Kennesaw State University

Department of Computer Science

Object Oriented Modeling and Simulation CS4491 Section 2

Project: Object Oriented Traffic Simulation

Jordan White, Nicholas Powers  
jwhit144@students.kennesaw.edu

npowers2@students.kennesaw.edu

Contents

[Figures 3](#_Toc418583683)

[Abstract 4](#_Toc418583684)

[Introduction 4](#_Toc418583685)

[The model 4](#_Toc418583686)

[Internet packet and vehicle analogy 6](#_Toc418583687)

[Vehicle routing 6](#_Toc418583688)

[Preventing deadlock 7](#_Toc418583689)

[Simulation GUI 7](#_Toc418583690)

[User interface 8](#_Toc418583691)

[Simulation output 8](#_Toc418583692)

[Source code 10](#_Toc418583693)

[Conclusion 10](#_Toc418583694)

[References 11](#_Toc418583695)

# Figures

[Figure 1: Illustration of two intersection nodes in a simulation model with LaneQueus (Blue), and Intersections.The white arrows represent the paths vehicles can make 5](#_Toc418578704)

[Figure 2: Image showing the expanded build menu 7](file:///C:\Users\Nick\Desktop\TrafficSim\Final%20Project%20Report\finalprojectreport.docx#_Toc418578705)

[Figure 3: JSON file containing statistical and physical information about the model and simulation. 9](#_Toc418578706)

# Abstract

This document describes the software produced to model and simulate street networks. The algorithms used to simulate vehicle behavior will also be mentioned. The Simulation Project required modeling, development, testing, and modifications to make it fully functional. Traffic simulations require a high degree of attention to functionality. Also, since Unity Engine was part of this project, the process functions needed to cooperate under certain conditions with the user interaction on the interface of the simulation. The simulation required tools to model the roads, intersections, and points of interest using separate Unity Engine classes connecting back to the controller classes.

# Introduction

The project in question was to create an accurate traffic-based simulation involving intersections, roads, and points of interest. What we had to do was think hard about how to simulate traffic accurately in the most simplistic manner that we could think of. The Unity3d engine scripted with C# was chosen over OOSimL because of its UI and graphics features. Without these features developing such a GUI as the one that follows would not have been possible in the constrained development time. A process-based simulation is used to model time and events within the simulation. To accomplish the development process, models and outlines of how real traffic models had to be produced. Unity3d provides support for C#, Jscript, and Boo scripts, C# was chosen as it provided the most reliable code, and had built-in support for concurrent software features. By using these tools, a product which simplifies the model construction process will be developed.

# The model

The model contains three primary types of objects. Those are **LaneQueues**, **Vehicles**, and **Intersections**.

The **LaneQueue** is a passive object that represents a section of a road, specifically a single lane. It functions similar to a standard queue data structure. The capacity of the internal queue is dependent on the length of the section of roadway. Also the **LaneQueue** has a *CurrentLength* field that represents the sum of the length of the contained vehicles and the space between them.

The Vehicle is a passive object that represents a road vehicle. At any given time a vehicle is either stored inside a **LaneQueue** or temporarily within an intersection. In the latter case the vehicle is currently turning or switching lanes. When a vehicle is not turning it is either moving or stationary within a lane, in this case the vehicle is stored within a **LaneQueue**.

The **Intersection** is the only active object in the simulation. An **Intersection** can represent any kind of transition in the roadway structure, for example an extra lane is added or an intersection. An intersection represents a node in the user generated street network graph. When the user places a point of interest along the roadway they place an intersection. That intersection then instantiates or removes vehicles from the simulation. Currently all points of interest behave the same. Vehicles are randomly dispatched from one point of interest to another.

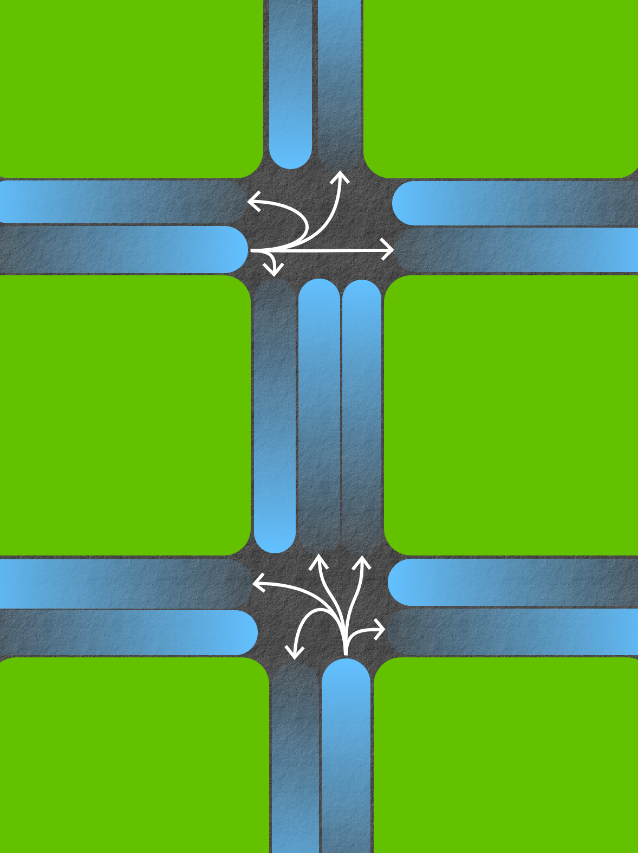


Figure 1: Illustration of two intersection nodes in a simulation model with LaneQueus (Blue), and Intersections.The white arrows represent the paths vehicles can make

# Internet packet and vehicle analogy

In order to accurately model traffic, vehicles need to navigate as they would in reality. A vehicle’s destination depends on where the vehicle starts. To most efficiently get to its destination a vehicle needs to make a specific set of turns along its way. Similar to how packets in the internet make a specific set of hops between routers, vehicles are routed from one intersection to another.

# Vehicle routing

Finite Automata is used to route vehicles to their destination. Before running the simulation a transition matrix is constructed. Vehicles are routed to minimize the distance travelled. To begin constructing the transition matrix the shortest path between every pair of nodes is first found using the Bellman Ford algorithm. Using this information the transition matrix is constructed.

The columns of the matrix represent the destination, while the rows represent the starting point. To figure out the next node to visit using this matrix simply query the cell at the row that is your current node index, and the column that is the destination’s index. A jagged array is used to store this matrix since not all nodes in the graph will be a destination. To find the value of the ith component in the jth column simply compare the shortest path value of all adjacent nodes to the ith node and pick the one with the smallest value for destination j. The transition function uses this matrix to determine the next node the vehicle will go to.

# Preventing deadlock

In situations where cycles exist within the graph it is possible for vehicles to get deadlocked. To prevent this from happening, a second transition matrix is used to provide a detour. To construct this matrix an algorithm similar to the one used to construct the optimal transition matrix is used, however instead of choosing the node with the smallest shortest path it instead picks the one with the second smallest shortest path.

When a vehicle waits at an intersection for too long the next node is chosen from the second transition matrix. Currently the simulation only supports nodes with a maximum out-degree of 4. That means a total of 4 transition matrices can exist each one representing a better option.

# Simulation GUI

To reduce the complexity of the model construction and modification process a GUI is used to provide tools the user can use to modify the street network. After the user has constructed a high level representation of the model an algorithm will construct the final model. To construct the final model the user’s graph of streets and intersections is transformed into a new graph consisting of **LaneQueues**. The new graph is much larger in terms of node and edge count. Each edge contains length information.

A vehicle’s movement is influenced by physics. Exchanging vehicles between intersections needs to be done carefully to accurately model the movement of a vehicle.

# User interface

 The street graph construction tools include a *Build Intersection*, *Build Road*, and *Build Point of Interest*. To use any of these three tools press the “Build” button on the bottom left of the screen. Currently intersections and roads can only be added. Select the solid circle in the submenu after pressing on the *Build Intersection* button*.* Before placing an intersection give it a name. Type the name into the “Intersection name” input field at the bottom of the screen. Type a floating point number into the “Speed modifier” input field to set the simulation speed as a multiple of real-time. Slower speeds should be used when simulating large models. Press “Start” to run the simulation, and then “Stop” to end the simulation. Once this is done review the “save.json” file produced by the application. For best results use a JSON object viewing tool.

Figure 2: Image showing the expanded build menu

# Simulation output

Statistical and physical information about the model and simulation can be found within a file name “save.json”. As the file extension suggests, this file contains JSON objects. When the user presses the “Save” button in the top-right corner of the GUI this file is written, and contains all of the information about the model. An intersection can have any name including no name. Contained within the “intersections” object within the main object is an array of fields where each one is a JSON object defining one of the intersections. A “pathTimes” variable is used to store another object that contains a collection of floating point fields. These fields represent the average time it takes for a vehicle to travel from the parent object’s node to the node whose name is in the field name.

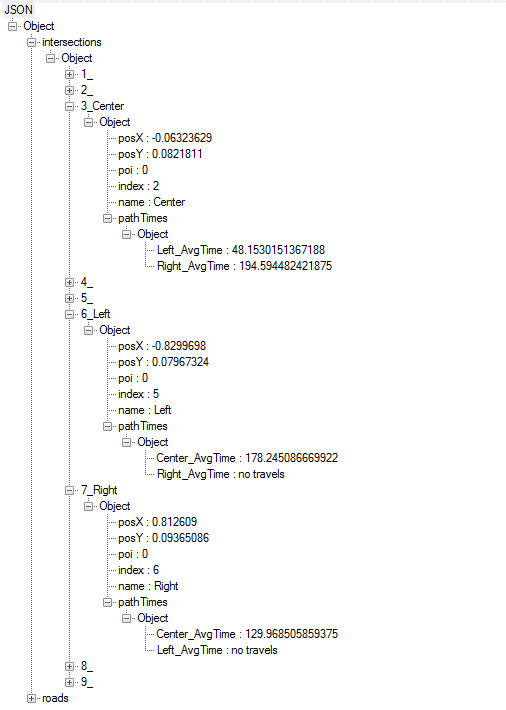


Figure 3: JSON file containing statistical and physical information about the model and simulation.

# Source code

The **MainCamera**, **Intersection**, and **Road** classes all inherit the **MonoBehaviour** type provided by Unity3d making those components. It links these components to an instance of a **GameObject** in the Unity3d engine. User input is handled by the **MainCamera** component/GameObject to create the model by instantiating all other types. A collection of tools were developed to provide polymorphic behavior to the tools that the user would need to construct the street network.

The **Simulation** class provides some helper functions for time management. The **Intersection** class references this class to sleep for specific durations of time that would match the time within the simulation. Everything contained within the **SimpleJSON** namespace [1] provides the JSON loading and saving functionality to the software. References to this class are made within the **MainCamera** class to perform file IO. The **NumberGenerator** namespace [2] contains the **SimpleRNG** class that is used to produce random numbers with an assortment of probability distributions.

The source code is located in the directory “Assets/Scripts/” within the project directory. In the event that Unity3d editor software is available the “MainScene.unity” file can be opened to view the Unity3d assets and debug the project software.

# Conclusion

By providing tools to construct a high level model of the street network, and writing an algorithm to convert that high level model to a low level representation, the model construction process is greatly simplified. The user is able to construct any variation of graphs of two lane roadways. By treating points of interest as intersections the model is kept simple and development time is shortened. Curved roadways are also possible by creating a dense sequence of nodes and edges that curve. Obstacles such as accidents and railroad crossings are also implementable using this simple model of intersections and roadways.

# References

1. "Simplejson/simplejson." GitHub. N.p., n.d. Web. 05 May 2015.
2. Cook, John N. "Simple Random Number Generation." CodeProject. Code Project, 26 June 2014. Web. 05 May 2015.