Traffic Condition Tracking and Visualization in Virtual City Testbed

by

Boyuan Zhu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 20 2011

Certified by .... [illegible]
Professor Emilio Frazzoli
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Certified by ....
Ketan Savla
Research Scientist
Thesis Supervisor

Accepted by .... [illegible]
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Traffic Condition Tracking and Visualization in Virtual City Testbed

by

Boyuan Zhu

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Computer traffic simulation is a tool widely used to understand how humans behave under varying traffic conditions. The Virtual City Testbed is a traffic simulation framework built to closely model human behavior by allowing direct user interaction in the simulation. Using the testbed, human subjects can remotely control vehicles in the virtual environment. A virtual positioning system (VPS) is displayed with the testbed client. It tracks and visualizes traffic, disruptions, and tolls local to the user's position. As traffic conditions shift, the VPS dynamically updates to reflect the changes. Together, the testbed and VPS provide an environment for studying how traffic conditions affect decision making.

Thesis Supervisor: Professor Emilio Frazzoli
Title: Associate Professor of Aeronautics and Astronautics

Thesis Supervisor: Ketan Savla
Title: Research Scientist

Contents

List of Figures

Chapter 1

Introduction

Designing a modern city is an exceedingly complex task that requires an understanding of human behavior in the urban environment. One such behavior is the pattern of traffic in a city. Traffic patterns can have long-reaching effects in other domains, such as property value[7]. As a result, the behavior of traffic can reflect the effectiveness of a city's design.

1.1 Motivations for simulation

We are interested in using computer simulation to understand traffic behavior in cities. Simulation is our weapon of choice for two reasons. First, we would like to study and predict system and individual behavior during natural disasters or other disruptions. Data collection may be difficult or impossible during an actual event. A simulation can be used to discover the resulting behavior and optimize the emergency response.

Secondly, simulation can be used to predict the effects of policy changes to the road network. For instance, the careful application of tolls may alleviate congestion in the network[6]. Implementing a defective policy can be expensive and dangerous[5], and we would like to minimize the chances of doing so. Traffic simulation gives us the ability to design and fine-tune policies to have the intended effect.

Chapter 2

Background

There are a number of existing tools available for performing traffic simulation. The design and scope of each project often differs; some projects simulate at the level of traffic flow and provide a large, statistical model. Others are microscopic simulations and focus on the movements of individual vehicles.

An example of an existing microscopic simulator is the Simulation of Urban Mobility (SUMO) [4]. The project can manage fairly complex networks with many edges and vehicles. Analysis can be done with or without 2D visualization. Like many other projects, it provides a comprehensive toolset for making measurements of the system variables. However, there is no way for a human agent to join the simulation and participate by controlling a vehicle.

Human control over vehicles provides an opportunity for us to study actual human behavior in the traffic system. While existing simulations do model how humans behave, these models may differ from reality, especially under disruptive conditions. Observing how humans actually behave in the simulation allows us to evaluate and improve existing models. The challenge to observing human behavior is building a sufficiently close to real driving experience. A human in a car will not able to see the entire city, or be aware of universal traffic conditions, and must make decisions with incomplete information.

2.1 Description of testbed

In this thesis project, we have built a Virtual City Testbed to perform real-time simulation of traffic behavior in an urban environment that allows human participation and mimics an actual driving experience. Recently, Kozhushnyan and Kochhar contributed to building the framework [8].The testbed is comprised of a server that manages traffic information, and a client that can remotely interact with the server. The server can communicate traffic conditions to multiple clients. The client allows the user to control a vehicle in the virtual city.

The server can introduce disruptions into the system. The administrator of the server can place any number of disruptions in realtime throughout the city to affect behavior. The server also logs traffic information: the position of cars, the number of cars on each road, and the average speed on each road. The server provides clients the traffic data and the ability to connect and assume control of a vehicle.

When connected, the client gains a 3D view of the city in the driver's seat of the vehicle. A virtual positioning system (VPS) displays nearby traffic conditions and tolls. As the user drives around, the VPS is updated with the changing road speeds, disruptions, and tolls. Together, the server, client, and VPS form a platform to monitor human behavior under disruptions and toll policy changes.

The focus of the paper is the design of the VPS. The design of the testbed simulator is addressed by Uh[9].The structure of this paper is as follows:

Chapter 2 discusses the contributions made by this testbed in comparison to existing traffic simulators.

Chapter 3 provides an overview of the system architecture. It explains the client-server model, and the database-VPS interactions.

Chapter 4 dives into the implementation of the VPS in the client. It also catalogs the structure of the traffic information database.

Chapter 5 describes the results from testing the platform.

Chapter 6 reiterates the key points of the discussion, and details ideas for extending the testbed in the future.

Chapter 3

Testbed Design

3.1 System Overview

The testbed is designed around a client-server model (3-1). The server runs the simulation and controls the AI vehicles in the road network. Generated traffic information is stored into the database and written into logs. Multiple clients can connect to the server to gain a view and control of a given car. Clients can then use the VPS to fetch and view local traffic information from the database for their assigned cars.

3.2 Client Overview

3.2.1 Virtual Positioning System (VPS)

The VPS is designed to be a modular component that can displayed separately from, or integrated with, the 3D simulator (3-2). The purpose of the VPS is to display the local road network around the assigned car as a map. The VPS engine interacts with a number of connectors to pull data from the server database and visualize it.

The connectors read the assigned car position, the average speeds on roads, and the position of disruptions. Knowledge of the car position allows the VPS to mimic a GPS. The VPS centers the map view on a car icon at that location. As the simulation proceeds and the values are updated, the engine reconnects at intervals to update its

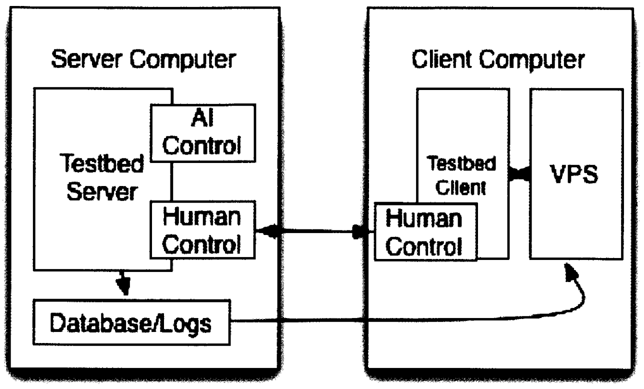

Figure 3-1: Testbed Client-Server Data Flow

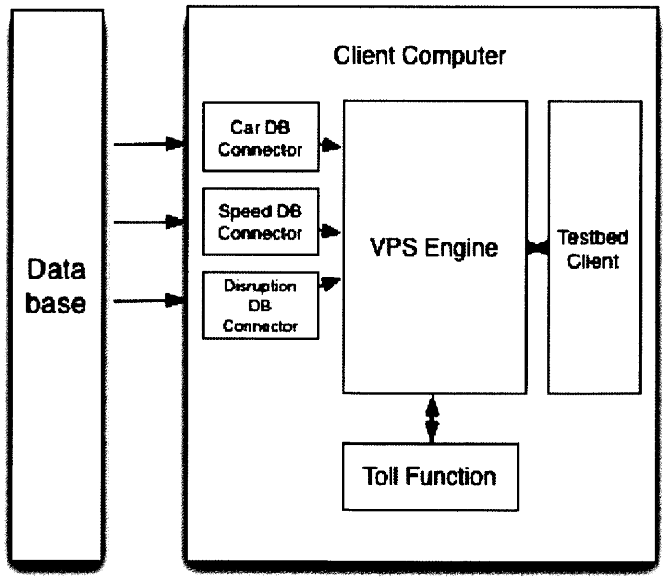

Figure 3-2: VPS Engine Data Flow

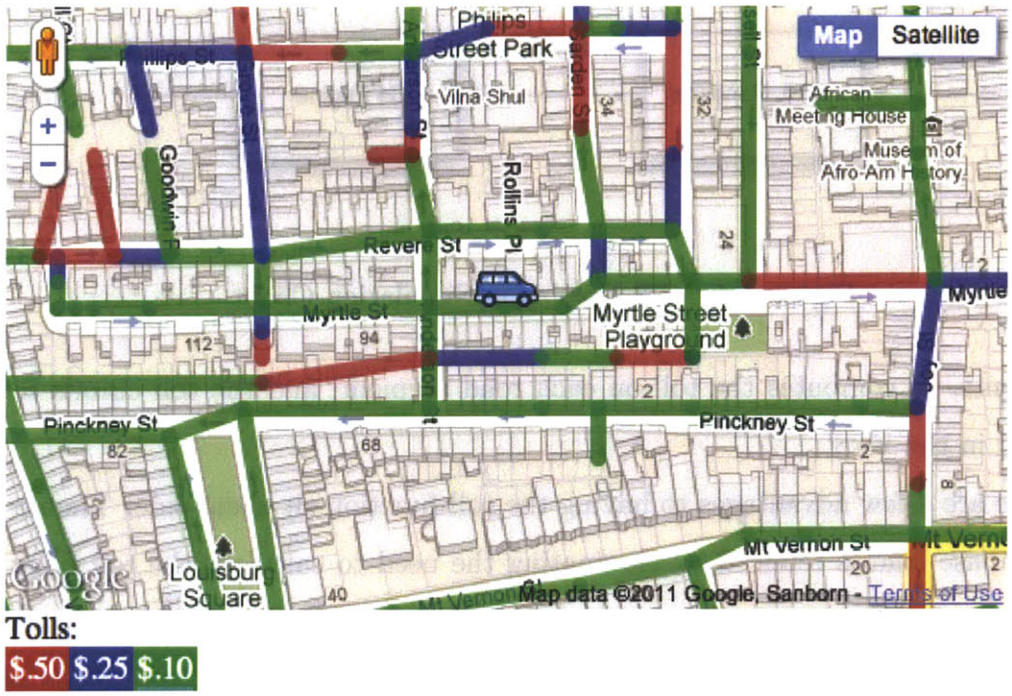

Figure 3-3: VPS Engine Running with Overlay

stored value of car position. The map view is animated to follow the icon as it advances to new positions.

The average road speed data is used in a visual overlay on top of the road network (3-3). Each road segment is overlayed with a colored line to represent the current speed. A wider overlayed line represents faster traffic. A narrower line represents a congestion, or slower traffic. To improve performance and reduce database load, a loader preloads the coordinates of the road network.

Any disruptions are also marked based on the locations in the database. The display of any element, road network or disruption, is limited by the zoom level of the VPS. Note that changing the amount of information displayed, such as zooming out to give full information, may result in markedly different user behaviors.

The final piece of VPS visualization is the display of current tolls for the road network. The tolls are reflected by the color of the road overlay. Higher tolls are red, and lower tolls are green. The values of the tolls are determined by a modular toll function that takes road network data from the VPS. A toll function has two

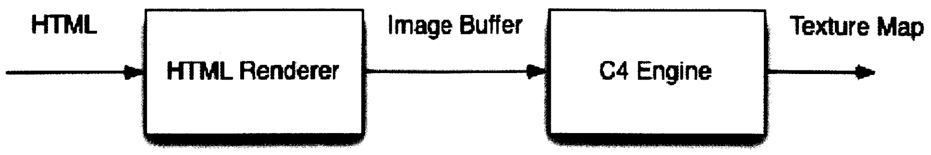

Figure 3-4: VPS Display in Simulation Client

components. It computes the toll on each road segment, and the toll cutoffs to be displayed in the VPS.

There are a few advantages to having the toll function sit client-side. The server and database load is reduced by eliminating the need to calculate the toll function and record values in the simulator testbed. It is also trivial to change the toll function for any given client, simply by replacing the module; a simulation can be run with multiple clients responding to different functions.

3.2.2 VPS-Client Integration

The VPS can also be integrated into the 3D testbed client to display the map inside the client. The client is able to render objects and display images, but is unable to parse a VPS webpage. We can use an external browser library to generate an image buffer of the webpage. The image buffer can then be mapped onto a texture that the client can display (3-4). Since the VPS information is constantly updating, we need to rerender the texture at intervals update the client with it.

Chapter 4

Implementation

4.1 VPS Engine

The VPS engine was built in HTML/JavaScript using the Google Maps API v3[3]. Initially, we had to choose between implementing the VPS directly in the 3D C4 engine or outside of C4[2]. Drawing a 2D map alone in C4 proved to be cumbersome; the need for extensibility to include tolls and other maps pushed us to favor a dedicated mapping solution like Google Maps.

4.1.1 Map API

The basic building block of map positions is the point, represented by `google.maps.LatLng`. These points are used to define the position of every object: markers, the view, and the roads. Given a center view point, the VPS map is generated as a `google.maps.Map`. Finally, the map needs to be configured to reside in a specific HTML element on screen.

Points are also used to determine the location of car and disruption markers. Markers are created with `google.maps.Marker` at a given `LatLng`. In order to display the marker, we configure its setting to assign it our map.

Road overlays are created by drawing an translucent line through the use of `google.maps.Polyline`. Each `Polyline` is determined by an array of `LatLng` and

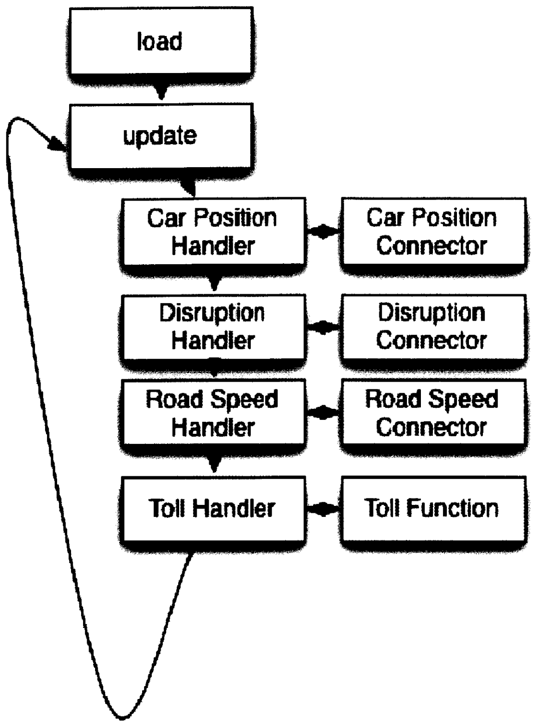

Figure 4-1: VPS Engine Main Loop

stroke and opacity options. We can draw lines of different colors for different tolls, and of varying weight for different traffic conditions. Like the marker, every `Polyline` is set to our map.

4.1.2 VPS Engine Loop

The engine works as follows (4-1). During `load`, the VPS preprocesses initial conditions, including the starting position of the car and viewport, and the positions of the road network. It uses the map API to set up the appropriate map, and draws the overlay for the road network and the car. The engine then sets up `update` to be called at intervals.

4.1.3 VPS Handlers

The update function is the primary loop of the engine. Each cycle, it calls handlers to update the assigned car position, update disruption positions, and refresh the road overlay. Each handler does two things: retrieve data through its corresponding database connector, and control the logic to parse and visualize the data.

The handler for disruptions is the most straightforward. It retrieves from its connector the XML that contains the coordinates of all system disruptions. The handler then loops and creates new `Markers` with the appropriate icon for each disruption. After creating the new markers, the handler reads the old markers from an array. The old markers are cleared, and the new ones are stored into the array.

The car position handler behaves similarly. Instead of reading multiple locations, it only retrieves one, the location of the user's assigned car. The handler takes the point information, and reads a dirty variable to check if the marker has already been created; if not, a new `Marker` is created with the car icon. The `Map` is then set to `panTo` the new location.

The road overlay handler grabs a XML of each road segment and the average speed on the segment. It loops through the array of road overlays and sets the width according to the speed observed.

Lastly, the toll handler passes the known speed data to the toll function. The toll function then calculates the overall toll cutoffs, and the individual value for each road segment. The handler receives a mapping of price to color, and sets the road overlay for each segment to the appropriate color. HTML elements for a toll/color legend are also generated to reflect the determined tolls cutoffs.

4.2 VPS Connectors

All database connectors share the same basic structure. They initiate a connection to the database and create a DOM tree for the data (4-2). The connector queries the database and fills the tree with data in each row. Each row becomes a new child node in the tree, and each attribute becomes a leaf.

```
<roads>
<road id = "0" speed = "18"/>
<road id = "1" speed = "9"/>
<road id = "2" speed = "27"/>
...
</roads>
```

Figure 4-2: Structure of the XML Reply

The connectors are written in PHP and form a server-client interaction with the VPS engine. The connectors sit on a webserver and are accessed by the engine client. The engine uses `downloadURL` to send a request to the webserver to process the PHP, and retrieve the XML.

4.3 Toll function

The toll function is a separate module from the VPS. It can be modified independently and plugged back into the VPS engine. It has two components: `tollPrice` to determine the overall prices of the tolls, and `tollRoad` to determine the toll price for each road. `tollRoad` receives an array of speeds on the road, and computes the value of the toll for that segment. It can be extended to consider additional information, such as car or disruption locations. The result is sent back the handler as a mapping of road to toll value. Currently, a simple inverse function is used for calculating tolls.

The logic for calculating the overall prices resides in `tollPrice`. Like `tollRoad`, it takes the array of road speeds as input. Depending on implementation, the two can share overlapping code. Although the two methods form a single a policy, they are split so that the engine can call them selectively.

4.4 Client-VPS Integration

The graphical 3D client is built in C++ using the C4 engine. While the C4 toolset provides a number of useful features like networking, it doesn't provide a straightforward way to draw and modify a map. For instance, one possible solution in C4 is to provide an overhead camera view of the city as a "map". Unfortunately, any changes to the "map" would also be reflected in the city. This solution is not very flexible especially if we want to have different VPS behaviors for different clients.

As discussed earlier, these difficulties propelled us to develop the VPS for browser display. The result is a VPS that can be viewed by any software that can render HTML/JavaScript. We would still like a way for the 3D testbed client to display the VPS, without the use of an external browser. The challenge is that our client is only capable of rendering objects and textures; however, we can manipulate our VPS output to be compatible.

4.4.1 Browser Framework

We use the Awesomium web browser framework to render the HTML as an image buffer[1]. The framework is built around the singleton class `WebCore`. One copy is instantiated and then used to create a `WebView` with given dimensions.

We call the `render` method of the view to retrieve a `RenderBuffer` of the page. We take the resulting image buffer and pass it to the C4 engine.

4.4.2 Buffer Display in Client

However, the client cannot directly use the image buffer. We need to create and display a texture map from the image. In order to display a texture, we need to create a panel node and add an `ImageWidget` to it.

We add the panel node and widget to the constructor of `TrafficWorld` in the simulation testbed, and insert the image update logic to its `update` function. Every update, we ask the `WebCore` to check if the view has changed. If so, we need to rerender and create a new image buffer.We then ask the widget to `SetTexture` to the latest buffer.

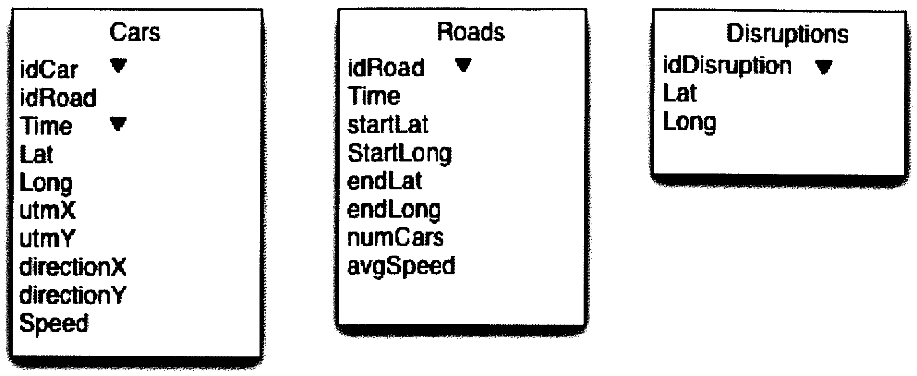

Figure 4-3: Tables in Simulation Database

4.5 Database

The database both acts as a log for the server, and as a means for the server to pass information to the VPS. Therefore, tables often log additional attributes that the VPS doesn't immediatedly need (4-3). However, we ran into some problems using the database as the exclusive logging system. Excessive database access slowed the testbed server down to a crawl. Our solution was to create additional logs that were written to text files. All that remains in the database are the critical tables for car positions, road speeds, and disruptions. Note that the car position table has two keys so that we can track the same car id at different points in time (otherwise they end up in the same row).

4.5.1 Coordinates

Coordinates in the database are stored as longitude and latitude coordinates. The simulator itself reads longitude and latitude coordinates from file, but performs computations after converting to UTM coordinates; the UTM system allows the use of meters and simplifies position and speed calculations. However, the map engine only accepts longitude and latitude coordinates, so the internal representation is converted once more before writing to the database. The `GeographicLocation` class facilitates the conversion both ways.

4.5.2 Speeds

Speeds are essentially quantized by the simulation. They appear in some multiple of 9, as all cars slow and accelerate to these values. This quantization is reflected as they are written into the database, and needs to be taken account of in the relevant handlers. The toll function, in particular, needs to be cognizant of that speed values are not continuous when determining the appropriate tolls.

Chapter 5

Evaluation

The evaluation of the VPS is complicated by the absence of a definitive metric for performance. It is both important that the system reflects the simulation state accurately, and that it is comprehensible and usable by human subjects. We approached users to drive the client, and noted both the accuracy of the VPS as they drove around, and their reaction to the VPS.

5.1 VPS Accuracy

To check accuracy, we followed a number of different cars around the city, noting the layout of the roads in the 3D client. In general, the car position on the VPS matched what we saw in the 3D client. As we entered traffic jams, those segment overlays were narrow, and the tolls were red/high as per our simplistic toll function. We would expect the opposite (wide, green/low) for segments with low traffic density. We saw the expected results when passing through such segments.

More rigorously, we were able to design a number of test scenarios: one in which cars flooded the entire network by adding in a high number of vehicles; ones in which a specific loop in the network was congested by modifying AI pathing. We observed the VPS for the networks. In all scenarios, we saw the appropriate speed and toll visualizations for both affected and unaffected segments. We used the simplistic inverse toll function.

5.2 Response to UI

Users quickly grasped the visual language that the VPS uses. The idea of line width being the flow, and the color being the cost was intuitive. The response about those features was positive. However, some users were confused and disoriented by the direction of the car marker. In its current state, the marker doesn't rotate to match the direction that the user is headed in the simulation. Users observed that they had to account for the actual direction of the car when looking ahead on the VPS. This is a visual improvement we look to make to the VPS.

We look forward to further evaluation of the entire testbed as we run comprehensive simulations with human users.

Chapter 6

Conclusion

In this thesis we have discussed the design and implementation of a traffic condition tracking system in a virtual testbed. Our system communicates with the testbed server to retrieve and present non-obvious traffic conditions to simulation subjects. Researchers can further explore the response of subjects through easily accessible and modifiable toll functions. We have also observed reasonable accuracy between the state of the simulation and the output of the tracking system. In the future, the system can be extended to display additional variables that may affect driver decision making. The further development of visualizing virtual traffic conditions can bring us closer to the goal of understanding which factors affect real human response.

Appendix A

Modification Guide

A.1 VPS

The main VPS file is vps.html. It can be viewed locally or from a webserver. Note that for the VPS to work, the corresponding PHP files need to be processable.

Additional handlers can be added to the `update` loop.

A.2 Database Connectors

Database connectors are the `*xml.php` files. They connect to specific tables in the C4 database. If the server IP changes, they need to modified to connect to the IP and port of the database. To be accessed by the VPS engine, they need to be on an active webserver with PHP processing.

The username and password for the database are contained in `phpsqlajax_dbinfo.php`.

A.3 Toll Functions

Toll functions are in the tollfunctions folder as the `tollfunction.js` file. Additional functions can be coded and left in that folder. Note that new files need to be added to the include section of `vps.html`.

Both methods in a toll function take an array of floats that represent the speed on that road segment. The nth value in the array corresponds to the nth road segment. The input to these functions can be extended to support new logic, but the VPS will need to be changed.

The `tollRoad` method returns an array of toll values for each road segment. The `tollPrice` returns a mapping of toll price to color. Logic in either method can be completely rewritten.

Bibliography

[1] Awesomium. http://awesomium.com/.

[2] C4 engine overview. http://www.terathon.com/c4engine.

[3] Google maps apiv3. http://code.google.com/apis/maps/documentation/javascript/.

[4] Simulation of urban mobility. http://sumo.sourceforge.net/.

[5] P. Paruchuri et al. Multi agent simulation of unorganized traffic. *AAMAS '02*, 2002.

[6] P. Ferrari. Road pricing and network equilibrium. *Transportation Research Part B-methodological*, 1993.

[7] W. Hughes. Traffic externalities and single-family house prices. *Journal of Regional Science*, 1992.

[8] Amrik Kochhar. Simulation and verification of autonomous route planning behavior. Master's thesis, Massachusetts Institute of Technology, 2010.

[9] Jason Uh. Virtual urban traffic simulator. Master's thesis, Massachusetts Institute of Technology, 2011.