            // Return vehicles[index]
            // int getVehicle(int index);

            // Add a vehicle to this road
            // void addVehicle(int index);
#endif

#if 0
            /* Read line */
            bool readLine(FILE *fptr);
#else
            bool fillRoad(const char *type, float fromX, float fromY, float
toX, float toY);
            bool fillRoad(float limit, float fromX, float fromY, float toX,
float toY);
```

```
#endif

            /* Generate a random position on the road */
            void getRandomPosition(float& x, float& y, float& z);

            // Add lane jitter
            void addLaneJitter(float& x, float& y);


            // returns the time simulated. Normal case will return time ==
remainingTime
            // however, if the end of the road is reached the a time <
remainingTime is
            // returned. x and y are starting positions and after moving
towards the
            // 'to' end of the road with 'speed' speed and for
'remainingTime' time the
            // x,y coordinates are returned through the reference variables
x and y.
            float updatePosition(float& x, float& y, float remainingTime,
float speed);

            /* Print this road */
            void dump(FILE *fptr);

};

class CityRoads
{
            public:
                        static CityRoads* getInstance()
                        {
                                    if(instance_ == 0)
                                    {
                                                instance_ = new CityRoads();
                                    }
                                    return instance_ ;
                        }

                        /* Add a road by reading it from the input file
denoted by 'fptr'
                         */
                        void addRoad(FILE *fptr);

                        /* return roads[index] */
                        Road* getRoad(int index);

                        /* return the road between intersections given by
indices 'from' and
                         * 'to'
                         */
                        Road* getRoad(int from, int to);

                        /* Return total number of roads */
                        int roadCount();

                        /* Get the speed limit from the type of road */
                        float getSpeedLimit(char *type);

                        /* Dump road stats */
                        void dump(FILE *fptr);
```

```
                        // STEADY STATE: get the average speed of city roads
                        float averageSpeed(void);

            private:
                        /* Singleton class */
                        CityRoads();
                        static CityRoads* instance_;

                        /* List of roads */
                        Road* roads[MAX_TOTAL_ROADS];

                        /* Total road count */
                        int roadCount_;

                        /* How many types of roads are present */
                        int typeCount_;

};


#endif
```

### A.1.3 Intersection.h

```
#ifndef _INTERSECTION_H_
#define _INTERSECTION_H_

#include <stdio.h>
#include "common.h"

/* Forward declaration of class Road
 */
class Road;

/* This class denotes a Node. In other words it denotes all those places
such as
 * intersections, parking lots, etc. where a road might end
 */
class Intersection
{
            /* Unique identifier of this intersection
             */
            int id_;

            /* The X Y and Z coordinates of the intersection
             */
            float x_;
            float y_;
            float z_;

            /* Number of outgoing roads
             */
            int outgoingRoadCount_;

            /* Number of incoming roads
             */
            int incomingRoadCount_;
```

```
            /* When a vehicle reaches at this node, it has to wait for these
many
             * minutes. For example if this node is simulating a parking lot
then the
             * waiting time is very large (say around an hour)
             * whereas if this is an intersection then the waiting time is
on the order
             * of minutes.
             */
            float waitTime_;

            /* In order to help with Dijkstra's algo we keep the neighbor
list. i.e.
             * other intersections that this intersection is directly
connected to
             */
            int neighbors[MAX_NEIGHBORS];
            int neighborCount_;

public:

            Intersection(float x, float y, float z);

            /* List of incoming roads
             */
            Road* incomingRoads[MAX_ROADS_AT_INTERSECTION];

            /* List of outgoing roads
             */
            Road* outgoingRoads[MAX_ROADS_AT_INTERSECTION];

            /* Accessor functions */
            int id();
            float x();
            float y();
            float z();
            int outgoingRoadCount();
            int incomingRoadCount();
            float waitTime();
            int neighborCount();


            /* Helper functions */
            void addToOutgoingRoads(Road *r);
            void addToIncomingRoads(Road *r);

            /* Get the i'th neighbor */
            int getNeighbor(int index);

};

class CityIntersections
{
            public:
                        static CityIntersections* getInstance()
                        {
                                    if(instance_ == 0)
                                    {
                                                instance_ = new
CityIntersections();
                                    }
```

```
                        return instance_ ;
                }

                // return intersection at x,y,z
                Intersection* getIntersection(float x, float y, float
z=0.0);

                // return intersections[index]
                Intersection* getIntersection(int index);

                // If no intersection present at x,y,z then add one,
else return pointer
                // to existing intersection.
                Intersection* addIntersection(float x, float y, float
z=0.0);

                // Accessor for intersectionCount_
                int intersectionCount();

                // Dump all roads
                void dump(FILE *fptr);

                // Compute dijkstra's shotest path between 'src' and
'dst' and fill the
                // indexes of the roads to take in 'directions'.
Check that the number
                // of roads does not cross 'maxRoads'. Return the
actual number of
                // roads.
                int BFSDijkstra(int src, int dst, int *directions,
int maxRoads);

                // Set bounding box coordinates
                void setBoundingBox(float x1, float y1, float x2,
float y2);

                // Check if coordinates within bounding box or not
                bool withinBoundingBox(float x1, float y1, float x2,
float y2);

                // check if coordinate is withing bounding box.
                bool withinBoundingBox(float x, float y);

                // Check if coordinate is within limits
                bool withinXLimits(float x);
                bool withinYLimits(float y);

                // get span of X and Y of bounding box
                void getXYSpan(float& x, float& y);

                // STEADY STATE
                float distanceBetweenIntersections(int src, int dst);
                float maximumDistanceInGraph(void);
                float timeBetweenIntersections(int src, int dst);
                float averageTimeMobile(void);

        private:
                /* Singleton class */
                CityIntersections();
                static CityIntersections* instance_;
```

```
                        /* List of intersections */
                        Intersection* intersections[MAX_TOTAL_INTERSECTIONS];

                        /* Total interesection count */
                        int intersectionCount_;

                        /* Extract min function for dijkstra's algorithm */
                        int extractMin(int *Q, int &qCount, float *distance);

                        /* The function for the updation of distance[] and
previous[]
                         * for dijkstra's algo
                         * This function is equivalent to the "RELAX()"
function in Cormen.
                         */
                        void updateDistanceMatrix(int u, float *distance, int
*previous);

                        /* list out the shortest path from src to dst and
return the length of
                         * the path */
                        int fillUpShortestPath(int *previous, int src, int
dst,

                int *directions, int maxRoads);

                        /* Get the weight of the edge between two
intersections. Looseley
                         * dependant on the speedlimit of the road connecting
the two
                         * intersections
                         */
                        float weight(int from, int src);

                        /* Bounding box coordinates */
                        float bbX1;
                        float bbY1;
                        float bbX2;
                        float bbY2;

};


#endif
```

### A.1.4 KmlReader.h

```
#ifndef _KMLREADER_
#define _KMLREADER_

#include <string>
#include <fstream>
#include <vector>
#include <cstdlib>
#include <iostream>

using namespace std;

class kmlReader
```

```
{
public:
            kmlReader();

            /*          opens a KML-file and reads its contents
                        returns 0 on success, -1 on failure
            */
            int openKML(char* filename);

            /*          makes every road offset from the smallest value
                        multiplies values so that largest is in the range
                        999 < x < 9999
                        also setts bounding box variables
                        returns -1 on error
            */
            int normalize();

            /*          get road segment (fromX, fromY, toX, toY)
                        returns -1 when there are no more roads to be read
            */
            int getRoadSeg(double &startX, double &startY, double &endX,
double &endY);

            /*          resets the index counter used by getRoadSeg
            */
            void resetIndex();

            /*  get bounding box values
            */
            void getBoundBox(double &bx1, double &by1, double &bx2, double
&by2);

            /*          returns the size of the start/end vectors (only one
                        value needed since start.size() == end.size())
            */
            unsigned int getRVectorSize();

private:
            /*          takes line (string) from the KML and splits it into
                        elements and data. Appends result to pased vector
            */
            void getElements(string line);

            /*          extracts roads from the elements vector and puts
                        them in vectors start and end. returns -1 on error
            */
            int extractRoads();

            /*          stores elements and data from KML file
                        */
            vector<string> elements;

            /*          stores start and end points of road-segments
            */
            vector<double> start;
            vector<double> end;

            /*          the boundning box
            */
            double bndSX, bndSY, bndEX, bndEY;
```

```
            /*          index used by getRoadSeg
            */
            unsigned int index;
};

#endif
```

### A.1.5 KmlReader.cpp

```
#include "kmlReader.h"

kmlReader::kmlReader()
{
            index = 0;
}

int kmlReader::openKML(char* filename)
{
            string str;
            char line[100];

            fstream kml(filename, fstream::in);
            if (!kml.is_open())
                        return -1;

            while(!kml.eof())
            {
                        kml.getline(line,100);
                        str.assign(line);
                        getElements(str);
            }

            kml.close();

            if(elements.empty())
                        return -1;

            if(extractRoads() != 0)
                        return -1;

            bndSX = 0, bndSY = 0, bndEX = 0, bndEY = 0;

            return 0;
}

void kmlReader::getElements(string line)
{

            string retStr;
            size_t start, end, wSpace;
            bool finished = false;

            while(!finished){
                        start = line.find('<', 0);
                        end   = line.find('>', 0);

                        if (start == string::npos || end == string::npos){
//no tag on current line
                                    if (line.length() > 0){
```

```
                                        wSpace =
line.find_first_not_of(" \t\n\r");
                                        if (wSpace != string::npos)
//found something other than whitespaces

            elements.push_back(line.substr(wSpace));
                                }
                                finished = true;
                        }
                        else{
                                if (start > 1){ //any data preceeding next
tag?
                                        wSpace =
line.find_first_not_of(" \t\n\r", 0); //skip inital whitespaces

                                        if (wSpace != string::npos &&
wSpace < start)

            elements.push_back(line.substr(wSpace, (start - wSpace)));
                                }

                                elements.push_back(line.substr(start + 1,
(end-start) - 1)); //skip first and last char (< and >)
                                end++; //skip last '>'

                                if (line.length() > end)
                                        line = line.substr(end,
line.length() - end);
                                else
                                        finished = true;
                        }
            }
}

int kmlReader::extractRoads()
{
            size_t pos1, pos2;
            string tmpStr;


            for (unsigned int i = 0; i < elements.size(); i++)
                        if (elements[i] == "coordinates"){
                                i++;        //step to first coordinate
                                while(elements[i] != "/coordinates"){
                                        pos1 =
elements[i].find_first_of(",");
                                        if (pos1 != string::npos){

            start.push_back(atof(elements[i].substr(0,pos1).c_str()));
                                                pos2 =
elements[i].find_first_of(",", pos1 + 1);
                                                if (pos2 !=
string::npos){
                                                        tmpStr =
elements[i].substr(pos1 + 1, (pos2-pos1-1));
                                                        tmpStr =
tmpStr.substr(tmpStr.find_first_of("+") + 1);

            end.push_back(atof(tmpStr.c_str()));
                                                }
                                                else
```

```
                                                                    return -1;
                                                            }
                                                            else
                                                                    return -1;
                                                            i++;
                                                    }
                                                    start.push_back(-1); end.push_back(-1);
            //padd with -1 to signal end of segment
                            }

            return 0;
}

int kmlReader::normalize()
{
            for (unsigned int k = 0; k < start.size(); k++){
                            if (end[k] != -1)   end[k] += 180;
                            if (start[k] != -1) start[k] += 180;
            }

            double smallestS = start[0], smallestE = end[0], largestS = 0,
largestE = 0;
            double temp;
            int mult = 1;

            if(start.size() != end.size())   //if this fails something is
wrong with the code =)
                            return -1;

            //determine largest and smallest values
            for(unsigned int i = 1; i < start.size(); i++){
                            if (start[i] != -1 && end[i] != -1){
                                            if (start[i] > largestS)  largestS
            = start[i];
                                            if (start[i] < smallestS) smallestS =
start[i];
                                            if (end[i]   > largestE)  largestE
            = end[i];
                                            if (end[i]   < smallestE) smallestE =
end[i];
                            }
            }

            if ((largestS - smallestS) > (largestE - smallestE))
                            temp = largestS - smallestS;
            else
                            temp = largestE - smallestE;

            while (temp < 250){
                            mult = mult * 10;
                            temp  = temp * 10;
            }

            //adjust values of the arrays
            for(unsigned int j = 0; j < start.size(); j++){
                            if (start[j] != -1 && end[j] != -1){
                                            start[j] = (start[j] - smallestS) * mult +
1;          // +1 so that smallest value is not 0
                                            end[j]      = (end[j]   - smallestE) * mult
+ 1;        //    -||-
                            }
```

```
            }

            //now we can set the values of the boundning box
            bndEX = (largestS - smallestS) * mult + 1, bndEY = (largestE -
smallestE) * mult + 1;

            return 0;
}

int kmlReader::getRoadSeg(double &startX, double &startY, double &endX,
double &endY)
{
            bool finished = false;
            while (!finished){
                        if (index + 1 >= start.size() || index + 1 >=
end.size()) //"eof"
                                    return -1;
                        if (start[index] != -1 && start[index+1] != -1 &&
end[index] != -1 && end[index+1] != -1){
                                    startX = start[index]; endX =
start[index+1];
                                    startY = end[index]  ; endY =
end[index+1];
                                    index++;
                                    finished = true;
                        }
                        else
                                    index++;   //if padding (-1) was found,
step forward
            }

            return 0;
}

void kmlReader::resetIndex()
{
            index = 0;
}


void kmlReader::getBoundBox(double &bx1, double &by1, double &bx2, double
&by2)
{
            bx1 = bndSX, by1 = bndSY, bx2 = bndEX, by2 = bndEY;
}

unsigned int kmlReader::getRVectorSize()
{
            return start.size();
}
```