Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

ATLAS: A Language For Modeling And Simulating Urban Traffic

Visualization Of Traffic Models

By

Shannon Borho and Jan Pittner

Supervisor: Professor Gabriel Wainer

A report submitted in partial fulfillment of the requirements
of the 94.498 Engineering Project

April 4, 2003

Abstract

The report Visualization Of Traffic Model (hereinafter referred to as the report) written by Shannon Borho and Jan Pittner is the culmination of work done on the fourth year project ATLAS: A Language For Modeling And Simulating Urban Traffic - Visualization Of Traffic Models. The work was done under the supervision of Professor Gabriel Wainer. Work on the project began in late September 2002 and was completed with the delivery of the final report in April, 2003.

The report addresses several problems with the ATLAS Traffic simulation (ATV) system. The system had required the manual generation of ATLAS files, a tedious process that did not lend itself for rapid changes to the system input. The output of the system also suffered from a non-user friendly interface. The simulation output was converted into several different file types with primitive ASCII drawings of the simulation results. An earlier attempt by another project produced a primitive VRML program that was extremely limited in its capabilities. Thus, it was not easy for a user to define the input for the ATV system, or easily absorb the simulation results. These problems led to another issue – the slow and unwieldy process of testing the main simulation engine, known as cell-DEVS, with urban traffic simulations.

The solutions to these problems were addressed in two parts. For the front-end, the option of using existing GIS systems was to be explored. No existing system was found to meet all the requirements. The details of this investigation are outlined in the report. This result leads to the creation of a graphical front-end. The program allows the user to draw a small city section complete with roads, intersections, and decorations, and then parse the drawing to create a valid ATLAS file. The report gives a brief summary of the ATLAS language. The front-end, known as MAPS, was built on the JHotDraw framework. The frameworks details are outlined in the report, as well as the changes and new classes along with some relevant algorithms that were created for MAPS. Jan Pittner completed the MAPS subsystem. Working in parallel on the output side, Shannon Borho upgraded the visual output system. Starting with the primitive existing program,

the output went from a single segment of road with blocks as cars to a full-blown city section with realistic 3D graphics. Parsing the ATLAS file, building the city section in a VRML world and then mapping the simulation output results onto the system accomplished this result. Details of VRML, the parsing, and the processing of the simulation results into a visual display of cars moving on a city section are discussed. A brief overview of other components in the ATV system are covered as well, specifically the TSC and the simulation engine.

The report outlines the accomplishments made by Shannon Borho and Jan Pittner on the visualization of the ATLAS urban traffic simulation system. These accomplishments entail the creation of a graphical input program MAPS by Jan Pittner. MAPS provides the core ATLAS functionality using the drawing capabilities of the JHotDraw framework. As well, the output of the system can now be easily understood by watching it unfold in the 3D VRML world created by Shannon Borho. Finally, the final report itself provides a solid completion to the work on the project.

Acknowledgements

Jan Pittner would like to acknowledge all the members of the JHotDraw team at http://jhotdraw.sourceforge.net. Their determination, creativity and generous donation of time have led to a valuable drawing framework. I'd also like to acknowledge all those that have been involved in the ATLAS project up to this point, including Professor Wainer, Mariana Lo Tartaro, César Torres, et al.

Shannon Borho would like to thank the other members of the VRML GUI output team who provided the previous version of the VRML GUI and offered support when it was needed especially early in the term for the new version of the VRML GUI. A big thanks is in order for Huayi Du and Wenhong Chen. I would also like to express thanks to Professor Wainer for his support throughout the term.

Table Of Contents

1.0 INTRODUCTION (Written by Shannon Borho and Jan Pittner) ... 1
2.0 EXISTING GIS SYSTEMS (Written by Jan Pittner) ... 4
2.1 MapMaker ... 4
2.2 MapInfo ... 5
2.3 ESRI MapObjects ... 5
2.4 Conclusions Regarding Use of Existing GIS Systems ... 6
3.0 ATLAS (Written by Jan Pittner) ... 7
3.1 Segments ... 7
3.2 Crossings ... 8
3.3 Railnets ... 8
3.4 JobSites ... 9
3.5 PotHoles ... 9
3.6 Control Elements ... 9
3.7 Sample ATLAS File ... 10
3.8 ATLAS Conclusion ... 11
4.0 JHOTDRAW FRAMEWORK (Written by Jan Pittner) ... 12
4.1 JHotDraw Background ... 12
4.2 JHotDraw Design ... 12
4.3 Package Structure ... 13
4.4 Model View Controller Paradigm ... 14
4.5 Key Design Patterns Used ... 14
4.6 Core Classes ... 15
4.7 JHotDraw Design Conclusion ... 18
5.0 MAPS (Written by Jan Pittner) ... 19
5.1 Key Features of MAPS: ... 19
5.2 Plan for Building MAPS ... 19
5.3 MAPS Classes ... 20
5.4 Parsing the Drawing ... 22
5.5 Creating Segments from Lanes ... 23
5.6 Known Issues ... 25
5.7 Future Enhancements ... 25
5.8 MAPS Summary ... 26
6.0 TSC - TRAFFIC SIMULATOR COMPILER (Written by Shannon Borho) ... 27
7.0 N-CD++ SIMULATOR (Written by Shannon Borho) ... 28
7.1 Log Messages ... 29
7.2 DRAW Files ... 29
8.0 VRML (Written by Shannon Borho) ... 30
8.1 The VRML Node ... 30
8.2 Basic Shapes ... 32
8.3 Complex Shapes ... 32
8.4 Events ... 33
9.0 THE EXTERNAL AUTHORING INTERFACE (Written by Shannon Borho) ... 34
9.1 Obtaining a Reference to a VRML Node ... 34

10.0 THE VRML GUI SUBSYSTEM (Written by Shannon Borho) ... 36
10.1 Requirements ... 36
10.1.1 Functional Requirements ... 36
10.1.2 Actors of the System ... 37
10.1.3 Use Cases ... 37
10.1.4 Use Case Diagram ... 40
10.1.5 Initial Analysis Objects ... 40
10.1.6 Nonfunctional and Pseudo Requirements ... 41
10.2 Analysis ... 41
10.2.1 Entity Objects ... 41
10.2.2 Boundary Objects ... 42
10.2.3 Control Objects ... 43
10.2.4 Sequence Diagrams ... 44
10.2.5 Identifying Associations ... 48
10.2.6 Identifying Attributes and Operations ... 49
10.3 Design ... 50
10.3.1 Segment, Crossing and Car VRML Objects ... 50
10.3.1.1 The Car ... 51
10.3.1.2 The Segment and Crossing ... 51
10.3.2 Attributes of Crossings and Segments ... 52
10.3.3 Overcoming Visualization Problems with the PLAN file ... 55
10.3.4 Performing the Initialization ... 58
10.3.5 Overcoming Problems While Visualizing the Output ... 58
10.4 Implementation ... 61
10.5 Testing ... 65
10.5.1 Unit Testing ... 65
10.5.2 Integration Testing ... 66
10.5.3 System Testing ... 67
10.6 VRML GUI Conclusions ... 68
11.0 CONCLUSIONS (Written by Shannon Borho and Jan Pittner) ... 70
12.0 REFERENCES (Written by Jan Pittner and Shannon Borho) ... 73
13.0 APPENDICES ... 74
Appendix A - The MA (Model) File ... 74
Appendix B - An Example of a DRW File ... 76

List Of Figures

Figure 3.1 - Sample city section........ 10
Figure 3.2 - ATLAS code for the figure 3.1 city section........ 10
Figure 4.1 - JHotDraw Class Diagram........ 15
Figure 5.1 - A small city section.. 22
Figure 5.2 - RoadView of a road with parking, a stop sign, and roadwork 23
Figure 8.1 - The VRML group node 30
Figure 8.2 - Nested group nodes 31
Figure 8.3 - The transform node's key fields........ 31
Figure 8.4 - Adding a box to the VRML world 32
Figure 9.1 - A VRML world with a red box of size 1x3x5........ 34
Figure 9.2 - Java code to change the colour of a box to blue........ 35
Figure 10.1 - The use case diagram for the VRML GUI system........ 40
Figure 10.2 - The sequence diagram for the View City use case........ 45
Figure 10.3 - The sequence diagram for the Initialize City use case 46
Figure 10.4 - The sequence diagrams for the Move Cars use case........ 47
Figure 10.5 - The sequence diagrams for Load File and Invalid File respectively........ 48
Figure 10.6 - A simplified class diagram for the new VRML GUI system........ 49
Figure 10.7 - The class diagram for the VRML GUI with attributes and operations shown........ 50
Figure 10.8 - The car, segment and crossing VRML objects 51
Figure 10.9 - Two crossings, one with a stop light, the other with a stop sign........ 52
Figure 10.10 - ATLAS representation of a two-way street........ 55
Figure 10.11 - A two-way four-lane segment 56
Figure 10.12 - An example of two two-way four-lane segments that overlap at a crossing........ 57
Figure 10.13 - ATLAS representation of two two-way, two-lane segments 57
Figure 10.14 - An output message indicating that a car has appeared........ 59
Figure 10.15 - An output message indicating that a car has left a cell 60
Figure 10.16 - The VRML hierarchy for the VRMLNode 64
Figure 10.17 - The static view of the city with just segments and crossings 69
Figure 10.18 - Cars moving within the city........ 69

List Of Tables

Table 2.1 – Pros and Cons of MapMaker 4
Table 2.2 – Evaluation of MapMaker 5
Table 2.3 - Pros and Cons of MapInfo 5
Table 2.4 – Evaluation of MapInfo 5
Table 2.5 - Pros and Cons of MapObjects 5
Table 2.6 – Evaluation of MapObjects 6
Table 4.1 - JHotDraw Design Patterns 15
Table 4.2 – JHotDraw Contributed Package 16
Table 4.3 - JHotDraw Figures Package 16
Table 4.4 - JHotDraw Framework Package 16
Table 4.5 - JHotDraw JavaDraw Package 17
Table 4.6 - JHotDraw Util Package 17
Table 4.7 - JHotDraw Standard Package 17
Table 5.1 – MAPPropertyPanel Class 20
Table 5.2 – RoadFigure and LaneDirection Figure Classes 20
Table 5.3 – IntersectionFigure Class 21
Table 5.4 – MAPDecoration Class 21
Table 5.5 – CityDrawingView Class 21
Table 5.6 – RoadDrawingView Class 21
Table 5.7 – ATLASParser Class 21

1.0 INTRODUCTION

Traffic in an urban area is a problem in almost every major city in North America. Gridlock occurs on a daily basis on major routes into and out of a city and traffic jams happen all the time within a major centers. This large amount of traffic can be predicted ahead of time, as there already exists sensors reading the amount of traffic on particularly busy roads. The existing problem is how to determine where the traffic is going to build up and how soon and what would be the optimal route to follow to avoid the gridlock.

Another problem in any urban city is construction. Construction happens on a yearly basis in many urban cities and affects not only traffic at the point of the construction but kilometers ahead and behind as traffic can build up at the bottleneck. People want to know what will be the result of setting up a construction zone ahead of time to determine if it would be a good decision to shut down a lane of traffic for weeks at a time to complete construction.

The best way to determine results is through a simulation. If we can predict the amount of traffic that is flowing into a specified city section, and can predict how individual drivers are going to react on city streets, then we can predict how the city traffic is going to develop in advance. A simulation can give city planners the ability to make modifications to an existing city map or even create their own city section and see the results based on parameters that they set that determine the amount of traffic that will be entering the city section and the normal destined route for the traffic when the get to the city section.

If a simulation is performed according to sensors that are currently set up ahead of a known location for traffic buildup, we can predict the best possible route a driver can take to get to a specific location and avoid the hassles of heavy traffic. Once we determine the best route for the drivers, then via traffic signs on the side of the road or even existing GIS (Geographical Information System) systems that are becoming predominant in today's cars we can inform the drivers of what was determined to be the most efficient route to a particular location.

This report discusses the visualization of the ATV (ATLAS Traffic Visualization) system that is based on ATLAS (Advanced Traffic LAnguage Specifications). ATLAS is a traffic specification language used to generate simulation specifications for the cell-DEVS (cellular Discrete EVent System specification) system. Using ATLAS, a small city section can be modeled on which to simulate urban traffic. These city sections are composed of streets, intersections, railnets, construction sites and various decorations such as potholes and stop signs. However, generating the ATLAS files that describe these city sections is a tedious chore to do by hand. This slows down the entire process of simulating urban traffic, which in turn slows down the process of testing the cell-DEVS engine. We need a way to allow designers to focus their attention on creating the city instead of the semantics of ATLAS. The developed city can then be compiled and simulated and produce a results file from the simulation. From this results file, it is very difficult for a human to make sense of the results. People would be more apt to use the system if the results contained a graphical output representing traffic flowing through a city. This project's aim was to make the system more visual and thus user-friendlier. A user should be able to easily draw a city section, have the simulation file automatically generated, and the output displayed graphically. The visualization of ATLAS would thus make simulation of urban traffic using the cell-DEVS engine a simple process with more understandable results.

The solution involved two main aspects. One, a program had to be developed to allow users to easily generate ATLAS files by letting the user draw and decorate the city section. The program should also eliminate the need for the user to comprehend ATLAS specific abstractions (such as segments). The program would have to then parse this drawing to create the ATLAS file, which in turn could be compiled and simulated. Next, the output of the simulation engine had to be parsed and displayed in a human friendly manner. A program utilizing VRML was extended upon to give the user a three-dimensional visual output, displaying car shapes moving along the defined city section removing the need for the user to read text based results.

The group managed the following accomplishments, adding extra visualization subsystems to the current ATV system. A beta graphical drawing program known as MAPS was developed. MAPS is based on the JHotDraw graphical drawing framework. A user can draw a small city section with several supported ATLAS components on which to model urban traffic. The

program parses the drawing to generate a valid ATLAS file. The visual output component called the VRML GUI subsystem is able to display the text-based simulation results in a 3D virtual reality world. This allows the user to easily understand the results of the simulation on a web-based interface allowing the use of the tool over a network. This report is the culmination of the project. It outlines the problem, its solution, and the accomplishments made by the group members starting with the MAPS subsystem and progressing to the VRML GUI subsystem.

The report starts by describing the results of our exploration of existing GIS systems. GIS was explored to determine if there were any existing systems that could be imported by the ATV system giving a starting structure for a city section. Next we describe ATLAS, its semantics and uses for the MAPS subsystem. The TSC (Traffic Simulator Compiler) compiler and N-CD++ simulator are briefly touched upon before getting into the VRML GUI subsystem. Finally, we will state our conclusions and give some references to works cited.

2.0 EXISTING GIS SYSTEMS

Prior to work on the MAPS program, several existing GIS systems were examined for potential use with the ATLAS system. The requirements for such an existing system were set out by the authors as:

1. Low cost or open source
2. Can access existing urban maps
3. Maps can be exported and then parsed into the ATLAS format
4. The system must also allow the user to easily add/modify decorations such as stop signs, pot holes, road work, parking etc.

The following programs were examined for use with the ATLAS system.

2.1 MapMaker

This program comes in two forms. The free version, known as Gratis, and the Pro version. The free version will be examined by virtue of it being free. The source is not available for MapMaker, so direct enhancement for ATLAS requirements would require another program. This automatically increases complexity for the end-user. There are however, benefits with MapMaker that could make it a worthwhile addition to the ATLAS system, such as its ability to provide a 3D perspective view of a location.

PRO	Can import existing GIS data, Available for free
CONS	Many extra features that distract user from sole focus of roads and their decorations. Takes several hours to learn as it is a full featured map making program No easy way of adding decorations such as potholes, parking, etc. Would require the user draw the map, save the output, load another program to parse that map onto which the user could draw or manually input parameters such as parking and other decorations (eg road speed, etc.)

Table 2.1 – Pros and Cons of MapMaker

Grade	*Requirement*
PASS	Low cost or open source
PASS	Can access existing urban maps
PASS*	Maps can be exported and then parsed into the ATLAS format
FAIL*	Allow the user to easily add/modify decorations such as stop signs, pot holes, road work, parking etc.

* indicates that an additional program would have to be used, increasing complexity.

Table 2.2 – Evaluation of MapMaker

2.2 MapInfo

A full featured enterprise level mapping application.

PRO	Supports GIS databases, importing of existing maps.
CONS	Expensive, not open source, does not allow easy addition of ATLAS specific functionality.

Table 2.3 - Pros and Cons of MapInfo

Grade	*Requirement*
FAIL	Low cost or open source
PASS	Can access existing urban maps
FAIL	Maps can be exported and then parsed into the ATLAS format
FAIL	Allow the user to easily add/modify decorations such as stop signs, pot holes, road work, parking etc.

Table 2.4 – Evaluation of MapInfo

2.3 ESRI MapObjects

$5000 (USD) collection of Java objects for GIS systems. A framework/collection of specifically designed JavaBeans for GIS systems.

PRO	Readily allows for use of existing GIS data such as ESRI ArcView StreetMaps extension
CONS	Very expensive, contains many more objects than needed

Table 2.5 - Pros and Cons of MapObjects

Grade	*Requirement*
FAIL	Low cost or open source
PASS	Can access existing urban maps
PASS*	Maps can be exported and then parsed into the ATLAS format
PASS*	Allow the user to easily add/modify decorations such as stop signs, pot holes, road work, parking etc.

* indicates that an additional programming and design would be required.

Table 2.6 – Evaluation of MapObjects

2.4 Conclusions Regarding Use of Existing GIS Systems

After analysis of the aforementioned products, it was concluded that a solution could be created to meet the requirements in a much more affordable manner. The existing GIS systems analyzed (except for ESRI MapObjects) also didn't provide an easy way to integrate ATLAS functionality. Additionally, the discovery of the JHotDraw framework gave even more reason to create an in-house solution for the visualization of ATLAS.

3.0 ATLAS - ADVANCED TRAFFIC LANGUAGE SPECIFICATION

ATLAS is a specification language for describing small city sections. The relevance of ATLAS in the ATV system has already been discussed, so this section will briefly cover the formal specification itself. For a complete detailed look at ATLAS, the reader is referred to the papers "Definición de un language de especificación para simulación de tráfico urbano siguiendo el paradigma Cell-DEVS" by Davidson et al and an English paper "TSC – Traffic Simulator Compiler" by Tartaro et al [2]. The following section contains excerpts from that paper.

Of main interest to the authors of this paper is the specification of the city section itself. The following ATLAS components are described: segments, crossings (also known as intersections), rail-nets, jobsites (roadwork), holes (potholes) and control elements (stop signs, etc.). Their implementation in the MAPS program is discussed in the MAPS section of this report.

3.1 Segments

Segments are the only mandatory components in an ATLAS file. They describe the general layout of the roads, as well form the basis on which ATLAS decorations (such as stop signs, potholes, and parking) are added. A segment can be a single lane or a road with several lanes. The segment can have parking or no parking (but it may not have parking in one location but not another). Segments cannot cross other segments. A crossing (see below) must be placed between two or more other segments. The names of segments are important, as they are used to describe where various decorations are placed.

This is the format for an ATLAS segment [2]:

id = p1, p2, lanes, shape, direction, speed, delay, parkType

where,

id*: alphanumeric, is the segment identifier.*

p1*: point, indicates the begining of the segment.*

p2*: point, indicates the ending of the segment.*

lanes*: integer, amount of lanes of the segment.*

shape*: [****curve|straight****], defines the shape of the segment.*

direction: *[**go**|**back**], defines the car movement sense of the segment.*
speed: *integer, defines the maximum car speed in the segment.*
delay: *integer, defines the car delay in the parking lanes.*
parkType: *[**parkNone**|**parkLeft**|**parkRight**|**parkBoth**], defines the parking lanes of the segment.*

A collection of segments is delineated in the ATLAS file by the "begin segments" and "end segments" statements.

3.2 Crossings

Crossings, also known as intersections, are located between two or more segments. In cell-DEVS they are modelled as roundabouts. The following is the format of an ATLAS crossing [2]:

id = p, speed, tLight, crossHole, delay, pout

where,
id: *alphanumeric, is the crossing identifier.*
p: *point, defines the placement of the crossing.*
speed: *integer, defines the maximum car speed in the crossing.*
tLight: *[**withTL**|**withoutTL**], indicates if the crossing has traffic lights.*
crossHole: *[**withHole**|**withoutHole**], indicates if the crossing has holes*
delay: *integer, defines the car delay produced by the presence of a hole inside the crossing.*
pout: *integer, defines the probability of getting out from the crossing.*

Similiarly to segments, crossings are grouped in the ATLAS file and surrounded by "begin crossings" and "end crossings".

3.3 Railnets

Railnets are used to model railroads crossing segments. Railnets do not split a segment in two. The following describes a railnet in ATLAS [2]:

$$id = (t_1, d_1)\ \{,(t_i, d_i)\},\ delay$$

where,
id: *alphanumeric, is the railnet identifier.*
t_i: *alphanumeric, indicates a segment identifier crossed by the railnet.*
d_i: *integer, indicates the distance between the beginning of the segment* t_i *and the railnet.*

delay*: integer, indicates the car delay crossing the segments cells affected by the railnet.*

3.4 JobSites

Jobsites, also known as road work, indicate where there is construction on the road. [2]

in t : firstlane, distance, lanes, delay

where,

t*: alphanumeric, is the segment identifier where the jobsite is placed.*

firstlane*: integer, indicates the first lane affected by the jobsite.*

distance*: integer, indicates for the first lane the distance between the central column of the jobsite and the begining of the segment.*

lanes*: integer odd, indicates the amount of lanes affected by the jobsite.*

delay*: integer, not used and reserved for future utilization.*

3.5 PotHoles

The holes section is delimited by the sentences **"begin holes"** and **"end holes"** and contains one or more sentences with the following syntax [2]*:*

in t : lane, distance, delay

where,

t*: alphanumeric, is the segment identifier where the hole is placed.*

lane*: integer, indicates the segment lane where the hole is placed.*

distance*: integer, indicates the distance between the hole and the begining of the segment.*

delay*: integer, indicates the car delay crossing the segments cells affected by the hole.*

3.6 Control Elements

The control elements section is delimited by the sentences **"begin ctrElements"** and **"end ctrElements"** and contains one or more sentences with the following syntax [2]:

in t : ctrType, distance, delay

where,

t*: alphanumeric, , is the segment identifier where the control element is placed.*

ctrType*: [****sawhorse****|****depression****|****intersection****|****saw****|****stop****|****school****], defines the type for the control element.*

distance*: integer, indicates the distance between the control element and the begining of the segment.*

delay*: integer, indicates the car delay crossing the segments cells affected by the control element.*

3.7 Sample ATLAS File

The following (figure 3.2) is a sample ATLAS file from "TSC – Traffic Simulator Compiler" by Tartaro et al. It describes theb city section in figure 3.1 As the reader can see, it would be a tedious process even to make relatively simple changes to the layout of the roads, or adding parking to a certain section of a road.

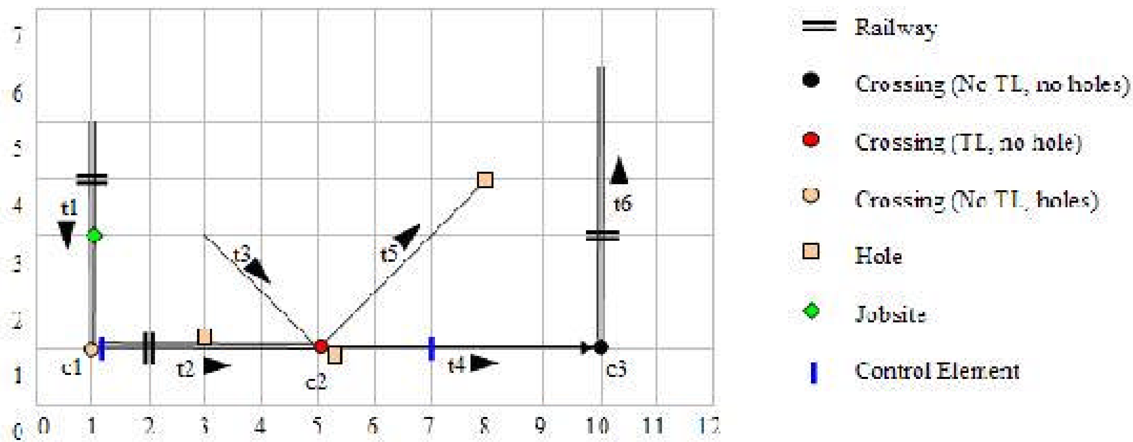


Figure 3.1 - Sample city section taken from "TSC – Traffic Simulator Compiler" by Tartaro et al [2].

```
begin segments
  t1 = (1,5),(1,1),2,straight,go,21,1100,parkNone
  t2 = (1,1),(5,1),2,straight,go,22,1200,parkRight
  t3 = (5,3),(5,1),1,straight,go,23,1300,parkNone
  t4 = (5,1),(10,1),1,straight,go,24,1400,parkNone
  t5 = (5,1),(8,4),1,curve,go,25,1500,parkNone
  t6 = (10,6),(10,1),2,straight,back,26,1600,parkLeft
end segments

begin crossings
  c1 = (1,1),11, withoutTL, withHole, 221, 111
  c2 = (5,1),12, withTL, withoutHole, 222, 112
  c3 = (10,1),13, withoutTL, withoutHole,223, 113
end crossings

begin railnets
  rn1 = (t1,1),(t2,1),(t6,2),331
end railnets

begin jobsites
  in t1 : 1,2,1,441
end jobsites

begin holes
  in t2 : 1,2,553
  in t4 : 1,3,557
  in t5 : 1,3,559
end holes

begin ctrElements
  in t2 : stop,2,651
  in t4 : sem,2,655
end ctrElements
```

Figure 3.2 - ATLAS code for the figure 3.1 city section [2]

3.8 ATLAS Conclusion

ATLAS provides a much simpler to read and work with format for describing city sections than pure cell-DEVS specification. However, ATLAS can be simplified even further by eliminating the need to work with the ATLAS language itself. This can be done by letting a user draw the city section and having a program automatically parsing the drawing to create a valid ATLAS file.

4.0 JHOTDRAW FRAMEWORK

The JHotDraw Framework [1] forms the foundation for the MAPS project. As such, it requires key representation in this document to give the reader a firm understanding of how it was used to form the MAPS program. Its background, design, core classes, and key design patterns used will be discussed. Understanding the JHotDraw framework was the first requirement to be met before MAPS could be developed.

4.1 JHotDraw Background

HotDraw was initially designed by Erich Gamma (of Group of Four fame) and Thomas Eggenschwiler. First written in SmallTalk, HotDraw's was initially just an exercise in design. It was since then ported to Java and renamed JHotDraw. The code was released as open source. Its goals, from the JHotDraw homepage [1], are:

- *gain a wider audience for this framework among developer*
- *build new applications based upon JHotDraw*
- *let application development influence the development of JHotDraw*
- *add new and advanced features*
- *drive its further development*
- *port JHotDraw to new Java GUI toolkits*
- *enhance and refactor the exisiting code*
- *identify new design patterns and refactorings*
- *make it an example for a well-designed and flexible framework*
- *examine the relevance of new Java APIs to JHotDraw (e.g. usefulness of Java 2D API for JHotDraw)*
- *learn and to have fun.*

4.2 JHotDraw Design

The simplicity of JHotDraw's design should not in any way lead one to conclude that JHotDraw is a weak drawing program. Nor should one conclude that its implementation is

trivial. Understanding its design will however allow a user to extend its functionality to meet his or her new requirements.

The design will be covered firstly by looking at the package structure. Next, key design patterns will be analyzed. This is followed by the core JHotDraw classes, and how they fit into the aforementioned design patterns. Armed with this knowledge, extending the framework to have ATLAS functionality proves to be an interesting challenge in the application of design patterns and implementation.

4.3 Package Structure

Knowing what goes where is the first step in learning an existing system that will be extended. Fortunately, the JHotDraw package structure is designed so that cohesion is high between files in each package. The files are logically placed in the following structure [1]:

CH.ifa.draw.util

This package provides generally useful utilities that can be used independently of JHotDraw.

CH.ifa.draw.framework

The framework package includes the classes and interfaces that define the JHotDraw framework. It doesn't provide any concrete implementation classes.

CH.ifa.draw.standard

The standard package provides standard implementations of the classes defined in the framework package. It provides abstract classes that implement a framework interface and provide default implementation. The convention is to prefix such classes with Abstract, e.g., AbstractFigure, AbstractHandle. Classes that implement a framework interface and can be used as is start with Standard, e.g., StandardDrawing, StandardDrawingView.

CH.ifa.draw.figures

A kit of figures together with their associated support classes (tools, handles).

CH.ifa.draw.contrib

Classes that where contributed by others, such as the MDI classes.

CH.ifa.draw.application

The application package defines a default user interface for standalone JHotDraw applications.

Additionally, there are several other packages of lesser importance not discussed here.

4.4 Model View Controller Paradigm

The basis of almost any program that interacts with the user is the Model View Controller paradigm (MVC for short).

In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task. The ***view*** *manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. The* ***controller*** *interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the* ***model*** *manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).*

Quoted from "How To Use Model View Controller" [6]

How each of these three parts of the MVC are handled in JHotDraw is discussed in the Core Classes section of this paper.

4.5 Key Design Patterns Used

There are several key design patterns used by the JHotDraw framework. Understanding these patterns is essential to understanding how the framework functions. There are other patterns used in addition to those listed here (such as Strategy and Singleton) however these four patterns are the absolute core and thus will be discussed further.

Pattern	***Description***
Observer	The observer pattern allows you to have an object notify other objects that its state has changed.
Command	The command pattern abstracts user input as an object. Menus and parameterized calls can thus be quickly and easily created. This pattern also allows for undo/redo to be implemented, critical for a program with heavy user interaction and where the user can be prone to change their minds.

Pattern	*Description*
Composite	The composite pattern makes it possible to treat individual objects and compositions of these objects the same way. Drawings will often be composed of many smaller drawings, which themselves can be composed of drawings as well. It is thus possible to treat a drawing of a square the same way as a drawing of a square with a square in it.
Prototype	The prototype pattern allows for the specification of an instantiation of a class and for that instantiation to be copied to create a new object.

Table 4.1 - JHotDraw Design Patterns

See [7] for more info on each pattern.

4.6 Core Classes

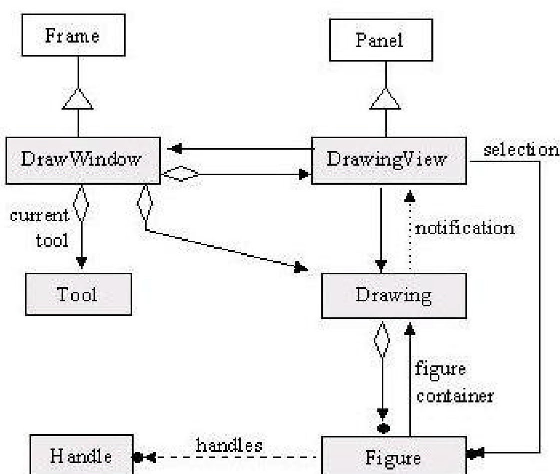


Figure 4.1 - JHotDraw Class Diagram (from JHotDraw documentation [1])

The following classes are of utmost important in the operation of JHotDraw. As such, they are discussed to give the reader a firm understanding of their design and capability.
It is these classes that must be understood in order to be able to expand upon the JHotDraw framework. The classes not mentioned play a supportive role which for the most part the user need not concern themselves with.

Package	*Class Names*	*Description*
contributed	CustomSelectionTool	Tool used to allow the user to select different drawings. Notifies selection listeners of changes. The is extended to allow further capabilities such as right clicking pop-up menu's, double clicking, etc.
	MDI_DrawApplication	Allows for multiple documents to be opened in the same program, as well as multiple views.

Table 4.2 – JHotDraw Contributed Package

Package	*Class Names*	*Description*
figures	FigureAttributes LineFigure AttributeFigure RectangleFigure AbstractFigure	Assortment of figures that store important information about themselves: what they look like, where they are drawn, how they are drawn, etc. The foundation is the AbstractFigure which stipulates what a figure class must have in order to be a figure.

Table 4.3 - JHotDraw Figures Package

Package	*Class Names*	*Description*
framework	Drawing	The interface that defines a drawing.
	DrawingChangeEvent	The event that is broadcast when a drawing is changed.
	DrawingEditor	the interface for the editor for drawings.
	DrawingChangeListener	the observer that receives broadcasts of DrawingChangeEvents
	FigureChangeEvent FigureChangeListener FigureSelectionListener ViewChangeListener	Assortment of various events, listeners that implement the Observer pattern to allow the Model to update the view when the controller affects a change to the drawing, or which current view is used.

Table 4.4 - JHotDraw Framework Package

Package	***Class Names***	***Description***
JavaDraw	JavaDrawApp	a skeleton application that can be modified to meet a user's particular needs.

Table 4.5 - JHotDraw JavaDraw Package

Package	***Class Names***	***Description***
Util	Command	implementation of the Command pattern.
	CommandButton	brings the Command features to Button form

Table 4.6 - JHotDraw Util Package

Package	***Class Names***	***Description***
Standard	AbstractCommand	the implementation of the Command pattern, forms the basis of the commands present in programs (eg, copy, cut, paste, etc.)
	CompositeFigure	the implementation of the Composite pattern as an abstract class. A figure can contain other figures.
	CreationTool	the abstract factory implementation, allows for the dynamic creation of figures such as rectangles, lines, and ellipses (among others) on the fly.
	GridConstrainer	class that serves the need to constrain user input to a grid.
	DrawingView	interface of the View component of the MVC paradigm. StandardDrawingView provides a ready to use implementation.
	ToolButton	Class that encapsulates the behavior of a button which can have different tools assigned to it.

Table 4.7 - JHotDraw Standard Package

4.7 JHotDraw Design Conclusion

Understanding the interfaces of these core classes, knowing how they apply the discussed design patterns, and having a plan on how to extend and re-use the framework was the first step in building MAPS. Although documentation is sparse for the JHotDraw framework, by understanding the key design elements the user can learn the system quite rapidly. Understanding how to fix, adapt, or upgrade existing systems is a common problem for newly graduated engineers in junior level engineering positions. JHotDraw proved to be a well designed training ground for future work with existing small to medium sized systems. Additionally it proved to be an excellent framework for the basis of the MAPS program.

5.0 MAPS

The goal of MAPS is to provide a visual front-end for the ATV. MAPS allows users to draw small city sections which are then automatically parsed into ATLAS files. Users can quickly and easily change the layout of the city section, as well as ATLAS specific parameters. MAPS eliminates the need to know the ATLAS language, and it dramatically reduces the time it takes to create ATLAS .plan files. This allows for rapid simulation of urban traffic, which in term tests the cell-DEVS engine.

5.1 Key Features of MAPS:

- Intuitive interface allows user to quickly draw streets
- Intersections are automatically generated for the user
- Roads, instead of segments, allow the user to ignore ATLAS abstractions
- Decorations can be easily added, changed, or removed
- ATLAS parameters can be easily modified to change simulation parameters
- Save/Load files
- Parses user's drawing into ATLAS format
- Undo/Redo
- Multiple document interface

5.2 Plan for Building MAPS

The first step was understanding the JHotDraw framework. This was done mainly by running the provided samples, reading through the code and learning about the underlying design patterns used. The requirements for MAPS were simple – allow the user to quickly and easily create a small city section which will then be parsed by the program to create valid ATLAS output. The user should be able to control the layout and attributes of the streets, as well as the decorations on those streets.

Given these requirements, the required classes were devised and implemented. Most are inherited or implement interfaces from the JHotDraw framework.

The following sections will briefly outline the more important classes created, which classes they inherit from, which interfaces they implement, and their key role. Should the algorithm used be of interest (specifically parsing for ATLAS) it is also discussed.

5.3 MAPS Classes

Class	**Superclass**	**Implements**
MAPPropertyPanel	*JPanel*	FigureChangeListener, ViewChangeListener, DrawingChangeListener, SelectionChangeListener
Function	Property panel which is constantly updated with the current view/selection (it is an observer of these events). User has to be able to input the various ATLAS parameters for the object.	

Table 5.1 – MAPPropertyPanel Class

Class	**Superclass**	**Implements**
RoadFigure *LaneDirectionFigure*	*LineFigure*	
Function	*RoadFigure* is the primary figure used for ATLAS functionality. It stores the road information such as name, speed, curvature, and so on. It extends the store/read functionality dictated by the Storable interface so that a user can save their map for later use. *LaneDirectionFigure* overrides certain methods in order to prevent the arrow from being moved around on the screen, but still be accessible via mouse clicks to change the direction.	

Table 5.2 – RoadFigure and LaneDirection Figure Classes

Class	**Superclass**	**Implements**
IntersectionFigure	*EllipseFigure*	
Function	Just like *LaneDirectionFigure*, the primary purpose of extending this class was to override certain methods to prevent moving the figure around (all crossings are automatically generated in MAPS) but still be accessible via mouse clicks.	

Table 5.3 – IntersectionFigure Class

Class	**Superclass**	**Implements**
MAPDecoration	*Object*	serializable, storable
Function	Simple class used to store MAP decorations and their information (eg, location on the road, affected lanes, etc). If available, the figure associated with the decoration is stored as well (eg, RectangleFigure with a blue "fillColor" for parking.)	

Table 5.4 – MAPDecoration Class

Class	**Superclass**	**Implements**
CityDrawingViev	*StandardDrawingView*	
Function	Provides the background view with grids, rulers.	

Table 5.5 – CityDrawingView Class

Class	**Superclass**	**Implements**
RoadDrawingView	*CityDrawingView*	
Function	Extends the CityDrawingView by painting on the road that the user has selected. The user can then draw road decorations onto that road.	

Table 5.6 – RoadDrawingView Class

Class	**Superclass**	**Implements**
ATLASParser	*Object*	
Function	Parses the user's drawing. Validates the drawing and generates the ATLAS code.	

Table 5.7 – ATLASParser Class

5.4 Parsing the Drawing

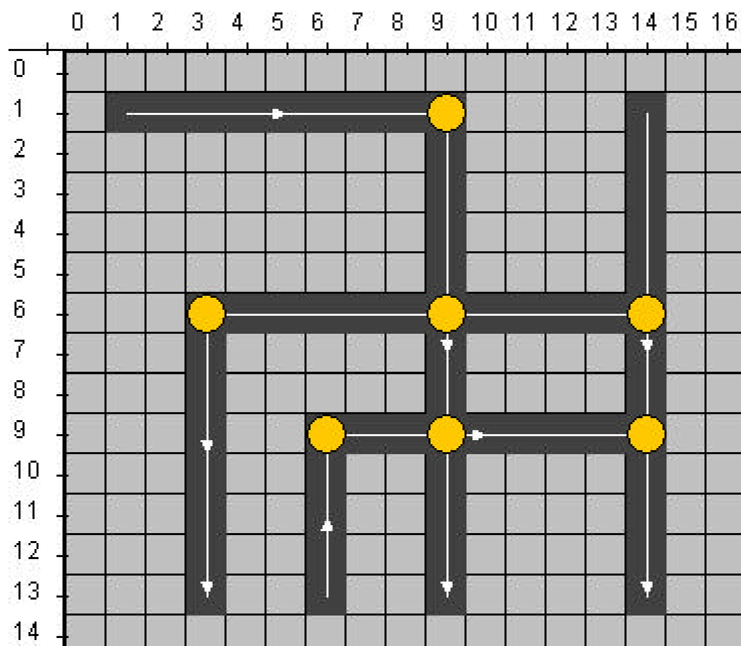


Figure 5.1 - A small city section. The orange circles are crossings. Note that the arrow indicates user-drawn direction of the road, not necessarily the direction of the lanes in that road.

The parser first removes and stores old crossings to preserve their settings (such as *pout*). City level decorations are then stored (e.g. rail-nets). All roads are then stored.

The parser then loops through each road to see if it intersects with other roads. If a previously generated crossing exists at the intersection point it is used. If it isn't, a new intersection is created. The parser also checks to see if the road contains a rail-net. If it does, a boolean value is set to inform the parser to check which segment the rail-net belongs to as the segments are created.

A new list of breakpoints (a simple class that stores the location of the cut, and the type – eg, start of the road, end of the road, intersection, parking start, parking end) will determine how to cut up the road into segments. This list does not contain intersections that do not form segments (eg, at the start and end of the road being segmented).

Breakpoints can also created by parking, as the parking can be on only certain parts of the road. The parser loops through the parking decorations of that road for each lane to create breakpoints for that lane. Each lane is its own segment, which can be further segmented by parking decorations on that lane. Each segment must have a unique identifier. This unique identifier is tagged to other decorations that that lane is affected by (eg, roadwork spanning multiple lanes, potholes, etc).

The lane breakpoints are then sorted and the segments are created, named and decorated. The process repeats for as many lanes and as many roads. The creation of segments from lanes is discussed further below. The segments and decorations are stored in vectors for each. The parser goes through the vectors for the segments and various decorations.
The crossings are parsed and their ATLAS code is added to the vector which will then be looped through to generate the ATLAS file.

5.5 Creating Segments from Lanes

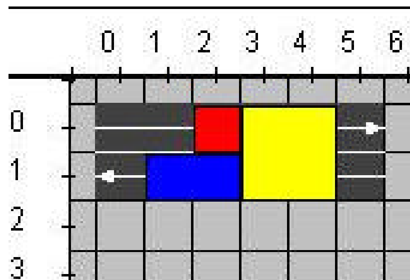


Figure 5.2 - RoadView of a road with parking, a stop sign, and roadwork

A road may have multiple lanes, multiple intersections, multiple places to park each with different parameters. Additionally each part of the road can have other decorations like potholes and stop signs. The process of portioning a road is described above. This section describes how the segments themselves are created from a lane. Note the above figure contains four ATLAS segments – one lane with the direction “go”, and three with reverse. Two of the “reverse” segments have no parking, while one (the part of the lane that is colored blue) does. The user need not worry about creating the segments, naming them, and ensuring the decorations are attached to the correct segment. This is all done automatically by MAPS.

To parse the lane valid intersections found for the road are added to the lane's breakpoints. Next, breakpoints are created from parking. The breakpoints are then sorted to be in ascending order (from the start of the road to the end).

The following rules apply to parsing a lane's decorations:

- There is no parking available at the start and end of a lane (the first and last grid units may not have parking)
- Parking objects may not overlap.
- Parking objects may not be intersected by another road – that is, there is no parking allowed in an intersection.
- Segments with parking may not contain a rail-net

These rules were formed by looking at typical rules for real life streets, as well as make parking parsing logic simpler. The user is informed if any decorations violate the rules, and the locations of the invalid objects are displayed.

The segments are created if the parking decorations are valid. The segments are then decorated by looping through the decorations for that lane and checking to see if the decoration (such as potholes, stop signs, etc) lay on that particular segment. If the user created a decoration that has a length of more than one grid size, then multiple decorations are added as many times as the length requires. The decorations differ in their position from the start of the road.

5.6 Known Issues

There are currently several known issues with the latest build of MAPS:

- missing option in ATLAS menu to generate intersections
- property panel layout needs adjusted to give cleaner looking JTextField edit boxes
- several decorations have not been implemented yet
- if a user closes the Road View without applying the changes, the changes are lost. Likewise, if a user makes changes to properties of a road, but then selects another figure, those changes are lost.
- "Update" button does not support undoable operations.
- Graphical artifacts occur when the user draws an object. The image must be manually be refreshed (by moving the entire window) in order for the artifacts to be removed.
- Multiple road views can be opened for the same road.
- Transparent/translucent objects are required for decorations where multiple decorations must be used on the same grid square. That, or priority in the Z level for an object should be established (eg, stop signs and potholes have a higher Z level than parking).
- It can be difficult to properly click an object (due to the use of LineFigure)
- this release has not been tested with the ATV system due to technical issues with the TSC (Traffic Simulation Compiler, see section 6.0).

5.7 Future Enhancements

Future enhancements to MAPS may include the ability to parse existing GIS data from existing map formats.

5.8 MAPS Summary

MAPS was created to do one thing – allow a user to quickly and easily draw a small city section with decorations and have the program automatically create an ATLAS file. This solved two problems. First, a user no longer has to understand ATLAS in order to manually generate ATLAS files (a tedious and slow process). Secondly, ATLAS files can be quickly generated with different parameters in order to test the entire ATV system, at the heart of which exists the cell-DEVS engine. With valid simulation files for urban traffic, the cell-DEVS engine can be tested in yet another operational field. This can potentially broaden the wide number of applications for which cell-DEVS is a practical and useable simulation engine.

- The *need*: An easy to use graphical tool to quickly provide valid input for the ATLAS-based urban traffic simulator.
- The *solution*: MAPS allows a user to quickly and easily generate the valid input.
- The *accomplishment*: MAPS provides the core ATLAS functionality with the drawing capabilities of the open source JHotDraw framework.
- The clever *name*: MAPS Makes ATLAS Pretty Simple.

6.0 TSC - TRAFFIC SIMULATOR COMPILER

The main goal of the TSC (Traffic Simulator Compiler) is to take an ATLAS input representation of a city section, generated either by the MAPS subsystem or written manually, and generate, as an output, an ma file containing the appropriate simulation models. The models generated will be written to a file that follows the supported structure of an input file for the N-CD++ simulator. Once the model file is executed using the N-CD++ simulation tool, the output file from the simulator can be analyzed to determine the traffic flow within the ATLAS defined city section. Manually defining the model file generated by the TSC would be very difficult and tedious due to the high level of details that go into a city section. The TSC uses a set of code generation templates for each element of the model to come up with an overall output ma file [2]. A portion of an ma file has been included in appendix A to illustrate the complexity of the model file. This compiler was developed several years ago by a Master's student but still has design issues and bugs that have to be dealt with.

7.0 N-CD++ SIMULATOR

N-CD++ is a tool that was also developed several years ago for simulating cellular space models using the DEVS (discrete event system specification) paradigm and asynchronous cellular automates. DEVS is a hierarchical simulation method that separates modeling from simulation enabling correct, efficient, event-based, distributed simulation. Cell-DEVS (Cellular DEVS) is an extension to the DEVS formalism that allows for defining cellular models that can be integrated with other DEVS models. Each cell of a cellular space is defined to be an atomic DEVS model with its own particular value [2]. At some specified point in time, each cell will use inputs from neighboring cells to compute the future value of the cell. The new value of a specific cell is dependant upon a set of rules that are defined in the model file along with each cell's neighborhood.

The cell-DEVS formalism is used extensively in the simulation of ATLAS models. We can think of a road segment containing a specific number of cells that can hold at most one car with the value of each cell depending whether or not the cell currently contains a car. The future value of a cell will depend on the current value of the cell and the value of the cell to the right and left of it. For example, imagine we have a one-lane segment with five cells. The future value of any one cell depends on its neighborhood. The neighborhood for cell (0,0) could be defined to be cell (-1,0), (0,0) and (1,0). The rule that determines the next value within a cell could be defined to be if a car exists in the neighboring cell to the left (-1,0) and no car exists in the current cell (0,0), then after the next event time a car will exist in the current cell. Another rule could be if a car is present in a specified cell (0,0) and no car exists in the cell of the neighbor to the right (1,0), then there is no longer a car is present in the specified cell (0,0). For example, if a car exists in cell (2,0) but no car exists in cell (1,0) or (3,0), then after a delay time, the cell (3,0) will contain a car, and cell (2,0) will no longer contain a car. The rules get more complex as the number of lanes is increased because the neighborhood is extended. If there are three lanes, then the neighborhood for cell (0,0) in the middle lane could be (-1,1), (-1,0), (-1,-1), (0,1), (0,0), (0,-1), (1,1), (1,0), and (1,-1). There will be rules for many different situations and they also can get complex. The simulator evaluates the entire model of segments and crossings

using the two-dimensional cell-DEVS based simulation tool N-CD++ following the rules and neighborhoods defined in the model file for each model.

7.1 Log Messages

Whenever the simulator detects that the value within a cell has changed from a one to a zero or vice versa, an output event is generated. These messages can be recorded in a log file. The output messages are indicated by the prefix Y message at the beginning of the statement. The message informs the user of the model name that was changed, the cell location and the new value that the cell contains. The simulator actually keeps track of all messages sent between cells. At time 0, there are many initialization messages that are passed between and within models. Also during the simulation, there is input messages passed into the system along with other messages passed between segments. These messages are logged by the simulator as well but they are marked by either an I, *, X or D delimiter and with regards to viewing the output, can be ignored. The only relevant message for understanding what is happening in the output is the Y messages.

7.2 DRAW Files

Another application that has been developed in the past is the DRAWLOG program. This program will interpret the log file generated by the N-CD++ simulator and generate a text-based output for individual segments. The draw file shows the time and the status of the cells in a segment. A sample draw text file has been included in appendix B. From this sample draw file, the limitations of the text-based outputs are obvious. For starters, only one segment can be viewed at one time. Generally, city designers want to be able to see how the traffic is progressing in their entire city. They would want a tool that easily allows them to choose a location to focus their attention and possibly watch traffic flow from segment to segment. The draw file does not allow for this. Also the text-based output is not easy to follow. A graphical tool with car shapes representing a car and no car present for a car not present in a cell, showing the entire city at once would be ideal for the user visualizing the output of a simulation. To make this possible, a modeling language that is capable of outputting car shapes to the screen is needed.

8.0 VRML

VRML stands for Virtual Reality Modeling Language and according to the VRML Consortium is "an open standard for 3D multimedia and shared virtual worlds on the Internet. [8]" It is considered an open standard because the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) recognized it as an international standard in December 1997, hence the name VRML97. VRML "worlds" got their name from the original inspiration; shared virtual worlds on the Internet. VRML was really the first 3D application that has the ability to use the Internet to share 3D objects and scenes. VRML is a scene description language that describes the geometry and behavior of a 3D scene or world but it isn't a general purpose programming language like C++. A world can range from simple objects to very complex scenes [8]. The VRML modeling language is appropriate for a system that outputs traffic results from a simulator because it can allow the user to view a city section in three-dimensions with the ability to browse around the roads and crossings to witness traffic congestion under certain conditions.

Browsing around VRML worlds is very simple. There are three main modes that the browser has; fly, walk and point. The fly mode gives the users the ability to fly around the 3D world as a pilot would in an airplane. The user can also walk on an imaginary flat plane and then by pressing a specific button, the user can move up and down. Lastly, point mode allows the user to point and click on an object and the browser will move the user directly towards that object [9].

8.1 The VRML Node

After looking at the purpose and history of VRML, it is time to look at the actual structure of a VRML file. In a nutshell, a VRML world is made up of many nodes. A node is simply a type of object with fields or attributes that are properties of the object. The simplest node is a Group node that just groups together other nodes [10]. Figure 8.1 shows a group node written in VRML.

```
Group {
        children []
}
```

Figure 8.1 - The VRML group node

The fields of the group node go inside the curly braces. In this case there exists only one field, the children field. The group node is an example of a grouping node. Grouping nodes are nodes that can contain other nodes in their children field. Other examples include the transform node, the collision node and the anchor node [10]. Figure 8.2 is an example of a group node nested in another group node.

```
Group {
   children [
      Group {}
   ]
}
```

Figure 8.2 - Nested group nodes

This gives somewhat of a hierarchy of nodes sometimes called the scene graph and in general the Scene Graph can reach several levels in the hierarchy [10].

Another useful grouping node, especially in the VRML GUI system is the transform node. The transform node has several fields that are useful for transforming and rotating nodes around a world [10]. Figure 8.3 contains a list of key fields.

```
Transform {
   eventIn        MFNode       addChildren
   eventIn        MFNode       removeChildren
   exposedField   MFNode       children          []
   exposedField   SFRotation   rotation          0 0 1 0
   exposedField   SFVec3f      scale             1 1 1
   exposedField   SFVec3f      translation       0 0 0
}
```

Figure 8.3 - The transform node's key fields

We see that the children field is of type MFNode, which represents a list of nodes. This means that the transform node is like the group node in the way the programmer can nest other nodes inside the transform node. Other useful fields for the project are the rotation, scale and translation fields. The rotation field will rotate the nodes that are in the children field of the transform node by a specific number of radians about an axis. The translation field translates the objects in the children field to a new location defined by the X, Y, and Z coordinates in the corresponding SFVec3f field. Lastly the scale field will scale the object in the children field of

the transform node by the amount specified in the SFVec3f type. There are two other fields of interest; the addChildren and removeChildren eventIn's. Most of the grouping nodes have the addChildren and removeChildren event fields but no other nodes contain these fields [10]. Events will be explained later on in the report.

8.2 Basic Shapes

There are four basic shapes: box, cone, cylinder and sphere. These are the basic geometry nodes. There are other geometry nodes but they are used to allow the user to create their own shapes, as will be done in this project to create car and traffic light shapes. The dimensions of the box can be defined in the box node's only field, the size [10].

Geometry nodes can only be placed inside the geometry field of a shape node. The shape node has only two fields, the aforementioned geometry field that defines the 3D polygon and an appearance node that defines how the shape looks in the field [10]. To add a box to the world with the default values in the fields, we would simply build a hierarchy as in figure 8.4.

```
Shape {
   geometry Box {}
   appearance Appearance {
      material Material {}
   }
}
```

Figure 8.4 - Adding a box to the VRML world

8.3 Complex Shapes

The basic shapes that VRML offer are just that, basic. For the most part, anything advanced will require the user to either combine basic shapes together or designing their own shapes. The developer can use a number of other geometry nodes to build up arbitrary or complex shapes to enhance their world. There are five other geometry nodes that can help the user create complex nodes: IndexedFaceSet, IndexedLineSet, ElevationGrid, Extrusion, and PointSet. The IndexedFaceSet node is the most useful of the five complex geometry nodes and consists of a set of faces, which is defined manually by the user, to build up complex shapes.

8.4 Events

Along with fields, most of the nodes also contain events. There are two types of events, eventIn and eventOut. eventOut events are outgoing events that inform other nodes of changing values in the source node, such as a value of a field changing or a time change. eventIn events are incoming information from nodes in the world that may cause the receiving node to change. Each event has a data type associated with it [10].

There are fields in nodes that are events of type exposedField. This means that there are two events associated with that field, the set_fieldName and fieldName_changed. These are the eventIn and eventOut for the field. If we look back at the transform node, we can see that there is an exposedField called rotation. If we wanted to rotate a box based on the user clicking a button, we would set the world up so an eventOut was generated when the user clicked the button and route it to the set_rotation field of the transform node of the box. We would send values 0 0 1 3.14 to rotate the box about the z-plane by 180°, then a rotation_changed event would be generated that could be routed elsewhere to inform another object that the box has rotated. This is how a chain of events can happen based on a single user action.

9.0 THE EXTERNAL AUTHORING INTERFACE

The VRML GUI in this project will be implemented using Java with VRML supplying the 3D graphics. Because of this, learning how to use VRML in Java is essential. The External Authoring Interface (EAI) is an interface that allows other programs, in our case a Java applet, to communicate with VRML worlds. "[e]ssentially, to Java, the EAI is a set of classes with methods that can be called to control the VRML world. To VRML the EAI is just another mechanism that can send and receive events, just like the rest of VRML. [11]" This quote comes from VRML Is – Java Does – And the EAI Can Help and very accurately describes what EAI can be used for. In theory, the EAI is not directly tied to Java or Java applets, but it is used in practice by these tools [11].

The first step in using the EAI in a Java applet is to get a reference to the VRML browser. The EAI offers a Browser class with a static method getBrowser that will retrieve the VRML browser encapsulated in a Browser object [11].

9.1 Obtaining a Reference to a VRML Node

To make VRML do anything useful, we will want to be able to send information to the eventIn of a node of interest (see section 8.4). To do this, we will need to be able to get a Java reference to the node. Imagine we have some simple VRML scene such figure 9.1 containing a simple red box.

```
Transform {
   children [
      Shape {
         appearance Appearance {
            material DEF MAT Material {
               diffuseColor 1 0 0
            }
         }
         geometry Box {
            size 1 3 5
         }
      }
   ]
}
```

Figure 9.1 - A VRML world with a red box of size 1x3x5.

Now lets say that we want to change the colour of the box using the EAI. First we need to get a reference to the node that contains the colour field. The “DEF MAT” part of the material node definition just gives the material node a name so it can be reused, or in this case, so it can be referenced outside the VRML world. In this case the node named MAT contains the diffuseColor field of the shape that we want to change. Next, we need to get a reference to the diffuseColor field of the material node. In order to change the colour, we need to send a value to the referenced field that will change the shape to the desired colour [11].

The EAI offers a Node class that can encapsulate a VRML node, and an EventInSFColor class that has methods that can send SFColor values to the event fields. Figure 9.2 will change the colour of the box from figure 9.1 to blue.

```
Node material = browser.getNode(“MAT”);
EventInSFColor set_diffuseColour =
        (EventInSFColor) material.getEventIn(“set_diffuseColor”);
float[] colour = new float[3];
colour[0] = 0.0f; colour[1] = 0.0f; colour[2] = 1.0f;
diffuseColour.setValue(colour);
EventOutSFColor diffuseColour_changed =
        (EventOutSFColor) material.getEventOut(“diffuseColor_changed”);
float[] colour = diffuseColour_changed.getValue();
```

Figure 9.2 - Java code to change the colour of a box to blue, then get and store the colour of the box

The first two statements just get the references as mentioned. The diffuseColor field of the Material node is of type SFNode and in Java an SFNode is implemented as a three-element array of float values, in this case defining the new colour. We then set the colour to blue and send the value to the Material eventIn diffuseColor and the colour of the shape is changed to blue. We can then get the color value of the box by using the getValue method on the diffuseColour_changed eventOut [11].

10.0 THE VRML GUI SUBSYSTEM

The VRML GUI is the subsystem that I was required to reengineer for the ATV system. The purpose of the VRML GUI is to allow the user to view the simulation results of the city section that the user created using MAPS, after compiling with the TSC and simulating with the N-CD++ simulator. The previous VRML GUI subsystem could only display basic shapes to represent cars and had a limitation of allowing just one segment displaying at a time. The new version of the VRML GUI will show the entire city as the user created it using MAPS, will display road shapes to represent the segments and crossings, and will display car shapes moving along the roads for every segment as the simulation output file specifies. The VRML GUI uses VRML as its graphics representation of cars, segments and crossings. The software engineering process will be described in detail starting with the requirements and analysis, then the design and implementation, finishing with a description about the testing that was done on the VRML GUI subsystem.

10.1 Requirements

10.1.1 Functional Requirements

The functional requirements of any software system describe the interactions between the system and its environment without describing the unnecessary details of implementation [12].

The VRML GUI is a graphical user interface that shows traffic flowing through a predefine city based on the results of a simulation. The VRML GUI will use the created plan file to determine a static view of the city without cars present. The GUI uses the results file from a previous simulation by the N-CD++ simulator, and determines the location and direction of specific cars at a particular point in time. A car shape will be displayed on the screen in the appropriate cell on a segment for the amount of time specified in the log file. When that time expires, the car will move to a new cell as per the results file. The entire city will operate in this manner with cars moving within segments and from segment to segment. The user will be able navigate around the city as they wish, watching cars pass through the various segments. The time will be displayed as it changes so the user has an idea of the time as cars are moving.

10.1.2 Actors of the System

The main user will be the person interacting with the VRML GUI. The user will indicate to the GUI where the files are located that define the city and simulation. They will also have the ability to navigate around the city when it is loaded, running or finished the results.

10.1.3 Use Cases

Use Case Name: *View City*

Includes the Load File use case
Extends Invalid File use case

Participating Actor Instance: The GUI user (initiator)
Entry Conditions:

1. An ATLAS file has been created that represents a city section
2. The VRML GUI has been loaded

Flow of Events:

3. The user clicks the ATLAS panel
4. The VRML GUI informs the user that it is waiting for the user to input the plan file
5. The user clicks the load ATLAS file button
6. The system invokes the Load File use case to get the plan file
7. The system loads the plan file, extracts the segments and crossings and stores them
8. The system takes the segments and displays a VRML road shape in the appropriate location for each segment
9. The system then takes each crossing and determines if a light or stop sign is present. If so, a VRML crossing shape is displayed with a traffic light or stop sign object, if not, then just a VRML crossing shape is shown.

Exit Condition:

10. The system indicates that it is ready to load the ma file.

Exceptional Conditions:

The file loaded is not a valid plan file - The Invalid File use case is invoked

Special Requirements:

The user is able to navigate through the city section at any time.

Use Case Name: *Initialize City*

Includes Load File use case
Extends Invalid File use case

Participating Actor Instance: The GUI user (initiator)
Entry Condition:

1. The city has been loaded and displayed by the VRML GUI

Flow of Events:

2. The VRML GUI informs the user that it is waiting for the user to input the ma model file
3. The user clicks the load ma file button
4. The system invokes the Load File use case to get the ma file
5. The system loads the ma file and verifies that all the segments and crossings from the plan file are present in the city, that they are of type cell and that their width and height dimensions are correct

Exit Condition:

6. The system indicates that it is ready to load the log file.

Exceptional Conditions:

The file loaded is not a valid ma file, the segments or crossings are not found in the plan file, or the dimensions of the segments don't correspond with the ATLAS plan file - The Invalid File use case invoked

Special Requirement:

The user is able to navigate through the city section at any time.

Use Case Name: *Move Cars*

Includes Load File use case
Extends Invalid File use case

Participating Actor Instance: The GUI user (initiator)
Entry Condition:

1. The system has initialized the city with the correct ma file

Flow of Events:

2. The VRML GUI informs the user that it is waiting for the user to input the log file
3. The user clicks the load log file button
4. The system invokes the Load File use case to get the log file
5. The system loads the log file
6. The system starts at the beginning of the log file and searches for a Y (output) message
7. When a Y message is found, the system verifies that it is a valid output message
8. If the file is a valid output message, the system determines which segment is affected by this message, which cell in the segment will be affected and whether a car is moving to or from the correspond cell. The system also checks the time associated with the message and if it is different than the previous message, then the new time is displayed.
9. If a car is moving to this cell (i.e. the message is a 1) a car shape is shown in this cell. If a car is moving out of this cell (i.e. the message is a 0) then the car shape that was present is removed

10. If the Y message is not a valid output message, the message is skipped
11. The system repeats steps 7 - 10 for the entire log file until the end is reached (i.e. the simulation ended)

Exit Conditions:

12. The system has reached its final state
13. The load plan button has been enabled again so another city can be loaded

Exceptional Conditions:

1. The file loaded is not a valid log file - The Invalid File use case invoked
2. The user clicks the stop displaying results button - The display stops at the time the user clicks the button so they can analyze the city at that time.

Special Requirements:

The user is able to navigate through the city section at any time.

Use Case Name: *Load File*

Participating Actor Instance: The GUI user
Entry Conditions:

1. A box that allows the user to browse their computer for a file is displayed

Flow of Events:

2. The user browses their drives to locate the proper file
3. The user locates the file and double-clicks it
4. The location of this file is stored in the system

Exit Condition

5. The box is no longer present

Exceptional Conditions:

The user clicks the cancel button - The system will go back the state it was in before the load file button was pressed

Use Case Name: *Invalid File*

Participating Actor Instances: None
Entry Conditions:

1. A file that is invalid has been loaded by the system

Flow of Events:

2. A message is shown to the user indicating that the file was invalid
3. The load plan file button is re-enabled so the user can restart the process and the button that was pressed is re-enabled to allow the user to attempt to load the file again

Exit Condition

4. The user clicks one of the two enabled buttons or closes the GUI

10.1.4 Use Case Diagram

Figure 10.1 is the use case diagram for the functionality that will be added to the current VRML GUI system. The use cases are described in section 10.1.3 above with details on the entry conditions, flow of events, exit conditions and special circumstances. The load file functionality is used by the three main use cases, View City, Initialize City and Move Cars, in the new system, hence the include relationship between the use cases. Also, the three main use cases perform exceptional actions if the file that was inputted from the user did not contain the proper information. For this situation, all three main use cases extend the Invalid File use case.

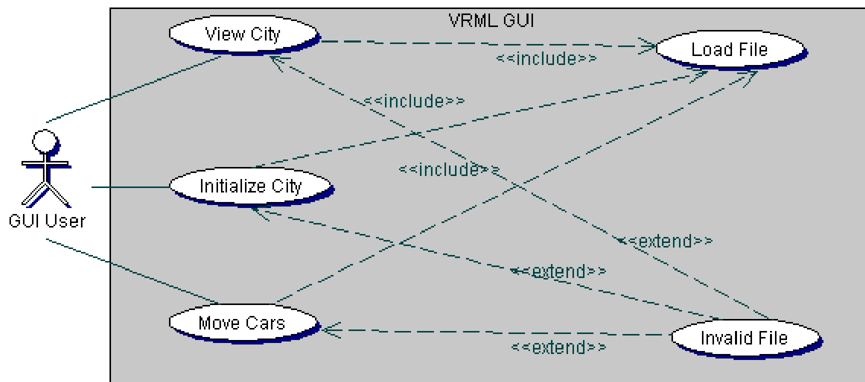


Figure 10.1 - The use case diagram showing the added use cases to the VRML GUI system

10.1.5 Initial Analysis Objects

Near the end of the requirements phase, it is important determine initial analysis objects from the use cases [12]. For this system, I have determined several objects that fall into this category.

Button - Represents the buttons that the user presses to indicate to the system that they want to load a city, initialize a simulation, or run and view the results of a simulation.
Segment - Represents a road that was built using the ATLAS specifications.
Crossing - Represents an intersection that joins multiple segments in the ATLAS specification.
Car - This object represents a car being present in a specific cell in a segment or crossing.
Log Output Message - A message from the log file indicating that an event has happened in the system.

10.1.6 Nonfunctional and Pseudo Requirements

As part of any software system, there are pseudo requirements called for by the consumer, client, or manager that really have no direct affect on the users' view of the system. These generally restrict the implementation of the system and are usually requirements on aspects such as the language and the platform that the software is to be implemented. There are also nonfunctional requirements that describe user-visible aspects of the system that are directly related with the functional behavior of the system such as quantitative constraints or accuracy [12]. For the VRML GUI system there are few nonfunctional and pseudo requirements.

The reengineered system must be an extension of the current Java applet. The system is required to use VRML as its virtual reality output and must work under the Cosmo Player virtual reality tool. The future system must include distributed execution via the Internet using a web-based interface. It must show the time of the simulation as per the results however, it currently does not have to run in real-time but a future version of the software may be required to do so.

10.2 Analysis

Now that the requirements have been determined for the reengineered system, it is time to produce a model of the system that is "correct, complete, consistent, and verifiable" [12]. The analysis phase is used to verify the system specification produced by the client during the requirements phase. The first step in the analysis phase is to determine the entity, boundary and control objects of the system [12].

10.2.1 Entity Objects

These are the objects that store the persistent information tracked by the system [12].

Segment - Represents a road that was built using the ATLAS specifications. The segment includes information on the start and endpoint, speed limit, the parking along the side of the segment, number of lanes, and the direction the traffic travels along the segment.

Crossing - Represents an intersection that joins multiple segments in the ATLAS specification. A crossing object includes information about the crossing such as its location, the speed limit as cars pass through the intersection, whether the crossing includes a traffic light or a pothole, and the probability that a car within the crossing will exit the crossing at the next possible segment. Recall from section 3.2 that crossings are represented as roundabouts by the N-CD++ simulator.

VRMLNode - A VRMLNode is an object that gets sent to the VRML browser, the simuUserClient, to eventually be displayed. There are three different types of VRMLNodes for the ATLAS output, a car shape, a segment shape or a crossing shape. The crossing shape has a traffic light or stop sign present if the field in the crossing object indicates so.

LogOutputMessage - A message indicating to the system that a car is either moving to a specific cell in a segment or crossing, or a car is moving out of a specific cell. The message indicates the segment or crossing name, the time from the start of the simulation at which this event occurred, the lane and cell number that is affected by the message, and whether a car is moving to or from this cell.

Model - This object represents a model in the simulation. These models represent both crossings and segments prior to the simulation being executed. This object stores the dimension of the model (i.e. the length and number of cells in a segment) as well as other information related to the simulator.

10.2.2 Boundary Objects

Boundary objects represent the interactions between the actor and the system [12].

Button - These objects represent the physical buttons that the user presses to load the three different files that are necessary to the system. There will be three instances of the Button object: LoadCityButton is the button the user presses to indicate that they want to input a city ATLAS file, InitializeCityButton is the button that the user presses to inform the system that they want to initialize a city and prepare to show the output, a RunCityButton that indicates that the

user wants to show the results from the simulation, and a StopRunningButton that freezes the output so the user can analyze the traffic in a city at any instant in time.

ATLASPanel - The interface object with which the user interacts. This is the object that allows the user to view text results, instructions and press buttons.

SimuUserClient - The object that allows the user to view the VRML output of the system. Also, the SimuUserClient object allows the user to navigate throughout the city section by zooming in and out, rotating the city or panning along segments.

10.2.3 Control Objects

These objects represent the tasks that are performed by the user and supported by the system [12].

ReadPlanFile - This object will take a valid plan file, parse through it and extract the segments and crossings, and store them in a Segment object.

ReadMaFile - This object will take a valid ma file, and store important information such as the length and number of lanes for each segment in a Model object.

ReadLogFile - This object will take a valid log file, and store each Y messages in a LogOutputMessage object.

CityControl - This is the main control object of the system. It handles user input received from the boundary objects and makes the system run using the different entity objects. The CityControl object handles the four different situations:

- *ShowCity* - The CityControl object will take the segments and crossings from the ReadPlanFile object and display a segment object for each segment and a crossing object for each crossing to the SimuUserClient object.
- *InitializeCity* - The CityControl object will take all the Model objects and ensure that they match up with the segments and crossings from the plan file.

- *RunCity* - The CityControl object will take the LogOutputMessage object and determine if it is valid, then if the message indicates that a car is moving to a cell, it will display a Car object to the SimuUserClient object. If the message shows that a car is leaving a cell, then the CityControl object will remove the car in that cell from the SimuUserClient object.
- *StopCity* - The object will stop the RunCity use case from processing LogOutputMessages. This will allow the user to view the city at a frozen moment in time.

10.2.4 Sequence Diagrams

The sequence diagrams are an essential part of the software design process. Once all analysis entity, boundary and control objects have been determined, the sequence diagrams give the engineer an idea of the messages that will be passed amongst the objects. The sequence diagram helps bridge the gap between analysis and system design since it really helps define the relationships between the different objects of the system [12].

First we have the sequence diagram for the Load City use case. In section 10.2.3, we have determined the entity, boundary and control objects. Figure 10.2 shows the objects that are involved in this use case as well the messages that are passed amongst these objects. The user only interacts with boundary objects, which in turn pass the events that the user generate to the control objects that modify the entity objects and send events back to boundary objects for the user to view [12]. The user clicks the load plan file button and control is passed to the CityControl control object. This CityControl object will extract segments and crossings using the other control object, ReadPlanFile, create entity objects, and send messages to the SimuUserClient browser to display for the user.

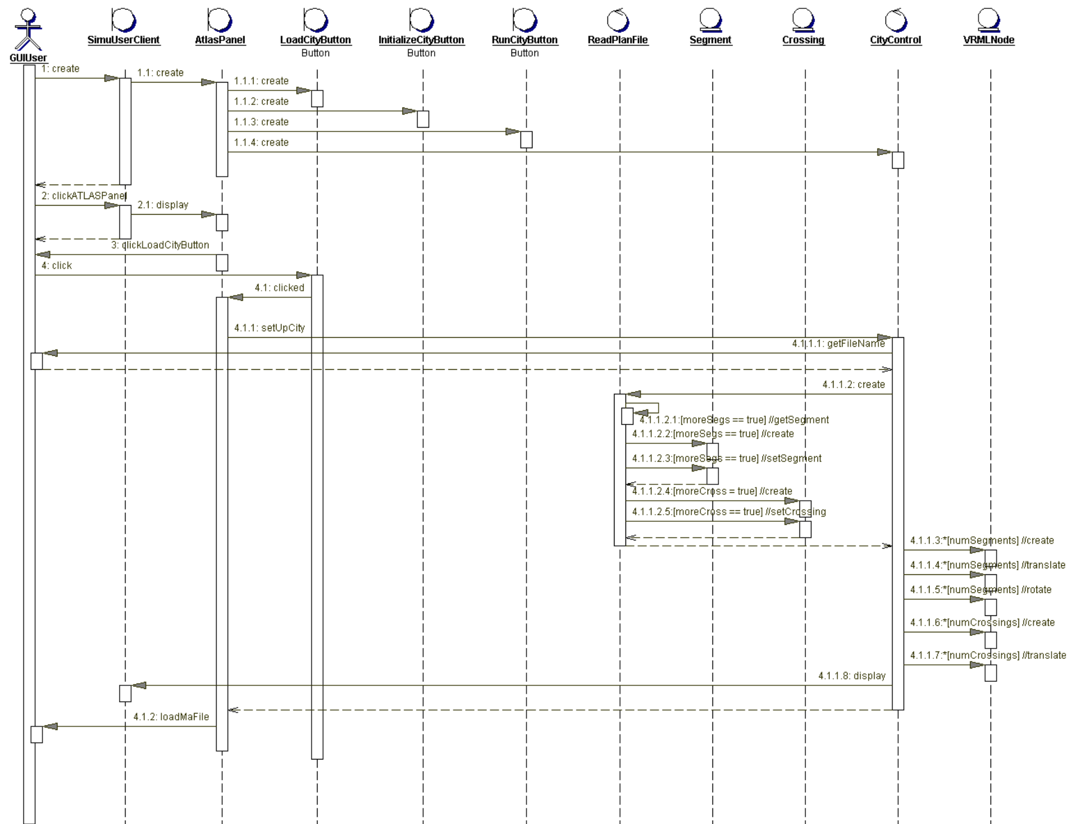


Figure 10.2 - The sequence diagram for the View City use case

The next use case is the Initialize City use case. As described in section 10.1.3, the Initialize City use case takes all the segments and crossings from the plan file and ensures that they match up with the cell models from the ma file that the user inputs. As explained in section 6.0, the ma file is the compiled plan file that describes all the components of the simulation including crossings and segments. The sequence diagram for this use case is shown in figure 10.3.

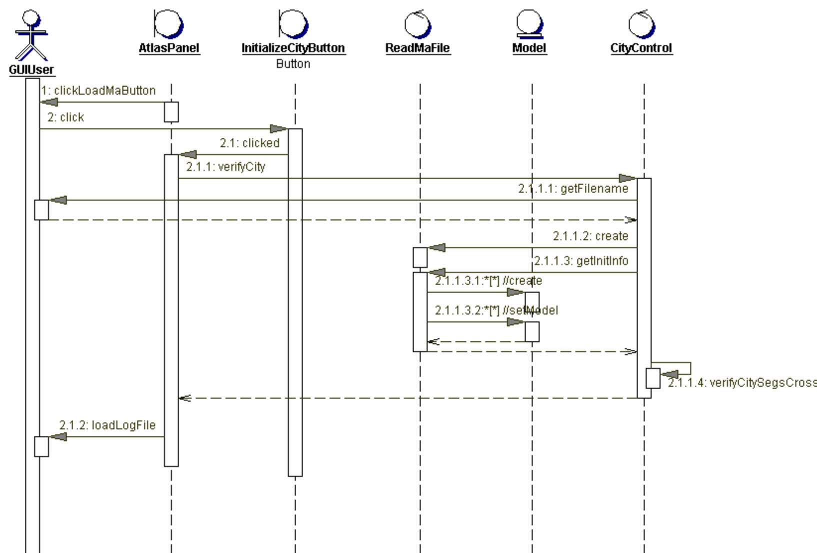


Figure 10.3 - The sequence diagram for the Initialize City use case

After the plan and ma files have been verified, the results from the simulation can be shown on the city that is displayed on the screen. Once the user inputs the log file, the system will extract the relevant output message that indicate cars moving through the city. The system will make the necessary changes to the car VRMLNode entity objects on the screen that reflect the results of the output messages. Figure 10.4 shows the sequence diagram for the Move Cars use case.

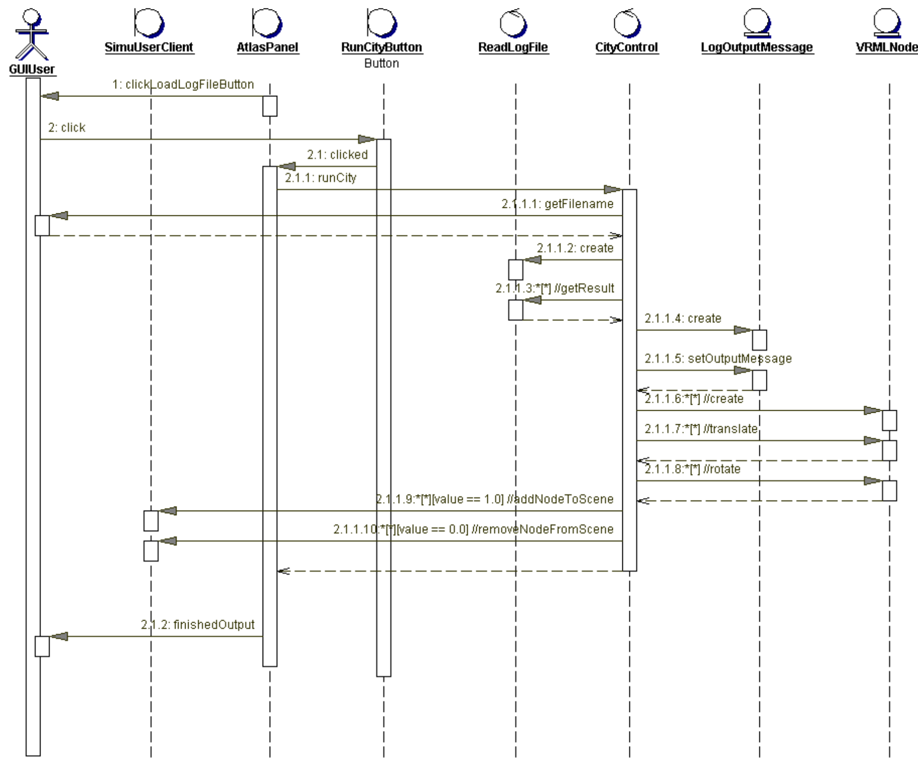


Figure 10.4 - The sequence diagrams for the Move Cars use case

Finally, there are two small use cases: Load File and Invalid File. The Load File use case is included in the above three use cases, which is the reason it exists as a separate entity. The Load File use case is instantiated by another use case, unlike the other use cases that are instantiated by the VRML GUI user. The Invalid File use case is an exceptional use case that only gets executed when the user inputs a file that is not a type the system was expecting, is inconsistent with a previous file inputted, or contains errors. Figure 10.5 shows the sequence diagrams for the Load File and Invalid File uses cases respectively.

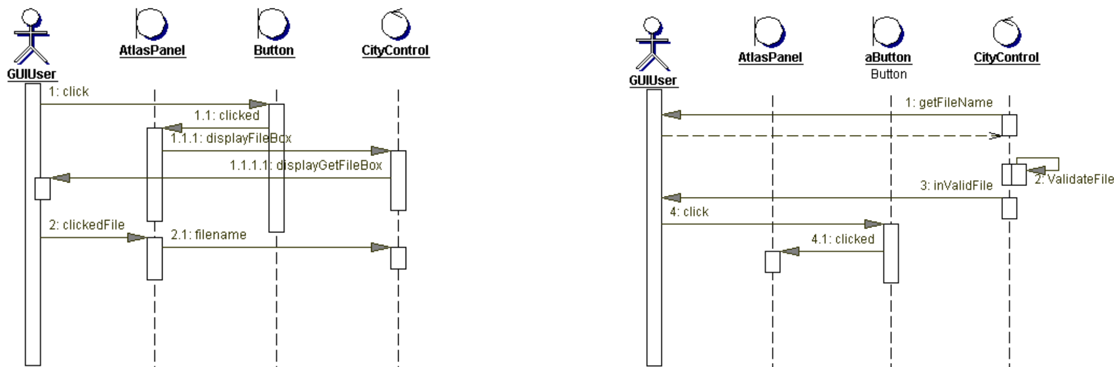


Figure 10.5 - The sequence diagrams for Load File and Invalid File respectively

10.2.5 Identifying Associations

It is important to be able to determine the associations between two or more classes. The sequence diagram can help show which objects need to communicate with one another. An overall system diagram, like a class diagram, shows the various classes that represent objects, and their associations. Also the multiplicity of associations is shown to give the developers an idea of the cardinality between two classes. For example, from figure 10.2, the sequence diagram for the View City use case, it is obvious that the CreatePlanFile object creates and modifies Segment entity objects. However, it should be noted that the CreatePlanFile would create and modify at least one but usually many Segment objects. Because of this, there is an association between the CreatePlanFile class and the Segments class with a multiplicity 1 to one or more as shown in figure 10.6, the simplified class diagram for the VRML GUI. Figure 10.6 shows all the associations between all the classes in the VRML GUI system. The diagram only shows the classes that are necessary for the modifications that are being done in this version of the subsystem. The details of the existing system for the most part have been left out for simplicity.

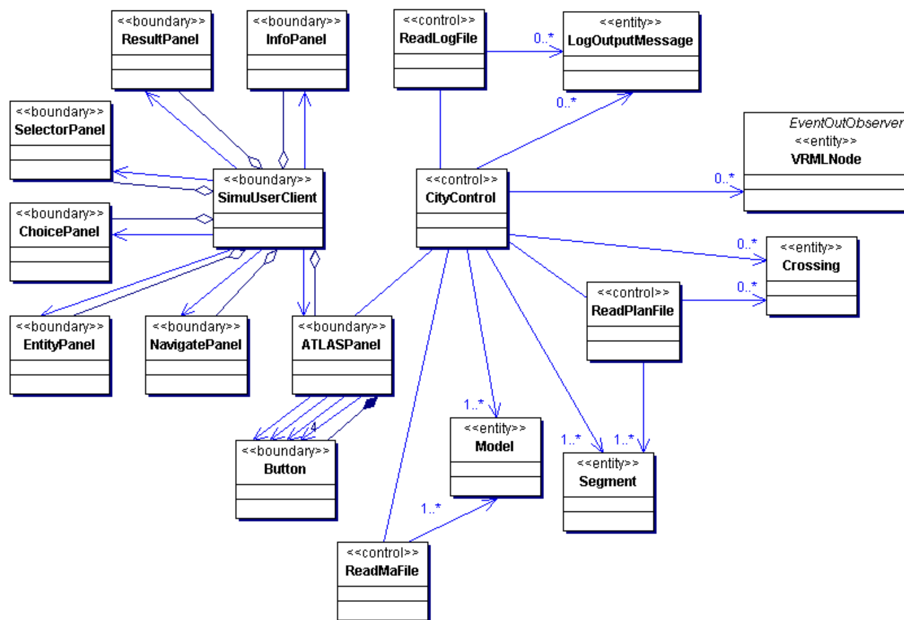


Figure 10.6 - A simplified class diagram for the new VRML GUI system

10.2.6 Identifying Attributes and Operations

Each class from the class diagram in figure 10.6 has important attributes. Attributes are properties of individual objects and define that object [12]. For example, each Crossing object contains a name, two-dimensional Point location, and variables representing properties of the crossing such as whether there exists a light, a pothole and the normal speed of traffic passing through the crossing. Another example is a LogOutputMessage. Each message contains the time the message was processed by the simulator, the segment or crossing name that is affected by the message, the lane and cell number that is affected and a variable indicating whether this message represents a car now present in the cell or not present. Figure 10.7 below gives the attributes in a class diagram for every modified or new class in the VRML GUI system. Also, each class contains operations that represent atomic behavior that is provided by a class. Figure 10.8 also shows the operations for each class in the class diagram.

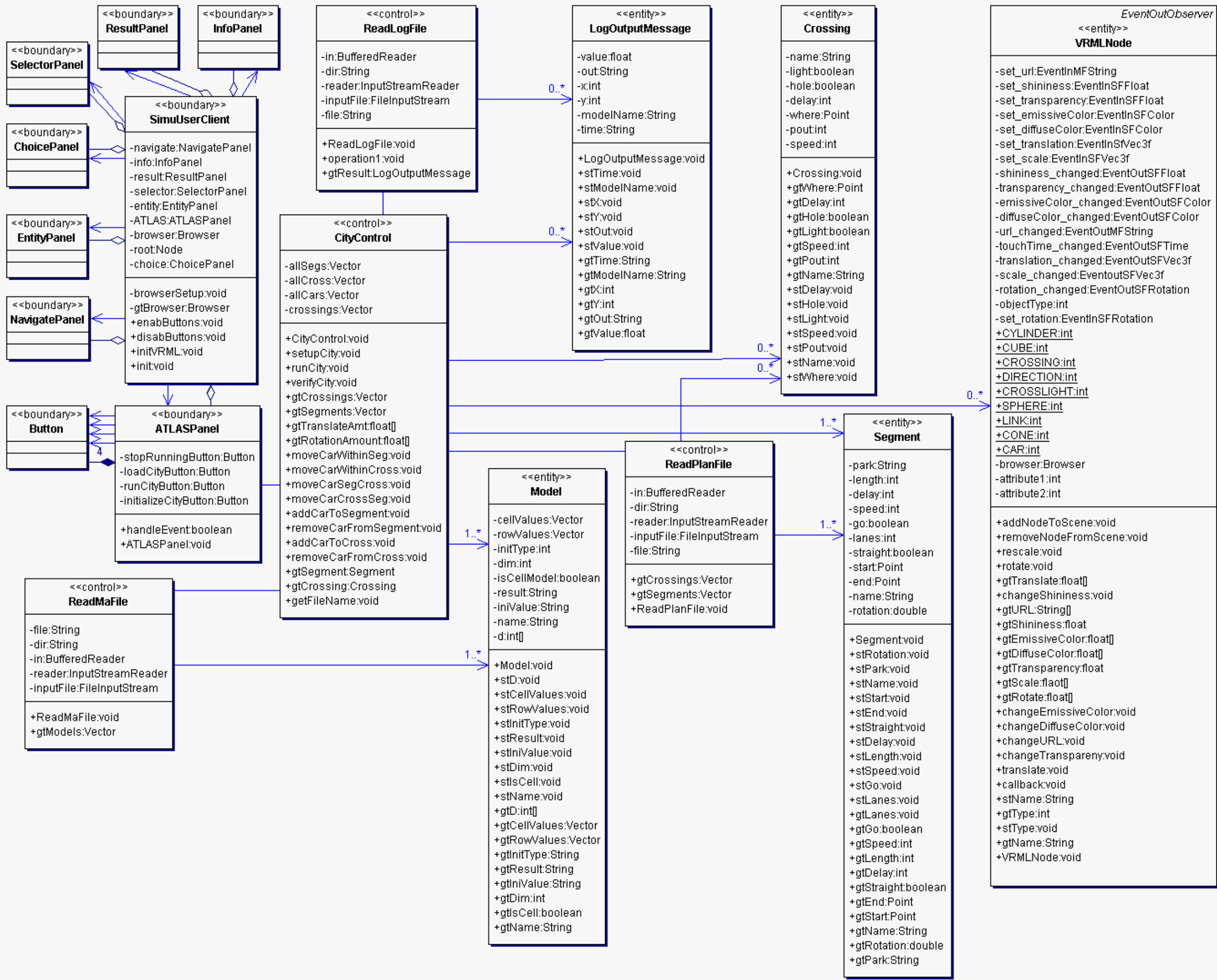


Figure 10.7 - The class diagram for the VRML GUI with attributes and operations shown

10.3 Design

There were many design decisions that needed to be made before to the reengineered system could be implemented. The basic structure for the Java applet was already developed so the visualization tool itself did not need to be designed. The design decisions that had to be made were related to the data that needed to be stored, how it was going to be presented to the user and what was going to be shown.

10.3.1 Segment, Crossing and Car VRML Objects

The requirements of the project stated that the city section is to be view statically when the plan file was loaded, then when the log file was loaded, the user would be able to see cars moving throughout the city. One of the nonfunctional requirements was to use VRML as its

virtual reality output and must work under the Cosmo Player virtual reality tool. In order for the system to follow this condition, it was essential to find or create VRML objects that represented cars, segments and crossings.

10.3.1.1 The Car

In order to make the output look realistic, the car shapes should look like real cars. With the downfall of VRML in the past several years, finding an existing car was more difficult that expected. Several websites offered some different shapes and objects but the best came from a VRML textbook that used the car as an example of VRML code on its accompanying compact disc. The size of the file is very large however (more than 350kB). Having a VRML object such as this car shape loaded in memory could have a negative effect on the speed of the output however there was no restriction in the nonfunctional requirements about the speed of the system so I decided to proceed with this car shape shown in figure 10.8.

10.3.1.2 The Segment and Crossing

In the requirements, it was determined that a static view of the city was required. To do so, we needed to have a shape that represents a segment and crossing. After browsing many websites and other resources, finding a road shape was more difficult than originally thought. Because of this, I decided to design a simple road shape for the project. To make it simple, I elected to make each cell in a segment an object. If a segment has five cells, the same road shape will appear five times consecutively. The road will just have white lines on the outside to indicate where a segment lane begins and ends. Figure 10.8 shows the final road shape for the segment. Similarly, a normal crossing will look very similar to the segment but is surrounded by a white line as shown also in figure 10.8.

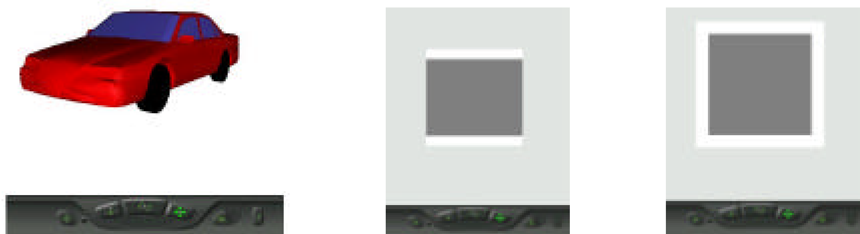


Figure 10.8 - The car, segment and crossing VRML objects

Some crossings in the plan file have traffic lights or stop signs. When the plan file is inputted into the VRML GUI system, the crossings are encapsulated in a Crossing object. There are attributes in the Crossing object indicating whether a crossing contains stop signs or traffic lights. To make the city section realistic, I have decided to have crossings with stop signs and traffic light shapes attached to them. Figure 10.9 shows two crossings, one with a traffic light and the other with a stop sign.

Figure 10.9 - Two crossings, one with a stop light, the other with a stop sign

10.3.2 Attributes of Crossings and Segments

In order to show a static view of the city, the plan file needs to be inputted to the system, read, the segments and crossings extracted and then displayed on the screen. Reading the file and extracting the segments and crossings will be left for section 10.4, implementation but a problem exists when the segments and crossings are extracted, how to display them to the screen. VRML works on a three-dimensional Cartesian system and the ATLAS plan file is implemented using a two-dimensional Cartesian system. I decided to display the segments using just the two dimensions because roads and crossings do not have a z-dimension.

An object that is 1-by-1 in size will be used to represent a cell in a segment. Each segment has a start and end point. Using these points, the number of cells a segment contains has to be determined. A segment that begins at (0,0) and ends at (0, 5) is quite obviously 5 cells long per lane. However, determining the number of cells for a one-lane segment that starts at (12,5) and ends at (-14, 26) is not so obvious. To determine the number of cells in a segment with a starting point (P_{1x}, P_{1y}) and an end point (P_{2x}, P_{2y}), the equation of the length of a line can be used as shown in equation 10.1

$$length = \sqrt{\left(P_{1y} - P_{2y}\right)^2 + \left(P_{1x} - P_{2x}\right)^2} \qquad (10.1)$$

Generally, unless the segment goes in the x or y direction, the length will not be an integer value. Since the N-CD++ simulator assumes that a full cell is required, the length must be rounded down to make an integer length. So using the example presented above with a segment with starting and ending points (12,5) and (-14, 26) respectively, using equation 10.1 gives a length of 33.42. After rounding down, the segment ends up being 33 cells long. Also, as seen in section 10.2.6, the analysis phase, each segment has a rotation field. This value is not given in the ATLAS plan file for the segments so it has to be calculated. The rotation of a segment defines the angle from the starting point that the segment is rotated. A segment starting at (2,2) and ending at (5, 2) has a rotation of 0 since it runs parallel to the x-axis whereas a segment starting at (2,2) and ending at (2, 5) has a rotation of 90° or p/2 radians. This rotation is important because the segments must run smooth from the starting point to end point. To determine a rotation for two points that don't have the same x or y coordinate, equation 10.2 was used.

$$rotation = \tan^{-1}\left(\frac{P_{2y} - P_{1y}}{P_{2x} - P_{1x}}\right) \qquad (10.2)$$

Equation 10.2 gives the rotation in radians, about a set of axes originating at the starting point. Again using the previous example, the rotation of a segment starting at (12,5) and ending at (-14, 26) using equation 10.2, is -0.68rad. This result is actually incorrect as the correct solution should be -0.86 + p/2 = 0.71rads because of the CAST rule. The CAST rule states that the tangent of an angle is positive when the angle is between 90° and 180° or between 270° and 0° [13]. If the direction of the segment were reversed (i.e. starting point (-14, 26), ending point (12, 5)), the rotation would still be calculated to be -0.68rad, which is the correct value in this situation. In order to figure out the correct value of the angle, the individual coordinates have to be examined as in equation 10.3.

if ($P_{1x} > P_{2x}$ and $P_{1y} < P_{2y}$) or ($P_{1x} >$ P2x and $P_{1y} > P_{2y}$)
    rotation = rotation + p/2 radians (10.3)
Where the rotation has been calculated as in equation 10.2.

Once the length and rotation of the segments have been calculated, the segment objects can be put together to form a full road. The objects have to be translated and rotated to their appropriate location to make the city section accurate. Equations 10.4 and 10.5 are the x and y translations for the segment objects with starting point (P_{1x}, P_{1y}) and ending point (P_{2x}, P_{2y}).

$$x = \frac{P_{2x} - P_{1x}}{length} \cdot (cellNumber + 1) + P_{1x} + \frac{P_{2y} - P_{1y}}{length} \cdot laneNumber \qquad (10.4)$$

$$y = \frac{P_{2y} - P_{1y}}{length} \cdot (cellNumber + 1) + P_{1y} - \frac{P_{2x} - P_{1x}}{length} \cdot laneNumber \qquad (10.5)$$

where cellNumber is the cell number in the segment that we are translating starting from 0 and laneNumber is the lane of the cell that we are translating starting from 0.

Continuing the example presented above, if the segment started at (12,5) and ended at (-14, 26), the first cell would be centered at (11.21, 5.63) using equations 10.4 and 10.5 with cellNumber and laneNumber both being zero. The length was calculated above to be 33 and gets stored in the Segment object. The second cell is centered at (10.42, 6.27), calculated using the same equations but with cellNumber one. The distance between the centers of these two cells should be equal to one since 1x1 cells are being considered. Using the same length equation in equation 10.1, the length between the two center points is close to one with some slight rounding errors. There was no requirement on the accuracy of the location of the segments in the nonfunctional requirements so this small error is acceptable as long as the transition between segment objects is realistic. Using equations 10.4 and 10.5 with cellNum = 32, the final segment object will be centered at (-13.21, 25.36) which is close to the destination point (-14, 26). It is not exact due to the rounding of the length down from 33.42 to 33 when we calculated the length. These equations were also tested when a translation of a segment in a lane greater than zero was desired.

Once the translation is calculated for every segment object, they have to be rotated by the amount calculated using equation 10.2 so the segments look continuous. Once the segments have been rotated and translated, they can be displayed on the screen.

10.3.3 Overcoming Visualization Problems with the PLAN file

One drawback with the plan file is that there is no room for crossings that join segments when visualizing a city section. If a segment goes from (0,0) to (10,0), generally there is a crossing at both the start point (0,0) and the end point (10,0) since other segments be connected to this segment like an intersection in a city. The problem that exists is that the segment is 10 cells long and if the first segment starts one cell away from the (0,0) crossing, then the last cell in the segment will be translated directly on the other crossing at (10,0). If there is another segment that joins the crossing at (0,0), there will not be room for the crossing at (0,0) either since the joining segment's last cell will be located at (0,0) and visualizing overlapping VRML objects is not possible. Ideally, a segment that starts at (0,0) and ends at (10,0) would be only 9 cells long so the first cell would translate to (1,0) and the last cell to (9,0). This would allow room for the crossings at (0,0) and (10,0). This is not possible at the current time since the TSC would compile a segment with these start and end points to have 10 cells.

Another issue is visualizing two-way streets. As we know from section 3.1 above, each entry in the segments section of the ATLAS file defines a one-way street. If the city designer wants to have a two-way street, they just add multiple entries of the segment with the same start and end point but change one of the fields in one of the segments from go to back like in figure 10.10.

```
Segment_A = (0, 0), (10, 10), 1, straight, go, 40, 300, parkNone
Segment_A1 = (0, 0), (10, 10), 1, straight, back, 40, 300, parkNone
```

Figure 10.10 - ATLAS representation of a two-way street

When the VRML GUI displays the first segments from the start point to the end point, there is no room for another segment that has the same coordinates but the opposite direction. I came up with three alternatives for a solution. First, the two-way street could share the cells from the start point to the destination. This would result in the cars traveling along the two-way street being half the size as they would be on one-way streets. This size limitation could be solved by having all one-way streets also half the normal size and leaving the other half of the cell blank (i.e. without a road object present). Another solution would be to have multiple layers along the path from the start point to the end. This could be done by using the z-dimension of the VRML world and have the one direction above the z=0 plane and the other direction below the z=0 plane. The

final solution was to have one lane in the two-way segment go to the left of the path from the start to the end point and the other lane going to the right of the path.

Another problem that goes hand-in-hand with the previous problem is how to show a multilane segment in the VRML GUI. A multilane segment is described in ATLAS using the lanes field of a segment definition. For this problem, I will look at the same three solutions again. The first was to divide the lane from the start to end points. In order to use this solution, we would have to split the path from the start to end point into four equal parts because the segments contain four lanes but if this segment was desired to be a two-way segment, the path would have to be split into eight equal lanes. This would cause the lanes to be extremely small when visualized by the user on the VRML GUI. The second solution was to have multiple layers. For the same reasons, this solution is not preferred since a two-way four-lane segment would require eight layers, which would be difficult for the user to visualize, especially when cars change lanes since the car objects would then be changing levels. This would also not be very realistic, which was one of the functional requirements. The final solution seems to be the best in having the one direction to the left of the path from start to end point and the other to the right of the path. With the multiple lanes, this solution will just have the extra lanes continue further away from the nominal path. The lane that flows from the segment's start point to the end point will not actually belong to either direction so it will act as a median. An example of a two-way four-lane segment is shown in figure 10.11. The problem with this solution is that the example in figure 10.11 has lanes that expand quite far away from the actual segment path. Two separate two-way, four-lane segments that meet at a crossing are going to inevitably overlap at some point like in figure 10.12. We do have to consider however that in real life, if two four-lane segments meet at a crossing, the crossing will be larger than the size of one cell to make room for all eight lanes to enter and exit the crossing.

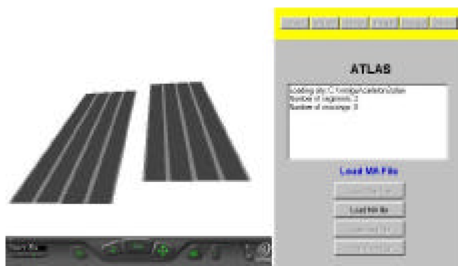


Figure 10.11 - A two-way four-lane segment

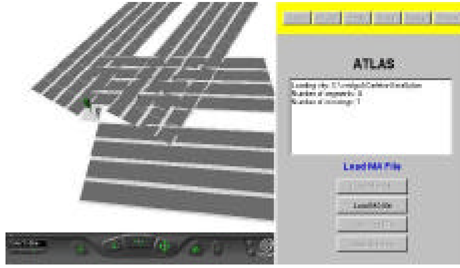


Figure 10.12 - An example of two two-way four-lane segments that overlap at a crossing.

Another problem could exist if the developer is not careful about where they place segments in the city when multilane, two-way segments are involved like in figure 10.13.

```
Segment_B = (0, 0), (10, 0), 2, straight, go, 40, 300, parkNone
Segment_B1 = (0, 0), (10, 0), 2, straight, back, 40, 300, parkNone
Segment_C = (4, 0), (4, 10), 2, straight, go, 40, 300, parkNone
Segment_C1 = (4, 0), (4, 10), 2, straight, back, 40, 300, parkNone
```

Figure 10.13 - ATLAS representation of two two-way, two-lane segments

The code in figure 10.13 will result in two two-way segments, each having two lanes in each direction. The only problem that exists is that the outside lanes from Segment_B1 will overlap with the outside lanes of Segment_C. This is because Segment_B1's lane 0 goes from (0,1) to (10,1) and lane 1 from (0,2) to (10,2). Segment_C's lane 0 starts at (3,0) and ends at (3,10) and lane 1 goes from (2,0) and ends at (2,10). From this example, it can be seen that lane 1 of Segment_C overlaps with lane 1 of Segment_B1. This has to be avoided, and if attempted, give the user an error somewhere in the overall system, either in the MAPS or TSC subsystems or both.

The crossings are being modeled in the VRML GUI as a one-cell object even though a crossing is treated as a roundabout by the simulator (see section 3.2). The simulator will simulate cars traveling around the roundabout and send output message when cars move to another location within the crossing. When these messages are received that indicate a car moving within a crossing, the car will just remain in the crossing until a message is received that indicates a car moving from the crossing to the segment. The crossings will not be rotated since segments can be joining the crossing from any direction. Crossings will simply be translated to the location that is stored in the 'where' attribute of the Crossing object.

10.3.4 Performing the Initialization

Each segment and crossing from the plan file will have to be stored in the system for the Initialize City use case. These segments are simply stored in an array in memory and cross-referenced when the ma file is loaded in the system and the models are stored. The verification that needs to be made is that all the segments and crossings match up with a cell-DEVS model. The Model object stores the number of lanes and length of the model in the 'd' attribute that corresponds to an array with length equal to the number of dimensions; in ATLAS segments and crossings are two dimensional. The first element of the d attribute holds the number of cells in the x direction of the model and matches up with the length of the segment. The second element holds the number of cells in the y direction of the model, which corresponds to the number of lanes in the segment. The model also stores the name of the model and whether the model is a cell-DEVS model. There must exist a cell-DEVS model that has the same name, lanes and length for every segment or crossing in the city section. If there is a cell-DEVS model that does not have a corresponding segment or crossing, or a segment or crossing does not match up with a cell-DEVS model, then the files entered by the user are not correct and the system should stop and make the user reenter the plan and ma file, or restart the VRML GUI.

10.3.5 Overcoming Problems While Visualizing the Output

After initializing the city is complete, the simulation results can be shown. The simulation results are stored in a log file that contain among many messages, output messages that indicate cars entering and exiting cells of certain segments and crossings. The best solution would be to create one VRMLNode object of type car and have it pass through the segments by just translating the same node or even better, animating the object from one location to another. However, having the cars move via animation was a difficult task because animations in VRML are based on timer sensors. The problem with basing animation on timing is that every simulation result is going to run in VRML GUI at a different speed depending on the number of segments, crossings, how busy the segments are, the speed of the computer on which the system is running among other factors. If the cars were to move at a consistent speed, say move one cell every second, then we could create a VRML node called a TimerSensor that sends an event to a PositionInterpolator node every 0.1s, that relays an event to all the car VRMLNodes to move 1/10 of its way towards its next location. This way, after one second, the cars would be at their

next location. Currently, this is not possible since there could be hundreds or even thousands of output messages every second for cars to move to new cells, and the system would have to determine ahead of time the destination cell of each car. This solution is beyond the scope of the project as animation was not part of the functional requirements but could be part of a future version.

Another solution would be to create a new car VRMLNode when a car is introduced into the city section and have it move throughout the city as the simulation output messages specify. This would have been the best solution in terms of memory usage (the number of VRMLNodes created would only the number of cars currently in the city), ease of implementation (would only have to store the location of each car, then translate the car VRMLNode), and execution time (only output messages that have a car appearing in the cell would have to be processed and all others could be ignored, and nodes wouldn't have to be added and removed from the scene). The only drawback with this solution is that the output messages don't indicate where the car came from. If a message indicated that a car is now in a cell, the system would not know which car node to move to the cell specified in the output message. Determining where the car came from would be difficult to determine, as the system would have to look at the rules for cars moving amongst cells in the ma file for each segment. The translation rules are difficult to read and would take a lot of execution time to read them every time a car moves within a segment and they would take up a considerable amount of memory if they were stored by the system.

The solution that seemed most appropriate was to create a car VRMLNode for every cell in the city, then add the node to the scene if a message indicates that a car is moving to a specific cell, or remove it from the scene when the output message says that a car is moving away from the cell. The output messages' form is shown in figure 10.14.

```
Mensaje Y / 00:00:00:200 / t1(0,0)(15) / out /      1.00000 para t1(14)
```

Figure 10.14 - An output message indicating that a car has appeared

This message is an output message as indicated by the Mensaje Y. The time that the message occurred within the simulation was 200ms after the simulation started. The message states that the segment or crossing that is affected is called t1, and at cell 0, lane 0, a car is appearing (as

indicated by the 1.00000). The message in figure 10.15 occurs at time 2 seconds after the simulation started and indicates that a car has left cell 3, lane 0 of segment or crossing t2.

```
Mensaje Y / 00:00:02:000 / t2(0,3)(08) / out /      0.00000 para t2(04)
```
Figure 10.15 - An output message indicating that a car has left a cell

When the messages in figure 10.14 is found, the system will look to an array of VRMLNodes and find the one corresponding to cell 0, lane 0 of segment t1 and add the car VRMLNode to the scene at the appropriate location in the VRML world. When the second message from figure 10.15 is reached, there is a car present in segment t2 in lane 0, cell 2. The system will remove this car VRMLNode from the world indicating to the user that the car has left the cell. The system will also display the time the message was received with respect to the beginning of the simulation. In this case, the time displayed will be 00:00:02:000.

One problem with this solution is the sheer number of VRMLNode object that will exist in the system after a long period of displaying results. Initially, no car VRMLNodes are created until a message indicates that a car is moving to a cell. At that time, a car VRMLNode is created with a location and direction equal to the translation and rotation of the road VRMLNode that the car will be displayed on top of. After creation, the car will be displayed on the screen indicating to the user that a car is present in that cell. When a message is sent indicating that the car has moved to another cell, the car VRMLNode corresponding to that cell is not destroyed from the system but instead just removed from the scene, thus making it invisible. The next time a message indicates that a car is moving into the cell, the node already exists so a new object does not have to be created, but just added back to the scene, which saves a considerable amount of time and makes the transitions of cars among cells smoother.

A drawback of this solution is that the first time a car moves along a segment, a car VRMLNode for each cell that the car enters needs to be created as the car travels along the segment. This first transition along each segment will be choppy as the system is creating several VRMLNodes. With a fast system and large amount of memory, these rough transitions can be avoided. The second pass however, will be smooth since the nodes have been pre-created. All the VRMLNodes could have been created at the same time during the initialization but this would

have consumed much time, so creating the VRMLNodes when the city is being displayed is a better solution. Also if there exists a cell that a car never enters, the VRMLNode object for that cell will not be created, thus saving memory and time. Another drawback is that once a car has traveled along every segment lane, there will be as many twice as many VRMLNodes in the system as there are cells; one for the road VRMLNode and another for the car VRMLNode. This could put a strain on the main memory and cause a slowdown in the system performance but once again, there was not a nonfunctional requirement on the amount of memory consumed by the system.

Another issue that needs to be addressed is how to deal with the crossings. In the simulator, all crossings were implemented as roundabouts (see section 3.2). However, as mentioned in section 10.3.3, we are showing the crossings as just one cell. This means that there will be messages indicating that the car is leaving a crossing where in reality, it is just moving to another cell within the crossing. This problem is addressed by the proposed solution of adding and removing car VRMLNodes because if this occurs, the car will be removed from the crossing when the message indicates that the car is leaving the crossing. Immediately after there should be another output message indicating that a car is still in the crossing since it moved to a new cell within the crossing, so the system will add the same node back to where it was removed.

Another feature of the system is the ability to stop the results from being displayed and allow the user to navigate around the city in a standstill. When the user clicks the stop-displaying button, the system stops displaying the results by halting the ReadLogFile control object from reading the log file. This will leave the cars in the position that they were in when the button was pressed so the user can navigate around the "frozen" world.

10.4 Implementation

Once the requirements, analysis and design were complete, the implementation of the system was relatively easy. As stated in the pseudo requirements in section 10.1.6, the implementation was to be done using Java. The output needs to be converted to a VRML world for the user view and navigate.

The structure of the GUI was already completed in the original VRML GUI system so this was not part of the requirements. I added a panel to the Java applet for ATLAS called the ATLASPanel. The ATLASPanel has four buttons, allowing the user to load, initialize, run and stop a city simulation. There is also a text output window for instructions to the user and for debugging. The ATLASPanel extends the Java Panel object that inherits the handleEvent method from the Component class. Because of this, by overwriting the handleEvent method, actions can be specified to the control classes when a button is clicked. In our case, as specified in the sequence diagrams, the ATLASPanel just passes control to the CityControl object.

The CityControl class is the brain of the operation as it is responsible for the majority of the actions of the ATLAS portion of the VRML GUI. When a filename and location is required from the user, a FileDialog object is created that allows the user to browse through their disks and find the appropriate file graphically. When the filename has been identified, the CityControl creates the appropriate file reader control object that is responsible for reading the important information from the inputted file. To parse the text files, the file reader control objects create a StringTokenizer object that makes parsing strings easier by reading portions of strings up to a specified delimiter. The CityControl object is then responsible for creating, storing and modifying the entity classes in the system.

There is a considerable amount of data to be stored in this project so the method of storing data is an important issue. The Vector class offered in the java.util package provides the necessary data structures for storing segments, crossings and VRMLNodes.

The segments read in from the plan file needed to be stored so they could be later verified in the Initialize City use case. Also we needed to keep track of the length of each segment to determine how many road VRMLNode objects to create and where to translate them. Two vectors were created by the CityControl class, one storing Segment objects and the other storing Crossing objects. Using these two Vectors, the model objects read from the ma file could be cross-referenced against the segments and crossings of the city section. To find a segment with a specific name, a loop is performed through the list of segments, and attempting to match the

name of the segment requested with the name of each segment in the vector. This could similarly be done for the list of crossings.

As described in the design section 10.3.5, the car VRMLNodes need to be stored so they can be easily added or removed from the scene without having to create the VRMLNode over and over again. There needs to be a storage position for every cell in the city section. There also needs to exist a map to get from a specific cell in a segment to the array of VRMLNodes stored in the system. To store the car VRMLNodes, I elected to place them in a two-dimensional vector (i.e. a vector of vectors). The vector called allCars contains many other vectors. The number of vectors that allCars will contain is equal to the number of crossings and segments in the city section combined. The vectors in allCars will be indexed according to the segments vector. Then depending on the lane and cell number of the car required, a car VRMLNode will be retrieved, then added or removed to or from the scene.

To clarify this process, an example will be used. The CityControl object has just received a message to add a car to the Segment_E segment in lane 2, cell 5 (both indexed from 0). We can assume that Segment_E has a length of 10. The first step is to find the index in allCars where the VRMLNodes for Segment_E are located. To get the index, a scan is performed through the segments vector until Segment_E is found. This index into the segments vector is then used to retrieve a vector in the allCars vector. The retrieved vector from allCars could contain many VRMLNodes. To find the correct car node, the mapping in equation 10.5 is used.

index_into_segment_vector = laneNumber * segmentLength + cellNumber (10.5)

So lane 0, cell 0 would give an index of 0. In the example presented, lane 2, cell 5 will give an index equal to 15. The 16th element (index 15 because of 0 indexing) in the vector of car VRMLNodes for Segment_E will contain the correct car VRMLNode to be added. If there is no VRMLNode in this location (i.e. this is the first time a car has reached this cell), then a new car VRMLNode must be created and added to the scene, and then added to this location in the vector for the next time a car arrives at this cell.

Most of the entity classes were easy to implement since classes such as Segment and Crossing just store the data from a text file. Since the objects are just storage, the only methods that had to be implemented were getters and setters that simply return or set the attributes for that type of class.

The VRMLNode class however is an exception; it does more than just store the data. This class is more of an entity and control class at the same time since there exist methods to add and remove nodes to and from the scene and getters and setters for the attributes of the class. When a VRMLNode is created, the control class must indicate what type of VRMLNode they want to create. For the ATLAS visualization, there will be car, segment and crossing VMRLNodes. The crossing VRMLNodes will contain a traffic light, stop sign, or neither. When then node is created, a VRML hierarchy is built as described in the background section 9.1. Five nodes are created using the createVrmlFromString method offered with the vrml.external.Browser package. The Transform, Shape, Material, ImageTexture, Appearance, and TouchSensor nodes all get created and built in a hierarchy as in figure 10.16.

```
Transform {
   children [
      Shape {
         Appearance appearance {
            Material material{}
            Texture texture {}
         }
      }
      TouchSensor {}
   ]
}
```
Figure 10.16 - The VRML hierarchy for the VRMLNode

Once the hierarchy has been built, we can add an external VRML file to the node using the createVrmlFromURL method. The method takes a URL parameter that contains the location of the VRML file. The second parameter is the destination node that the VRML code in the file is sent to. In this case, it is the transform node. The final parameter is the name of the MFNode eventIn location to send the nodes to. Since we are adding the code to the transform node, then we want to send an event to the transform node's eventIn called addChildren. With this call we will have a hierarchy the same as figure 10.16 but with the code in the external file added to the children of the transform node. After the hierarchy has been built, it is necessary to obtain a

reference to the events of the transform node so the cars, segments or crossings can be translated, rotated or scaled. If we want to translate, rotate or rescale the node, then we just call the setValue method on the eventIn of interest passing a parameter that defines the location, rotation amount, or scaling values. If the current location, rotation or scale values are needed, then the getValue method can be called on the eventOut of interest and the value will be returned [11].

As described in the design section 10.3.5, the car nodes need to be added and removed from the VRML world with ease. Adding and removing nodes to and from a scene is very similar to adding and removing nodes to or from other nodes. When adding an object such as a car to the VRML world, the car object contained in the VRML transform node as in figure 10.16 has to be added to the root node. The root node is the top most node of every VRML world and has just two eventIns, addChildren and removeChildren. To obtain a reference to the root node's add and removeChildren eventIns, we first need to obtain a reference to the root node itself. The root node can be obtained from a Browser object by using the Browser.getNode("ROOT") method call to the Browser class similarly to the way nodes were retrieved in section 9.1. This method will return a reference to the root node and hence allow subsequent VRML additions to the root node. If we want to add a car VRMLNode to the scene we send an event to the addChildren eventIn field of the root node with the node that needs to be added as the parameter. Similarly, we can remove existing nodes from the root by sending an event to the removeChildren eventIn.

10.5 Testing

Testing is necessary in every system. Finding differences between the expected behaviors specified by the system models and the observed behavior of the system happens in every software engineering project. The testing phase began with unit testing, followed by integration testing and finally system testing [12].

10.5.1 Unit Testing

This phase of testing is done to find faults by isolating an individual component using specific test cases. Unit testing was done extensively in the VRML GUI system. When a new class such as the Segment class was implemented, the class was first tested to ensure that all attributes contained the proper values after a test segment was constructed, and that the attributes

could be modified as they were designed. The Segment, Crossing, Model, and LogOutputMessage classes are all derived from strings, and parsing strings can be very touchy if the string from a file is not formatted the same every time, so the classes had to be tested carefully. Several different segments, crossings, models and log output messages were created and tested with different parameters to ensure that they reacted as predicted. Also involved in this testing was attempting to create invalid objects of these types and ensuring that the system responds properly. In general, since the parameter passed is a string generated by another subsystem, the objects should be created properly. The MAPS subsystem creates the plan file that generates the segments and crossings (although users can create these as well), the TSC compiler will generate the ma file that is read for initialization and the simulator generates the log output file that contains the log output messages.

The control and boundary classes were unit tested as well to ensure that when they were created, the proper method invocations were made and the proper entity objects were created and modified at the appropriate time according to the sequence diagrams from section 10.2.4.

10.5.2 Integration Testing

Integration testing is the task of integrating components together to ensure that they operate together as a unit. The two main classes that had to be tested initially were the ATLASPanel boundary class and the CityControl control class. The ATLASPanel object takes events from the user and passes them on to the CityControl class to handle the events. Testing was done to ensure that the proper method invocations in the CityControl class were being made when each of the four buttons were pressed. Once all the bugs were ironed out of the integration of these two classes, the integration of the CityControl control object with the entity objects had to be ensured.

The first action that happens within the VRML GUI system is showing the static view of the city. Many different segments and crossings were inputted into the system to ensure that the segments were showing up in their correct location in the VRML world and that the proper objects were present. Segments with a long and short length, multiple lanes, one and two-way segments and segments that are rotated at different angles were inputted and tested thoroughly. Also multiple

segments were added to the world either joined together with a crossing or separate to ensure that they displayed on the screen in the proper location. As a test, cars were added to the world in different locations in a test city section to ensure that they were placed in the proper lanes and cells of the appropriate segments or crossings. This test was to ensure that the cars were rotated by the same angle and translated to the same location as the segment within which they were contained. Additionally, it had to be ensured that when a car VRMLNode in a certain segment with specific lane and cell number needed to be retrieved from the data storage, the correct one returned. Similarly, a newly created car VRMLNode had to be stored in the correct entry of the data storage. Also tested were the file reading control classes to ensure that they created the proper entity objects with the correct attributes when they read in a string from a file.

10.5.3 System Testing

The system testing focuses on the system as a whole, ensuring that it's functional, nonfunctional and pseudo requirements are met as defined in section 10.1.1 of the requirements. This is the first time that the entire system including the GUI is tested to ensure full functionality of the VRML GUI (this version of the subsystem). At this point, unit and integration testing has been completed so theoretically the functionality of the individual components should work flawlessly. The first step was to go through the testing of the functional requirements. This was done using the use case descriptions from section 10.1.3. When the five use cases pass sample tests successfully as they were designed, then the system is functionally correct. Incorrect user input is also tested to ensure that the system doesn't go into an inconsistent state, and that the user receives proper feedback from the system to inform the user about their error. This also tests the recovery ability of the system when the user inputs incorrect data.

The test input that was used were cases derived as part of a final project for a graduate class SYSC 5807 - Advanced topics on Computer Systems: Methodological Aspects of Modeling and Simulation instructed by Prof. Gabriel Wainer. There were several plan, ma and log files generated by the TSC and simulator that were used to test the ATLAS system. The examples ranged from large city sections, to medium sections like Carleton University campus, to examples with only a few segments and crossings to test the functionality for many city size ranges. These variations test that the system is still functional when there exists many

VRMLNodes stored in the system. The only major problem encountered was when large city sections were loaded, the performance of the system degraded and it was difficult to navigate around the city when running. This however can be eliminated by running the program on a fast system with a large amount of memory. Also when many cars are moving around the city at the same time, the amount of real time that passes compared with the amount of simulation time is quite different. This is acceptable since timing issues were not part of the nonfunctional requirements.

10.6 VRML GUI Conclusions

The above sections have shown the process of software engineering for the VRML GUI subsystem. It started with the requirements to determine the needs of the client. This was done by coming up with functional requirements, use cases and nonfunctional requirements for the VRML GUI system. After the requirements were determined, the analysis phase came next where the VRML GUI system was modeled as a complete solution. The analysis phase focused on formalizing the system requirements by identifying problem domain objects, and determining their behavior, relationships, and classification. A sequence diagrams for each use case and system class diagram were realized to define the static and dynamic aspects of each object, along with their properties. The design phase looked at the analysis of the system and determines how to put the pieces together. Many design decisions had to be made to determine how the information was going to be stored, how to access that information as well as how to present the information to the user according to the specifications given in the requirements. The implementation stage took the design decisions and put the system together as a whole following the nonfunctional and pseudo requirements defined in the requirements phase. Finally, the testing phase was used to ensure that all functional requirements were followed to allow for a product that was robust, complete and reliable.

The development of the VRML GUI subsystem was a success according to the agreed upon requirements. The system allows the user to input a city plan file, and a static view of the city is displayed as in figure 10.17. The system then needs to initialize prior to showing the results of a previous simulation. To initialize, the system requires the user to input the model file used for simulation to ensure that the plan file matches up with the simulation performed. Finally, the

user can input the log results file from a previous simulation to view the city and how the cars proceeded throughout the segments and crossings as in figure 10.18.

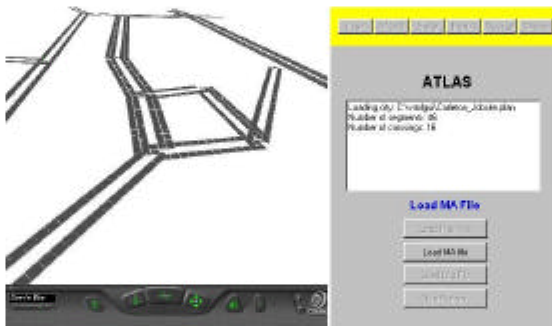


Figure 10.17 - The static view of the city with just segments and crossings

Figure 10.18 - Cars moving within the city

11.0 CONCLUSIONS AND RECOMMENDATIONS

The ATV system can be a very useful software product. It can allow users to develop a city section, simulate it under certain conditions and view the results. If a city is considering construction or foresees a delay in traffic on a specific road, they can design the city section affected by the delay using the MAPS subsystem, add in the construction site, simulate, and view the city using the VRML GUI to determine how traffic will flow after the delay has been added. If the city planner is considering adding a new highway or expanding lanes to an existing highway that is traffic plagued and want to know ahead of time whether adding the road or lanes will relieve some of the traffic, they can simulate the city with the changes and view the results to give them a rough idea of the effectiveness of the changes.

Prior to this project, the ATV system was purely text based. If a city developer wanted to create a city section, they had to learn the ATLAS semantics, then, by hand design the city without knowing exactly how the city looked. In order to get a rough idea of how the roads in the city were made up, the user would have to plot all the segments, crossings and control elements by hand. Even then visualizing the roads in the city would be difficult. Assuming that the city developer managed to design the city as they desired, they could set parameters as to how busy the city was, and the amount of traffic that would flow throughout. They could then compile the ATLAS code into a model file suitable for simulating. Once the model file was simulated, the user could get a text-based results file of all the messages that were passed amongst segments and crossings, but deciphering this file would have been a nightmare. A program was developed to help the user visualize the log output file from the simulator. The program called DRAWLOG takes the log output file and generates a text file that shows the results of individual segments with ones representing a car present and zeros when a car is not present as show in appendix B. DRAWLOG was a definite step forward towards visualizing the city but still was not adequate since the user could not see multiple segments at the same time, the time of the simulation was difficult to follow, and the results still were not as user friendly as they could be. Overall, only those experienced ATLAS programmers and those who could make sense of a log output file could use the system.

Adding two extra subsystems to the ATV system was a definite step in the right direction with regards to visualizing the city and the simulation output. The MAPS subsystem developed by Jan Pittner allows the user to graphically input a city using a graphical user interface. MAPS allows the user to venture away from the ATLAS abstractions and focus their attention on designing the city itself. Using MAPS, the user can add full roads that intersect and MAPS will automatically add in the crossings and divide the road up into segments for the user. The user can easily add construction sites, potholes and control elements by simply dragging and dropping the elements onto created roads. Once the city has been designed to the satisfaction of the user, MAPS will parse the drawing and create an ATLAS plan file that can be compiled, and then simulated generating output results for traffic flowing in the city. From there, the user can activate the VRML GUI subsystem designed and developed by Shannon Borho that will display the results of the simulation using VRML's 3D virtual reality capabilities. The VRML GUI will first show the city statically showing all the roads and crossings. After the initialization of the city has been completed, the user can input the log results file and the city will come to life, showing cars flowing throughout the roads and crossings as traffic would in the actual city. The tool gives the user the ability to move around the city to focus their attention on specific areas of concern, possibly a specific busy crossing or a road that has construction. If the user was not happy with the results of the traffic flowing throughout the city, they can go back and change the design of the city again using MAPS, re-simulate and view the results again.

Also investigated for this project was the possibility of using existing GIS systems to automatically input current city structures into the MAPS program. This would allow a city planner to have a map of their city ahead of time, without having to even graphically input the roads and crossings. From there, the user could add extra parameters like jobsites, potholes, crosswalks, or other control elements that GIS systems cannot provide. The user could then simulate the modified city and view the results using the VRML GUI to determine if the control elements added to the city were feasible without delaying traffic. In the distant future, it may be possible to have sensors reading in information about the flow of traffic in a city to MAPS along with other factors such as lanes being closed or traffic lights going out. With real-time information, we can simulate and view the results on the VRML GUI. From the output results, if it is determined that drivers could take an optimal path to a particular destination, avoiding the

traffic jam that is likely to occur based on the information provided, then that path could be provided to the drivers via signs on the side of the roads or even signals being sent to GIS systems that exist inside many of today's cars. Innovations such as the ATV could drastically help the traffic factor that plagues virtually every major city in the world.

12.0 REFERENCES

[1] JHotDraw website, available at http://jhotdraw.sourceforge.net as of April 2nd, 2003.
[2] M. Tartaro, C. Torres, G. Wainer, Departamento de Computacion, Universidad de Buenos Aires “TSC - Traffic Simulator Compiler”, 2000.
[3] MapMaker website, available at http://www.mapmaker.com/ as of April 2nd, 2003
[4] MapInfo website, available at http://www.mapinfo.com as of April 2, 2003
[5] ESRI MapObjects website, available at http://www.u-i-s.com/software/descesri_other.htm as of April 2, 2003
[6] “How To Use Model View Controller”, Steve Burbeck, no longer available at http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html as of April 2, 2003. A Google cache is available (Google for that website) as of April 2, 2003.
[7] “Design Patterns, Elements of Reusable Object-Oriented Software”, Gamma et al, Addison Wesley, 1995
[8] B. Crispen, “comp.lang.VRML FAQ Answers” [online document], December 21, 2000, (Version 2.53), Available: http://vrmlworks.crispen.org/faq/faq1.html#q1, (April 2, 2003).
[9] A Scott, Aerael, Inc. “Getting Started in VRML”, [online document], 1996, Available: http://www.vrmlsite.com/oct96/spotlight/qa/qa.html (April 2, 2003).
[10] J Smith, Vapour Technology Ltd. “Floppy’s VRML97 Tutorial”, [online document], November 15, 2002, Available: http://web3d.vapourtech.com/tutorials/vrml97/ (April 2, 2003).
[11] D.K. Schneider, S. Martin-Michiellot, Faculte de Psychologie et des sciences de l'education, University of Geneva, “VRML Primer and Tutorial”, [online document], March 18, 1998, (Version 1.1a), Available: http://tecfa.unige.ch/guides/vrml/vrmlman/vrmlman.html (April 2, 2003).
[12] B. Bruegge, A.H. Dutiot, *Object-Oriented Software Engineering - Conquering Complex and Changing Systems*, Upper Saddle River, NJ: Prentice Hall, 2000.
[13] P.Y. Woo, Biola University, “Trigonometry Formulas and Calculus Formulas”, [online document], August 27, 1999, Available: http://www.woopy.woo.org/math/formulas.htm (April 2, 2003)

13.0 APPENDICES

Appendix A - The MA (Model) File

```
components : University_A
components : Library_A
components : Library_CCons@Consumer
components : Library_C
components : Campus_AGen@Generator
components : Campus_A
components : Campus_B
components : c2
components : c3
components : c5
components : c4
link : y-co-hayauto012@Anniversary_A x-t-hayauto0@Anniversary_ACons
link : y-co-hayauto112@Anniversary_A x-t-hayauto1@Anniversary_ACons
link : y-t-hayauto0@University_AGen x-ge-hayauto00@University_A
link : y-t-hayauto1@University_AGen x-ge-hayauto10@University_A
link : y-co-hayauto022@Library_C x-t-hayauto0@Library_CCons
...
[Library_B]
type : cell
width : 27
height : 1
delay : transport
defaultDelayTime : 1
border : nowrapped
neighbors : Library_B(0,-1) Library_B(0,0) Library_B(0,1)
initialvalue : 0
in : x-c-hayauto00
out: y-c-haylugar00
in : x-c-haylugar026
out: y-c-hayauto026
link : x-c-hayauto00 x-c-hayauto@Library_B(0,0)
link : y-c-haylugar@Library_B(0,0) y-c-haylugar00
link : x-c-haylugar026 x-c-haylugar@Library_B(0,26)
link : y-c-hayauto@Library_B(0,26) y-c-hayauto026
localtransition : Library_B-lane0-rule
zone : Library_B-segment1-cell00-rule {(0,0)}
zone : Library_B-segment1-cell0n-rule {(0,26)}
zone : Library_B-segment1-lane0-rule  {(0,1)..(0,26-1)}

[Anniversary_A]
type : cell
...
```

```
[c2-cellIn-rule]
rule : {1 + send(1, y-t-haylugar)}  20 { (0,0) = 0 and ( (0,-1) = 1 or portvalue(x-t-hayauto) = 1 ) }
rule : {0 + send(0, y-t-haylugar)}  20 { (0,0) = 1 and (0,1) = 0 and (0,-1) = 0 }
rule : {0 + send(1, y-t-haylugar)}  20 { (0,0) = 1 and (0,1) = 0 and (0,-1) = 1 }
#Macro(c2-Default)

[c2-cellOut-rule]
rule : {1 + send(0, y-t-hayauto)}  20 { (0,0) = 0 and (0,-1) = 1 and ( portvalue(x-t-haylugar) = 1
or (portvalue(x-t-haylugar) = 0 and random < 3)}
rule : {0 + send(1, y-t-hayauto)}  20 { (0,0) = 0 and (0,-1) = 1 and portvalue(x-t-haylugar) = 0
and random >= 3 }
rule : {0 + send(0, y-t-hayauto)}  20 { (0,0) = 1 and (0,1) = 0 }
rule : {(0,0) + send(0, y-t-hayauto)}  20 { t }

[c3-cellIn-rule]
rule : {1 + send(1, y-t-haylugar)}  20 { (0,0) = 0 and ( (0,-1) = 1 or portvalue(x-t-hayauto) = 1 ) }
rule : {0 + send(0, y-t-haylugar)}  20 { (0,0) = 1 and (0,1) = 0 and (0,-1) = 0 }
rule : {0 + send(1, y-t-haylugar)}  20 { (0,0) = 1 and (0,1) = 0 and (0,-1) = 1 }
#Macro(c3-Default)
```

Appendix B - An Example of a DRW File

```
Time: 00:00:00:000
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:00:200
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:00:400
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:00:600
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:00:800
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:01:000
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:01:200
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:01:400
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  0  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:01:600
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0  1  0  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:01:800
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  0  1  0  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0
Time: 00:00:02:000
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  0  1  0  1  0  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0
```

...

ATLAS: A Language For Modeling And Simulating Urban Traffic

Visualization Of Traffic Models

ADDENDUM

Jan Pittner

Introduction

This document serves to correct and complement the final report submitted April 4th for the ATLAS Traffic Visualization system. Shortly prior to submitting the final report, a severe bug was found with the MAPS program. This document updates the changes made to MAPS and corrects the final report.

As with any software project, bugs are bound to occur. The bug in question relates to how MAPS parsed roads into segments. The submitted report refers to a method in which the roads are segmented by their lanes. In some cases, this method would not be functional with the TSC or the 3D visualization program written by Shannon Borho. The MAPS output was not tested with the TSC due to the TSC's current semi-functional state. Nor was MAPS tested (prior to the submittal of the final report) with the 3D visualization program ("VRML program ") written by Mr. Borho.

Testing has since occurred, the results of which are shown in this addendum. Please note that due to limiattions with the VRML program, testing is basic and only verifies the correct segmentation of the roads. It does not test some of the major components of MAPS, which is the addition of decorations (although since parking decorations create segments, this is the only decoration whose effects on the city section can be observed in the VRML program).

The Segmentation Problem

When the final report was submitted, each lane in a road was created as a segment. This lane-segment was further segmented based on the parking available in the lane. This is unexpected behaviour for the TSC and the 3D VRML program.

A road with two lanes
------------------>
--pppppppp----->

Created four segments, each with a unique segment ID corresponding to the lane number and the parking available on that lane.

New Segmentation Procedure

Roads are now segmented vertically, across all lanes, based on the intersections with other roads and parking decorations on the road. As such, the road in the previous example would create three segments:

-->pppppppp>----->
-->----------->----->

This is the expected behavior by both TSC and the VRML program.

The following side effects are to be noted with this method:

parkBoth feature is not implemented: that is, a user may not draw a road with 3 lanes (all in the same decoration) and having parking on both the exterior lanes.

Not Permitted:

--ppppppp->
-------------->
--ppppppp->

Decorations (such as roadwork) may not traverse two directions of traffic.

Not permitted:
-------------->
------**R-**----->
<----**R**-------
<-------------

Instead, the user must draw two roadwork decorations (shown here as one in bold, the other in italic):

-------------->
------**R-**----->
<----*R*-------
<-------------

Additionally, all decorations must have a length of 1 units, no more:

Not permitted:
-------------->
------**RR**---->

Instead, the user must draw two decorations (shown here as one in bold, the other in italic):

-------------->
------**R***R*---->

In case of incorrectly drawn decorations, the program may not behave as expected (in most cases it will give the user an error).

Finally, roads may not have a melee of lane directions. The direction of traffic must be uniform, and there can only be one change in the direction of traffic across the road:

Permitted:

------------->
<------------

Not permitted

------------->
<------------
------------->

The lane direction rule will generate a parse error should it be violated.

ADDENDA:

The BreakPoint class was updated to store the type of parking decoration (valid only for Start Parking points) and the delay associated with that parking.

A RailNet figure was added. Note that MAPS currently supports only one RailNet in the city section.

The magicPoint method had minor bugs and now correctly returns the point in between two points on a line at given distance *dist* from the start of the line. There is a self check that should never fail now that the code is correct. The check fails if the point located is greater than 1 unit away from the line (eg, a point that is 0.3 units away from the line is “on the line” whereas a point that is 1.1 units away from the line is off of the line). This check was important to verify the functionality of the magicPoint function so that decorations could be properly located and checked if they applied to a particular segment of a road.

The MAPSPropertyPanel and toolbar was updated to support all attributes available for ATLAS objects. New tools were added to support other ATLAS decorations.

Code was refactored to increase cohesion (parsing of CityDrawingView and RoadDrawingView is now done through the ATLASParser2 class.)

The type of parking is “guessed” for a segment. If it is in the first lane, it is assumed to be parkLeft. If it is in the last lane, it is assumed to be parkRight. This should be verified with the direction of the lane with respect to the direction of the road as it was drawn by the user. Clarifications in the user interface must be made to clearly indicate to the user how the parking will be classified.

A “Generate Intersections” menu item has been added to the ATLAS menu. This allows the user to generate the intersections and modify any of their properties prior to generating the ATLAS code. Should the user not wish to do so, the intersections will be automatically generated when the ATLAS code is created, with default values for the intersections.

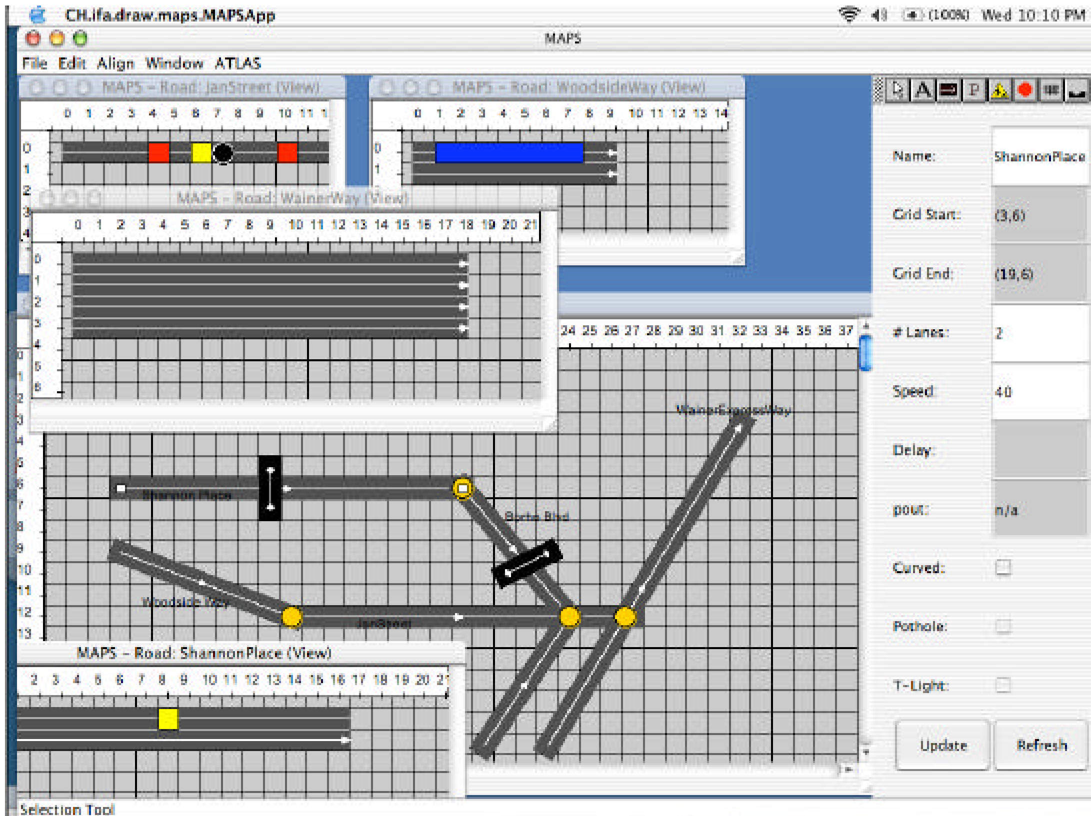
The number of lanes can be updated via the MAPS property panel. The user is warned that updating the number of lanes erases any decorations (and lane directions) previously set. As such it is advised that users set the number of lanes for roads prior to adding decorations and changing lane directions.

The program currently supports one railnet. The user must take care not to cross two roads with the same railnet figure, and it is advised that the delay is set for all railnet figures. (note that only one delay may apply so all delays should be the same).

Only the “Stop Sign” control element is supported. Additional support for other control elements was not added due to unfamiliarity with pop up menus which would have been the best way to select the type of control element. Additionally the problem arose of how to display the various types of control elements with the given figures without having to create more. This problem was deemed minor and not addressed.

All the tools are currently on one toolbar which does not enable/disable tools as the view changes (for instance, drawing parking on the CityDrawingView is not allowed, however the user can do it. These objects are ignored when the drawing is parsed). To rectify this, the toolbar would have to implement the ViewSelectionChangeListener interface and adjust accordingly. There was insufficient time to implement this.

Screenshot of MAPS:



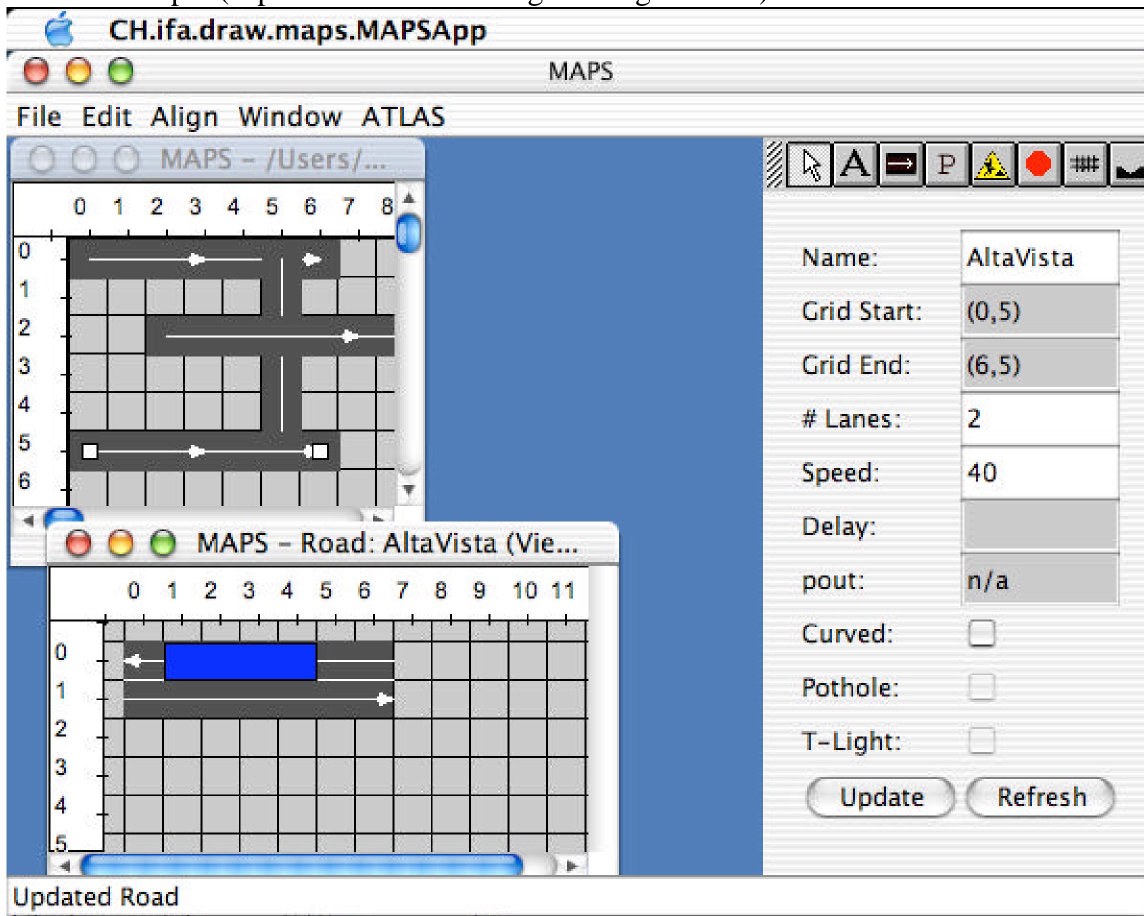
Note the presence of railnets (black rectangle with white line), crossings (yellow circles, automatically generated), roadwork (yellow squares), stop signs (red squares), parking sections (blue rectangles), multiple and bidirectional roads, ready access to ATLAS parameters such as speed, curvature of the road, etc.

Screenshot of generated ATLAS Code (partial):

Generating ATLAS Code...

```
176:    begin segments
177:    ShannonPlaceGOS1=(3,6),(19,6),1,straight,go,40,0,parkNone
178:    ShannonPlaceBACKS1=(3,6),(19,6),1,straight,back,40,0,parkNone
179:    BorhoBlvdGOS1=(19,6),(24,12),1,curve,go,55,0,parkNone
180:    BorhoBlvdGOS2=(24,12),(24,12),1,curve,go,55,0,parkNone
181:    BorhoBlvdBACKS1=(19,6),(24,12),1,curve,back,55,0,parkNone
182:    BorhoBlvdBACKS2=(24,12),(24,12),1,curve,back,55,0,parkNone
183:    JanStreetGOS1=(11,12),(24,12),1,straight,go,55,0,parkNone
184:    JanStreetGOS2=(24,12),(24,12),1,straight,go,55,0,parkNone
185:    JanStreetGOS3=(24,12),(26,12),1,straight,go,55,0,parkNone
186:    JanStreetGOS4=(26,12),(27,12),1,straight,go,55,0,parkNone
187:    WainerWayGOS1=(23,18),(26,12),4,straight,go,55,0,parkNone
188:    WainerWayGOS2=(26,12),(32,3),4,straight,go,55,0,parkNone
189:    WoodsideWayGOS1=(3,9),(4,9),2,straight,go,70,0,parkNone
190:    WoodsideWayGOS2=(4,9),(10,11),2,straight,go,70,20,parkLeft
191:    WoodsideWayGOS3=(10,11),(11,12),2,straight,go,70,0,parkNone
192:    PittnerPlazaGOS1=(20,18),(24,12),2,curve,go,90,0,parkNone
193:    PittnerPlazaGOS2=(24,12),(24,12),2,curve,go,90,0,parkNone
194:    end segments
195:
196:
197:    begin crossings
198:    ShannonPlace&BorhoBlvd = (19,6),40,withoutTL,withoutHole,0,0.5
199:    BorhoBlvd&JanStreet = (24,12),55,withoutTL,withoutHole,0,0.5
200:    JanStreet&WainerWay = (26,12),55,withoutTL,withoutHole,0,0.5
201:    JanStreet&WoodsideWay = (11,12),55,withoutTL,withoutHole,0,0.5
202:    end crossings
203:
204:
205:    begin ctrlElements
206:    in JanStreetGOS1 : stopsign,4,30
207:    in JanStreetGOS1 : stopsign,10,20
208:    end ctrlElements
209:
210:
211:    begin jobsites
212:    in ShannonPlaceBACKS1 : 0,8,1,666
213:    in JanStreetGOS1 : 0,6,1,80
214:    end jobsites
215:
216:
217:    begin railnets
218:    rn1 = (ShannonPlaceGOS1,7),(ShannonPlaceBACKS1,7),(BorhoBlvdGOS1,2),(BorhoBlvdBACKS1,2),0
219:    end railnets
```

Another sample (captured before crossings were generated):



MAPS generated the following plan file:

```
begin segments

BankGOS1=(0,0),(5,0),1,straight,go,60,0,parkNone
BankGOS2=(5,0),(6,0),1,straight,go,60,0,parkNone
BankBACKS1=(0,0),(5,0),1,straight,back,60,0,parkNone
BankBACKS2=(5,0),(6,0),1,straight,back,60,0,parkNone
LibraryGOS1=(5,0),(5,2),2,straight,go,55,0,parkNone
LibraryGOS2=(5,2),(5,5),2,straight,go,55,0,parkNone
LibraryBACKS1=(5,0),(5,2),2,straight,back,55,0,parkNone
LibraryBACKS2=(5,2),(5,5),2,straight,back,55,0,parkNone
AltaVistaGOS1=(0,5),(5,5),1,straight,go,40,0,parkNone
AltaVistaGOS2=(5,5),(6,5),1,straight,go,40,0,parkNone
AltaVistaBACKS1=(0,5),(1,5),1,straight,back,40,0,parkNone
AltaVistaBACKS2=(1,5),(4,5),1,straight,back,40,45,parkLeft
AltaVistaBACKS3=(4,5),(5,5),1,straight,back,40,0,parkNone
AltaVistaBACKS4=(5,5),(6,5),1,straight,back,40,0,parkNone
BronsonGOS1=(2,2),(5,2),1,straight,go,75,0,parkNone
BronsonGOS2=(5,2),(12,2),1,straight,go,75,0,parkNone
end segments

begin crossings
Bank&Library = (5,0),60,withoutTL,withoutHole,0,0.5
Library&AltaVista = (5,5),55,withoutTL,withoutHole,0,0.5
Library&Bronson = (5,2),55,withoutTL,withoutHole,0,0.5
end crossings
```