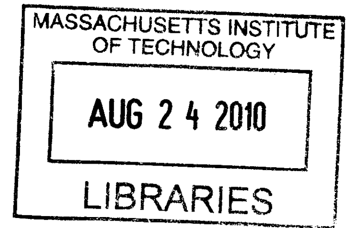
**Virtual City Testbed**

by

Oleg I. Kozhushnyan

S.B. Electrical Engineering and Computer Science
S.B. Mathematics
Massachusetts Institute of Technology, 2009


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 2 4 2010
LIBRARIES

ARCHIVES

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology

May, 2010
[June 2010]

©2010 Massachusetts Institute of Technology
All rights reserved.

Author ______________________________________
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by ______________________________________
Emilio Frazzoli
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by ______________________________________
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

**Virtual City Testbed**
by
Oleg I. Kozhushnyan

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

# ABSTRACT


Traffic simulation is an important aspect of understanding how people move throughout various road systems. It can provide insight into the design of city streets and how well they can handle certain traffic patterns. There are various simulators available, ranging from free tools such as TRANSIMS to commercial implementations such as TransCAD. The available tools provide complex, large scale and very detailed simulation capabilities. The Virtual City Testbed addresses aspects that are not available in these tools.

Primarily, the test bed provides the ability for interaction with the traffic system in real time. Instead of basing the simulation solely on automated vehicle models, we allow for human participants to interact with individual cars via a remote simulation client. Thus we are able to inject realistic human input into our simulation.

A second feature provided by our simulation is the ability to disrupt a simulation in progress. A disruption usually involves disabling access to a set of streets which forces the traffic to adapt as it moves around the road system. This yields a way to study the way traffic motion changes within a road system under the presence of unexpected events such as natural disasters or other real life disruptions.

Ultimately, we provide a test bed for studying traffic under varying environmental conditions.


Thesis Supervisor: Emilio Frazzoli
Title: C.S. Draper Professor of Aeronautics and Astronautics

# Contents

1. Introduction ........ 6
2. Background ........ 8
3. Contributions ........ 10
4. System Design ........ 12
  4.1 Simulation ........ 13
    4.1.1 Car Simulation ........ 14
  4.2 Server Design ........ 15
  4.3 Client Design ........ 16
  4.4 Design Patterns ........ 17
5. Implementation ........ 19
  5.1 Road System Specification ........ 19
    5.1.1 Coordinate System ........ 20
    5.1.2 Road Network Preparation ........ 21
  5.2 Testbed Server ........ 21
    5.2.1 Traffic Simulator ........ 22
  5.3 Client ........ 25
  5.4 Networking System ........ 28
6. Validation ........ 32
7. Conclusion ........ 34
  7.1 Future Work ........ 34
Bibliography ........ 35

## Table of Figures

Figure 1. High level system overview. ........................................................................................ 12
Figure 2. Illustration of the closest car at intersection rule. ........................................................ 15
Figure 3. Structure of the OSM library. ........................................................................................ 20
Figure 4. Diagram of traffic simulator classes. ............................................................................ 22
Figure 6. Diagram of traffic client classes. .................................................................................. 26
Figure 7. GPS overlay. ................................................................................................................ 28
Figure 8. Diagram of network system classes. ............................................................................ 29

# 1. Introduction

A good understanding of the way traffic flows within a city is vital in the study of the behavior of a modern city. The traffic patterns can reveal information about where people travel on a daily basis. Furthermore, the flow can provide a large amount of information about the effectiveness of the city design. Using the information about where people travel and how they make decisions about how to get there we will be able to improve the future of city design.

Collecting and analyzing traffic data is a monumental task. The complexity comes not only from the sheer amount of data, but also from the difficulty of gathering the data over large city sized regions. Even then, the traffic information that can be collected is limited to day to day activities. If we were to question what happens in the case of a natural disaster or some other unexpected disruption, we would have an even more difficult time in obtaining this information. As part of the project, we provide an easier method to collect the special case traffic information.

Natural disasters or disruptions rarely happen, but when they do, they have a great impact on traffic behavior. As such, we will not be able to effectively collect this information through the use of road sensors or other means in the real world. Thus we must turn to computer simulation to reach our goal. By simulating a city traffic system, filled with both computer and human controlled agents, we can build an approximation of the traffic patterns for any city. Thus we are able to simulate natural disasters in the system and record the responses of all the agents.

Over the course of this thesis project, we developed an application that is capable of simulating traffic in real time and a corresponding client application that allows for remote interactions with

the traffic system in a controlled manner. The role of the traffic simulator is to output the state of any vehicles traversing the road network as a function of time given any road network and a particular set of simulation parameters. The job of the client is to present a view of the traffic simulation to the participants of the experiment as well as to provide them with controls over their assigned vehicles.

The traffic simulator component acts as the server for the simulation and is intended to be run by those staging the traffic experiment. It is configurable with various car behaviors, custom road network and logging abilities to record experimental data for later analysis. It is an essential component of the test bed as it is the one that handles the actual computation of the simulation. When the simulator runs, it provides remote clients with the ability to connect and obtain control of one of the cars in the simulation as well as receive updates over the overall state.

The client component is intended to be used by the participants in the traffic simulation. It provides a visual representation of the road network and facilitates user input to alter the state of the simulation. Together, the simulator and the client make up the test bed that is the subject of this thesis.

The paper is structured as follows. Section 2 provides examples of other traffic simulators and describes the similarities and differences with this test bed. Section 3 outlines the contributions of this thesis and relates them to prior work. Section 4 discusses the design of the test bed system and goes into the details of its architecture. Section 5 describes the implementation of the design and discusses the details and limitations. Section 6 discusses a set of experiments and tests that were performed to test the validity of the test bed. Section 7 concludes the discussion by describing how the test bed can be extended and provides ideas for future work.

## 2. Background

The study of traffic systems and their effects has been important ever since vehicles became a primary form of transportation. The bulk of the research has been important to organizations such as the US Department of Transportation [1] as well as other governing bodies that determine road planning. These organizations use previous traffic data and simulations to improve on future designs of road networks.

Not all of the organizations have the need to develop their own traffic simulation systems. There is a wide range of tools, currently available, that can be used to simulate traffic on scales ranging from small towns to huge intercity regions. Some well known packages in the field are TRANSIMS [2], MITSIMLab [4], and TransCAD [5].

TRANSIMS is an open source community project, whose goal is to predict possible traffic conditions, congestion and pollution for city planning [3]. It is a multi level simulation environment that takes into account the distribution of population, zoning, common activities and micro simulation of traffic. The simulation is rooted in using statistical data to simulate the decisions that people make while moving about in their daily lives. It does not only consider vehicles as the only form of transportation but goes on to allow public forms of transportation as well. Ultimately, TRANSIMS is a very robust, but sometimes overwhelming with respect to usability [3], system for traffic simulation.

MITSIMLab is a project undertaken at MIT with a goal to simulate alternate methods of traffic management [4]. The ability of MITSIMLab to perform traffic micro-simulation (simulating

traffic at the level of a single vehicle) is similar to TRANSIMS. The advantage of MITSIMLab comes from its detailed system for simulating traffic control elements [4]. Furthermore, the simulator comes with a simple graphical user interface which is more accessible than its TRANSIMS counterpart.

TransCAD is a powerful commercial tool designed for traffic simulation and much more. As part of the package it provides analysis tools to study traffic flow. Furthermore it provides facilities to predict travel demand and public transportation usage to assist in city planning [5]. Unlike TRANSIMS and to a greater extent than MITSIMLab, TransCAD relies on a graphical user interface for much of its functionality which increases its ease of use.

Every one of the three tools is capable of simulating a traffic system and predicting possible future outcomes. They do this with two primary limitations. The first is that they do not support the introduction of a disaster event in real time. An experimenter cannot decide when an event, such as a road closing, should happen. This puts a limit on the ability to simulate unexpected disruptions. The second limitation is that it is difficult to use these systems to observe actual human behavior. There is no easy way to take human input and have it affect the simulation. In the next section we discuss how our system contributes to the field and provides a new method of simulation.

## 3. Contributions

This project makes multiple contributions in the area of traffic simulation software. These contributions come from the implementation of the traffic simulator and client application with features that are not present in currently available simulation software. Furthermore, our test bed provides certain levels of customizability that go beyond certain available packages.

The first major contribution is related to the "dynamic" nature of the traffic simulator. By "dynamic" we mean that the simulation happens in real time. Other simulators, such as the ones mentioned in the previous section, rely on long term offline computation. For example, the TRANSIMS simulator is made up of a large set of programs that run separately to perform processing on input data. This means that before any output is available, the whole tool chain must execute. Our test bed takes a different approach to traffic simulation. At the cost of complexity and scale, our test bed performs the simulation in real time. This means that the amount of simulation time that passes is directly related to the amount of real time that has passed.

The "dynamic" nature of the test bed has benefits over the more widely used offline approach in certain situations. In particular the advantages show up when the course of the simulation is not pre-defined but requires some random adjustments or modifications. Although this behavior can be approximated by offline simulators through some sort of event or rule based system there can be cases where this is not sufficient. For example, if the traffic conditions are unknown at any particular time, it may be hard to automatically trigger an adjustment in the simulation. The most likely option would be to run the simulation multiple times with various modifications.

Our system allows for real time observation of traffic conditions so at any given time the simulation state is known and can be adjusted.

The second major contribution of this project is the ability of the traffic simulator to connect to a remote client. The client, in our case, provides an interface for observing and altering the simulation state. There are similar features in commercial simulators, such as TransCAD, which provide a user interface to set up and observe a simulation and its results. The particular abilities provided by our client are something that is not available in any simulation package known to us.

Our client allows for users to take control of a single vehicle in the simulation. The idea behind this is that we are able to observe how a real driver can affect the traffic system. Furthermore, we can observe how the driver will respond to unexpected circumstances. Essentially, we allow for an easier way to perform complex traffic experiments on live subjects in real time scenarios. No other traffic simulation system, known to us, allows for the alteration of the behavior of a single vehicle.

# 4. System Design

The primary goals of the design of the test bed are to provide a flexible platform for testing various route choice systems and to allow for real time interactions with the simulation. Because of this, special care was given in implementing the traffic system so that it can be easily modified and extended in the future. Furthermore, the system was designed with a game development-like approach as there was need for real time interactions between participants of experiments and the traffic simulation.

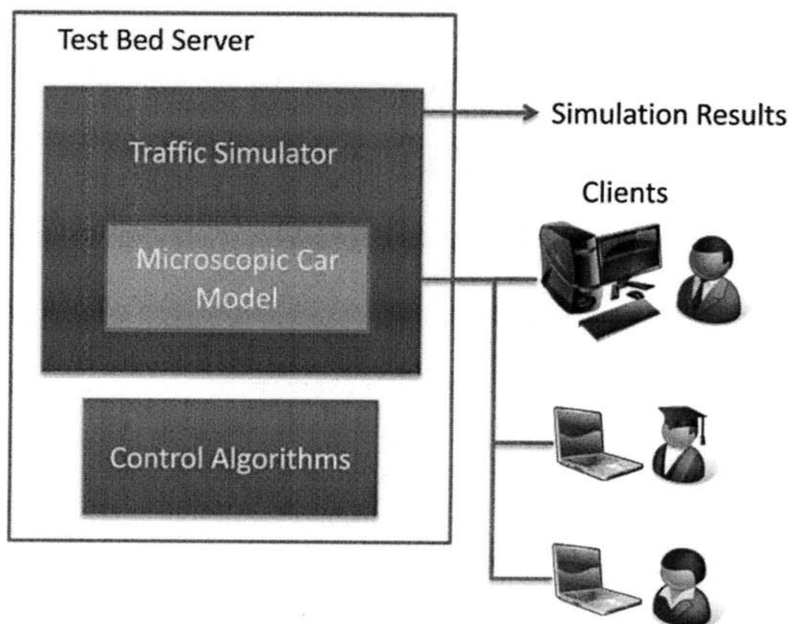



Figure 1. High level system overview.

In figure 1 we can see a high level overview of the test bed. The test bed server is made up of a traffic simulator that handles the control of all the vehicles in the environment. It also supports a set of algorithms that are used to navigate the vehicles around the road network. The server is

also able to maintain connections to multiple clients that act as a view for the state of the system. Finally, the server also records all important movements and events experienced by the vehicles and provides the record for later analysis.

## 4.1 Simulation

The traffic simulation model used by the test bed revolves around a microscopic car model. We simulate the cars at a low level and in a continuous environment to observe the traffic as an emergent behavior. By continuous we mean that the cars can occupy locations with coordinates limited by machine precision. This is unlike other simulators mentioned in the previous section that rely on a discrete environment, with a predetermined grid, or a very high level approach to traffic modeling. This tradeoff limits the scale of the simulation but in return allows for much greater control.

The test bed is designed with low level simulation and a continuous environment in mind. These aspects allow for it to quickly respond and adjust to input from the client. Also the low level allows us to parameterize the behavior of the cars which makes them independent of the path model that controls how they traverse the road network. This allows the simulated cars to be driven by a wide variety of methods while keeping the underlying logic identical in all cases.

In order for the vehicles to be able to drive around the road network, they must have access to the connectivity of all the roads. We achieve this by constructing a graph where the edges are roads and the nodes are the intersections. This representation allows for arbitrary road networks to be constructed. The intersections also support updates in order to allow them to act dynamically, such as when representing traffic lights. Each edge has parameters that are associated with it. Some example parameters are the length of the edge, the directionality of the edge and whether

the edge should be traversable by vehicles. The last parameter facilitates our ability to disturb the simulation as we can dynamically toggle the ability for cars to traverse an edge on or off.

### 4.1.1 Car Simulation

For the purposes of simulation, each car is considered a separate entity. As each vehicle drives around the road network, it follows a simple set of rules based on driving experience from real life:

1. The vehicle must maintain a straight path between the beginning and an end of any single road segment. This is vital for traffic to flow in a realistic manner.
2. The car can only belong to a single road at a time. This is important for correctness when roads cross each other without an intersection such as in the case of tunnels and bridges.
3. A vehicle must maintain a buffer zone, proportional to its speed, between itself and the vehicle ahead.
4. When a car approaches an intersection, the next closest car to the intersection is considered to be the closest car that is not on the same road. This rule makes sure that cars will not turn onto roads that are already full and instead mimic the real behavior of slowing down when approaching intersections. An example can be seen in figure 2.

By following these rules, the simulated cars exhibit real world behavior and allow for the traffic effect to emerge.

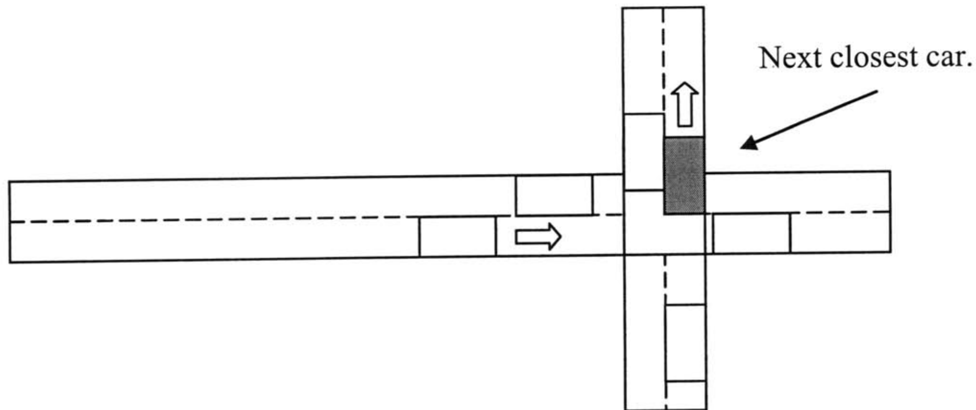



Figure 2. Illustration of the closest car at intersection rule.

In the real world, when a car makes a turn at an intersection it is never instantaneous. During the process there are times when a car does belongs to neither the road where it started to make the turn, nor the road on which it finished. This is not handled by our simulation as it adds unnecessary complexity. Each intersection is considered a point so that it is impossible for a car to be in between roads. These basic vehicle simulation principles are at the core of our test bed.

## 4.2 Server Design

The server application supports a variety of features that are needed to run experiments. Primarily, it runs an instance of the simulation. Just like a game server, it updates the state of the simulation even when no clients are present. The simulated vehicles controlled by the path finding models keep recording their state updates regardless of any player participation.

When a client connects to the server, it should be assigned a random car for it to control. No one else but the client should be aware that they are in control of this car. This prevents any unwanted interruptions for other clients that already may be in control. This is vital to the realism of the simulation as we cannot have cars appearing and disappearing when clients join and leave.

Once a connection is established, the server should keep the client updated on all the aspects of the simulation. This includes the state of disrupted roads as well as the positions of all the vehicles. Throughout this whole time the server should listen to client commands as well. These commands include direction changes and speed adjustments. Furthermore, it should keep recording the vehicle states for future review.

Upon the termination, the clients should be given a chance to disconnect cleanly. This involves the termination of the remote connection and the reassignment of the clients previously controlled vehicle. After this, the vehicle will function using some prescribed path finding model to traverse the road network. This also completes the requirement for giving the clients a seamless experience.

## 4.3 Client Design

The client presents participants of the experiment with a view of the environment as well as a set of controls for interaction with it. The presentation should be adequate for the experimental subjects to be able to ascertain the state of the simulation around them. This means they must be able to tell what are the possible route choices as well as what are the traffic conditions.

When a client connects to the server, they are presented with a view of their assigned vehicle already in motion. They are also given a GPS like overlay to be able to navigate the road network in a more efficient manner. Then the client is instructed to perform the actions that a particular experiment requires. During this whole time the server is recording their input and altering the simulation.

Over the course of the simulation, the client will be notified of the state of certain simulation vehicles. It is necessary to perform some post processing on that data before presenting it to the user. For example, since intersections are considered to be points, transitions between roads will be unsightly. The car jumps from one road to the next after hitting the center of the intersection. This effect is lessened by applying smoothing to the car position as it approaches the intersection.

When the simulation is over, the client should simply be able to disconnect by either notifying the server or just terminating the connection. In each case the server should substitute the proper automated controller for the vehicle to keep the simulation running.

## 4.4 Design Patterns

Our design uses a series of well known software design patterns to make sure the test bed remains modular. The patterns used are the Model-View-Controller pattern, the Visitor pattern and the Observer pattern.

The Model-View-Controller pattern is a design pattern that breaks up an object into three distinct components. The model component is the representation of the object that we want to model as data. If we take a car as an example, the model would consist of things such as the position, speed, direction, ID number and many other pieces of information. The view component can be considered as the visual representation of the object. To continue the car example, this would be the representation of the car as drawn by the 3D engine. The view may require access to some of the elements in the model of the object. Finally, there is the controller component of the pattern. The controller is the component that takes some sort of input, and applies changes to the underlying model. It can be seen as the public interface to the Model-View-Controller triplet.

All interaction with the object happens through the Controller. For the car example, the controller could be the remote client sending messages, or the local path finding algorithm.

The Visitor pattern is a design pattern that abstracts the notion of an operation that can be performed on an object. This pattern allows for new operations to be added to an object without changing any of the underlying code. The idea is that each operation that can be performed is defined in terms of an object. Then, a new method is added to the class that is to support the new operation. When the operation is needed to be performed, the new method is invoked with the object representing the operation. Thus, no changes need to be made to the object other than a few additions. This also guarantees that the old behavior will not be altered in any way by the new operation.

The Observer pattern allows for structured management of an event based hierarchy. The hierarchy can be made up of multiple objects where some of them are registered as observers of the other. When an event happens, each object notifies every other object that is registered as its observer. For a clear example, consider the situation where we may want to know when a car crosses an intersection from one road to another. One solution is to continuously poll the position of the car and compare it to all of the intersections. It is easy to see that this is an inefficient solution. Using the observer pattern, we can register as an observer of the car and have the car notify us when it crosses an intersection. This pattern eliminates any inefficiency in code that may be heavily event based.

# 5. Implementation

Over the course of the year we developed two implementations of the test bed. The first was developed in C++ with the use of the C4 engine [7] while the second used Java and the jMonkeyEngine [9]. The second revision is essentially a translation of the first in another language. It is slightly reworked to improve extensibility as well as support for the Apple OS X. The functionality of the server is also extended in the final version as we will discuss in the following sub sections. Thus we limit our discussion to the second and more robust implementation.

## 5.1 Road System Specification

One aspect of the simulator that has not changed over the course of the thesis is the decision to use the OpenStreetMap [6] format for specifying the simulation road network. The format is very robust and contains a lot of information in addition to road connectivity. Furthermore, it is freely available for anyone to use.

The format is based on XML and as such it is easy to read. In both versions of the simulator the road system is loaded from such a file. In addition to connectivity and geographic location information, the loaders attempt to extract speed limits, directionality of the road and the number of lanes when available. This data is made accessible to the simulator through a graph interface.

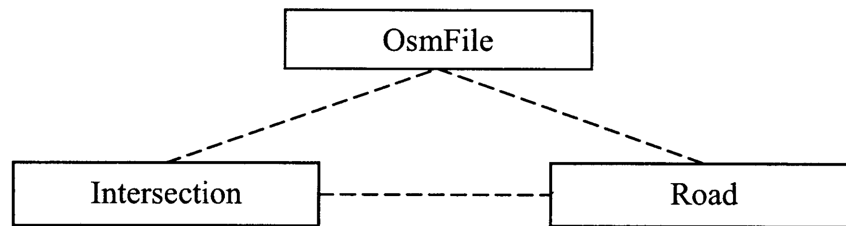



Figure 3. Structure of the OSM library.

The graph interface is made up of a list of nodes, the intersections, and the edges, the roads. Each node is assigned a unique ID by the file format. The roads, on the other hand, require some processing to extract their ID. The file format assigns a single ID to a road, which could be made up of more than one segment. This makes it impossible to reference the sub segments of the road, which is vital for the simulator. Thus, we assign custom ID numbers to the road segments in the order in which they are read from the file. This guarantees the same ID numbers as long as the file is read in the same order. As can be seen in the figure 3, the OsmFile stores the graph by having lists of intersections and roads. The intersections and roads also have references to each other to make it easier to convert from one graph element to another.

The simulator is able to extract the connectivity by extracting the endpoint intersections from the roads and then looking at the other roads that the intersections belong to. Other information, such as length and number of lanes, can be extracted from the roads as well.

### 5.1.1 Coordinate System

The coordinate system used by the simulator is based directly off the geographic locations extracted from the file format. The file format specifies all positions using their latitude and longitude. Since we are dealing with a much smaller scale of simulation, it does not make sense to remain in that coordinate system. We provide a GeographicLocation class that handles

longitude and latitude as well as UTM coordinates. It also allows us to convert all latitude and longitude coordinates into a more localized UTM coordinate system. This removes any issues due to the curvature of the earth and allows us to use a more accessible unit for length measurement, the meter.

### 5.1.2 Road Network Preparation

There are multiple ways of importing a road network into our simulation. The first method involves manually constructing an OsmFile object and the corresponding Road and Intersection objects. This is used primarily for testing in order to construct simple test networks that can be easily understood. The second and more robust method involves obtaining an actual OSM file and using it to establish the network.

The process begins with an OSM (OpenStreetMap) file containing the road network. These files can be easily obtained on the OpenStreetMap site where they provide an export interface. Their full data set is also available for download. It is then processed by a simple converter application. This application parses the file line by line and converts all latitude and longitude into UTM coordinates. We do this to prevent any issues that different coordinate converters may have so that their output locations are guaranteed to match up. Then the server and client are able to import the OSM file directly in order to construct the internal graph representation for simulation.

## 5.2 Testbed Server

The test bed server mimics closely the design established for it in the previous section. It is made up of a traffic simulation engine, a set of controllers, and a communication subsystem for interacting with the clients.

### 5.2.1 Traffic Simulator

The traffic simulator server was implemented with the intention of also acting as a persistent entity that could run continuously throughout the experiment. The following is a block diagram of the important modules of the simulator.

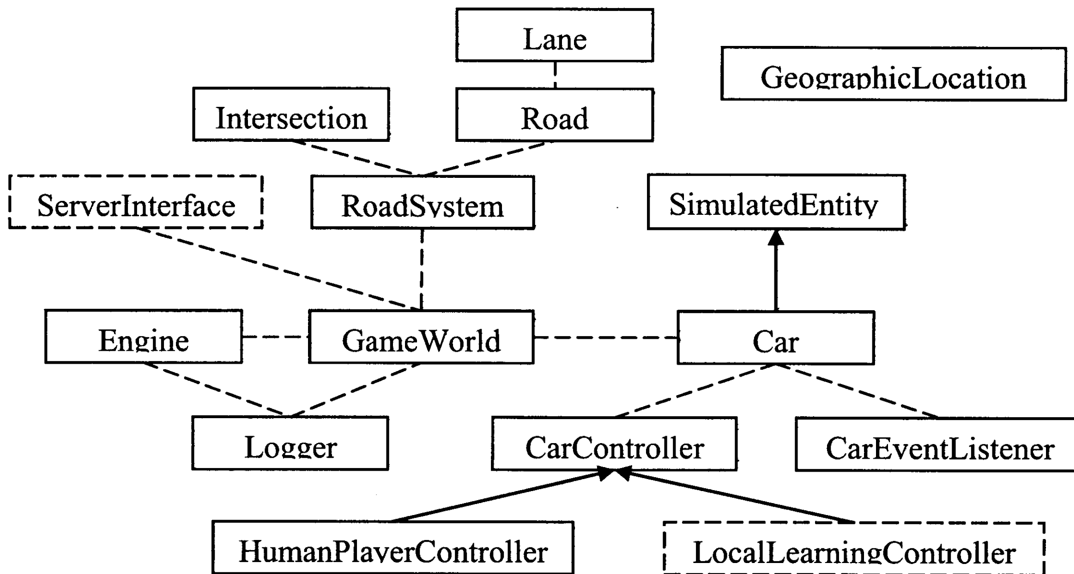



Figure 4. Diagram of traffic simulator classes.

The traffic simulator component of the server also closely follows the design in the previous section. The system is also abstracted in such a way that it is very easy to implement a custom type of car and have it be controlled in different ways.

The central component of the simulation system is the GameWorld class. This class maintains a list of all the vehicles in the simulation, a reference to the currently used road system, and a reference to the Logger object. This structure can be considered as the state of the simulation. It is also a signal handler for various simulation and network events. For example, when a player joins the server, this object is responsible for assigning and revoking its vehicles. The class also

maintains a reference to a ServerInterface which is part of the networking subsystem. This interface allows the simulator to send updates about its current state to all remote clients.

A simulated vehicle is represented by a Car class that inherits from an interface known as SimulatedEntity. The interface requires any of its implementations to create a method called update. This method handles the logic behind driving the car. Particularly, it checks for any nearby vehicles and then adjusts the acceleration of the car accordingly. If a car is within the buffer zone, which is adjustable through a parameter, the update method "applies the brakes" and sets a negative acceleration, otherwise it keeps accelerating until it reaches the speed limit. Turning is handled by setting a "next road" parameter for the car which it transitions to during the call to update on the next intersection. It is very easy to add a custom type of vehicle to the simulation. In order to do this, one must extend the Car class and override the update method. This will allow for any behavior of the vehicle to be changed.

In order to control vehicles through the Car class, we apply the model and controller components of the standard model-view-controller pattern. Each car is assigned a controller that sets up its next road and acceleration parameters. In our implementation, we have two controllers, one that listens for input from the remote clients and another that would follow a predetermined path. The HumanPlayerController and LocalLearningController are the two methods we have of taking control over the vehicles. The HumanPlayerController converts the network messages and converts them into the next road that the vehicle should take. Using this controller, a remote client can control any vehicle. The LocalLearningController is more advanced and is beyond the scope of this thesis. For a detailed description of the Local learning controller, refer to [8],

which is another thesis based on this traffic simulator. The controller allows for cars to follow and adapt paths between an origin and destination.

Another feature of the Car class is the use of the observer pattern. This is implemented through the CarEventListener interface. Each car has listener associated with it. The listener is notified every time the car changes roads. The listener has three main applications; the first is to record road changes undertaken by each car so the data can be analyzed later and the second is to generate an update message to be sent to the client and the third is to update any learning information for path finding as described in [8]. Using a listener provides significant improvement to the event based nature of road changes which otherwise would consume a large fraction of simulation time.

The Car class is the essence of the simulation. An instance of the test bed has an instance of the Car class for each vehicle participating in the simulation. Each car may have one of the two possible controllers over the course of the simulation as well. The cars also interact with one another by communicating through the road network graph represented by the RoadSystem object.

The RoadSystem object is the simulations representation of the road network and its geography. Notice that it looks very similar to the OsmFile class in the OSM library. In fact, the RoadSystem uses the OSM library to load an existing network file.

The RoadSystem object maintains a list of Intersection and Road objects. The Intersection is nearly identical to that in the OSM library and its sole purpose is to provide connectivity information to the nearby roads. The main difference is that this Intersection object also

provides methods to query the closest cars. This functionality is used by the Car class to make sure it does not speed into a full intersection and thus make the simulation unrealistic.

The Road object is very different from its counterpart in the OSM library. The major difference is that it is now contains several internal Lane objects. A Lane represents a single lane of traffic. Thus, for roads with multiple lanes, there would be more than one Lane object. There are two sets of lanes, ones that go from start to end and the other in reverse. They are also numbered with lane zero being closest to the median. When a car enters a road system, it registers itself in a particular lane which is then used by other cars to find nearest neighbors on the road. The Lane object is mainly an acceleration structure that maintains a linked list of all the cars in the order that they are driving in the lane. It allows for cars to find neighbors quickly by looking at the previous or next element in the list.

This implementation is extensible in many ways. The Car object can be extended to provide different behaviors for different vehicles. Different CarEventListeners can be developed to perform different tasks on car events. New CarController implementations can provide other methods for guiding the vehicles around the network. Overall the implementation of the test bed server is a very extensible and robust system.

## 5.3 Client

The test bed client is an application that is used to visualize the output from the server in real time. Using a 3D engine, it renders the current state of the simulation as it receives updates from the server. It also provides facilities for commands to be sent back to the server. The important modules of the client are displayed in the following block diagram.

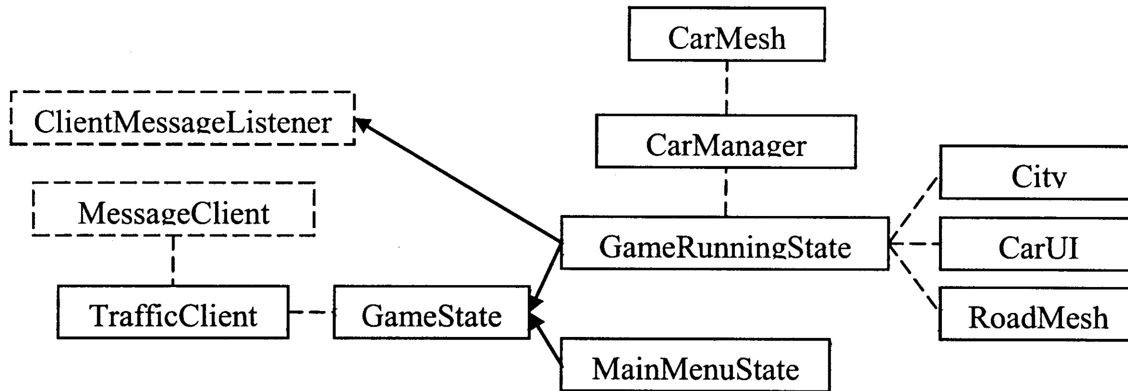



Figure 5. Diagram of traffic client classes.

The main class that is the entry point into the client application is the TrafficClient. It ties down the whole system by linking the various states with the networking subsystem. It also handles the set up of the keyboard and a logging system for use in the rest of the client. The main class maintains a reference to the MessageClient which is part of the networking system and will be described in the next section. The next component of the main class is the association with the GameState interface. This interface allows the client to seamlessly change its state depending on conditions. The benefit of this will shortly become clear.

The GameState interface provides an abstraction to multiple client states. It requires methods such as initialization, rendering, cleanup and the support for state transitions. Through this interface, the client can take on one of the currently available states. The use of this abstraction separates logically different states of the client. For example, the state in which the client renders the simulation is very different from a main menu state which does none of those things.

The two available game states are the MainMenuState and the GameRunningState. The main menu state is included to present some sort of information to the user of the client before

jumping into the simulation. This can be in the form of basic instructions or can be neglected all together.

The important part of the client processing happens in the state defined by the GameRunningState class. It is responsible for the whole presentation of the simulation. The first thing that it does is set up the presentation of the simulation. This includes loading and generating all the graphics needed to draw the road network and the vehicles on the screen. Then it handles the rendering of all the elements at every frame. Another responsibility of this state is to update all the client objects. This entails listening to the keyboard in order to alter the user interface, such as the GPS overview, or to prepare commands to be sent to the server.

The GameRunningState utilizes the City, CarUI and RoadMesh classes to render a representation of the simulation on the screen. The City class represents all the buildings that line the sides of the streets. The way it functions is essentially by taking all the roads in the road network and generating simple cubic buildings to line the sides of all the streets. It is a very simple and effective, but not very good looking solution. The CarUI class represents the GPS overlay that shows extra information about the road network. The following is a screenshot of the GPS overlay.

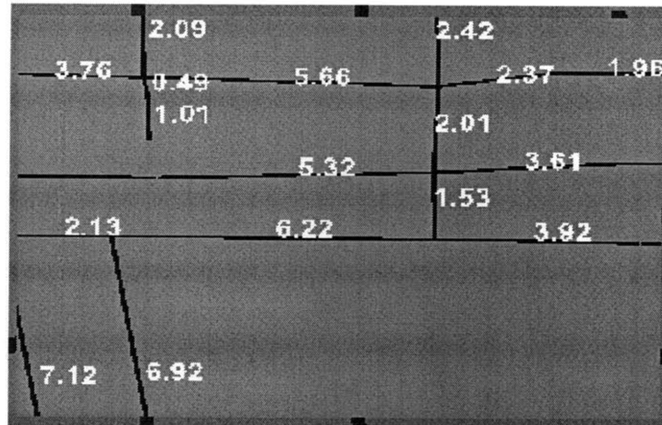



Figure 6. GPS overlay.

The purpose of the overlay is to present the user with the road network information in their vicinity. Additional information can also be displayed. In this example, the numerical values represent the scaled lengths of the road segments. Finally, the RoadMesh class handles the generation of the road network. As an input, it takes a list of roads from an OsmFile object and converts it into a useable mesh. The mesh is used to render the streets for the primary simulation as well as for the simplified display as seen in the GPS overlay above.

The final component that the game state uses for rendering is the CarManager class. The GameState notifies the CarManager of any updates that it receives from the server. In return, the manager instantiates graphical representations of those updates through the CarMesh object. The car manager is capable of adding new vehicles when they arrive, deleting them when they are removed and updating their positions when they move about, all in response to network messages.

## 5.4 Networking System

The networking system is the vital link between the test bed server and the client. Its various components are used by both elements in order to communicate their state to the others. The module diagram of the networking subsystem is as follows.

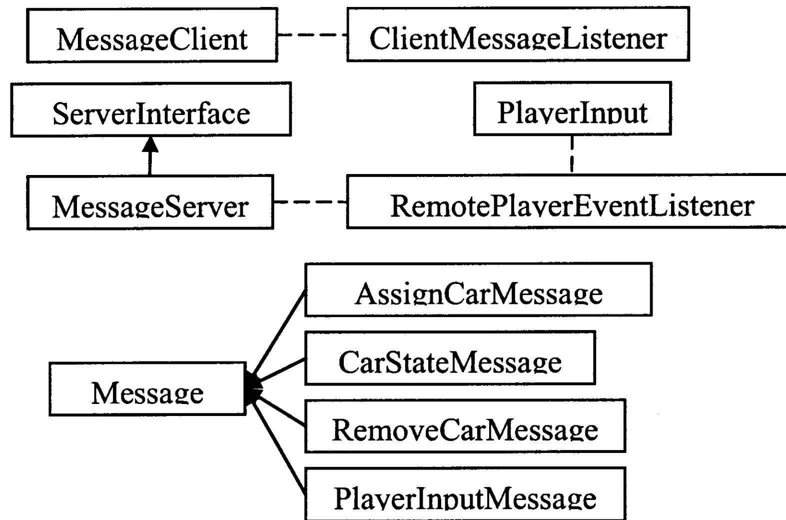



Figure 7. Diagram of network system classes.

The networking system revolves around the MessageServer and MessageClient classes. These two classes communicate to one another over channels established through the networking component of the jMonkeyEngine [9]. These two classes instantiate their own threads that run in parallel with the rest of the application. In these threads they listen to messages and generate notifications for the rest of the system. This is advantageous in that the networking subsystem has no observable impact on the performance of the rest of the test bed.

The MessageServer is the server component that resides in the test bed server. It accepts communication from the test bed servers GameWorld object through the use of the ServerInterface. The interface allows information about Car objects to be sent to the clients. The MessageServer also provides feedback to the GameWorld through the use of the

RemotePlayerEventListener interface. This interface allows the test bed to be notified about incoming connections, disconnections and input updates sent from the clients. The input that arrives at the server is translated and then passed on through the use of the PlayerInput interface. The interface abstracts the details and only presents the vital input information. In our implementation, the only necessary information is the desired vector of travel with respect to the current vector of travel. Using this information the server can direct the vehicle properly at the next intersection. With the effective abstraction of the server networking system we are able to extend it without altering any of the test bed code.

The MessageClient is the corresponding communications component that resides in the client. It communicates with the GameRunningState through the use of the ClientMessageListener interface. This interface notifies the client of the car assignment events, the car position updates and the car removal events. These three pieces of information are sufficient to communicate the simulation state over the network.

Both of the messaging systems depend on the four network message classes currently used by the simulation. The four messages convey the actions that are then converted into event listener method calls as stated above. If in the future more messages are needed, they can be easily added by extending the Message class. Out of the current messages, the AssignCarMessage is sent when a player initially connects to the server. The server picks a car for them and sends its ID number in the message. This tells the client which car they have control over. The CarStateMessage is sent with the position and ID of each car in the simulation to all clients. Using this information the client can extrapolate the current state of the simulation with very little error. The RemoveCarMessage is intended for simulations that require that cars be created

in removed. Normally an experiment would maintain a constant number of cars over its course, but in some cases, such as modeling flow, the removal of cars is necessary. Finally the PlayerInputMessage contains the input information sent from the client to the server to be acted upon. Together, these four messages make up the test bed communications protocol.

## 6. Validation

The validation of the test bed accuracy was a complicated process. Due to the generic nature of the simulation it is very difficult to make sure that it is realistic in all possible scenarios. The smaller scale of the simulation also makes things harder as most related traffic information is not available at its scale. Thus we were relegated to observing the simulation and making sure that it behaves in a believable manner when observed by someone with driving experience.

One particular experiment gives us confidence that the simulation is adequate. Consider a traffic jam where cars are dormant for most of the time with short periods of movement. When observed from a distance, the movement appears to be moving backwards through the traffic. Cars that are able to move use up their available space and are forced to stop but in the process create space for the cars behind them to do the same. Now consider if the cars were on a closed circular road. The wave of free space would be moving in reverse through the traffic. This is the condition we attempted to simulate.

We constructed a road network and deterministically drove all the cars into a single loop section. As the section began to fill up, a traffic jam began to form until the whole loop was almost completely full. At that point we could observe the wave of free space for cars to move travelling in the opposite direction of the traffic. Thus our simulation was able to successfully represent a realistic traffic event.

Further validation of the simulation was undertaken by Amrik Kochhar in his thesis "Simulation and Verification of Autonomous Route Planning Behavior." His thesis concentrated on extending the simulator with a complex car controller that allowed for adaptive paths [8]. He

then measured the distribution of cars on the road network as a function of time. The results of his simulation provided us with more confidence that our test bed is a good approximation for real traffic systems.

Overall the system must be continuously verified in the future as it is used for further experimentation with traffic simulation.

# 7. Conclusion

In this thesis we described the design and implementation of a new test bed for dynamic traffic simulation. We developed a distinct server and client component that communicated with one another over a network. Furthermore we performed initial verification of the system and concluded that it is accurate enough to be used for traffic simulation. We hope that this test bed system will be useful in accomplishing the goal of real time traffic simulation with both simulated and real drivers.

## 7.1 Future Work

The extensible nature of the test bed leaves it open to future enhancements. Some possible extensions to the server would be to introduce new path finding and car simulation models. On the client side, an improvement to the graphical representation of the simulation would be a very helpful addition. One of the principles behind this project was to make a test bed that is easy to extend in the future.

## Bibliography

[1] *Turner-Fairbank Highway Research Center,* http://www.tfhrc.gov/

[2] *The Transportation Analysis and Simulation System (TRANSIMS),* http://transims-opensource.net

[3] *TRANSIMS Overview,* http://tmip.fhwa.dot.gov/resources/clearinghouse/docs/transims_fundamentals/ch1.pdf

[4] *MITSIMLab,* http://mit.edu/its/mitsimlab.html

[5] *TransCAD,* http://www.caliper.com/tcovu.htm

[6] *OpenStreetMap,* http://www.openstreetmap.org/

[7] *C4 Game Engine Overview,* http://www.terathon.com/c4engine/

[8] Kochhar, Amrik, "Simulation and Verification of Autonomous Route Planning Behavior," M.Eng thesis, Massachusetts Institute of Technology, Cambridge, MA, 2010.

[9] *jMonkeyEngine,* http://www.jmonkeyengine.com/home/