# A SIMULATION AND REGRESSION TESTING FRAMEWORK FOR AUTONOMOUS VEHICLES

by

CHRISTIAN KURTZ MILLER

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Thesis Advisor: Dr. M. Cenk Çavuşoğlu

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

August, 2007

# Contents

# List of Tables

# List of Figures

# Acknowledgements

Thanks are long overdue to the many people who have helped me. Cenk deserves a medal of honor for sticking with me through three years of thesis mishaps, all the while showing infinite patience and constant encouragement. Prof. Branicky has been a source of interesting ideas and inspiration in innumerable ways, and Prof. Newman deserves thanks for having the guts to get us into the Urban Challenge in the first place. I owe Prof. Buchner a large debt of gratitude as well, as he has opened many doors for me.

Team Case has been an unbelievable group to work with, and I must give special mention to Amaury, Andy A., and Scott for their constant design ideas, feedback, and bug reports. This project is a once-in-a-lifetime opportunity, and we are all lucky to be working with such talented people. We have truly been Urban Challenged.

My friends from Case deserve thanks for filling the past five years to the brim with fantastic and memorable times, always keeping me on my toes, and making me who I am now. You will never be forgotten. As for Kate and Brooke, there's just too much to say... thanks for everything!

Last but not least, thanks to my family, for their encouragement and guidance gave me everything in life.

# A Simulation and Regression Testing Framework for Autonomous Vehicles

Abstract

by

CHRISTIAN KURTZ MILLER

This work describes the design and implementation of a software suite capable of automatically testing and evaluating autonomous vehicle behavior within the context of the Defense Advanced Research Projects Agency (DARPA) Urban Challenge. The first major component is a program called Dexsim, which simulates vehicle dynamics and sensor inputs for several dozen vehicles at a time. The second major component is a regression testing framework that is capable of performing evaluations of a test vehicle's performance in simulation. Two types of tests are discussed: functional tests, which flex specific aspects of a vehicle's behavior in targeted scenarios, and randomized tests, which stress the vehicle's stability and reliability in long missions. Several examples of functional tests are given, and the design of autonomous agents used in randomized tests is detailed. Results are discussed, and the suite's utility as a development tool is examined in the context of the Urban Challenge.

# Chapter 1

# Introduction

This thesis describes the design and implementation of an autonomous vehicle behavior simulation and testing system developed to support Case Western Reserve University's entry in the 2007 DARPA Urban Challenge [17], a vehicle named Dexter (Figure 1.1). The simulation component consists of a program called Dexsim, which provides an environment for the test vehicle to navigate and simulates both vehicle dynamics and sensors. The testing component consists of a scripting system capable of evaluating the vehicle's performance, a series of functional tests, an agent architecture to provide interesting rivals controlled by independent artificial intelligence (AI), and a framework that binds it all together into an automated regression testing system.

The point of this system is to help Case's autonomous vehicle development team identify faults in vehicle behavior and isolate their causes so they can be fixed. If used properly, it can help the team make the most of the available development time and produce a much more reliable vehicle, thus improving the team's chances of performing well in the competition. While the system was designed with the specific requirements of the Urban Challenge in mind (and thus is biased towards a very particular class of vehicles), the ideas used in its construction are applicable to the

Figure 1.1: Team Case's autonomous vehicle, Dexter.

field of autonomous robot testing in general.

## 1.1 Motivation

If autonomous robots are to ever achieve widespread use, we must first have some guarantee that they will display correct or, at worst, reasonable behavior when faced with any situation, especially those not anticipated by the designers. As any roboticist can attest, autonomous robots manage to mire themselves into the most outstanding situations, violating virtually every implicit and explicit assumption that may have been used in their creation. Regardless of the quality of its components and algorithms, an autonomous robot is only as good as its testing.

Testing certain aspects of a robot is easy. Actuators can be stress-tested in the lab or factory, sensors can be run in test rigs, and individual processing modules can be fed logged or artificial data. Such unit tests provide some verification that, at the very least, the components used to build the robot's software and hardware work in a controlled setting. However, integration testing of all the components in concert is


Figure 1.2: The Roomba, an autonomous floor vacuum. Image courtesy iRobot, Inc.

more difficult, and techniques vary wildly depending on the intended purpose of the robot in question.

Take, for example, an autonomous floor vacuum like the Roomba (Figure 1.2). After the robot has graduated from a laboratory environment, an effective means of integration testing is for the engineers to simply take prototypes home and try them out. There, they will encounter stairs, furniture, pets, shoes, a variety of surface conditions, different room shapes and sizes, and so on. Such environments are nearly identical to the final intended deployment areas for the robot, and serve as ideal grounds for verifying the robot's behavior.

As a robot's purposes become more esoteric or dangerous, the testing procedures must become more elaborate. An autonomous lawn mower, like the one seen in Figure 1.3, must be tested on a variety of different patches of grass of varying sizes, shapes, surroundings, and environmental conditions. It would be of interest to evaluate the robot's respose to moving cars on a highway, to dense woods, to hillsides, to swingsets, to rain, to pets, or to pedestrians. For good measure, it should also be tested on parking lots, roads, driveways, patios, or any other surface it might

Figure 1.3: An autonomous lawnmower. Image courtesy Miami University and The Institute of Navigation.

encounter in its quest for grass to cut. What happens when it runs over a brick and chips its blade? If it flips over? If it runs out of gas? Not only are a staggering variety of test conditions required to fully profile the robot's behavior, but until it reaches a certain assumed reliability, the test environments must be cordoned off to prevent unexpected encounters between the outside world and an misbehaving robot armed with rotating blades.

For an even more complicated example, look at the DARPA Urban Challenge. The robots in question are full-size ground vehicles (Figure 1.4), as powerful and dangerous as those driven by humans. It is unsafe, impractical, and illegal to deploy such vehicles on public roads, meaning that physical testing is relegated to rented or volunteered facilities. Even small tests require significant amounts of time, effort, and personnel to conduct, and must be executed efficiently to be of any use. Each team has at most a handful of autonomous vehicles [3], and thus traffic must be


Figure 1.4: TerraMax, a competitor in the 2007 Urban Challenge. Image courtesy Team Oshkosh.

approximated by human drivers, which carries significant risk. It is impossible to reproduce the environment in which the robots will ultimately be deployed, as the final Urban Challenge is the first time in history that such an environment will be created.

Under such circumstances, comprehesive testing cannot be conducted, and it is necessary to find ways to compensate. While there is no substitute for thorough physical testing, a common and useful method of providing alternative testing of vehicle behavior is simulation [37].

Simulation allows an autonomous robot's developers to preview its behavior without touching the physical hardware, thus avoiding the time, money, and personnel costs of running actual integration tests. With these burdens alleviated, testing can become a pervasive activity in the development cycle. Furthermore, simulations can be modified to the developers' liking, and thus can display informative overlays, pause, alter the environment, and more, greatly aiding the debugging process. These bene-

fits come at the cost of accuracy, but a carefully-designed simulation can ensure that the errors are not crippling.

As useful as simulations are, they still require at least one developer present to observe and evaluate the robot's behavior. If the simulation is incapable of running faster than realtime (possibly due to limitations of the robot's AI), then it requires a significant time investment to run lengthy stress tests. Small development teams may be unable to spare a person just for running tests, or worse, a tester might find the tests too insufferably boring to bother watching. Either way, critical faults which could be caught by simulation are missed due to human error. This thesis attempts to address the issue by providing a means of automatically evaluating robot behavior, and selectively reporting interesting events and failures to the developers as they happen.

## 1.2 Contributions

This work builds upon the well-explored field of vehicle simulation to create a novel system capable of automatically evaluating and verifying autonomous vehicle behavior. It consists of two main components: a mid-level simulator capable of capturing the vehicle dynamics relevant to urban driving, and a testing framework capable of running entire simulated Urban Challenges and evaluating Dexter's performance in them.

The simulator is an original piece of software that can reproduce Dexter's behavior and interactions with several dozen other vehicles in realtime. It wholly replaces the physical vehicle and sensors while using the same AI, thus limiting simulation inaccuracies to failures of actual hardware or control and errors due to simplifications of vehicle dynamics or sensors. It also displays debugging information that can be used to spy on Dexter's current state of mind. The goal is to reduce the need for

physical vehicle testing and to facilitate development by making it easy to preview Dexter's behavior in meaningful environments using only simulation.

The testing framework augments the simulation with a novel scripting system capable of evaluating Dexter's conformance to test criteria, such as competition rules and performance measures. This system is used to implement a number of specific functional tests that check Dexter against the DARPA technical evaluation criteria [2]. It also features robust intelligent agents that can navigate the simulated environment, making it possible to run several-hour-long virtual competitions and evaluate Dexter's stability in extended randomized tests. The framework automatically runs both functional and randomized scenarios as regression tests [8], and produces a webpage detailing the results, helping developers identify and correct failures in vehicle behavior. With such simulation testing facilities in place, the net result is a much more reliable and robust autonomous vehicle requiring less development time and manual testing.

## 1.3 Environment

All software was written and tested primarily on a dual-core AMD Athlon 64 based 3.2 GHz PC with 2 GB of RAM, and all performance statistics listed in this work are based on that computer. However, the simulator has been tested on a large variety of computers, ranging from laptops used in field tests to the PXI controllers used within Dexter.

The work in this thesis covers approximately nine months of development from October 2006 through June 2007, during which time both the simulation / testing framework and Dexter's AI were developed in parallel. The two projects bear the mark of each others' influence, and while this thesis focuses exclusively on the simulation and testing framework, knowledge of Dexter's architecture [17] might prove

useful in understanding it.

For scale, Dexsim itself is currently 28,193 lines of C++ code, although 13,268 of these are an autogenerated C-to-Lua language bridge generated by SWIG [15]. The testing framework consists of 24 virtual instruments written in the National Instruments LabVIEW environment, a 450 line Lua [26] script containing shared testing code, and approximately 400 lines of test-specific Lua code spread over 11 tests.

## 1.4 Organization

This thesis is organized into seven chapters:

**Chapter 1** contains an introduction to the simulation and regression testing system, and a description of its purpose and motivation.

**Chapter 2** gives relevant background on the 2007 DARPA Urban Challenge, describing the previous Desert Challenges, introducing Team Case, and giving a tour of their vehicle, Dexter. It also contains a history of vehicle simulation and testing, providing context for the work in this thesis.

**Chapter 3** details the design and implementation of the simulator and dissects every aspect of its workings, such as map file processing, communication with Dexter's AI, and physical simulation.

**Chapter 4** describes the architecture and implementation of the intelligent autonomous agents which inhabit the simulations and serve as traffic for Dexter in extended tests.

**Chapter 5** examines the evaluation system used to automatically grade Dexter's performance in regression tests. It details the scripting system, the runtime system, the criteria used in evaluation, and the regression tests themselves.

**Chapter 6** discusses the results and impact of the testing system and analyzes the system's impact upon Dexter's development.

**Chapter 7** draws conclusions about the utility and limitations of the simulator and regression testing system, and outlines several directions for future work.

# Chapter 2

# Background

The most cherished promise of the age of technology is that, through the aid of electronic devices and automated machines, our lives will be made easier and safer. Even as far back as the 1920's, futurists and scientists alike were anticipating a time when dishes clean themselves, planes fly without pilots, and wars are fought without the loss of human life. With the pervasive cultural influence of science fiction and a stream of encouraging scientific breakthroughs, the notion of a robot as an everyday substitute for humans in dangerous or menial roles became a powerful and popular theme.

Sadly, the scientific and engineering communities have yet to deliver on this promise. As of 2007, the most useful robots available to the common man can do little more than trundle around, vacuuming dust and not falling down stairs. Despite an excellent body of research to build upon and growing availability of affordable sensors and actuators, robots have yet to demonstrate human-level competence in any free-form task.

## 2.1 The Grand Challenge

With this discrepancy in mind, the Defense Advanced Research Projects Agency (DARPA), a branch of the US Department of Defense charged with driving technological development and based in Arlington VA, began its so-called Grand Challenge to stimulate robotics research. The following text is taken from DARPA's website [1]:

---

The National Defense Authorization Act for Fiscal Year 2001, Public Law 106-398, Congress mandated in Section 220 that "It shall be a goal of the Armed Forces to achieve the fielding of unmanned, remotely controlled technology such that, by 2015, one-third of the operational ground combat vehicles are unmanned." DARPA conducts the Grand Challenge program in support of this Congressional mandate. Every "dull, dirty, or dangerous" task that can be carried out using a machine instead of a human protects our warfighters and allows valuable human resources to be used more effectively.

---

The format is a series of competitions, open to any team led by a US citizen, to build and race car-sized autonomous vehicles in a variety of settings. The first competition, which took place in 2004, was the "Desert Challenge," in which vehicles were required to follow a 130-mile track through the Nevada desert while avoiding impassable terrain and static obstacles without any aid whatsoever from humans. No vehicle managed to make it past the 8-mile mark, so DARPA re-issued the challenge the following year. That time, five robots completed the course, with the Stanford Racing Team's Stanley (Figure 2.1) taking the $2 million first prize [16].

With the success of the second Desert Challenge, DARPA decided to raise the stakes by conducting the "Urban Challenge" in 2007: a roughly sixty-mile race through an urban environment in which all vehicles would interact with each other in traffic. Requirements include the ability to follow lanes with unreliable GPS, smoothly avoid both static and dynamic obstacles, and to obey traffic laws such as intersection precendence and proper u-turn procedures. With an estimated 20 teams contending, the final Urban Challenge event is the first time in history that so many autonomous


Figure 2.1: Stanford's winning entry in 2005 Desert Challenge, Stanley [16]. Image courtesy of the Stanford Racing Team.

vehicles will be interacting together. It is set to take place on November 3rd, 2007 at a currently undisclosed location in the American southwest.

## 2.2 Team Case and Dexter

Under the leadership of Prof. Wyatt Newman and Prof. Roger Quinn, Case Western Reserve University is fielding an Urban Challenge entry under the name Team Case. The team is comprised primarily of graduate students and a handful of undergraduates from the Electrical Engineering and Computer Science (EECS) and Mechanical Engineering departments, working with a robot inherited from the 6th place finishers in the 2005 Desert Challenge, Team ENSCO (Figure 2.2). The robot, named Dexter, is built on a custom dune buggy chassis conspicuously lacking a driver's seat, and is outfitted with an array of computers and sensors.

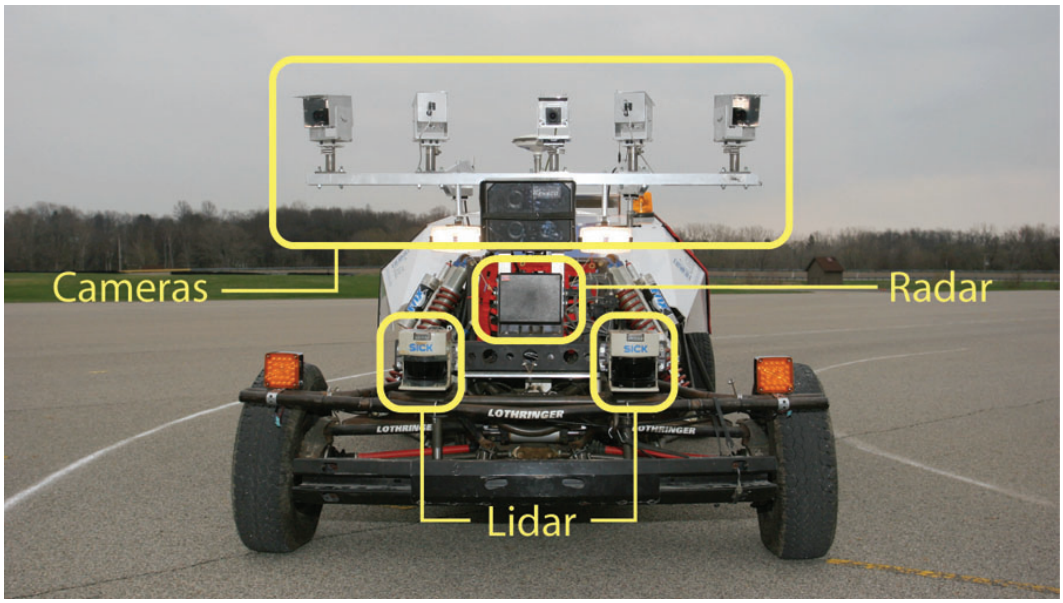
Dexter is equipped with a sensor load that has emerged as the defacto standard in autonomous driving: global positioning system (GPS) receivers, an inertial mea-


Figure 2.2: Dexter running in the 2005 Desert Challenge, under Team ENSCO.

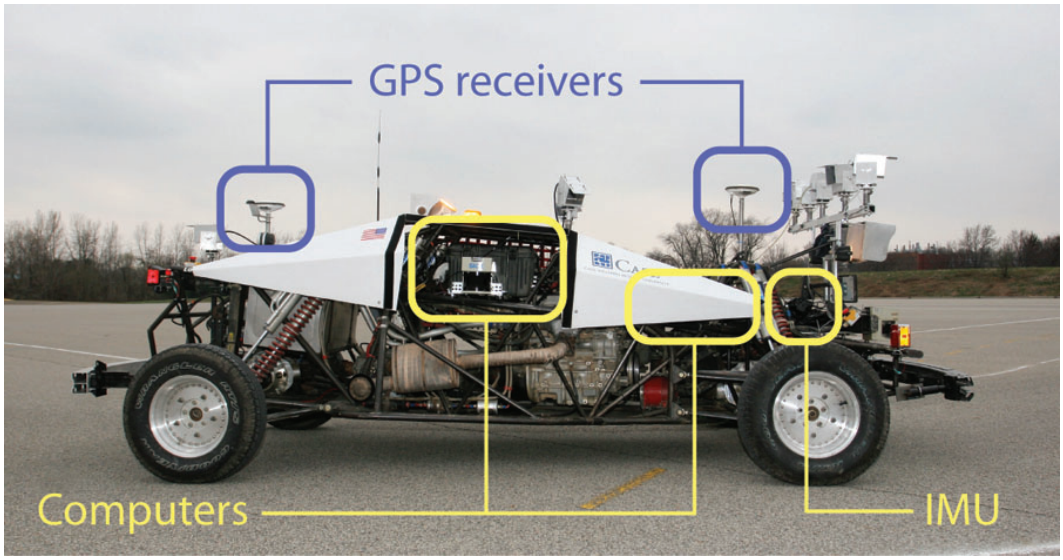
surement unit (IMU), laser range scanners (lidar), cameras, and radar. Differential GPS serviced by Omnistar provides vehicle position information to about 10 cm accuracy, which serves as the foundation for estimating the vehicle's physical state. The IMU works in conjunction with GPS by providing accurate acceleration measurements which are integrated into the position data with a finely tuned Kalman filter. This combination of position sensors works particularly well for smoothing noisy GPS signals, or even providing decent state estimations when GPS is lost entirely.

Dexter's primary environment sensing is provided by several lidars mounted on all sides of the vehicle. They read an accurate cross-section of the world surrounding the robot, and are used as the basis for obstacle detection. Cameras provide a wide variety of useful data, including but not limited to lane line detection, obstacle confimation, and stop line detection. Radar provides long-range detection of vehicles directly ahead. The configuration of sensors on the vehicle can be seen in Figure 2.3.

The AI systems for Dexter are programmed almost entirely in the National Instruments LabVIEW environment, chosen for its solid hardware-software interface support, excellent library, and ease of use. When mandated by efficiency concerns, some




(a) Front view




(b) Side view

Figure 2.3: Front and side views of Dexter's sensor and computer arrangement.

portions of the code (such as image processing) were ported to C++ and integrated into the system. The various modules of the AI form a reconfigurable distributed system, which can be deployed in various patterns across a network of National Instruments PXI and Apple Mac Mini computers.

As of this writing (June 2007), Dexter has successfully completed the DARPA site visit at the Transportation Research Center in Columbus OH, and is pending approval for admission to the next stage of the competition, the National Qualification Event (NQE). If Dexter is admitted into the NQE and passes, then Team Case will proceed to the Urban Challenge Final Event (UFE), and compete with the other teams on November 3rd.

## 2.3 Vehicle Simulation and Testing Background

Vehicles are among the best-studied subjects in the field of simulation. The earliest works describing simulated traffic appeared in the mid-1950s, and the quality and scope of the simulations has been increasing steadily ever since.

The first vehicle simulations were created to understand why traffic congestion happens, and treated cars as if they were flows on an abstract graph of roads. Early research (such as [20] and [21]) discovered three things: traffic congestion was directly rooted in the theoretical properties of network flow and resource sharing problems, simulation was indeed useful in exploring where congestion was likely to occur and how it could be mitigated, and the size and detail of the simulation is most often limited by the available computing power. Since then, researchers have been experimenting with different tradeoffs between size and detail, depending of the intended goals of the simulation. In 1963, for example, Jesse Katz simulated traffic for a 324-road network in Washington D.C. by discretizing the roads into cells and treating each cell as a queue of vehicles [28]. As computing power became more plentiful, the tradeoffs

became so marked that the field split into two classes of traffic simulations: macro and micro.

Macro simulations emphasize scale at the expense of detail. The general idea is to capture traffic patterns for the largest possible networks of roads, using whatever simplifications possible while maintaining roughly human-like driving behavior. To enable larger simulations, the most common tactic is to abstract away the behavior and dynamics of individual vehicles and use less expensive models. Several different techniques have been used, such as treating traffic as network flows [14], probability distributions [43], clustered particles [10], or a compressible fluid [35]. Work also has focused on making the simulation as separable, and thus parallelizable, as possible, to allow supercomputers to perform large city simulations [10]. In the most extreme example, the researchers in [36] were able to perform a traffic simulation for the entire country of Switzerland.

Micro simulations, on the other hand, emphasize detail, and usually attempt to predict, analyze, and optimize the traffic in a local area. This is most commonly done by implementing a multi-agent simulation with each car having separate AI, and many models have been proposed to make the vehicles behave as realistically as possible. One early attempt [29] focused on intersection behavior, running an elaborate model through simulations and comparing the results to an instrumented car at a real intersection. Another work simulated the traffic for the city of Garland, Texas, and derived a set of optimal signal timings to help improve the safety and speed of driving around town [40]. Several projects (such as [13] and [34]) have given the agents "personalities", and analyzed the resulting impact of overly aggressive or cautious drivers on traffic flow. General traffic simulators have been created for both highway [5] and urban environments ([25] and [37]).

Traffic has not been the only research direction for vehicle simulation. Another trend in recent years has been the creation of very high-detail vehicle dynamics simula-

tions, which are designed to replicate various aspects of vehicle behavior as accurately as possible. At the lowest level, many simulations have been created to explore the efficiency of power distribution for various types of conventional, fuel cell, battery, and hybrid-powered vehicles. These simulations, such as [4] and [32], model the tiniest details of engine dynamics and the resulting impacts on performance.

At a somewhat higher level, there has been great interest in creating simulators that allow humans to drive a virtual automobile. Much effort has been expended to derive appropriate dynamics models (see [23] and [42]), which have then been embedded into commercial software packages such as *SCANeR II* [33] and video games such as the *Gran Turismo* series [12]. A popular addition to such a dynamics simulation is a vehicle mockup that the user can sit in, complete with screen, sound, and force feedback via a Stewart platform. Several examples are documented in [11], [24], and [30]; commercial products doing the same can be seen at theme parks and auto shows all over the world.

Simulations have long been used for vehicle testing, but no matter how good the model is, it will never quite predict the actual behavior of a vehicle's hardware [31]. This is normally handled by taking the vehicle out for test drives after the development team feels confident that enough simulation has been done (see [38]), but some researchers have managed to blend hardware and simulation together into a type of test called "vehicle-hardware-in-the-loop." In [4], for example, the researchers use the hardware that controls the vehicle throttle and brake alongside a vehicle simulation, handling communication through encoders. Other projects, such as [7] and [41], have taken the next step and mounted real or scaled model vehicles on rollers in a lab. Tests are then run on the actual vehicle hardware, but without the hazards of a moving or uncontrolled environment.

# Chapter 3

# Design of the Simulator

The simulator created for the Urban Challenge project is a vehicle dynamics and micro traffic simulator called Dexsim (Figure 3.1). It was written from scratch in C++ and designed for the Windows operating system. The language and environment were primarily chosen for speed, familiarity, the widespread availability of free, useful libraries, and ease of integration with Dexter's AI. The software is capable of hosting only one virtual Dexter at a time, but can simulate dozens of traffic vehicles at the same time and still maintain a 60 Hz update rate [9]. It was developed in parallel with Dexter's AI, and functionality was added incrementally as Dexter grew. Several different components work in concert to make the simulation, and are described loosely in order of their construction.

The Urban Challenge places fewer demands on the hardware of the vehicle than the previous Desert Challenge. The target course will be a (possibly artificial) urban environment somewhere in the American southwest, meaning that the terrain will likely be rather flat, and DARPA rules cap the vehicles at a maximum speed limit of 30 MPH [2]. As a result, it is very unlikely that a vehicle will ever encounter a situation where it would leave the ground, roll excessively, drive on steep hills, or have to avoid large ditches. Accordingly, the simulator was designed to work in only two

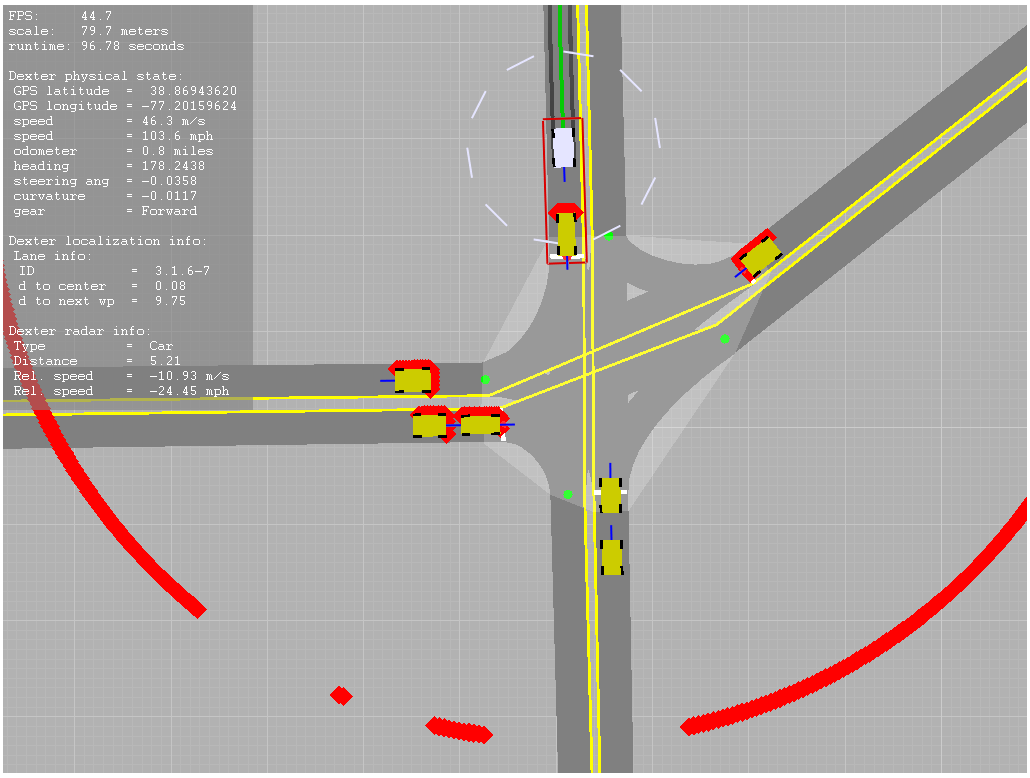



Figure 3.1: A typical scene from Dexsim. The white vehicle at top is Dexter, and the yellow vehicles are simulated traffic agents. The red markings are simulated lidar readings from a sensor mounted on the front of Dexter. The grid in the background is adaptive and scales as the user zooms in and out; in this scene, large squares are 10m on a side, and small squares are 1m wide. The roads are drawn with lane markings, and the transparent polygon represents the intersection area.

dimensions, estimating the third if necessary by treating the ground as a flat plane upon which all vehicles and obstacles are placed.

## 3.1 Software architecture

Dexsim was designed to be as architecturally simple as possible, and its code reflects this motivation. The highest-level aspects of its design can be summarized by examining its core class structure and logical flow.

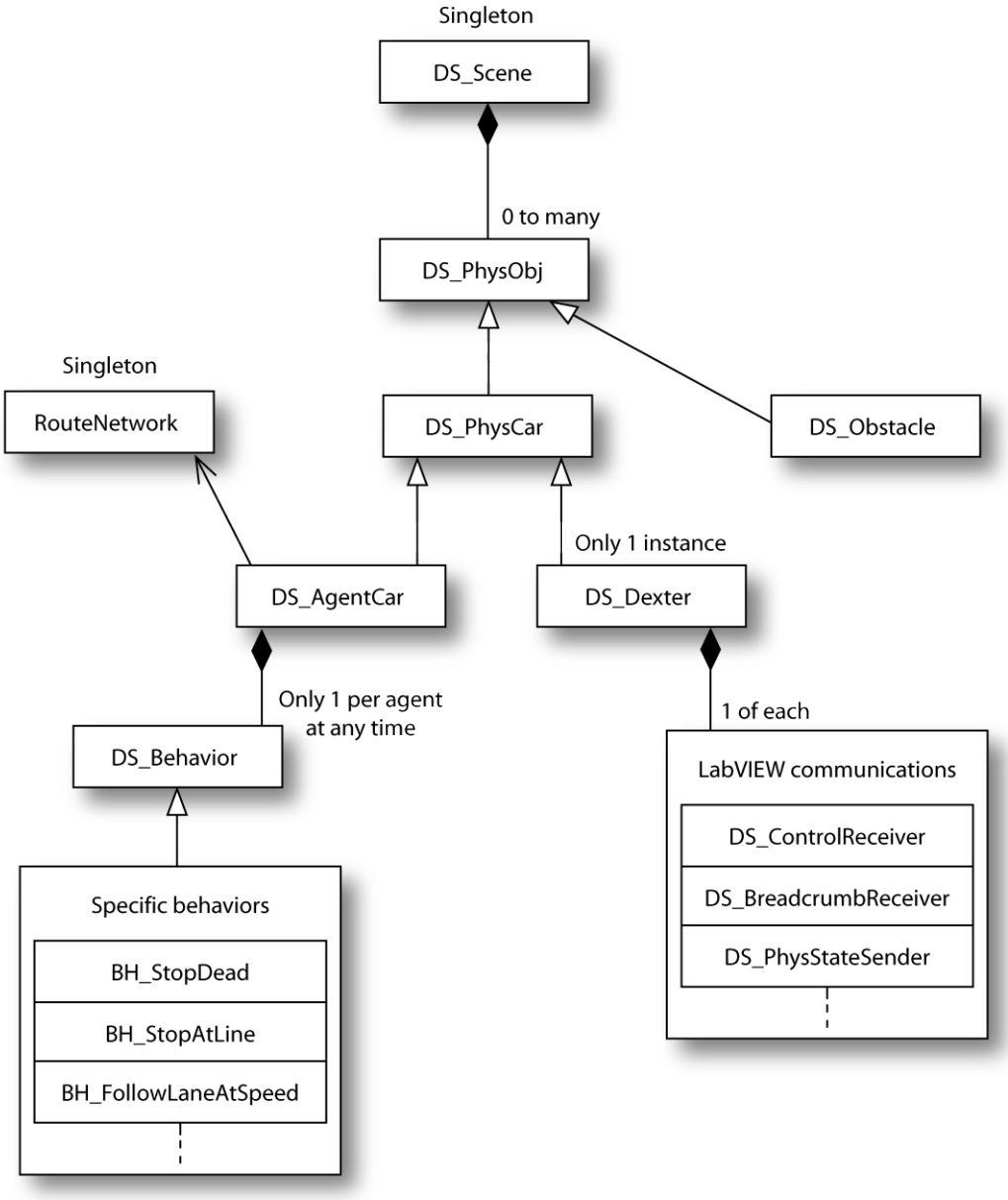



Figure 3.2: A class diagram showing the relationships between the primary components of Dexsim.

### 3.1.1 Class structure

Dexsim was designed in an object-oriented fashion, and a class diagram of the components relevant to simulation can be seen in Figure 3.2. Each of the classes has a role that will be described in subsequent sections of this thesis.

The `DS_Scene` class is a singleton instance of a quadtree, which serves as Dexsim's scene graph and is explained in Section 3.6. `DS_PhysObj` is simply a common base class for all objects that inhabit the simulation. The `DS_Obstacle` class represents static obstacles that Dexter may enounter in the environment (Section 5.1). `DS_PhysCar` encapsulates the vehicle dynamics for all vehicles in simulation, as described in Section 3.5. The `RouteNetwork` is Dexsim's internal representation of the map, containing roads, intersections, localization information, and such (Section 3.4).

Dexter itself is contained in the `DS_Dexter` class, which is no more than a simple interpreter for sending and receiving information with the robot's AI (Section 3.2) and drawing debugging information (Section 3.8). The `DS_AgentCar` class encapsulates the traffic agent AI (Chapter 4), and each agent has one currently executing behavior (Section 4.4).

### 3.1.2 Program flow

The flow of program control in Dexsim is documented in Figure 3.3. The program begins by loading and processing the map file (Section 3.4), then loads the environment if it exists (Section 5.1).

Next, the communication handlers for passing data back and forth between Dexsim and Dexter's AI are spun off into separate threads. These threads will wait for updated data from either side, and perform the necessary movement of information. Mutexes are used to ensure exclusive access to data shared between threads.

After the initialization phase, the program proceeds to the main simulation loop. The first step is to detect collisions between objects in the scene graph, which is

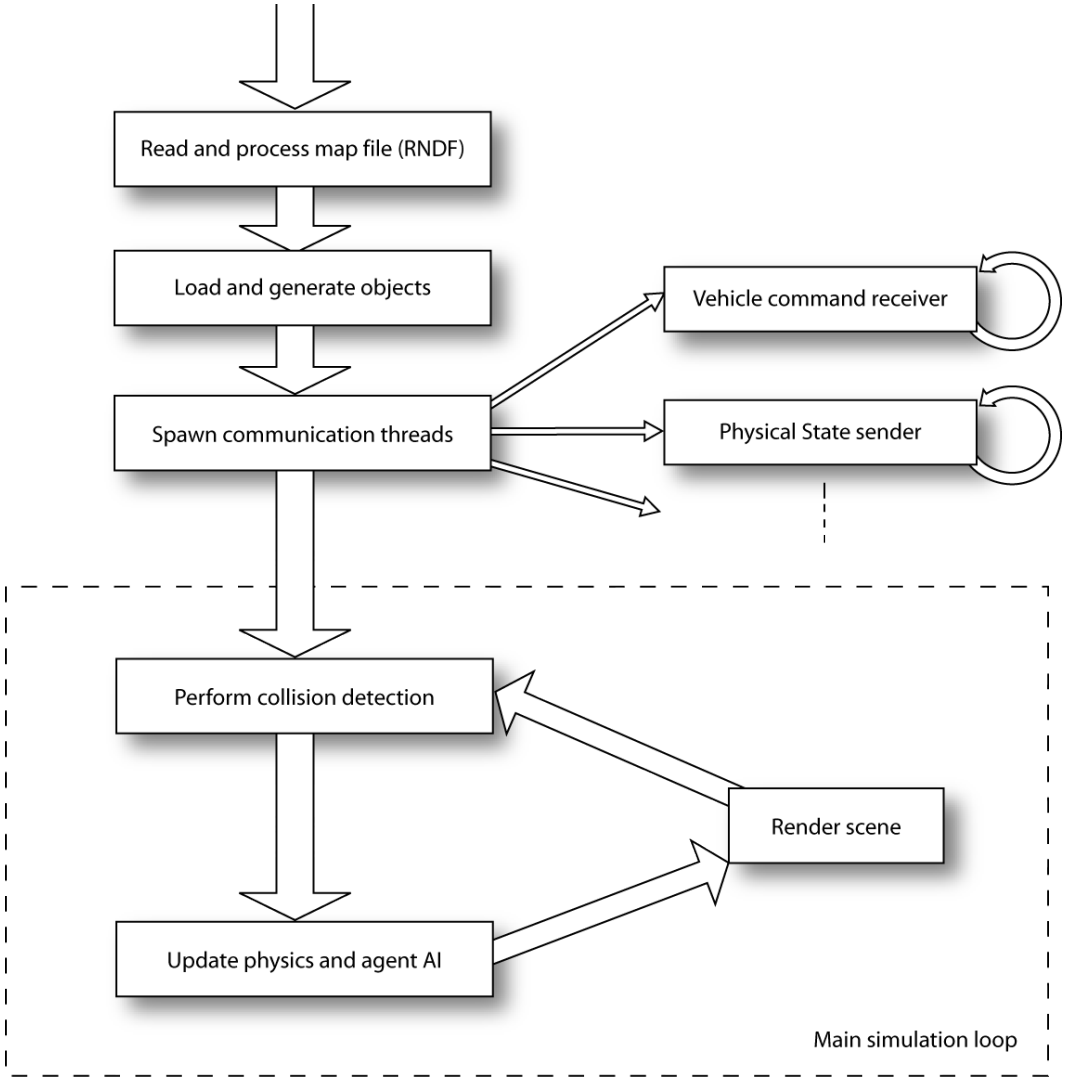



Figure 3.3: Flow of program control. Dexsim starts by loading the map and object files, starting asynchronous communication with LabVIEW, then beginning the main simulation loop. The loop iteratively computes collisions, updates vehicle AI and physics, and renders the scene graph.

described in Section 3.6. The next step is to call the update routine for all objects, which performs such operations as AI processing for agents and physics computations for all vehicles (Chapter 4 and Section 3.5). The final step is to render the scene for the user (Section 3.8). The steps are performed in this particular order to avoid situations where one object's update invalidates data maintained by other objects (such as collision information).

Dexsim iterates through the main simulation loop at 60 Hz, or as fast as possible given the available computing power. The program is terminated either by user request or test control options.

## 3.2 Interface with Dexter

In order to make the simulation as accurate as possible, the virtual Dexter receives commands from the same AI that controls the physical vehicle, and attempts to provide the same sensor information available on the real platform. In other words, the simulator replaces the environment in Dexter's control loop, which includes the vehicle, its sensors, the map, and all the other objects populating the world, such as static obstacles or rival vehicles (see Figure 3.4).

The means of communication between Dexsim and Dexter's AI is called the DataSocket Server, which is a program included by National Instruments in the standard distribution of LabVIEW [27]. It serves as a generic asynchronous data blackboard that can be accessed over a network, and is already used for Dexter's internal communication. A writer can send any block of data to the server with a name attached, and when received, it will overwrite any previous data under that name and persist until overwritten by the next such update. A reader sends a name request to the server, and will receive the most recent version of the data available. Writers must be exclusive, but any number of readers can connect to the same server.

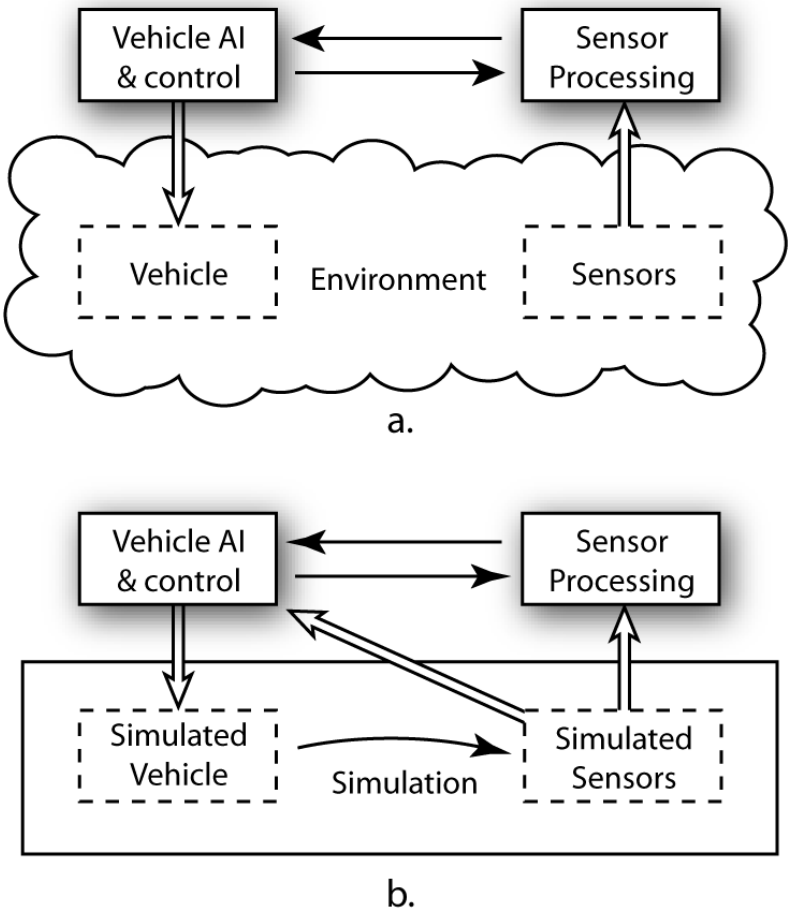



Figure 3.4: High-level information flow in the actual vehicle (a), and in simulation (b). The "Vehicle AI & control" and "Sensor processing" blocks are identical in both cases, aside from how communication is handled.

Dexsim simply taps into this existing service to communicate with Dexter's AI.

The input to Dexter's AI is large amounts of data about the environment harvested from the vehicle's suite of sensors as described in Section 2.2. On the physical vehicle, this raw information is received by several layers of "Observers," which process the data and convert it into forms useful for making decisions. The simulator mimics this arragement, feeding its virtual sensor data into mostly the same Observers. Some sensors, such as cameras, are impossible to simulate realistically, and thus more processed forms of information are delivered directly to Observers further

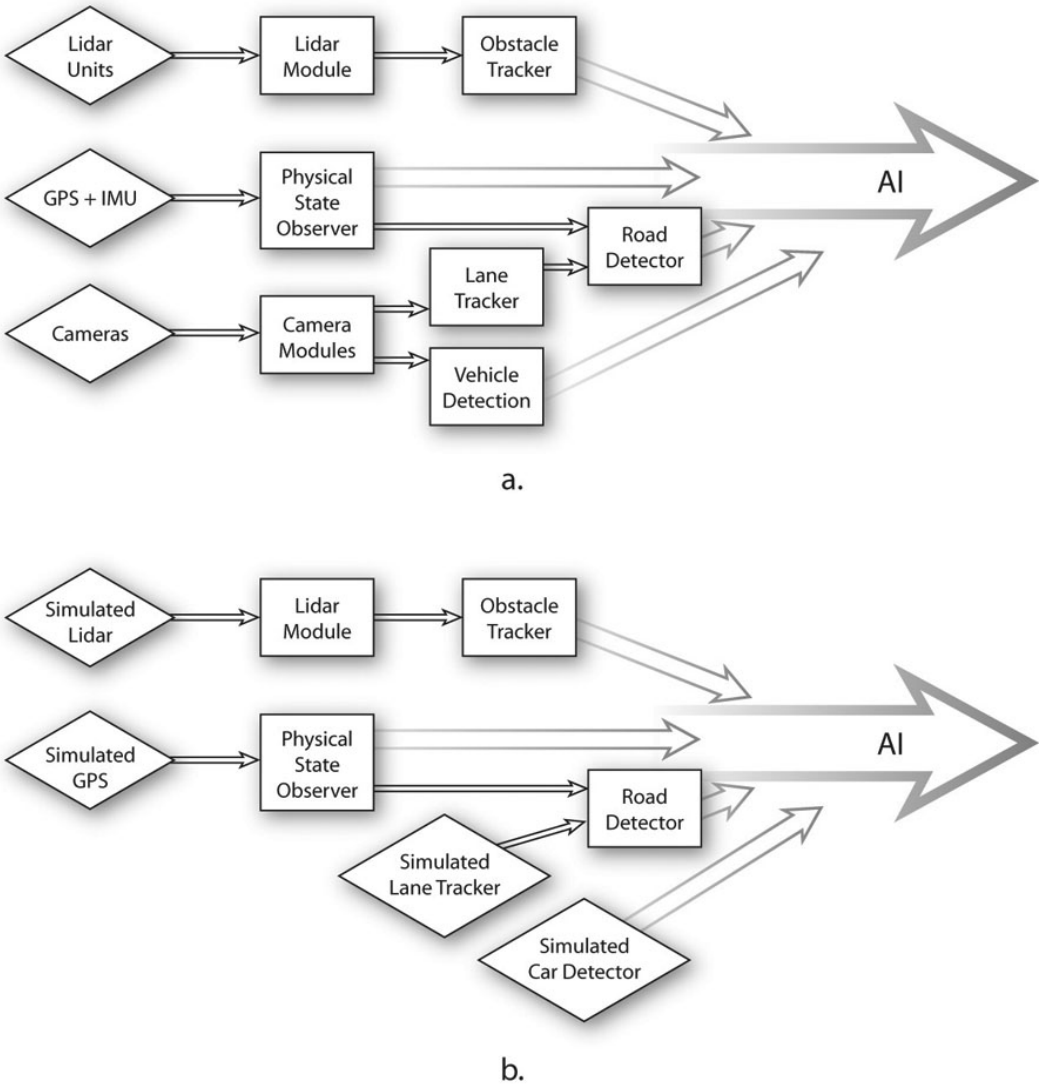



Figure 3.5: Input data from the actual vehicle (a), and in simulation (b). Rectangular blocks are sensor processing elements in Dexter's AI, and diamonds are either real sensors on Dexter or simulated sensors from Dexsim.

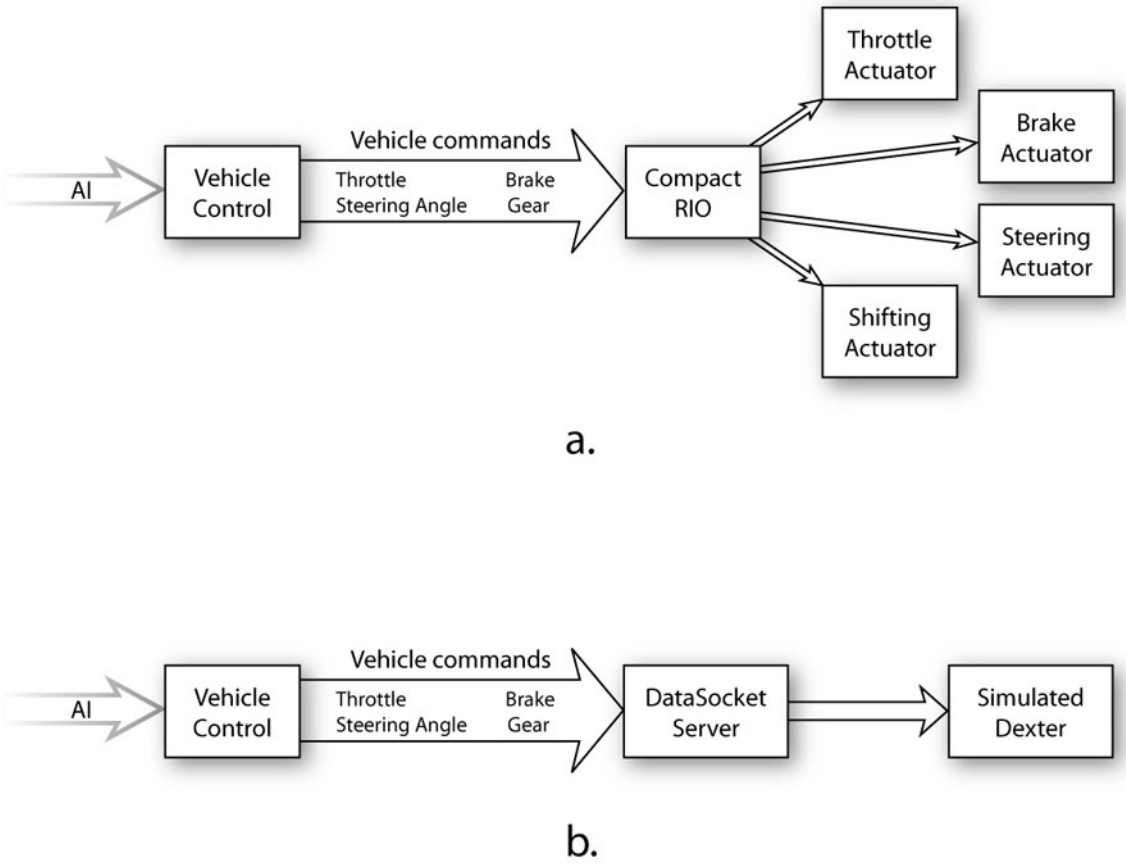



Figure 3.6: Output data from the actual vehicle (a), and in simulation (b).

down the processing chain (Figure 3.5). Note that as of this writing, Dexter does not use cameras for any processing or decision making; thus, the input diagram represents a potential design for the flow of camera information, and not necessarily the final implementation.

The output of Dexter's AI is a set of vehicle commands: throttle, brake, steering angle, and gear. These are published by the Vehicle Controller, the lowest level of Dexter's AI and the component responsible for directing the vehicle to drive a given path. On the physical vehicle, these values are fed into a Compact RIO controller, which issues the raw commands to the engine, brake pump, steering actuator, and shifting actuator. In simulation, these commands are sent to the DataSocket server, intercepted by the simulator, and used to drive the virtual vehicle instead (Figure 3.6).

## 3.3 Coordinate Transformation

The coordinate system used by both Dexter and DARPA is the standard WGS84 Global Positioning System (GPS), which specifies locations as pairs of latitude and longitude. This system is ideal for large-scale location finding on the globe, but is somewhat less useful for simulation, as dynamics formulae are typically expressed in a Cartesian space with an orthonormal basis. Thus, it is necessary to define a new coordinate frame for the simulator to use, and provide conversions to and from GPS.

All simulation takes place in a two-dimensional cartesian plane, using meters as units, which we will refer to as the "simulation world frame" or "sim frame" for short. It is defined by treating the earth as a sphere and picking a GPS point on the surface, which we'll call the "plane origin", then placing a plane tangent to it. The origin of the sim frame is the point at which the plane touches the earth, with the $x$ axis pointing east and the $y$ axis pointing north (see Figure 3.7).

Conversion from GPS to the sim frame is accomplished by locating the target

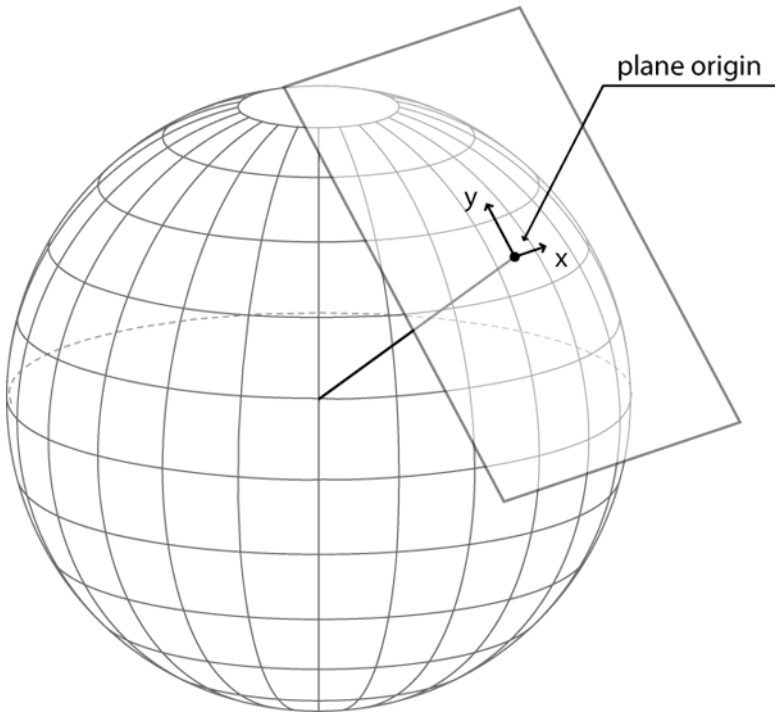



Figure 3.7: An example GPS conversion plane

GPS point on a unit sphere, projecting it onto the tangent plane, and scaling by the radius of the earth (6,378,137 meters according to WGS84). Conversion from the sim frame back to GPS is accomplished by the inverse operation, that is, dividing by the earth's radius, projecting the point from the plane onto the unit sphere, and computing its spherical coordinates.

As is bound to happen when approximating a sphere by using a plane, there are inaccuracies in the conversion. In particular, as distance away from the plane origin increases, there is a foreshortening effect due to the projection onto the plane, such that the ratio of plane distance to arc length is $\sin\theta/\theta$, where $\theta$ is the angle between the plane origin and the GPS point undergoing conversion (see Figure 3.8). This means that the conversion system can only work for half the world at a time, but anything less than 0.2422 radians (roughly 1544 km) away from the plane origin suffers less than 1% error. Fortunately, Urban Challenge maps are very unlikely to be more than a 10 kilometers across, meaning that by setting the plane origin as the

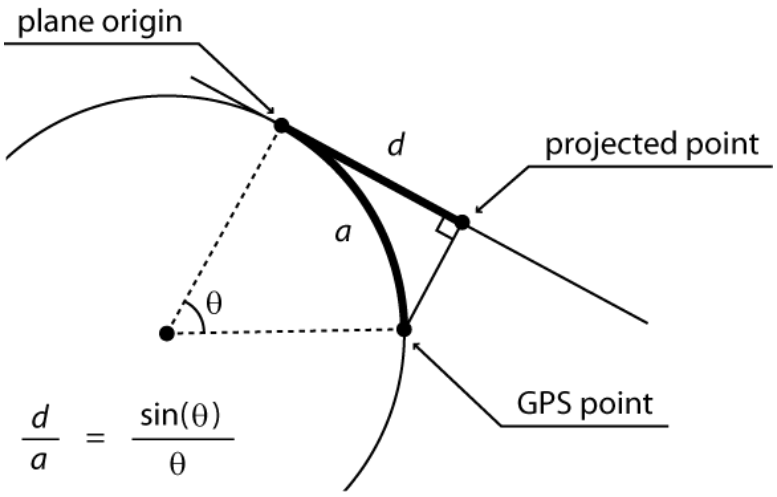



Figure 3.8: A demonstration of the foreshortening effect caused by projecting GPS coordinates onto a tangent plane.

center of the map, conversion errors will be less than $5.1 \times 10^{-4}$ meters.

Another potential source of error is the assumption that the earth is a sphere. In reality, the earth is slightly elliptical, meaning that Cartesian coordinates are slightly warped. The earth's inverse flattening according to WGS84 is approximately 298.257, which means that the difference between its major and minor axes is only 0.335%, or about 21 km. The difference is small enough that it is unlikely to cause significant errors.

For angle representations, the GPS system uses compass heading, which measures out the unit circle with 360 degrees going clockwise, with 0 degrees pointing north. The simulator internally uses an engineering representation with radians as units, and a counter-clockwise progression with 0 pointing east. Conversions are provided to and from GPS headings.

Once a plane origin has been chosen, all computations within the simulator are performed with respect to the sim frame, thus keeping units, measures, and angles consistent. Only when it is strictly necessary, such as for communicating with Dexter's AI or storing data to a file, is data converted back to GPS.

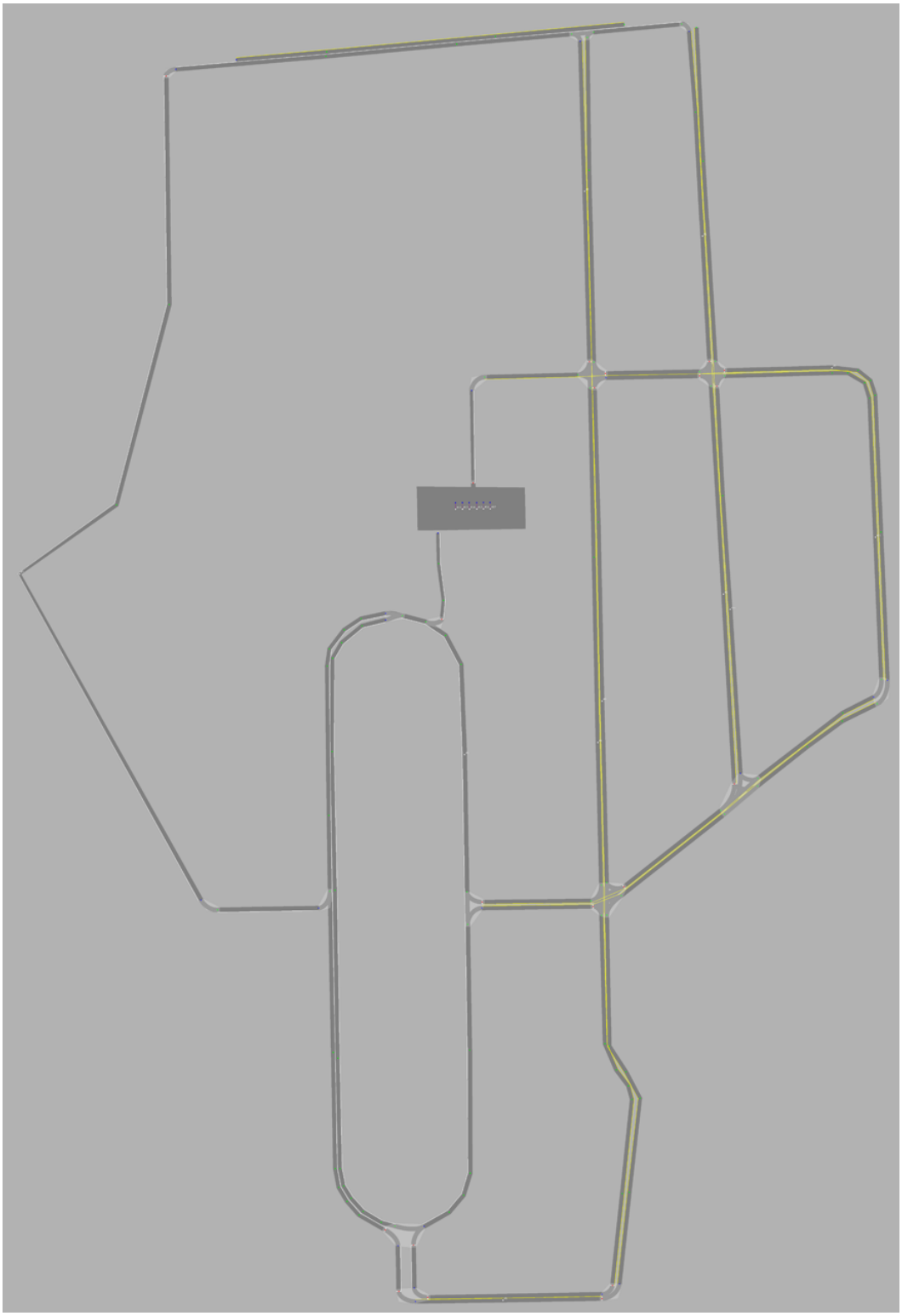


Figure 3.9: A rendering of the sample RNDF provided by DARPA.

```
segment 13
        num_lanes       2
        segment_name    Virginia_Rd
        lane    13.1
                num_waypoints   11
                lane_width      12
                left_boundary   double_yellow
                checkpoint      13.1.6  3
                stop    13.1.7
                stop    13.1.9
                exit    13.1.7  4.2.5
                exit    13.1.7  4.1.5
                exit    13.1.9  3.2.11
                exit    13.1.9  3.1.4
                exit    13.1.11 12.1.1
                13.1.1  38.870844       -77.198884
                13.1.2  38.871935       -77.198945
                13.1.3  38.872943       -77.199001
                13.1.4  38.873051       -77.199045
                13.1.5  38.873125       -77.199164
                13.1.6  38.873146       -77.199409
                13.1.7  38.873125       -77.200427
                13.1.8  38.873116       -77.200662
                13.1.9  38.873103       -77.201554
                13.1.10 38.873093       -77.201816
                13.1.11 38.873076       -77.202697
        end_lane
        lane    13.2
                num_waypoints   9
                lane_width      12
                left_boundary   double_yellow
                checkpoint      13.2.8  11
                stop    13.2.2
                exit    13.2.2  4.1.5
                exit    13.2.2  4.2.5
                exit    13.2.9  10.1.1
                13.2.1  38.873072       -77.201548
                13.2.2  38.873085       -77.200661
                13.2.3  38.873085       -77.200425
                13.2.4  38.873107       -77.199406
                13.2.5  38.873107       -77.199260
                13.2.6  38.873034       -77.199100
                13.2.7  38.872921       -77.199054
                13.2.8  38.871897       -77.198998
                13.2.9  38.870849       -77.198941
        end_lane
end_segment
```

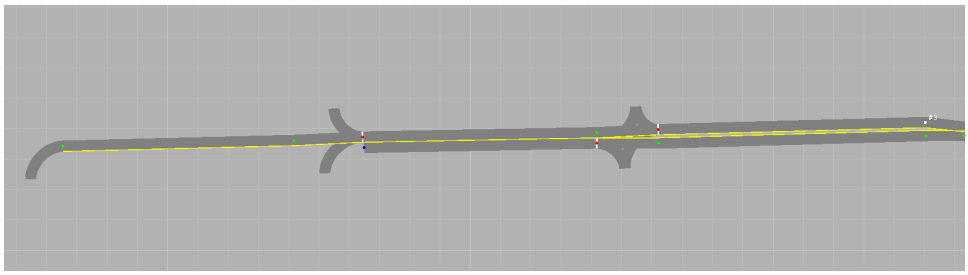


Figure 3.10: Text description and rendering of an RNDF road segment.

```
zone    14
num_spots       6
zone_name       Central_Parking_Lot
perimeter       14.0
num_perimeterpoints     6
exit    14.0.5  11.1.1
14.0.1  38.872271       -77.203339
14.0.2  38.872258       -77.202804
14.0.3  38.872264       -77.202315
14.0.4  38.871959       -77.202309
14.0.5  38.871948       -77.203136
14.0.6  38.871947       -77.203331
end_perimeter
spot    14.1
spot_width      16
checkpoint      14.1.2  12
14.1.1  38.872151       -77.202972
14.1.2  38.872103       -77.202971
end_spot
spot    14.2
spot_width      16
checkpoint      14.2.2  13
14.2.1  38.872152       -77.202907
14.2.2  38.872104       -77.202906
end_spot
spot    14.3
spot_width      16
checkpoint      14.3.2  14
14.3.1  38.872152       -77.202843
14.3.2  38.872104       -77.202840
end_spot
spot    14.4
spot_width      16
checkpoint      14.4.2  15
14.4.1  38.872152       -77.202772
14.4.2  38.872105       -77.202770
end_spot
spot    14.5
spot_width      16
checkpoint      14.5.2  16
14.5.1  38.872153       -77.202708
14.5.2  38.872106       -77.202707
end_spot
spot    14.6
spot_width      16
checkpoint      14.6.2  17
14.6.1  38.872153       -77.202645
14.6.2  38.872106       -77.202643
end_spot
end_zone
```

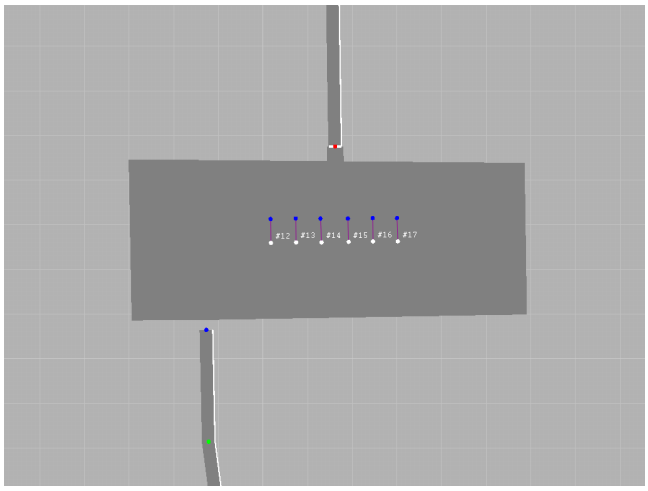



Figure 3.11: Text description and rendering of an RNDF zone.

## 3.4 Map Data

For each competition location, DARPA provides a Route Network Definition File (RNDF), a structured text file specifying the "map" that vehicles use for navigation [2]. It consists of GPS waypoints annotated with route connectivity information, as well as outlines of parking lots (called zones) and designations of available parking locations. It also contains a number of annotations designed to affect the vehicle's behavior, such as marks at the locations of stop signs, descriptions of lane line markings, and numbered "checkpoints" which are used to specify routes. These RNDFs are used as the base maps for simulation.

By and large, the RNDFs specified by DARPA are rather crude, requiring significant processing before they can be useful. For example, waypoints are sparsely spaced and may not accurately represent a drivable road surface when linearly interpolated. There is no explicit information in the file to specify intersections, which play a pivotal role in vehicle behavior, and parking lots are specified as nonconvex polygons, which must be tesselated before they can be useful. Dexsim uses the RNDF to create its internal representation of the roads, but the data must be processed in several ways before it is usable.

### 3.4.1 Parsing and Annotation

The RNDF is a simple tab-delimited text file, making parsing simple (see Figures 3.10 and 3.11). Once the data structure has been filled out, it is made easier to navigate by adding previous and next pointers to lanes and waypoints, lists of pointers to entrances and exits of each waypoint, and indexing structures for checkpoints. It is then converted into the sim frame using the techniques described in section 3.3, and travel directions are derived for each waypoint by normalizing the line connecting the waypoints immediately before and after (Figure 3.12).

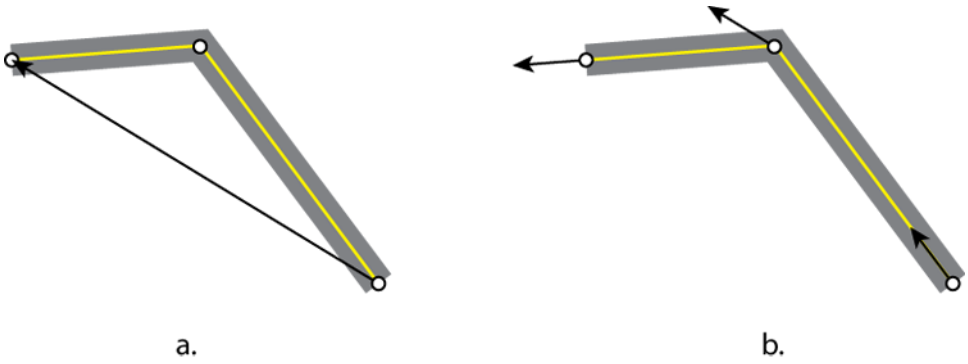



Figure 3.12: Waypoint direction computation. The direction for the middle waypoint is the normalized difference between the waypoints immediately before and after (a). The final directions are seen in (b)

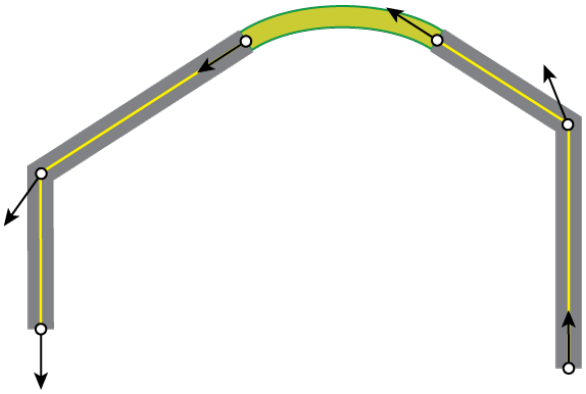


Figure 3.13: An example of a minimal acceleration Hermite spline connecting a gap between lanes.

The original C++ RNDF parser used in Dexsim was contributed by Nathan Wedge, and enhanced with the computations described in the remainder of this section.

### 3.4.2 Connector Generation

Every lane in the RNDF is specified by a sequence of waypoints, but turns between lanes are specified only by lane exit and lane entrance waypoints, with no predefined paths. Thus, it is necessary for the simulator to generate its own drivable connections between the waypoints. Cubic splines were an obvious choice due to their smoothness

```
initialize label to 0
initialize marks to -1
for each waypoint w ∈ W
  if marks[w] == -1 and w is in a connector
    RecursiveMark(w, label)
    label += 1

RecursiveMark(w, l):
  if marks[w] != -1
    return
  marks[w] = l
  foreach en ∈ (entrances[w])
    RecursiveMark(en, l)
  foreach ex ∈ (exits[w])
    RecursiveMark(ex, l)
  foreach wp ∈ (waypoints in lanes adjacent to w)
    if (wp is in a connector and distance(w, wp) < 20 meters)
      RecursiveMark(wp, l)
```

Figure 3.14: The algorithm used to detect intersections in the RNDF from connectivity information.

properties, and specifically minimum-acceleration Hermite splines [6] were chosen as they produce remarkably natural driving paths and are easily created from a pair of points and tangents (Figure 3.13). Unfortunately, the length of the spline must be found by expensive numerical integration, so the value is cached when first requested.

### 3.4.3 Intersection Generation

As described before, the RNDF specifies nothing about intersections other than their connectivity information. In order to be able to identify which waypoints and connectors are part of the same intersection, a recursive marking algorithm is used to identify related sets of exits.

The algorithm iterates over all waypoints having connectors attached, and clusters them according to the following two criteria:

- Waypoints attached to each other with connectors are in the same cluster.

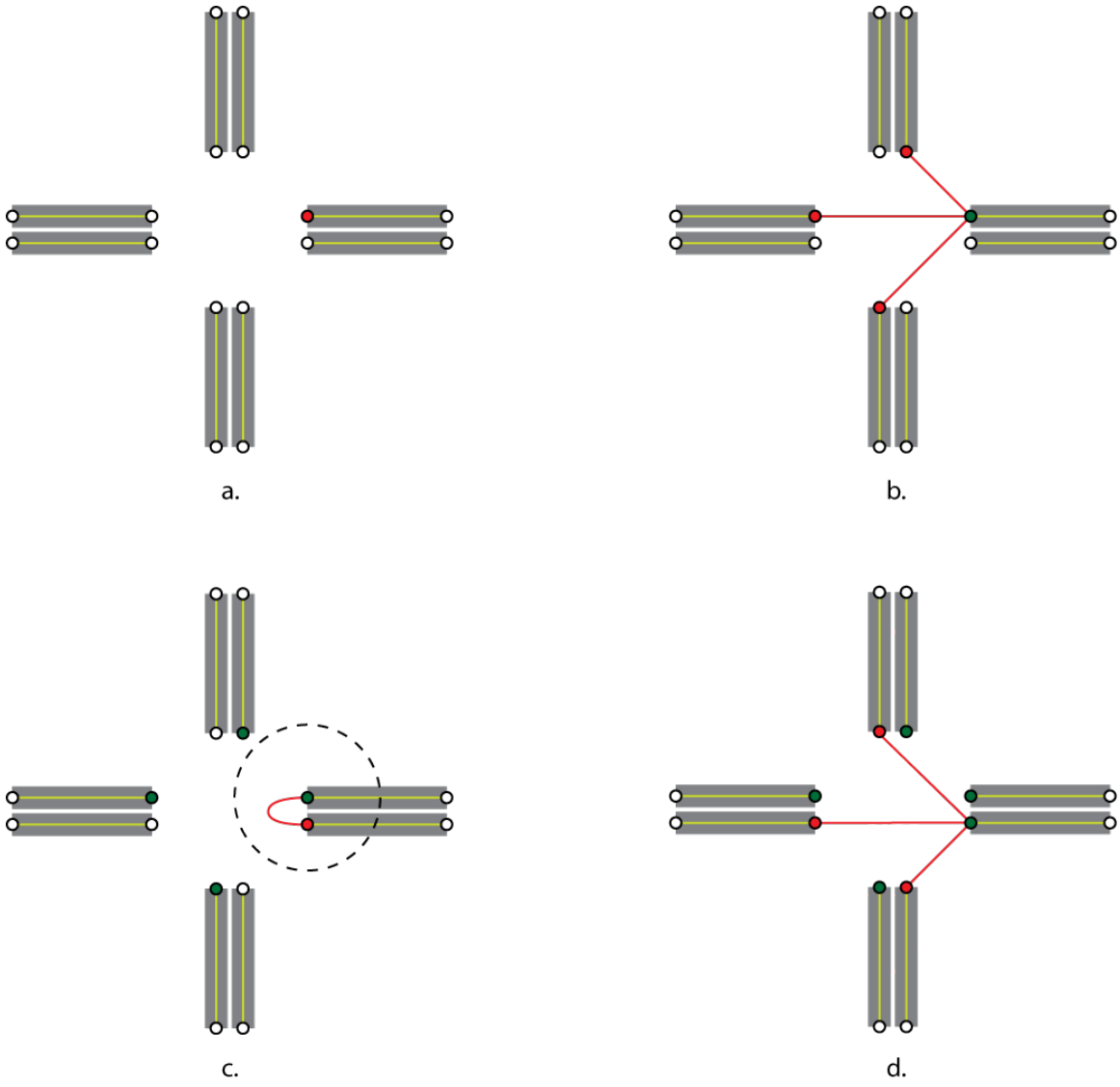



Figure 3.15: The marking algorithm for detecting waypoints in the same intersection. To begin, (a) shows a four-way intersection with one waypoint marked, and (b) shows the expanding of that waypoint by following its exits. Next, (c) shows expanding a waypoint by marking waypoints in adjacent lanes. Finally, (d) shows marking the entrances to a waypoint; at this point, all intersection points are marked.

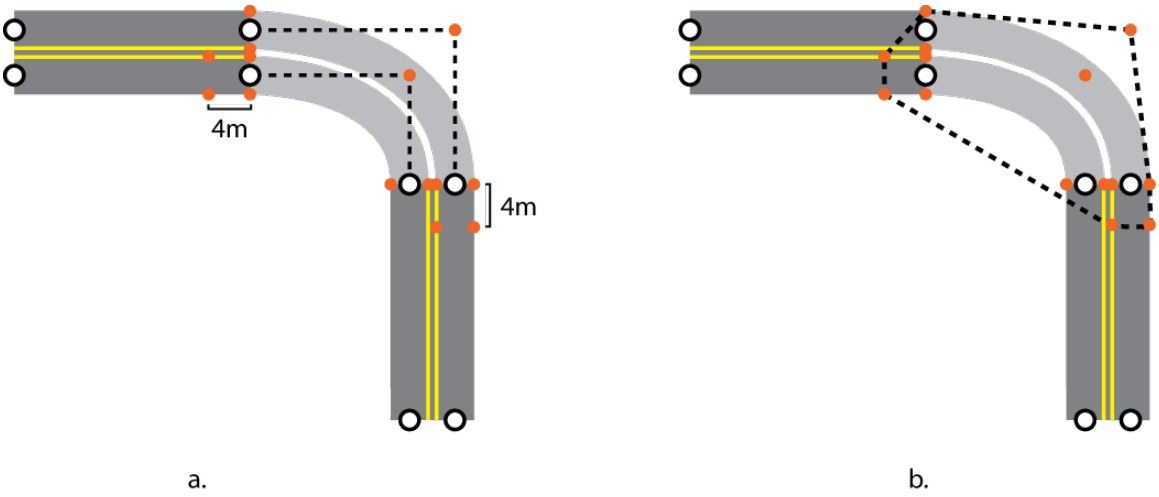



Figure 3.16: Intersection hull generation. Points of interest are shown in (a), and the convex hull is shown in (b).

- Waypoints within 20 meters of each other that are in the same segment but different lanes are in the same cluster.

The recursive algorithm is seen in Figure 3.14, and can be visualized in Figure 3.15.

Once the waypoints in each intersection have been identified, a convex polygon is created to encompass it. This is done by picking a number of points of interest in the intersection, namely the edges of lanes leading into and out of intersections, the edges of lanes 4 meters ahead of the intersection, and what would be the intersections of lanes if they continued straight into the intersection. Once the points are identified their convex hull is computed, generating a polygon that contains the intersection and portions of all lanes leading into it, which can then be used for AI and localization functions. The entire process can be see in Figure 3.16.

### 3.4.4 Localization Information

One of the most important functions of the map in simulation is localization, which determines which roads, intersections, or zones a car is currently occupying. Such information is extremely useful when evaluating Dexter's performance, or writing

intelligent agents to drive around the map. The procedures involved vary depending on what particular feature of the map is being localized against.

For localizing against roads, the simulator creates a database of all the road segments in the map during initialization, then does a simple brute-force distance comparison at runtime for each localization query. The localized road segment is simply the closest one found. (Note that it is possible to accelerate these queries with a spatial data structure, but they were not found to be a bottleneck in the application's performance.) Localization for lanes fails if the car is not within the bounds of its closest road segment, but information about the nearest lane is returned nonetheless for utility purposes.

Localization queries against intersections are accelerated by creating bounding circles for each convex hull. At runtime, cars are tested against each intersection's circle, and if they collide, the full collision test is run against the convex hull. The car is considered localized against an intersection if it is in contact with that intersection's hull. Localization for zones is handled in a very similar fashion, except the car in question is compared with each of the convex subregions of the zone.

### 3.4.5 Lane Change Information

To identify where lane changes are possible, the simulator will identify straight segments of lanes and loop over all waypoints in those segments, attempting to localize points 30 meters down the road and 4 meters to either side. If one of those points happens to land in an adjacent lane traveling in the parallel direction, and there is only a dashed white lane line separating the two lanes, then a lane change is possible, and the internal map representation is updated with this new connectivity information.

## 3.5 Physical Simulation

The heart of the simulator is its vehicle dynamics engine, which updates the physical state of all cars in the simulation given the steering angle, gear, throttle, and brake commands from their vehicle controllers. It is the component primarily responsible for the character and realism of the simulation, and was carefully designed to capture the aspects of vehicle behavior most relevant to driving with traffic in an urban environment.

As described before, the Urban Challenge will likely take place on a relatively flat course, and DARPA caps the maximum speed limit at 30 MPH. Under such mild conditions, most of the finer details of vehicle dynamics, such as the stiffness of the shocks, the grip of the tires, and the height of the center of gravity, can all be safely ignored. At those speeds, a vehicle's motion is governed primarily by the kinematic constraints imposed by its wheels, the rate it can change its steering angle, and its acceleration and braking power.

Accordingly, vehicles in simulation are treated as two-dimensional rigid rectangles of uniform mass with motion subject to bicycle constraints to keep the tires from slipping. More specifically, at each incremental update of the physics simulation, the point in the middle of the vehicle's rear axle is constrained to move parallel to the rear wheels, and the point in the middle of the front axle is constrained to move parallel to the front wheels. The parameterization of each vehicle can be seen in Figure 3.17.

To enforce the bicycle constraints, we modify the equations of motion from [19] to ignore vehicle roll and slip calculations, and use simple Euler integration to compute the vehicle's motion.

Besides the bicycle constraints, there are a few other dynamics effects that are relevant to the vehicle's large-scale behavior. In particular, the simulator imposes limits on every vehicle's steering angle and its rate of change, to match the range and responsiveness of actual steering systems. Shifting is also not instantaneous, so the

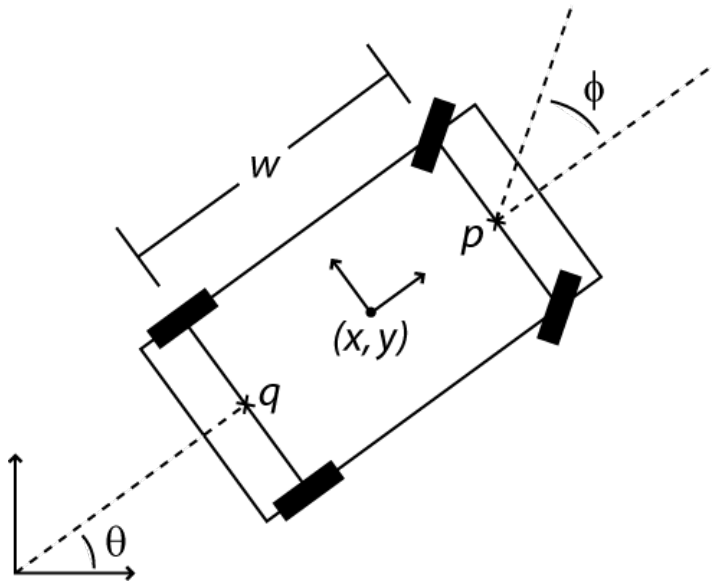



Figure 3.17: Parameterization of vehicle dynamics. $(x, y)$ is the center of mass, $\theta$ is the current heading, $\phi$ is the current steering angle, and $w$ is the length of the wheelbase. Point $p$ is constrained to move parallel to the front wheels and point $q$ is constrained to move parallel to the rear wheels.

vehicle is held in place for a specified time delay when changing gears between drive, reverse, and park.

Throttle and braking forces are read from the vehicle controller and fed into a first-order delay which mimics the responsiveness of the real vehicle. The time constant of the delay was measured from logs of performance tests in parking lots. The rolling friction in the model is a rough linear approximation, made by applying a resistive force to Dexter's motion proportional to the vehicle's velocity.

Vehicle parameters for Dexter such as mass, wheelbase, length, width, shifting delay, and steering rate were measured directly from the vehicle itself, and throttle and braking forces were estimated using tuning constants from Dexter's vehicle controller. They can be seen in Table 3.1. Vehicle parameters for other vehicles, such as AI agents, are randomly generated within reasonable ranges.

The simulator attempts to run at a constant 60 Hz, thus keeping integration error low and user interaction responsive. The update procedure described here is efficient

| | |
|---|---|
| **Mass** | 3200 kg |
| **Length** | 4.064 m |
| **Width** | 2.096 m |
| **Wheelbase** | 3.048 m |
| **Steering limit** | ±26 degrees |
| **Steering rate** | 35 degrees / sec |
| **Shifting delay** | 1.5 sec |
| **Throttle $\tau$** | 0.7 sec |
| **Max throttle** | 15000 N |
| **Max brake** | 15000 N |
| **Rolling friction** | 0.015 |

Table 3.1: Dexter's measured physical properties

enough that, on the test machine, it can be run for over two hundred vehicles at once without breaking the target framerate.

## 3.6 Collision Detection

A physical simulation is of little use unless one is able to make geometric queries against the environment, such as detecting collisions between cars or casting rays from simulated laser scanners. Unfortunately, brute force collision detection alone is an $O(n^2)$ problem with $n$ being the number of objects, and with hundreds of cars and objects potentially populating the simulation, a more efficient scheme is necessary to ensure acceptable performance.

The simulator implements its scene graph using a quadtree, which recursively subdivides a two-dimensional space into a hierarchical tree of squares of decreasing size [18]. The entire map is first bounded in a square, which is subdivided into four equal parts. Each of these squares is subdivided into four equal parts as well, and the recursion continues until a predetermined minimum square size is reached (see Figure 3.18a). While numerous spatial data structures are available in the literature [22], the quadtree was chosed due to the simplicity of its implementation.

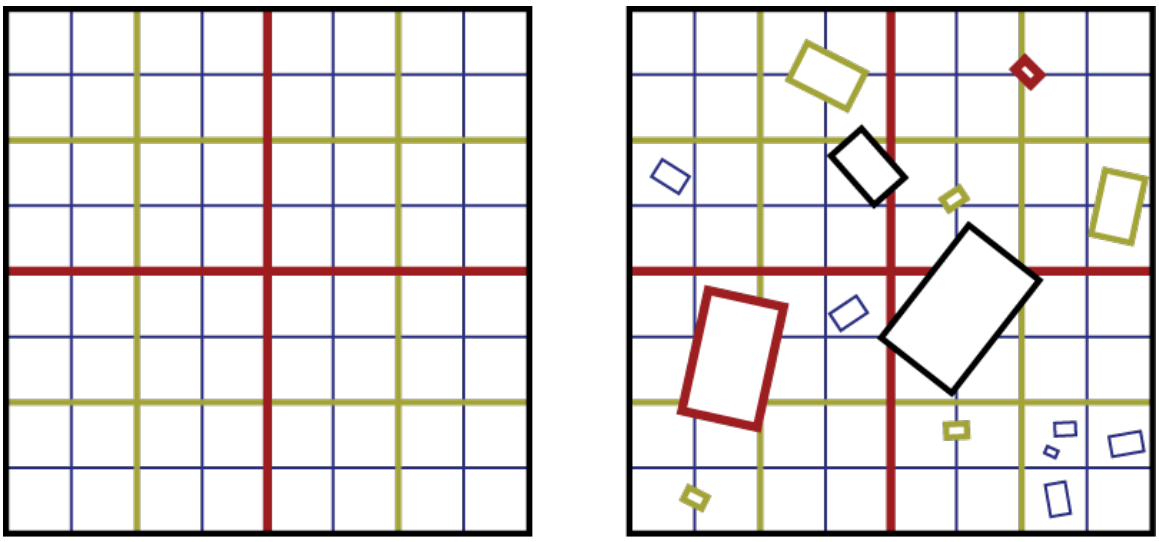



Figure 3.18: Visualizations of a quadtree. In an empty space (a), the black square is first subdivided into four parts by the red lines. Then each of those parts are subdivided by the yellow lines, and those once again by the blue lines. Each color represents a different level of the quadtree. The objects in (b) are color-coded by the level of the tree at which they are inserted.

Objects are inserted into the quadtree at the lowest possible level. If an object falls entirely within one of the smallest nodes, then it is inserted at a leaf. If it happens to straddle the boundaries of two nodes, then it is inserted one level higher. As objects move around the simulation, they are shuffled up and down the nodes of the tree to keep them at the appropriate level (see Figure 3.18b). Given a random distribution of objects and a well-chosen minimum node size, most of the objects will end up in the leaves of the tree.

When performing collision detection, objects are tested only against objects in the same node and any of its parents and children. This means that most objects will be able to avoid comparisons with most other objects, bringing the average case time complexity for collision detection down to $O(n\ lg\ n)$.

The quadtree can also be used to accelerate ray casting queries by only comparing rays against objects nodes that the ray will penetrate. Within each node, the object

comparisons in the closest node are done first, to avoid exploring farther nodes in the event that a collision in found.

## 3.7 Sensor Simulation

When simulating an autonomous vehicle, vehicle dynamics and collision detection are only half of the picture. We must also simulate the input to the vehicle's AI, namely, the sensor data. This means estimating the input that all of Dexter's sensors (see Section 2.2) would be receiving from the simulated environment and making it look as much as possible like data recorded from a real environment.

Useful sensor simulation is governed by the processing that the autonomous vehicle is actually capable of performing, and unfortunately, at the time of this writing, the only sensors Dexter uses for decision making are GPS, IMU, and lidar. As such, many potential simulation aspects of sensors were not implemented; however, the intended designs for these simulations are documented here for reference.

### 3.7.1 GPS and IMU

Under ideal conditions, the combination of high-precision GPS and an IMU give a near-perfect estimation of the vehicle's physical state. Thus, for basic purposes, it is entirely sufficient to export the vehicle's exact state in the simulation, which is then used in Dexter's AI in the same fashion that actual GPS readings are. This is the scheme currently used.

In less than perfect conditions, the sensors are vulnerable to a number of different errors. The GPS receiver requires a clear signal from at least four satellites before it can converge to a reliable solution, and any interference or occluders (both of which are common in urban environments) can cause the GPS signal to be lost. This causes the receiver to report erratic guesses of its location, which can wreak havoc on the

vehicle's position estimation.

The IMU is intended to fill in these dropouts by accumulating acceleration measurements, however, integration errors cause the readings to deviate from the true vehicle position gradually over time. In addition, once GPS is reacquired, the position estimation has a tendancy to "jump" back to an accurate reading, leading to a discontinuity in the vehicle's path as interpreted by the sensors.

In such situations, Dexter must rely on information from his cameras and lidars, such as lane line estimations and obstacle tracking data, to reconcile inaccurate readings from the GPS and IMU. However, Dexter does not yet have such navigation facilities in place and is only capable of assuming that the GPS information received is perfect. Thus, simulated GPS dropouts are currently meaningless tests, and have not been implemented.

### 3.7.2 Lidar

Laser scanners can be simulated by providing virtual mount points on Dexter, casting rays from them in a 180 degree sweep, and returning the distance at which they collide with something solid (Figure 3.19). This information is packaged and sent to Dexter's AI, where it is processed in the exact same fashion as lidar data from real sensors. This simulates ideal lidar readings, and is currently the extent of lidar simulation used.

Lidars work by measuring the round-trip times for beams of laser light emitted from the sensor, and are thus susceptible to any interference or physical phenomena that would prevent a beam from being measured accurately. Lidars can be blinded by light from bright sources such as headlights or the setting sun, and interpret the interference as total occlusion. Sharp incident angles may cause the beam to deflect at such an angle that it cannot be measured, and very dark objects may absorb the beam outright, both of which cause the lidar to erroneously report the absence of an

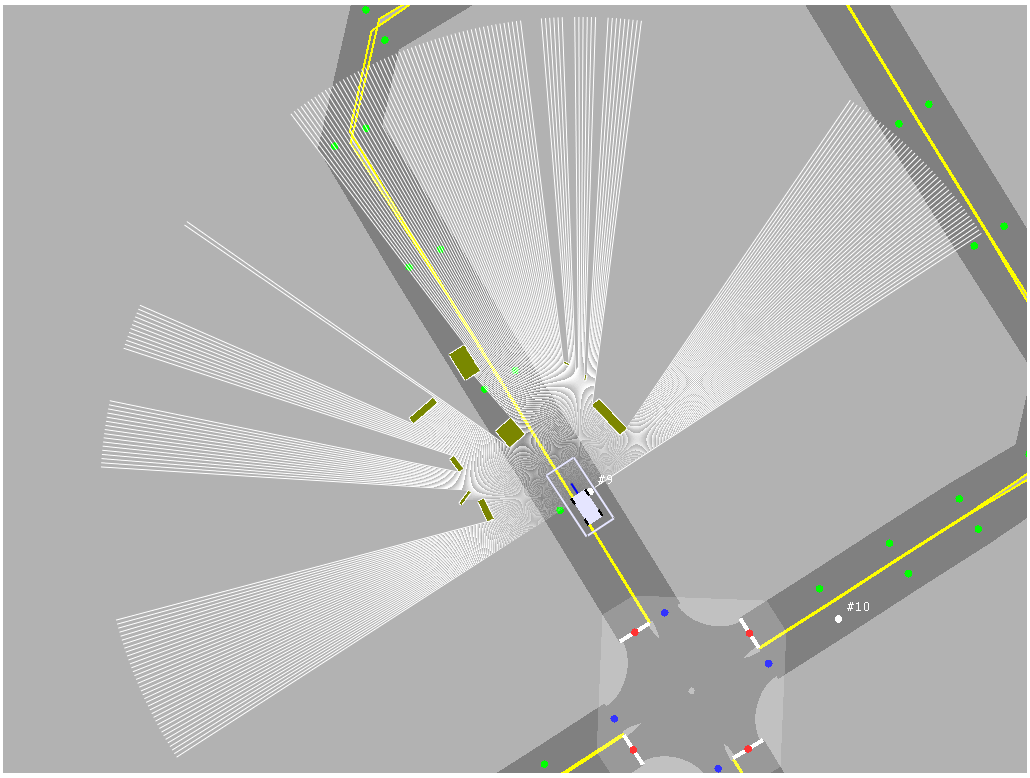


Figure 3.19: A screenshot showing lidar simulation. Rays are cast out in a 180 degree sweep from a virtual mount point on Dexter, and stop at the first object they encounter. The maximum distance is 60 meters, after which it is capped.

obstacle.

In such error cases, Dexter must use temporal information and camera data to account for mistakes made by the lidars. However, the AI currently does no such sensor fusion, rendering tests of lidar failures meaningless as well.

### 3.7.3 Cameras

While Dexter features a large array of cameras, they are not presently used for any decision making processes, meaning that Dexsim currently performs no camera simulation. However, the system as designed is documented here for reference.

Cameras are notoriously difficult to simulate, requiring an extremely complex rendering engine and excruciatingly detailed world data to create even roughly realistic

results. Thus, we avoid the brute-force approach and instead simulate the data that is harvested from cameras by image processing methods. In other words, we simulate the data that is typically extracted from camera images.

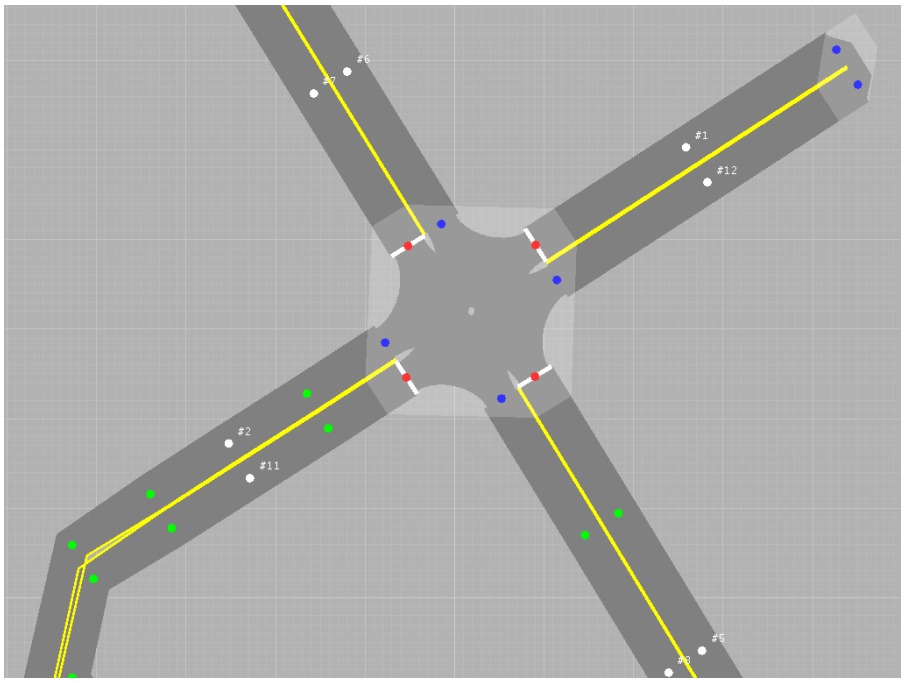
For example, cameras are commonly used to identify lane lines on the road, and subject captured images to several color classification and line fitting algorithms to extract the desired data. Instead of generating an image, the simulator would generate lane line information directly from its road description, feeding it straight to Dexter's AI and bypassing the camera processing step. This creates a hole in the simulation's accuracy and leaves plenty of room for the real vehicle to behave differently than tests would suggest, but given the impossibility of simulating full camera data, it is the next best option.

## 3.8 Rendering

Dexsim was designed as a development tool, and built to help the team creating Dexter's AI root out bugs from the system. As such, it features a number of debug overlays that visualize the salient points of Dexter's environment and current state of mind.

Roads from the RNDF are displayed naturally, with appropriate widths and lane markings. Waypoints are displayed as small circles, and are color coded to represent their properties. Checkpoints are labeled for easy identification. An adaptive grid is displayed in the background to provide reference for size, and optional satellite map underlays (obtained from image services such as TerraServer or Google Maps) can be added to get an idea of the real obstacles and topology of an area (Figure 3.20).

Dexter's GPS receiver (mounted between his front wheels) leaves a green position trail to visualize its recent position history, and tire tracks can be enabled to get precise measurements of his wheel placement. Dexter's immediate desired path (known

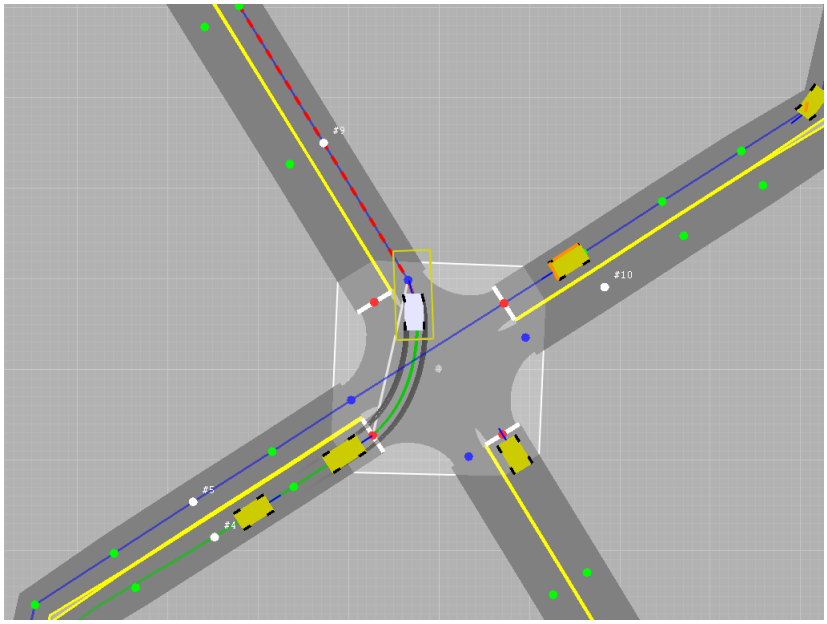



(a) Regular rendering

(b) Satellite map rendering

Figure 3.20: Regular and satellite views of map rendering. Waypoints are drawn as colored circles; green circles regular waypoints, reds are stop signs, blues are lane entrances / exits, and whites are checkpoints, labeled with their number. Stop lines are drawn across stop points. Each intersection's convex hull is drawn as a transparent polygon over the intersection.

(a) Dexter following a lane



(b) Dexter turning in an intersection

Figure 3.21: Views of Dexsim's debug overlays. The red dashed line is Dexter's breadcrumb trail, and the blue line shows waypoints in Dexter's route plan. The position trail is rendered in green, and tire tracks are black. The box surrounding Dexter represents the DARPA safety zone, and scales as the vehicle speeds up or slows down. The orange line segments represent detected obstacles from Dexter's obstacle tracker, which has been processing simulated lidar data in the same fashion that it would process actual lidar data.

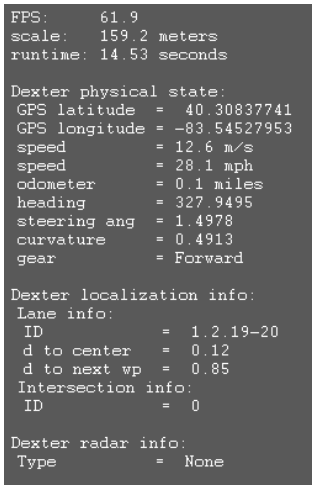



Figure 3.22: Dexsim's information overlay.

as the "breadcrumbs") is shown as a red dotted line, color coded by speed, and his route plan is overlaid as a blue chain of waypoints. The DARPA-specified safety zone is drawn as a bounding box, and a blue "whisker" denotes the current steering angle. Orange line segments denoting Dexter's obstacle tracking results are rendered on top of the actual objects (Figure 3.21).

There is also an information dialog available which displays crucial data about Dexter such as its position, heading, speed, odometer, localization, and the like (see Figure 3.22). When combined, these features give a clear representation of the most important aspects of the simulation, and facilitate debugging.

The rendering step is by far the slowest aspect of the simulator, and as it uses OpenGL for its graphics API, its speed is limited entirely by the underlying graphics hardware. Fill rate is often the culprit for slowdowns, and usually becomes a problem when the view is zoomed in on an intersection or zone. Rendering such scenes requires touching most of the pixels in the framebuffer, sometimes several times each. On the test machine, it is not uncommon for the framerate to drop to 30 frames per second in such cases, only to jump back to 60 frames per second upon zooming out. To mitigate these problems, graphics can be optionally disabled in the simulator, in which case

no test machine had any trouble keeping a 60 Hz update rate.

## 3.9 Live visualization

The development team found the debug information useful enough that a special version of the simulator called Dexviz was created. This version performs no simulation, instead receiving all physical state and sensor updates from the vehicle itself. Essentially, this allows one to spy on Dexter's state of mind over a wireless network connection while the real vehicle is running, using the same debugging tools as are available during simulation. This has been very beneficial during field tests, and helps highlight problem cases or sensor quirks that simulation may have missed.

# Chapter 4

# Simulated Autonomous Agents

For the simulator to be capable of testing anything beyond simple vehicle control, it is necessary to provide some traffic which can interact with Dexter. While it is impossible to predict what the behavior of other teams' vehicles will be (even if we had their full cooperation in the endeavor), we can make a reasonable approximation by populating the simulation with intelligent agents that obey the rules of the competition.

Creating such agents is technically as difficult as solving the Urban Challenge itself, but these agents are only designed to be used within simulation, and have several advantages over physical vehicles that make the job easier. In particular, the simulated agents have perfect sensor information, can ask arbitrarily detailed questions about the environment, and can communicate with each other to coordinate actions. In essence, the most difficult problems of the Challenge are solved by cheating. This is an infeasible strategy for designing real vehicle AI, but is entirely sufficient to create interesting test data.

Every agent is simulated as a virtual vehicle, using the same physics and control parameters as Dexter but with randomized vehicle parameters. They are each equipped with a variety of virtual sensors, many of which reflect the types of in-

formation available to real robots. The information from these sensors flows into a four-tiered control hierarchy, which is loosely inspired by the architecture of Dexter's AI [17]. The top tier, the *planner*, plots paths around the map to reach target checkpoints. On the next lower layer, the *state machine* is responsible for switching between behaviors as the vehicle drives around the map, and modulating allowable speeds to prevent collisions. The next tier, *behaviors*, turns low-level directives such as "drive in this lane" and "stop at this sign" into sequences of desired headings, positions, and speeds. The lowest tier, *vehicle control*, converts desired heading, position, and speed into the throttle, brake, and steering commands which directly drive the vehicle.

The agents described here are capable of fulfilling most of the requirements in the Basic Navigation and Basic Traffic sections of the DARPA Urban Challenge Technical Requirements [2], with the exceptions of performing u-turns and passing static obstacles. They are fully capable of queueing at stop signs and handling intersection precedence, and thus have the necessary features to test most of Dexter's basic traffic behavior.

## 4.1 Sensors

The sensor information available to the simulator's agents is perfect, detailed data about the environment that all agents agree upon. All virtual sensors operate on pools of information shared among all the agents, and thus queries will always be consistent between agents.

### 4.1.1 Localization

Each agent is provided with perfect localization information, and always knows which road, intersection, or parking lot it is currently occupying. This information is ex-

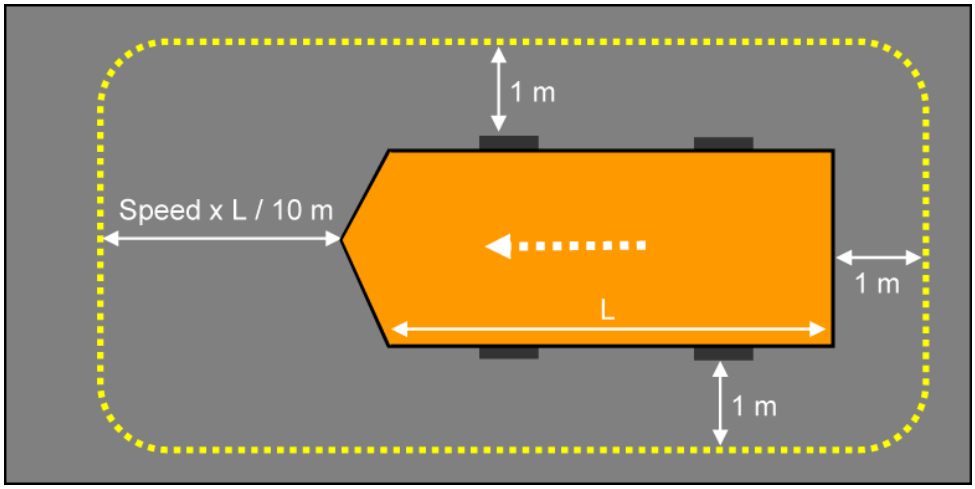



Figure 4.1: Diagram of a vehicle's DARPA safety zone

tracted from the vehicle's position and the map file, and uses the techniques described in Section 3.3.

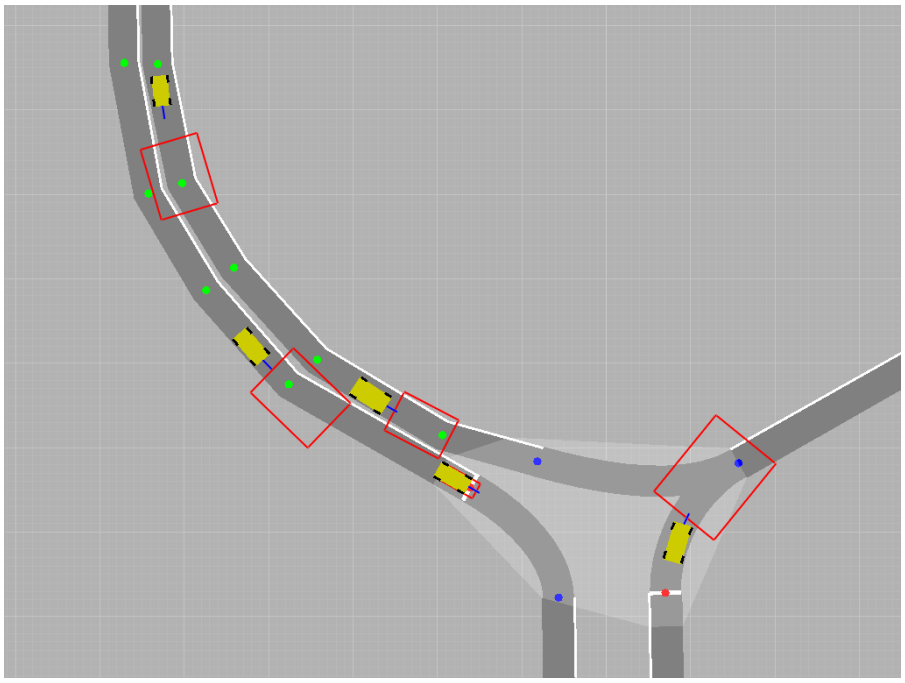
### 4.1.2 Safety zone

The "safety zone" is a rectangle specified by DARPA, surrounding each Urban Challenge competitor, which the vehicle is responsible for keeping clear. The zone extends from the back and sides of the vehicle by one meter, and extends from the front by one vehicle length per 10 mph of speed, with one vehicle length minimum (see Figure 4.1). An agent is alerted if any other object enters its safety zone, so it can take appropriate action.

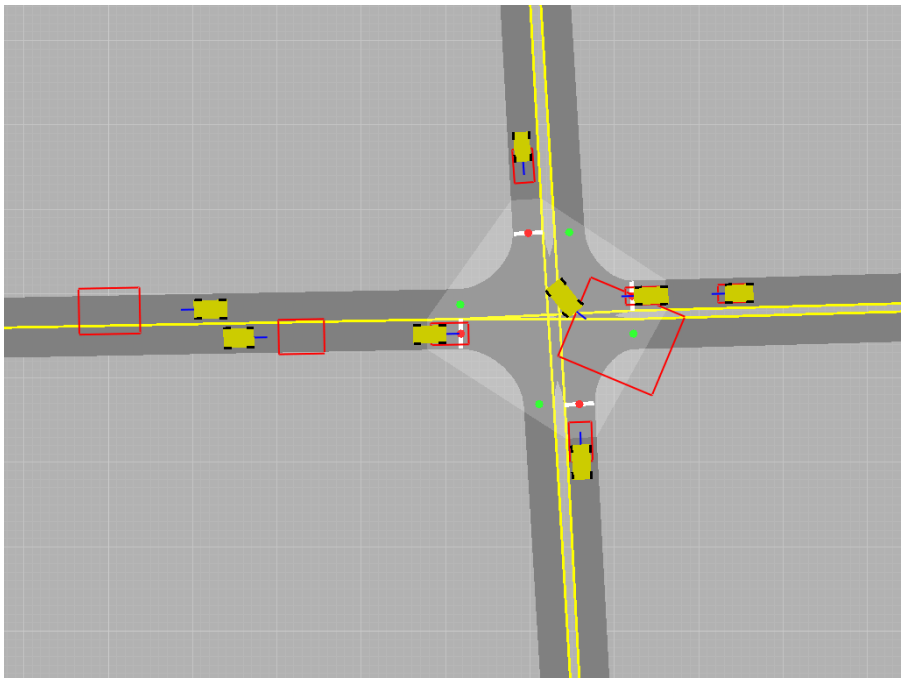
### 4.1.3 Prediction zone

The "prediction zone" is roughly defined as the area that an agent car would like to keep clear for safe driving. Using the car's current position, heading, speed, and steering angle, the prediction zone is calculated as an estimation of where the car could possibly be within the next second.

First, the center of the prediction zone is found by computing the vehicle's heading



(a)



(b)

Figure 4.2: Agent prediction zones, drawn as red outlines. The zones are centered around each vehicle's anticipated position one second in the future, and dialate when the vehicle is turning or moving fast.

one second from the current time and the distance traveled during that second. This information, along with the current radius of curvature, is then used to find the vehicle's position as if it had traveled at a constant speed and steering angle.

The vehicle's existing bounds are transformed to this predicted location, then dialated by a factor of its steering angle multiplied by its speed. Thus, higher speeds and steering angles cause the prediction zone to expand dramatically, essentially making the agent more cautious of other vehicles (see Figure 4.2).

An agent is alerted whenever its prediction zone collides with another car, or another car's prediction zone. This enables agents to make guesses of future collisions, and act to avoid them before they happen.

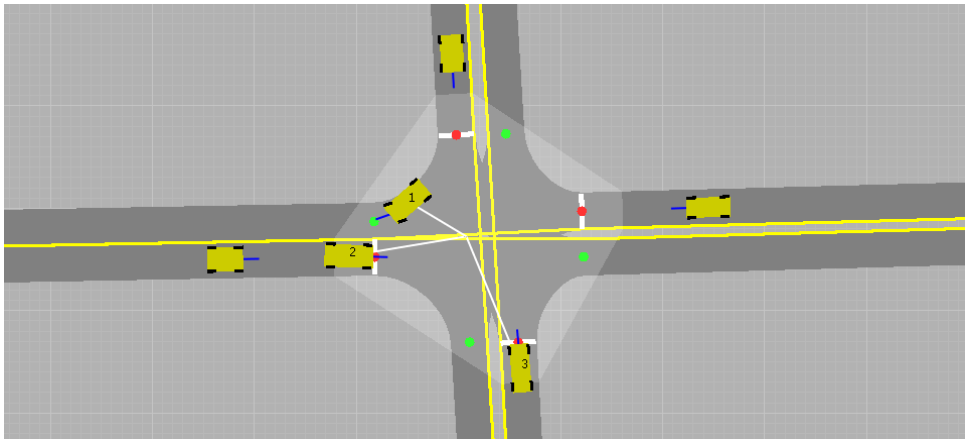
### 4.1.4 Ray casting

When desired, vehicles can fire rays in any direction and find out what they hit, in a similar fashion to a lidar sensor. However, the ray also returns information about whether its target it static or dynamic, as well as distance and relative speed data. This information is most frequently used to see if the vehicle has a clear path to its next waypoint, or to implement radar-style vehicle following.

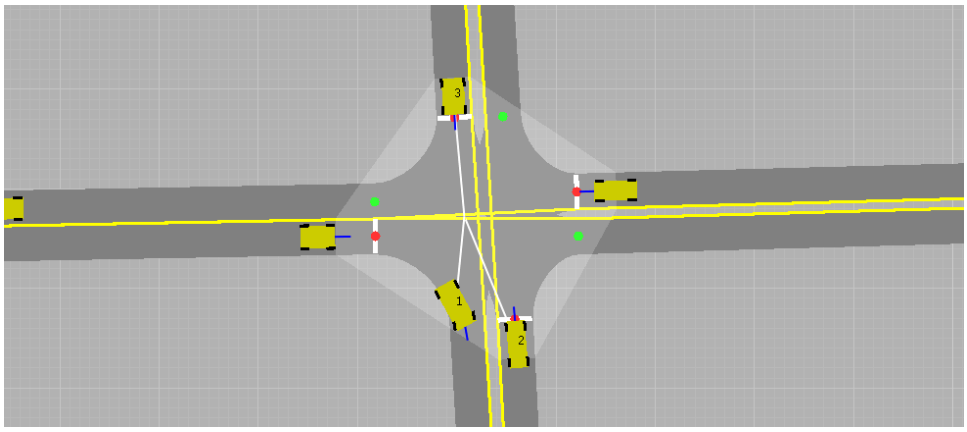
### 4.1.5 Intersection precedence

Agent cars handle intersection precedence through a "ticket" system, similar to the numbers passed out at a deli where large numbers of people are waiting to be served. If a vehicle is pulling up to an intersection with a stop sign, it is assigned a ticket once it has made a complete stop. Whichever vehicle has the lowest ticket has precedence until it clears the intersection, at which point precedence is passed on to the next higher ticket (see Figure 4.3). The intersection precedence "sensor" lets each agent car know whether or not it has precedence.

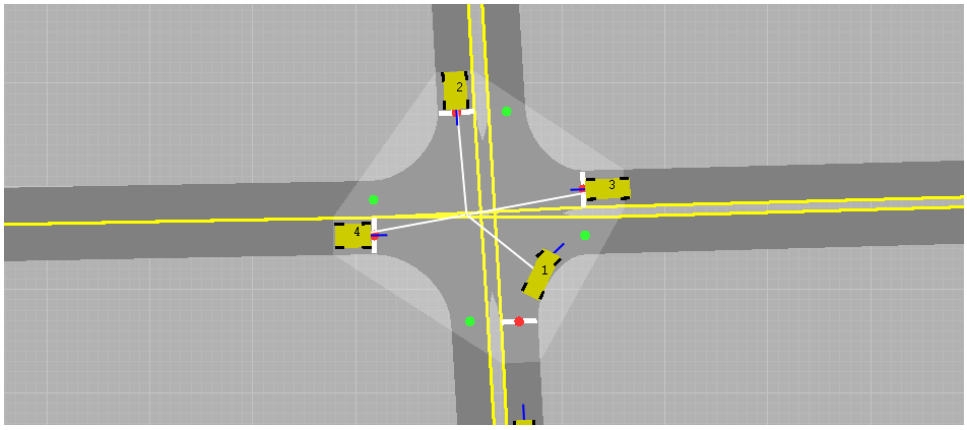
If a vehicle is approaching an intersection but does not have a stop sign, it is




(a)

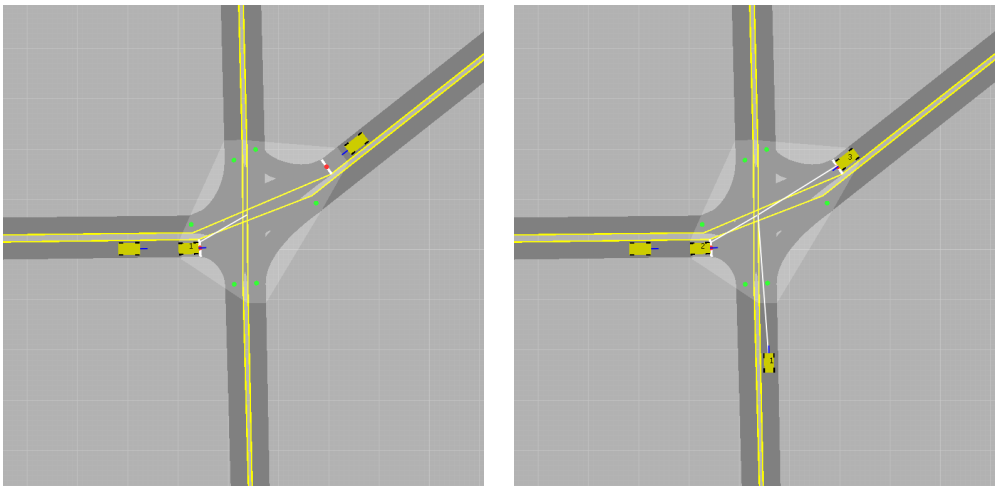



(b)




(c)

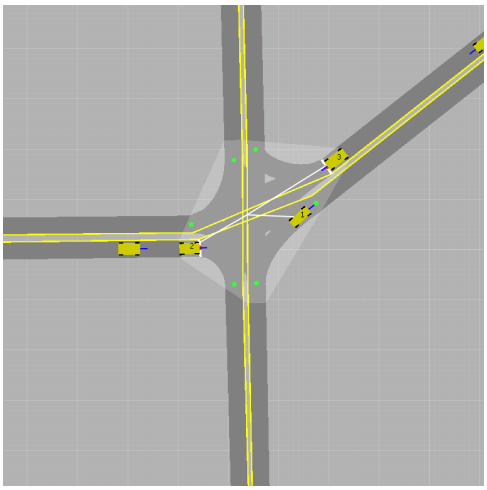
Figure 4.3: A time-lapse of several cars handling intersection precedence. Tickets are assigned as soon as each car stops, and released once the car clears the intersection. The lowest ticket (1) has precedence.



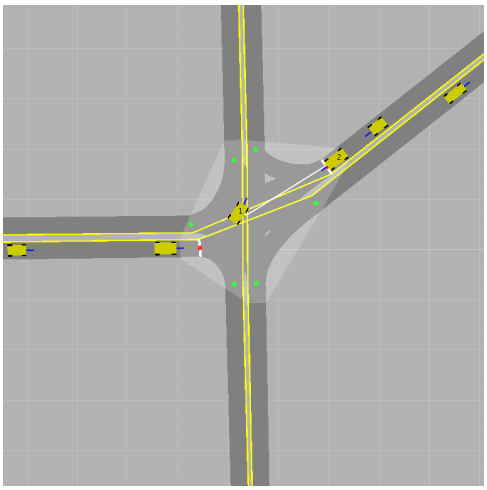
(a)



(b)



(c)



(d)

Figure 4.4: A time-lapse of several cars handling precedence at an intersection with stop signs in only one direction. The car moving in from the bottom jumps precedence ahead of the two cars at stop signs.

| Action | Weight |
|---|---|
| Stay in lane | $d$ |
| Make turn | $d + 50$ |
| Change lanes | $d + 20$ |

Table 4.1: Weightings of various actions for creating the route plan. $d$ represents the distance between successive waypoints of each action.

automatically assigned the lowest ticket once it is within 5 seconds of reaching the intersection at its current speed. This means that vehicles without stops will "jump in line" ahead of those with stops, thus implementing yielding behavior (see Figure 4.4).

Agents are capable of examining each others' precedence information and AI goals, but Dexter's workings are opaque to them. As a result, Dexter is also assigned tickets through the same scheme, thus allowing the agent cars to play well with Dexter at intersections. It is worth noting that the ticket system is very different from the precedence computations used on Dexter itself, and it is possible for the two systems to confound each other when they reach different conclusions. This is actually a useful feature, as real Urban Challenge vehicles will all have subtly different intersection behavior, and precedence confusions are likely to happen in competition.

## 4.2 Planner

The highest level of agent AI is the planner, which is responsible for orchestrating the long-term goals of the vehicle. These goals are codified in the *route plan*, which specifies a sequence of waypoints to hit in order to reach a desired checkpoint. It is up to the state machine to select the proper behaviors to complete the route plan.

The route plan itself is generated by selecting a checkpoint on the RNDF and using Dijkstra's algorithm to generate a sequence of waypoints connecting it to the vehicle's current location. At each waypoint, the are a number of possible actions that can be taken, and each is weighted differently (see Table 4.1).

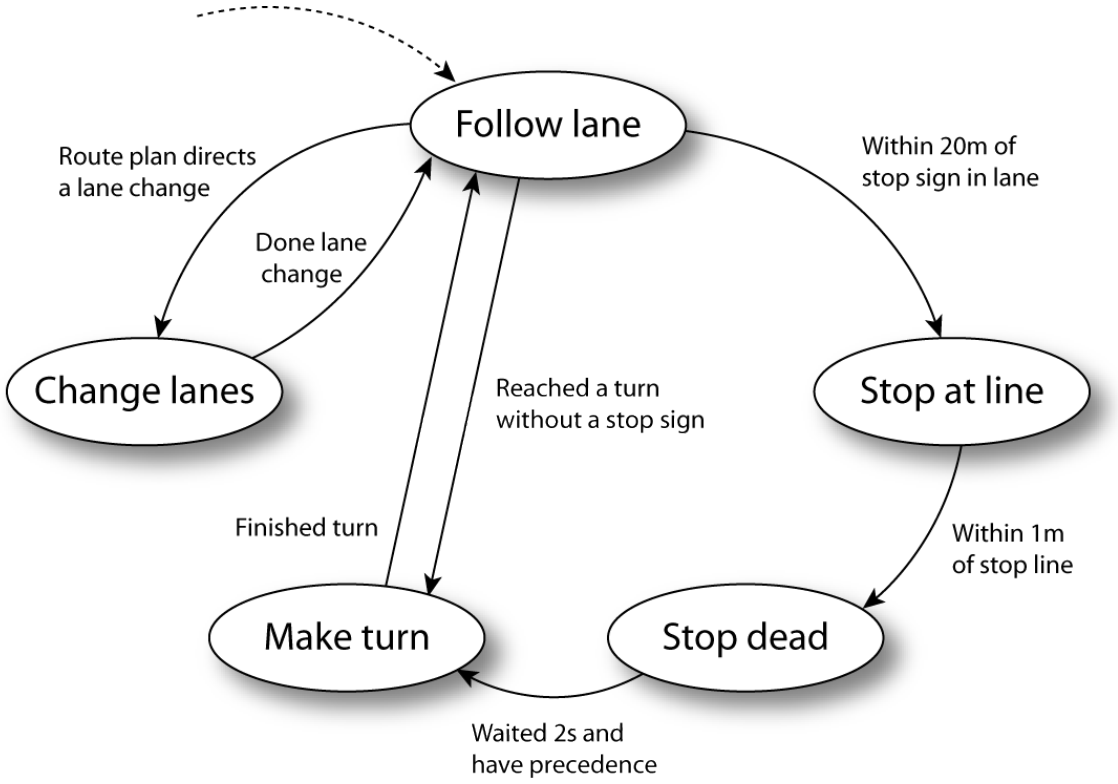



Figure 4.5: The state machine that controls an agent's decision making process. The agent begins in the "Follow Lane" state.

This weighting scheme encourages the planner to avoid passing through intersections or changing lanes unless absolutely necessary. Thus, the agent will opt for a slightly longer sequence of "stay in lane" actions over a path that makes lots of turns and lane changes. This is good driving behavior in general, and better mimics the decisions that humans and other robots are likely to make.

## 4.3 State Machine

### 4.3.1 States

The "state machine" layer of agent AI is responsible for switching between behaviors in order to make progress along the vehicle's current route plan, and for controlling the maximum speed at which the behaviors are allowed to drive. It is also the layer

which interprets the majority of the sensor information.

Cycling between behaviors is accomplished with a simple finite state machine, using sensor information and a few timers to define transitions. The state diagram can be seen in Figure 4.5, with the states corresponding to active behaviors. As stated before, the system is currently incapable of performing static obstacle avoidance or u-turns.

### 4.3.2 Speed control

Each one of the states (except "Stop dead") continuously monitors the available sensor information for potential collisions, and will modulate the agent's maximum speed to avoid them.

Potential collisions are detected by three sources: ray casting, safety zones, and vehicle predictions. First, a ray is cast from the vehicle's location in the direction it is currently steering, and any solid objects detected within 10 meters are considered potential colliders. Second, any objects colliding with the vehicle's safety zone (Section 4.1.2) are marked as potential colliders. Finally, any vehicle whose prediction box (Section 4.1.3) or current location is in contact with the agent's prediction box, then the encroaching vehicle is labeled a potential collider.

In the absence of any potential collisions, the agent will default to its preferred top speed of a random value between 25 and 30 mph. Otherwise, the vehicle uses its relationship to the potential collider to determine if and how much it should slow down.

It is not a good policy for both vehicles entering a potential collision to slow down, as doing so will never allow either car to pass. Instead, one should maintain speed while the other waits, effectively serializing the cars as they pass through the potential collision zone. In some cases, potential colliders can be flagged on vehicles that will not collide, such as two lane following vehicles proceeding around a turn in opposite

directions. Thus, a priority scheme is necessary to determine which vehicle gets to go first, or if either needs to slow down at all.

The priority scheme is implemented as a series of conditions; if one condition fails or is indeterminate, the decision process falls through to the next. The final condition is impossible to tie, so a decision is guaranteed.

- If the vehicles are in different lanes in the same road segment, and both are traveling the proper direction for their lane, and neither vehicle is changing lanes, then neither vehicle should slow down.
- If both vehicles are in the same lane, but one is traveling the wrong direction or changing lanes, the other vehicle must slow down (defensive driving).
- If both vehicles are in the same lane and performing simple lane following, the rear vehicle should slow down.
- If both vehicles are not in the same intersection and all conditions up to this point have failed, neither should slow down. From this point on, all conditions handle cases of intersection precedence for merges and turns across traffic.
- If either vehicle is about to stop at a stop sign, neither should slow down.
- If one vehicle is traveling more than 5 mph faster than the other, the slower vehicle should slow down.
- If both vehicles are traveling the same direction (headings within 90 degrees) and one is distinctly in front of the other, the rear vehicle should slow down.
- If the vehicles are traveling opposite directions, or one is not necessarily in front of the other, the vehicle making the sharper turn should slow down.
- Barring all else, the vehicle occupying a lower position in memory must slow down.

This list of conditions, while seemingly arbitrary, attempts to codify notions of good driving gleaned from experience. The list is knowingly incomplete and does not handle all cases perfectly, however, it is sufficient to provide the desired reliability for traffic in randomized tests.

In addition to adjusting speed to avoid collisions, all states will look ahead 30 meters ahead in the route plan for stop signs or sharp turns, and slow the vehicle to 20 mph in preparation for difficult maneuvers or stops.

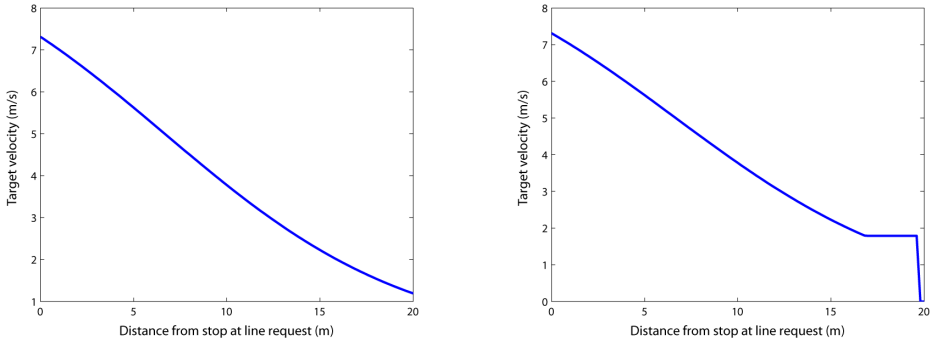
Finally, if the "Follow Lane" state detects a lead vehicle 30 meters ahead or closer, it will halve its speed in an attempt to maintain a 30 meter separation distance. If the lead vehicle stops, the agent will still creep up behind it until it has reached the minimum safety zone distance.

## 4.4 Behaviors

Behaviors provide low-level navigation, turning general directives such as "drive in this lane" into speed and driving point requests (see Section 4.5.2), which are then fed into vehicle control. Higher-level layers achieve their goals by stitching together sequences of behaviors and providing desired speeds. Behaviors treat the desired speed as an upper limit, and will drive slower if necessary to maintain control around turns.

### 4.4.1 Stop dead

When instructed to stop dead, an agent simply slams on the brakes and centers its steering. This brings it to a screeching halt in as little distance as possible, after which point it just sits still.




(a) Scaled and biased sigmoid for computing target speed

(b) The same sigmoid after clamping minimum speed and forcing a stop at the line

Figure 4.6: An example of the stop at line velocity profile bringing a vehicle moving at about 7 meters per second to a stop in 20 meters.

### 4.4.2 Stop at line

When instructed to stop at a line, an agent will attempt to smoothly decrease its current speed to zero in the exact remaining distance to the stop line. To create a more natural braking profile, the desired velocity is computed as a scaled and biased sigmoid function (see Figure 4.6a). This mimics the human behavior of braking soft, then hard, then soft again, which makes for a more comfortable ride and wears easier on the mechanical components of the car. The simulated cars have no concept of comfort or wear and tear, but must nonetheless act as if they do to produce realistic behavior.

The vehicle's speed is set to a minimum of 4 mph until it gets to the stop line, to ensure that it will creep forward enough to not obstruct traffic (see Figure 4.6b). If the car overshoots the stop line, it hits the brakes and attempts to come to a stop as quickly as possible.

### 4.4.3 Follow turn at speed

To make a turn, an agent will generate a spline (as described in Section 3.4.2) from the exit waypoint of one lane to the entrance waypoint of the next, and attempt to

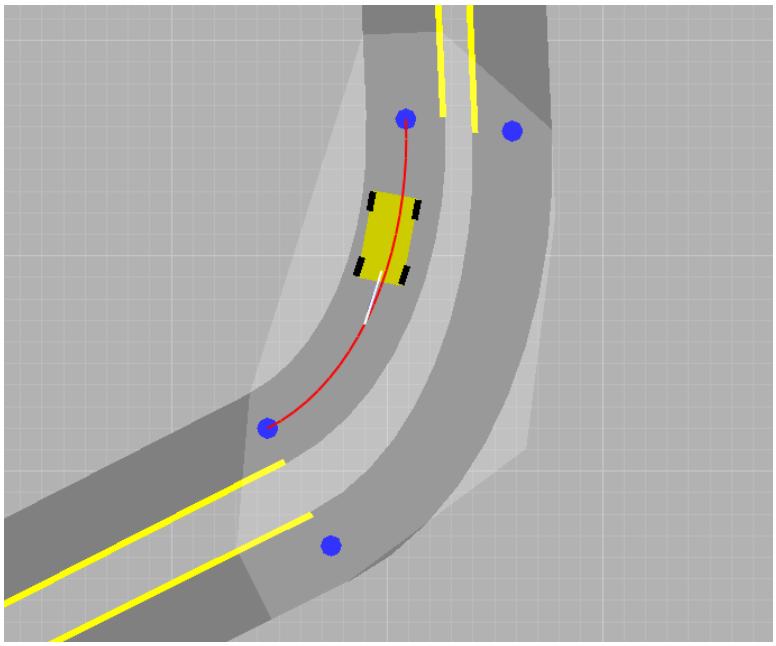
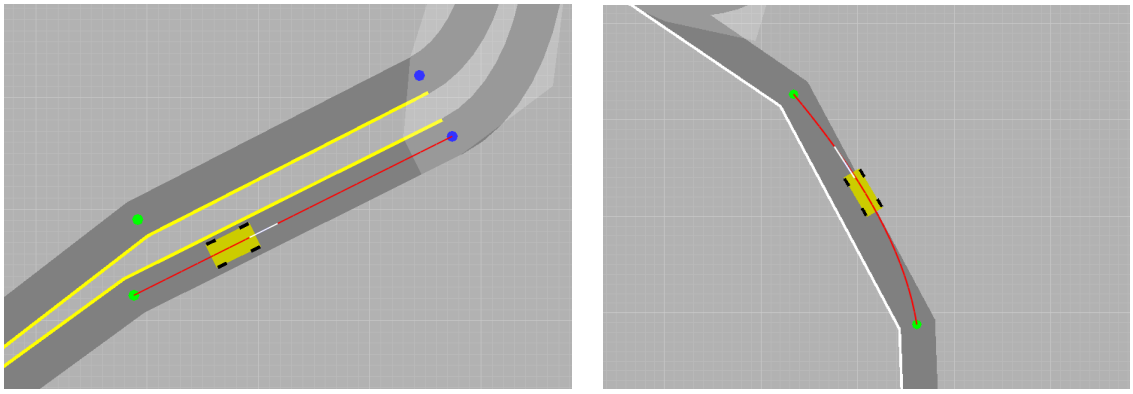


Figure 4.7: An agent making a turn. The spline is shown in red, and the agent's target driving point is at the end of the white line, 3 meters down the spline.

drive along it at the requested speed (see Figure 4.7). During the turn, it will track its position on the spline and use a lookahead to generate its steering target. In addition, it will adjust its speed to ensure that it doesn't take the turn too hard.

Due to the cubic nature of the splines and the fact that a vehicle will rarely be precisely on its driving path, finding a vehicle's progression along a spline is non-trivial, and must be solved using a numerical technique. For the agents in Dexsim, a conjugate gradient method is used to minimize the distance between the vehicle's position and a parameterized point on the spline. The initial guess for the parameter is derived by projecting the vehicle's steering point (see Section 4.5.2) onto a straight line connecting the starting and ending waypoints of the spline. Once the vehicle's approximate parameter on the spline is known, a point three meters down the spline is used as a target steering point. If this point extends past the end of the spline, the tangent of the end waypoint is used instead, causing the vehicle to straighten out at the end of the path.



(a) An agent following a straight lane

(b) An agent following a curved lane

Figure 4.8: Follow lane behavior, shown in both straight and curved lanes.

Speed control limits the target velocity to keep the vehicle's perpendicular acceleration below a certain limit, thus maintaining control through turns and simulating a comfortable ride. Once the vehicle's progression along its spline is known, we can compute the spline's unit acceleration at that point and subtract off the component parallel to the vehicle's direction of travel to obtain the spline's unit perpendicular acceleration $a_{perp}$. This is related to the radius of curvature of that location on the spline by $a_{perp} = 1/r$. By rearranging the centripetal acceleration equation $a_c = v^2/r$ and substituting in the previous equality, we arrive at the relation $v = \sqrt{a_c/a_{perp}}$. From this, we can compute the maximum allowable velocity that keeps the perpendicular acceleration experienced by the vehicle below the desired $a_c$. The acceleration limit $0.4 * g = 3.93\ m/s^2$, with $g = 9.82\ m/s^2$ was chosen for the Dexsim agents due to its natural looking results.

### 4.4.4 Follow lane at speed

Agents follow lanes by successively chaining together waypoints using splines, and driving along those splines using the exact same control as when making turns. Whenever a vehicle rolls over each successive waypoint in the lane, it will create a spline to the next and begin driving it. This way, the vehicle will progress smoothly down the

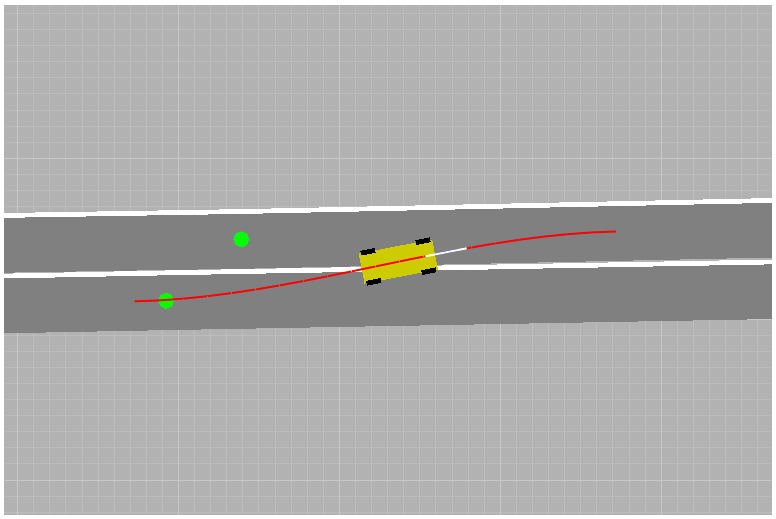


Figure 4.9: An agent changing between two lanes.

lane.

The smoothly curving nature of the spline is quite desirable when driving around bends in a lane, but has a tendancy to cause small wavy motions when driving along nearly straight segments. As such, a full spline is only generated between waypoints that have a heading difference of more than 10 degrees, otherwise a straight line segment is used. This produces much more natural driving patterns (see Figure 4.8).

### 4.4.5 Change lanes at speed

When instructed to change lanes, the agent will use the RNDF to plot a point roughly 30 meters ahead of its current location in the target lane. It then connects its current location to the target point using a spline, which it drives in the exact same fashion as the "Make turn at speed" behavior (see Figure 4.9).

## 4.5 Vehicle Control

The vehicle control layer is the lowest tier in the agent AI hierarchy, and is responsible for issuing the raw throttle, brake, and steering commands that drive the vehicle. It

accepts as input a "driving point," a location somewhere down the road that the agent is supposed to drive towards, and a target speed. Speed control and steering control are treated separately.

### 4.5.1 Speed control

Speed control is accomplished by using a PD controller to close the gap between the vehicle's current speed and target speed. The coefficients were drawn from the PID used in Dexter's vehicle controller [17], and the integral portion was removed due to undesirable windup, which would cause the agents to run stop signs. The output of the PD is split into mutually exclusive throttle and brake commands depending on its sign, which are then used directly to drive the vehicle.

### 4.5.2 Steering control

Steering control is performed relative to the steering point, defined as the point in the middle of the vehicle's front axle. The heading from the steering point to the driving point is computed, and compared against the vehicle's current heading to determine the desired steering angle. This request is clamped to the vehicle's available steering range, and is matched as quickly as the vehicle's steering delay will allow.

## 4.6 Stability testing

With the implementation described above, a set of 50 agents was left to explore a slightly modified version of DARPA's sample RNDF (seen in Figure 5.9) as a means to examine their long-term stability. The parking lot was removed, the sparse waypoints to the west were straightened out, and all three and four way intersections were modified to have stop signs in all directions. The agents were instructed to randomly pick checkpoints to visit, and to keep moving until a collision occurred. Dexter was

driven off the road, so as to not obstruct the test. The agents managed to last over 48 hours driving amongst themselves, well in excess of the length of any necessary regression test. As a result, they are able to successfully provide traffic with which Dexter can interact.

# Chapter 5

# Automated Testing

The simulator itself is a useful tool, but it is made far more useful as an element of a testing framework. By creating a system which can automatically test and verify Dexter's behavior in a variety of driving situations, the development team is given a powerful tool that can find failures without any human interaction. Once a failure has been found, the test provides an easy means to recreate it, thus allowing the developer to focus on isolating and resolving the problem.

The testing system provides two major types of tests: functional and randomized. In functional tests, Dexter is placed in a very specific situation designed to make the vehicle exhibit a particular behavior, usually to verify his conformance to a particular law or rule. Such tests are typically short and evaluated harshly for aberrant behavior. As each test is quite specific in its goals, large numbers of functional tests are created to exercise the robot's behavior for every rule. Randomized tests are much looser and longer, examining Dexter's performance in several-hour-long sessions with other simulated vehicles. These tests are designed to stress Dexter's long-term stability, and expose the AI to traffic situations that were not anticipated by the designers.

Each test is set up and run automatically by a regression testing framework [8], and evaluated by a test script specific to each test. The framework allows the simulation

to be paused or interrupted for debugging purposes, and logs significant events (as determined by the test script) by taking screenshots. These events, as well as the final judgement of Dexter's performance in each test, is output as a webpage for convenient review.

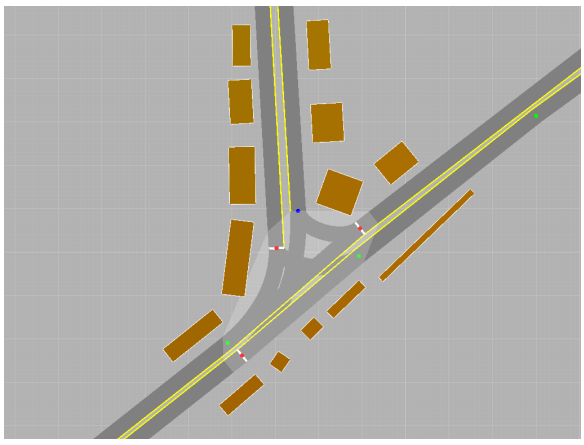
The tests implemented currently in the regression suite cover the "Basic Navigation" and "Basic Traffic" sections of the DARPA Urban Challenge Technical Evaluation Criteria [2], but do not touch upon the advanced sections. This is mostly a reflection of the state of Dexter's AI, which at the time of this writing has just completed the DARPA site visit. As of yet, none of the advanced features are ready to test.

## 5.1 Environment

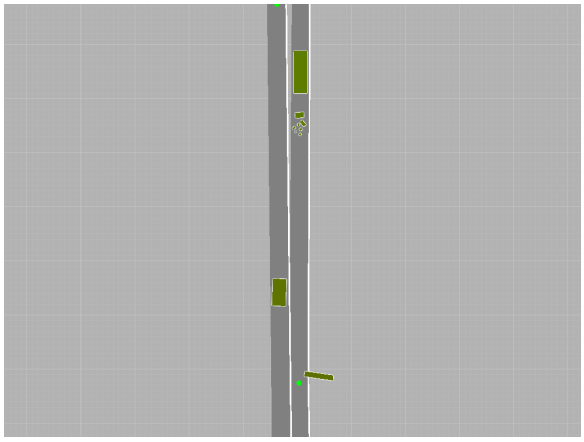
To create interesting tests, it is necessary to populate the test environment with more than just Dexter himself. Besides a map to use, there must also be static and dynamic obstacles to avoid. These are implemented as various objects placed in Dexsim's scene graph, and are used to calculate simulated sensor data.

Static obstacles are represented as simple rectangles of arbitrary sizes and rotations, which can be placed on a map by a user or a random distribution. As their name suggests, they will not move once placed, and can represent any type of obstacle that Dexter might encounter on the course (see Figure 5.1).

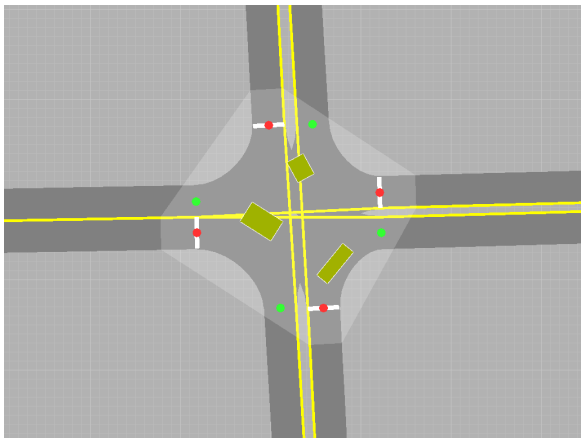
Dynamic obstacles are limited to other vehicles, due to the absence of pedestrians in the Urban Challenge. The simulator treats these vehicles as full-fledged cars, and uses the same dynamics equations as are used on Dexter, although with different vehicle parameters such as mass, length, width, braking power, steering limits, and the like. They are controlled by autonomous AI, as described in Chapter 4.



(a) Buildings



(b) Lane-blocking debris



(c) Traffic jam

Figure 5.1: Various configurations of obstacles, mimicking buildings (a), lane-blocking debris (b), and a traffic jam (c).

## 5.2 Driving Evaluation

When evaluating the performance of an autonomous vehicle, humans look for pre-conceived patterns of "proper" behavior, with any significant deviation from them deemed "improper" behavior. These patterns are sometimes codified into nicely defined laws and rules, but also come from more abstract notions of "good driving" gleaned from experience. No matter the source, these patterns break down into sequences of expected actions and failure conditions, and with a proper vocabulary, they can be recognized by finite state machines.

Judging a vehicle's actions for conformance to laws is straightforward, as the laws themselves provide explicit context and rules for evaluating behavior. Normally, it is sufficient to break down each rule into a series of conditions that can be identified by the simulator, and constantly monitor the test vehicle for violations.

In general, the more subjective judgments of vehicle performance are evaluations of the vehicle's ability to proceed smoothly to its destination with a minimum of extraneous or wasteful behavior. In such cases, it is often easier to recognize and reject bad behavior rather than attempt to codify good behavior. Bad behaviors include excessive delays (such as waiting too long at a stop sign), wasteful movements (such as swerving off a straight road), and risky actions (such as driving too close to an obstacle). Once identified, such judgments can be quantified and recognized in simulation.

In order to implement evaluations of driving behavior, it was necessary to create more flexible logic than that offered by a compiled language such as C++. Thus, the Lua language [26] was embedded into the simulator, and integrated with SWIG [15] such that simulation objects could be referenced and controlled from scripts. This flexibility allowed rapid tweaks to be made to tests, and greatly broadened the tests that could be created.

The evaluation criteria are implemented as yes-or-no conditions which are checked

by the test script during simulation. Each one can be enabled or disabled for particular tests, or combined into more complex conditions. The evaluations implemented are documented in the following pages, and define the vocabulary of the regression tests.

### 5.2.1 Hit waypoints in order

The test system checks that Dexter passes over a list of RNDF waypoints in order, and does not deviate from the expected course. Optionally, Dexter can also be given maximum time allowances to proceed from point to point. This condition is useful to check Dexter's route planning and lane following.

### 5.2.2 Hit checkpoints in order

A somewhat weaker version of the above, this condition checks that Dexter hits a particular sequence of checkpoints in order. There is an optional time delay as well, which in practice is set rather high (30 minutes) to ensure that Dexter has enough time to complete his route. This condition is useful to ensure that Dexter actually completes his mission without making any errors.

### 5.2.3 Stop and stare timeout

The DARPA rules document states that cars will be penalized for more than 10 seconds of "stop and stare" time, defined loosely as time when the robot just sits in place despite a clear means to proceed (at least, clear to a human) [2]. This condition starts a timer as soon as Dexter has stopped, and resets when he starts moving again. If the timer reaches 10 seconds, the test is failed. A contextual version is available which will not penalize Dexter if the vehicle is queueing at a stop sign, and not moving due to expected traffic conditions.

### 5.2.4 Run timeout

This condition checks the runtime of the simulation, and can be used to complete or halt the test. In functional tests, it is frequently used as a failure condition to cut off the test in case Dexter happens to be running around the track without making progress. In randomized tests, it is used as a success criterion if, for example, Dexter manages to run for six hours without breaking any laws.

### 5.2.5 Collision

If Dexter comes into direct contact with any obstacle or vehicle, the test is immediately failed. This condition's use is straightforward.

### 5.2.6 Speed limit

These conditions check if Dexter's speed is within a specified range. A high limit is used to enforce a global speed limit of 30 mph, and a lower limit is turned on and off to check if Dexter is making expected progress down a road segment.

### 5.2.7 Lost localization timeout

Throughout the duration of the competition, Dexter must stay in a lane, intersection, or zone. This condition will start a timer whenever Dexter loses localization, and reset when it is regained. If the timer exceeds a certain limit, the test is failed. This condition is critical to ensuring proper behavior, and is used pervasively in all tests. The timeout ensures that short periods of lost localization are okay; the RNDF is specified in terms of straight line segments which may not represent the exact curvature of the road, so Dexter will attempt to make smoother turns around sharp corners, and may occasionally leave the RNDF's straight lane for a short time. In practice, a useful timeout is 5 seconds.

### 5.2.8 Safety zone timeout

This condition operates like the other timers, but checks for violations of the DARPA-specified safety zone around Dexter (see Section 4.1.2). It is used to ensure that Dexter does not move too close to other vehicles or obstacles for very long, usually 2 seconds. The timeout is necessary to allow temporary violations, such as Dexter's safety zone passing through traffic that is stopped at an intersection.

### 5.2.9 Reverse limit

According to DARPA's rules, no vehicle is allowed to move in reverse for more than three vehicle lengths. This condition accumulates how far Dexter has gone in reverse, and resets whenever the vehicle begins moving forward again. If Dexter moves more than three vehicle lengths in reverse, the test is failed.

### 5.2.10 Does not break precedence

By assigning "tickets" to Dexter in the same fashion as AI agents (see Section 4.1.5), the test system can identify Dexter's precedence at an intersection. To handle ambiguous cases where the simulation's idea of precedence differs from Dexter's, the vehicle is allowed through the intersection if it holds the first or second ticket. If Dexter breaks precedence by more than one spot, the test is failed.

### 5.2.11 Does not run stop signs

This condition identifies if Dexter runs any stop sign, and requires significantly more logic than most others. It is implemented as a state machine, defaulting to the "good" state. In that state, it monitors Dexter's localization, and transitions to "before stop" if the next waypoint in the lane is a stop sign. Dexter is considered successfully stopped at the line if it is pointed in the same direction as the road, its

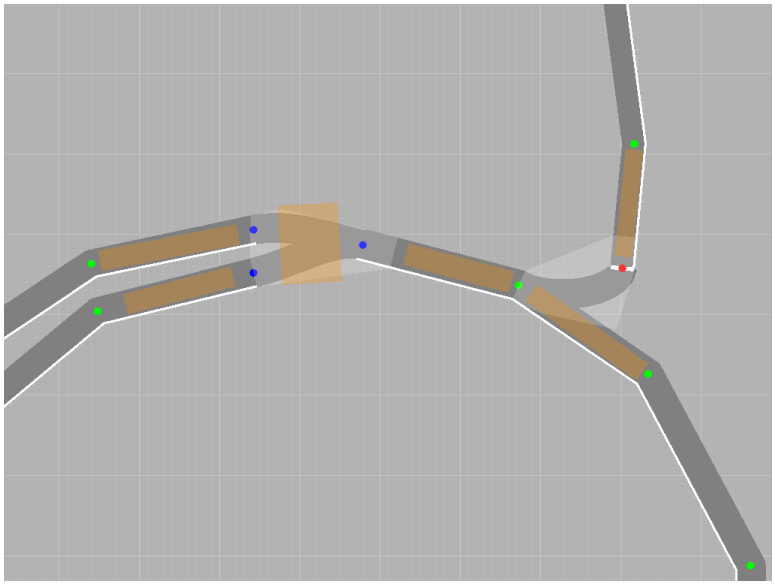


Figure 5.2: Visualization of a junction instrumented with sensors (orange or grey rectangles).

front bumper is within one meter of the stop line, and its velocity is below 0.01 meters per second. Upon a successful stop, the machine transitions to "after stop", which just waits until the stop sign has been passed before returning to "good". If Dexter drives over one meter past the stop line without coming to a successful stop, the test is failed.

### 5.2.12 Hit sensor

Test sensors are rectangular regions which do not obstruct traffic or show up on virtual sensors, and can be defined in the test setup in the same fashion as obstacles and AI agents (see Figure 5.2). These sensors can detect when a car (Dexter, an agent, or both) collides with them, and are used frequently to set up pass and fail conditions for functional tests. They can be set up to trigger repeatedly, or just once.

## 5.3 Tests

The evaluation conditions above can be used as primitives for recognizing and verifying vehicle behavior by stringing them together with state machines. By making specific environments and goals for Dexter, we created a suite of tests that automatically evaluate the vehicle's conformance to DARPA rules and the designers' more subjective judgements of good behavior.

Each test consists of five components: a Route Network Definition File (RNDF), a Mission Data File (MDF), an object file, a test script, and an options listing. The RNDF is the map to be used for the test, and mimics those created by DARPA. The MDF is another DARPA-specified file that directs the vehicle to hit a list of checkpoints in order, and specifies speed limits for all road segments. For functional tests, the MDF is usually designed to direct Dexter into the specific area of the RNDF where the test takes place; for randomized tests, it is randomly generated. The object file contains a listing of all static obstacles and vehicles to be included in the test, and may be edited using the simulator. The test script is the heart of the test system: a Lua file that contains the code which evaluates Dexter's performance, specifies the success and failure criteria, and houses the logic that determines when screenshots should be taken. An example test script for the u-turn test can be seen in Appendix A. The options listing contains any extra options that the test designer might want to pass Dexsim, such as visualization tweaks or requests for randomized vehicles and obstacles.

Creating a test is as simple as putting these five components in a directory, then instructing the test framework to run it. The following pages describe the tests implemented so far, although there is a large space of tests left to explore.

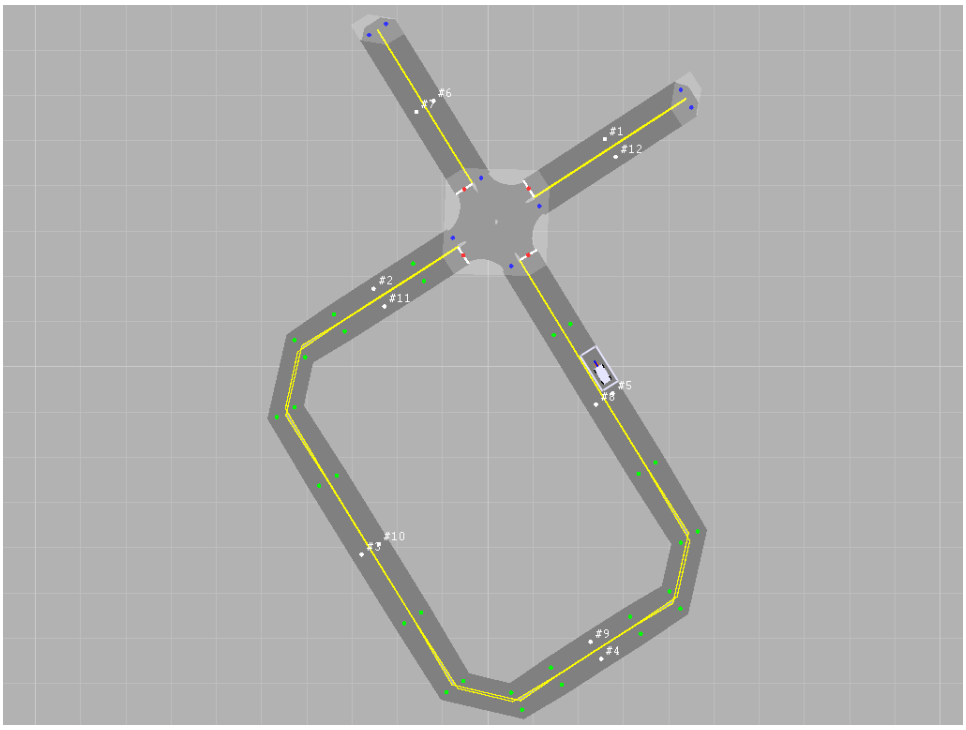



Figure 5.3: An overhead view of Dexter's site visit course This RNDF is used for several tests, including the MDF following test, in which Dexter is given a long sequence of checkpoints (white circles) to drive over in order.

### 5.3.1 MDF following

This functional test places Dexter on the site visit course (see Figure 5.3), which features an intersection and two u-turns, and checks that the vehicle hits a sequence of 52 checkpoints correctly without running off the road, halting, running stop signs, or experiencing unreasonable delays. There are no obstacles or other vehicles present. It is a simple test that exercises Dexter's route planning, lane following, and general stability.

The test script uses the "hit checkpoints in order", "lost localization timeout", "ran a stop sign", and "stop and stare timeout" evaluation criteria, and will take a screenshot every 15 seconds, when Dexter hits a checkpoint, or when the vehicle violates a timeout. This configuration gives the test reviewer a continuous history of Dexter's actions, and makes it easier to identify the causes of failures.

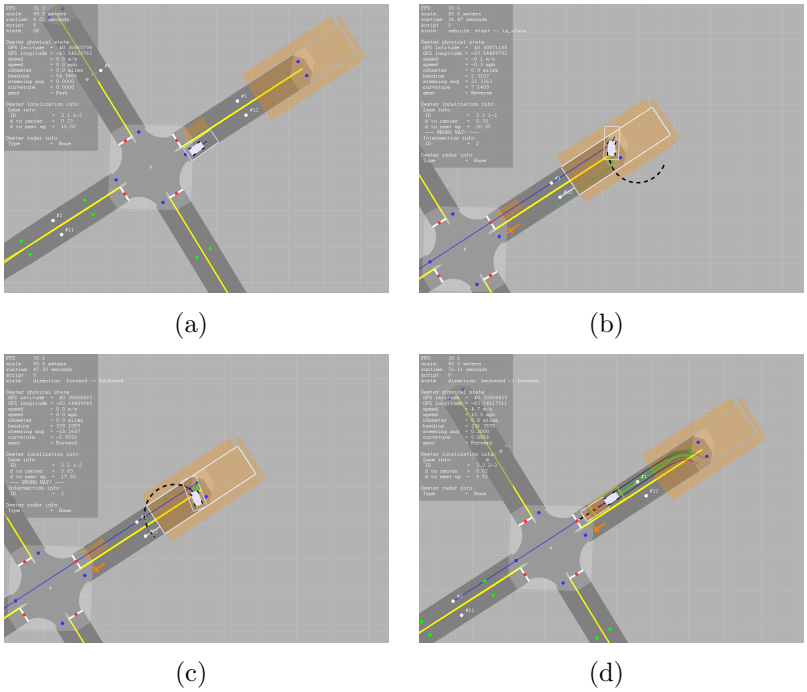
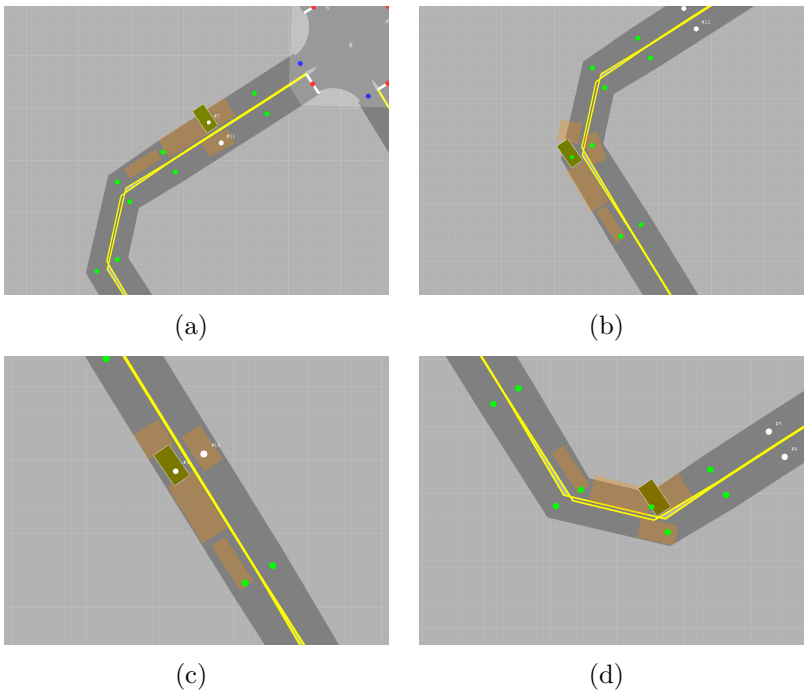


Figure 5.4: Snapshots of various stages of the u-turn test. In (a), Dexter is approaching the u-turn. In (b) and (c), Dexter is driving backwards and forwards to perform a 5-point turn. In (d), Dexter has exited the u-turn and successfully passed the test.

### 5.3.2 U-turn

This functional test checks very specific aspects of Dexter's u-turn behavior. DARPA rules specify that the vehicle is not allowed to leave a 30 × 9 meter rectangle around a u-turn once the turn has begun, so this test carefully checks Dexter's positioning throughout the entire maneuver (see Figure 5.4).

The test script uses the "stop and stare timeout", "reverse limit", and "runtime timeout" evaluation criteria, combined with a set of sensors around the edges of the u-turn area that will fail the test if Dexter touches them. Screenshots are snapped when Dexter enters and exits the u-turn, whenever the direction of travel changes (as in an n-point turn), and when a failure occurs. The test script for this test is described in Appendix A.



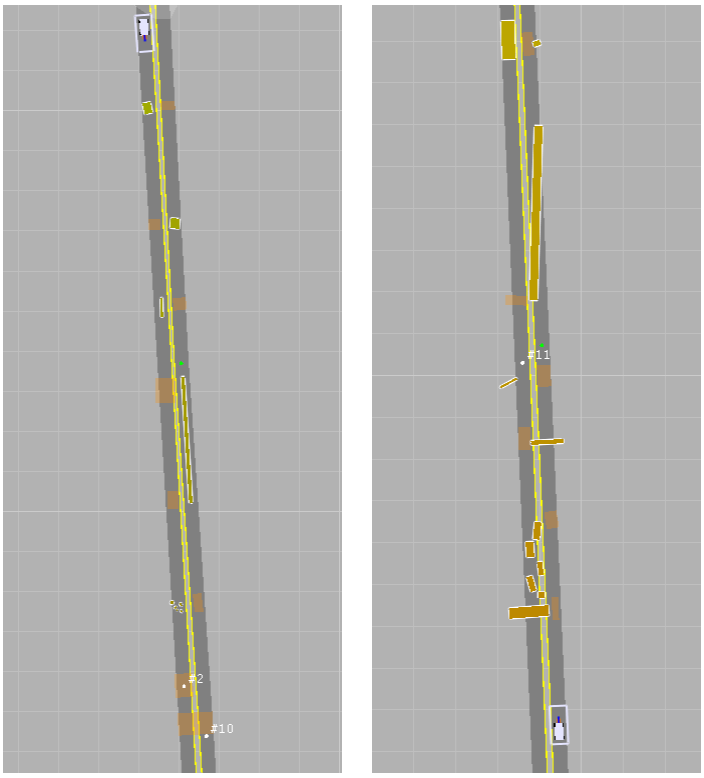
(a) (b) (c) (d)

Figure 5.5: Various configurations of obstacles and sensors for the Simple Obstacle class of tests. After an intersection (a), on an outside corner (b), in the middle of a lane (c), and on an inside corner (d).

### 5.3.3 Simple obstacles

These functional tests take place on the site visit course, and present Dexter with a single static obstacle to pass. The test checks that Dexter stops the required distance away, changes lanes, passes the obstacle, and returns to the travel lane within the required window. If the vehicle drives out of the lane, collides with the obstacle, or misses the required spacings, then the test is failed. Several different configurations of obstacles are presented, such as in the middle of a lane, on a corner, immediately after an intersection, and going the opposite direction around a corner (see Figure 5.5).

The test script uses the "stop and stare timeout", "runtime timeout", "lost localization timeout", and "collided" evaluation criteria, as well as a combination of sensors to check spacings. Snapshots are taken at each stage of passing the obstacle, and whenever a failure occurs.



(a) (b)

Figure 5.6: Obstacle course tests. One features obstacles entirely within their lanes (a), the other has obstacles crossing lane lines (b).

### 5.3.4 Obstacle course

These functional tests take place on the DARPA sample RNDF, and present difficult obstacles to navigate as Dexter progresses down the road. Currently, two versions are implemented: one with obstacles restricted to lane boundaries, the other with obstacles crossing lanes lines (see Figure 5.6).

The test script uses the "stop and stare timeout", "runtime timeout", "lost localization timeout", and "collided" evaluation criteria, as well as a combination of sensors to check for proper passing of obstacles. Snapshots are taken as Dexter passes each obstacle, and whenever a failure occurs.

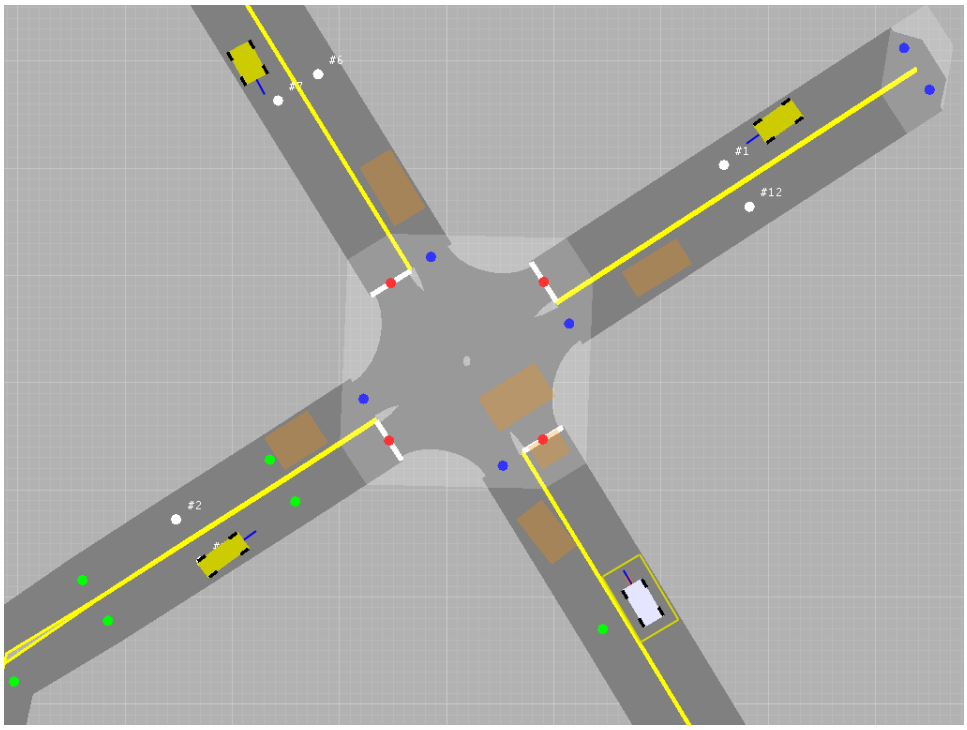


Figure 5.7: An overhead view of the intersection test setup, with three agents, Dexter, and a 4-way stop intersection. All cars, including Dexter, pull up and stop at the intersection, and must proceed through in order of arrival. In this case, the left agent arrives first, then Dexter, then the top agent, and finally the right agent.

### 5.3.5 Intersection

This class of functional tests examines Dexter's intersection precedence detection on the site visit course. In each test, several agents and Dexter arrive at an intersection in a specified order, and Dexter must take precedence and drive through at the appropriate time (see Figure 5.7). The agents can be tuned to arrive in different orders, or nearly identical times, to test ambiguous cases of precedence.

The test script uses the "stop and stare timeout", "runtime timeout", "lost localization timeout", "collided", and "intersection precedence" evaluation criteria, plus a combination of sensors to synchronize vehicles passing through the intersection. Snapshots are taken at each stage of stopping and proceeding through the intersection, and whenever a failure occurs.

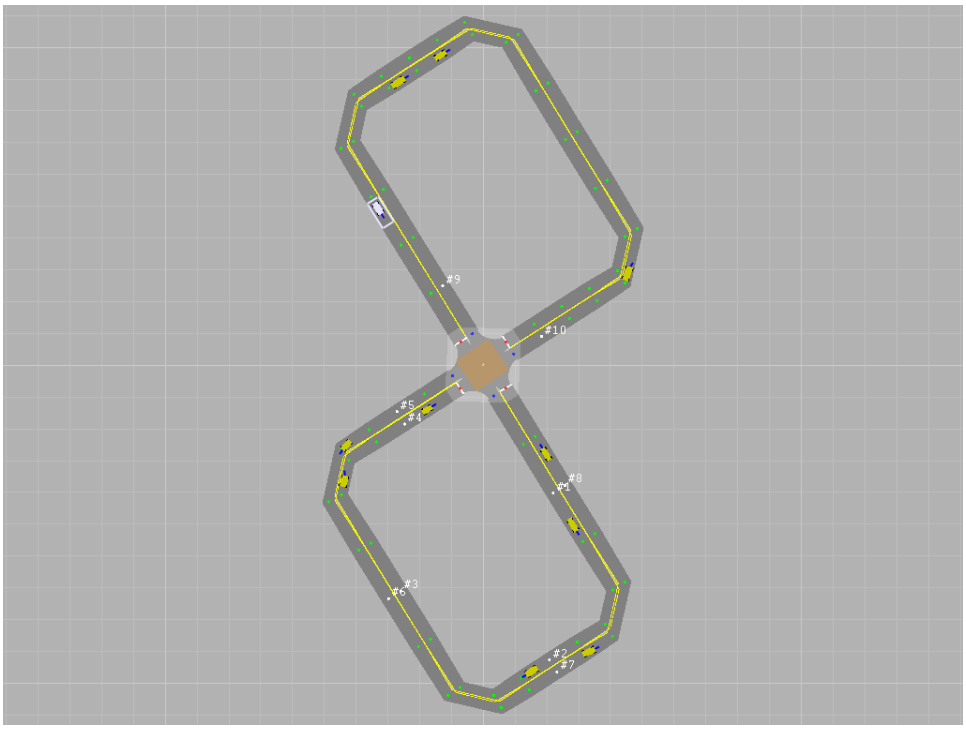


Figure 5.8: An overhead view of the random intersection test setup, with 10 agents and Dexter. The agents and Dexter drive around the figure-8 for 20 minutes, and must always obey precendence at the intersection.

### 5.3.6 Random intersection

This randomized test is similar in nature to the Intersection test, but acts as a stress test. Instead of sending a controlled sequence of vehicles into the intersection and measuring the order they come out, ten random vehicles are created and left to find their way around a figure-eight course with Dexter (see Figure 5.8). The test will run for 20 minutes, and if Dexter manages to stay out of accidents and continually make progress, the test is considered passed. Just by sheer variation of arrival times, this test covers a large number of intersection precedence cases.

The test script uses the "stop and stare timeout", "runtime timeout", "lost localization timeout", "collided", and "intersection precedence" evaluation criteria. Snapshots are taken whenever Dexter enters the intersection, every 15 seconds, and whenever a failure occurs.

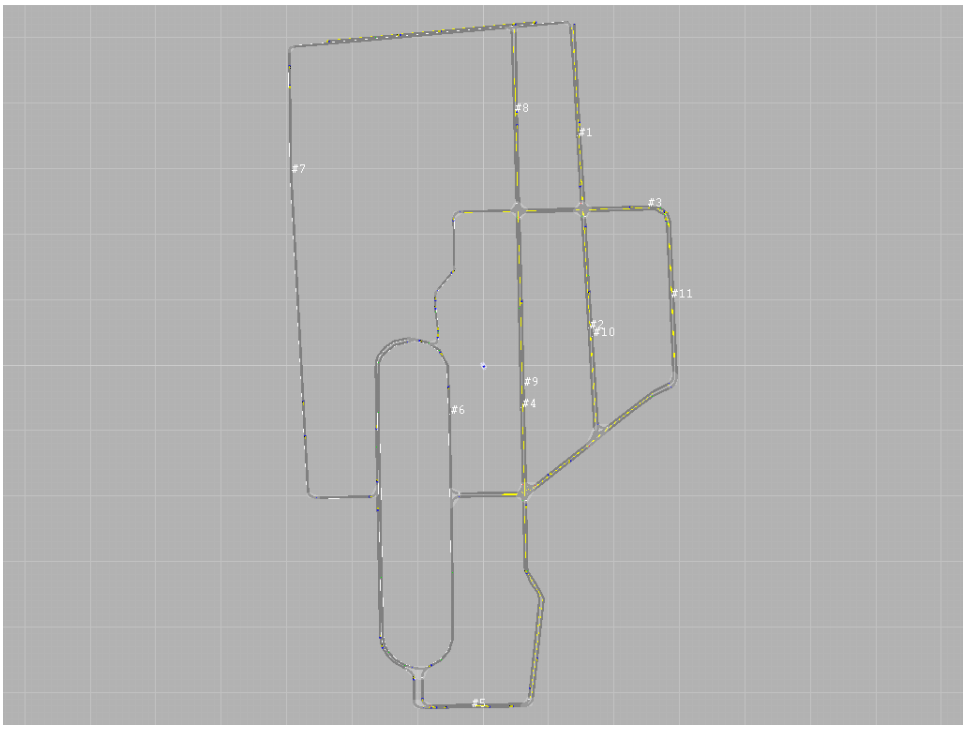



Figure 5.9: An overhead view of the random challenge test setup. The map is a modified version of DARPA's sample RNDF, with the parking lot removed and stop signs added to all 4-way intersections. 50 agents populate the course, and Dexter must navigate the checkpoints while obeying all traffic laws.

### 5.3.7 Random challenge

This randomized test challenges all of Dexter's current functionality to work in concert. It pits Dexter against 50 other agents for six hours on the simplified version of DARPA's sample RNDF (see Figure 5.9). The vehicle is given a long MDF, and must last all six hours without breaking precedence, running off the road, or colliding with another vehicle.

The test script uses the "stop and stare timeout", "runtime timeout", "lost localization timeout", "collided", and "intersection precedence" evaluation criteria. Snapshots are taken every 15 seconds, and whenever a failure occurs.

# Chapter 6

# Results

The ultimate goal of this work has been to improve the quality of Team Case's Urban Challenge entry by creating a simulation and testing framework capable of aiding the development team in writing and debugging vehicle code. The system has had a positive impact on Dexter's development, but the precise extent of its benefit is difficult to quantify. As there is only one development team, one code base, and one robot, it is impossible to compare this team's productivity and Dexter's reliability versus a control without such a simulation system in place. Comparisons against opposing teams cannot be made either, as the final Urban Challenge competition has not taken place yet, and all development is still under a veil of secrecy. We can analyze the accuracy of the simulator and make rough estimations of how many hours Dexter has spent in simulation, but the majority of the results analysis that can be performed is qualitative evaluation of the advantages that the simulation and testing suite has given our team.

To help gauge the benefit of the system, a survey was sent out to the development team asking how they used the simulation and testing functionality. Eight responses were received, six from core developers (out of eight total), and two from professors serving as advisors (out of four total). Their answers were used to create the discussion

below.

## 6.1 Simulation accuracy

Dexsim was not created with accuracy as a foremost goal; indeed, it was knowingly designed around several assumptions that would impair realism. Specificially, the simulation was limited to two dimensions (with flat terrain), dynamics effects such as vehicle roll were ignored, and the vehicle transmission was simplified into a first-order differential system. Nonetheless, it is still important that the simulator's behavior reasonably matches that of the real vehicle, otherwise there is little use to simulation at all.

To compare the results of simulation and the real vehicle, a controlled test was performed using Dexter's site visit course (see Figure 5.3). Logged data from a clean loop around the course, taken while Dexter was practicing for the site visit, is placed side-by-side with a nearly identical run in simulation. Dexter starts off stopped at an intersection, drives around the loop while slowing for corners, and finishes by stopping again at the intersection. The speed was limited to 15 mph, about 6.7 meters per second. To see how well the runs match up, plots were made of Dexter's desired and actual heading, speed, and steering angle for both simulation and the real test. The results can be seen in Figures 6.1–6.3.

Despite the simplifications used by the simulator, its results match up with actual data quite well. Simulated heading is almost identical to actual heading, and has similar delays between desired values and actual values. Steering angle is also extremely close, with one exception: a steering bug which shows up much more dramatically on the real vehicle than in simulation. Velocity shows quite a bit more variation, with the real vehicle demonstrating more sluggish performance than the simulated one. This is due in part to the fact that the actual vehicle's engine was rate-limited to

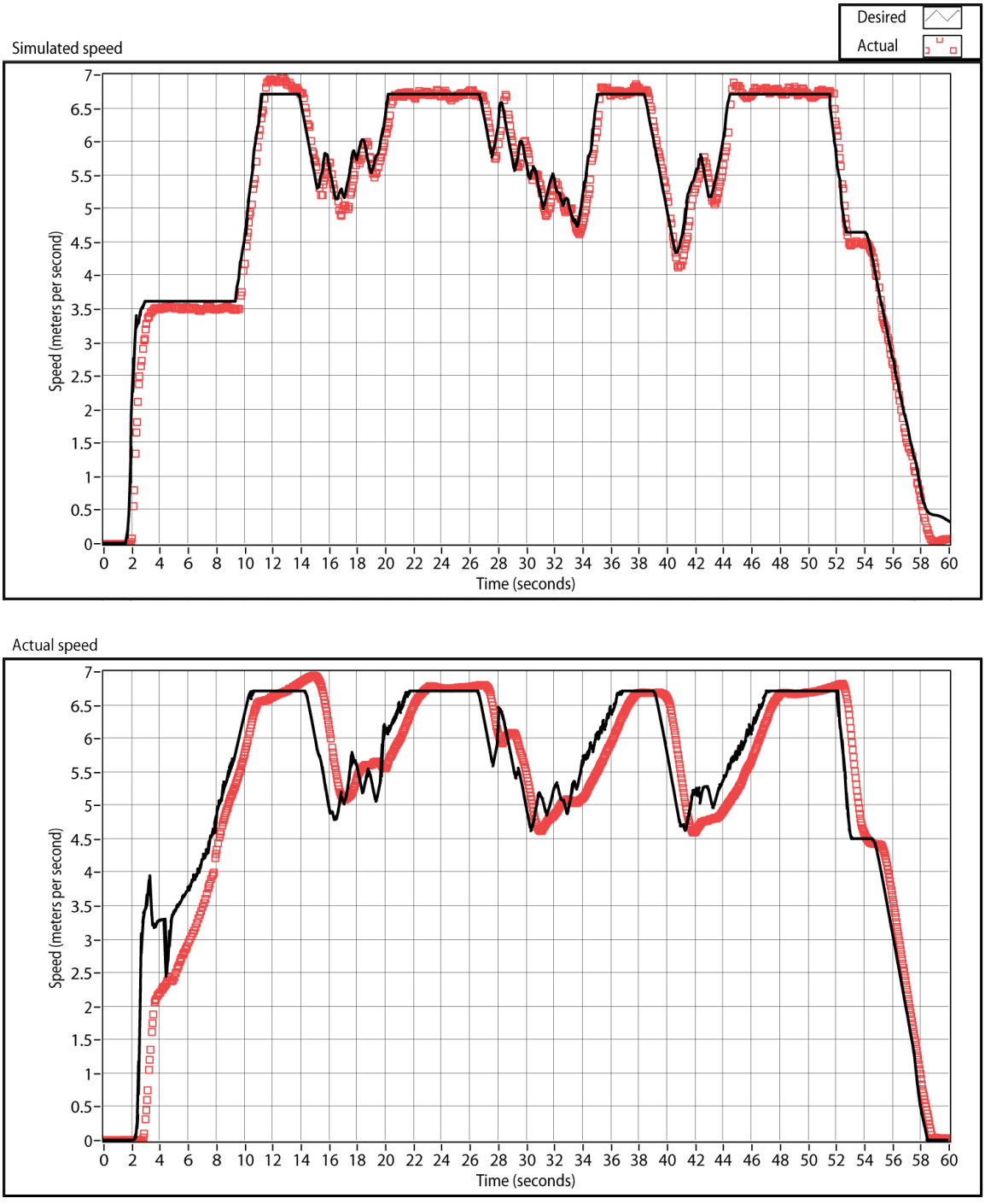



Figure 6.1: Velocity profiles of simulated (top) vs. actual (bottom) test run.

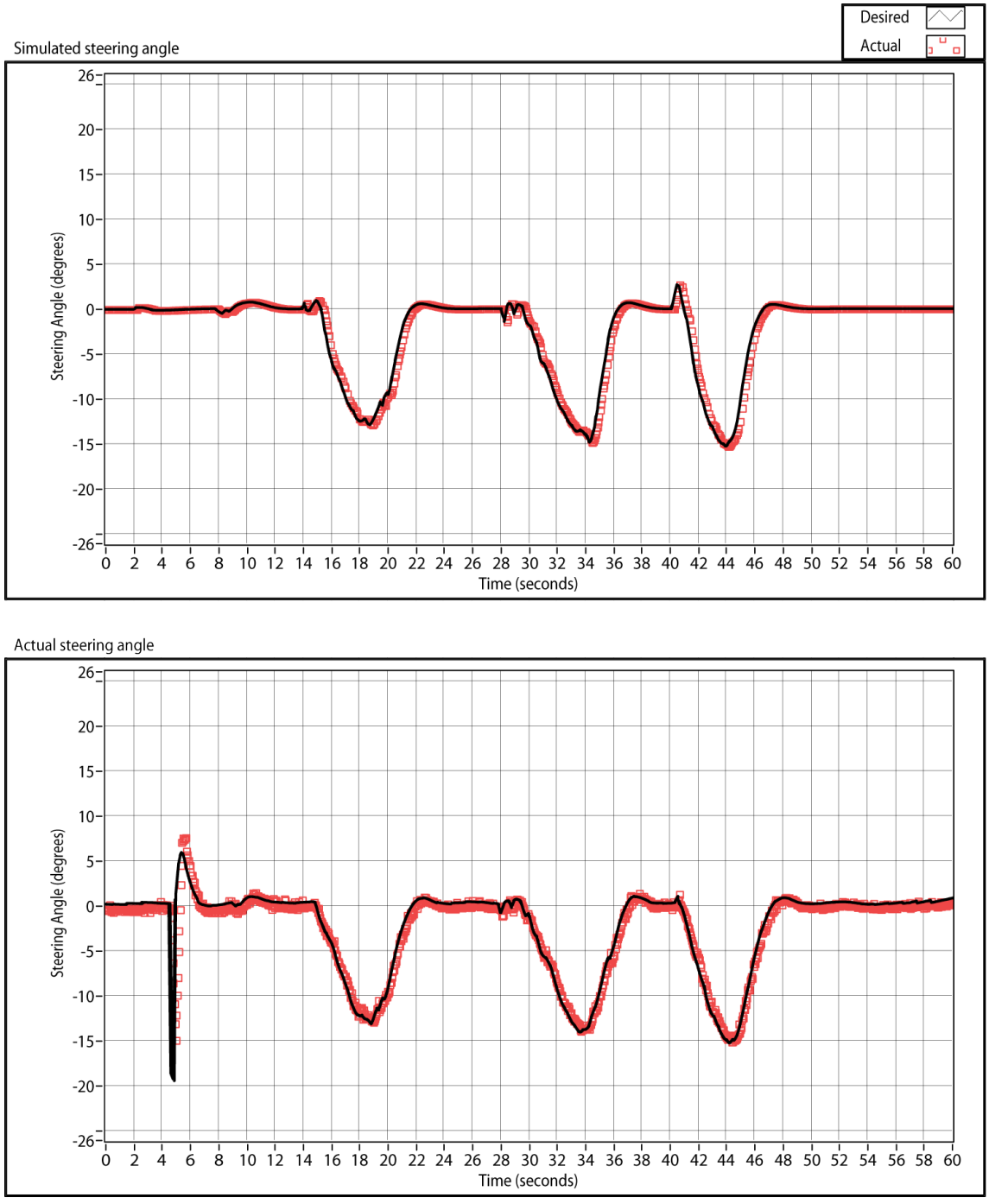



Figure 6.2: Steering angle profiles of simulated (top) vs. actual (bottom) test run. The spike in the actual run is a bug that is much less dramatic in simulation.

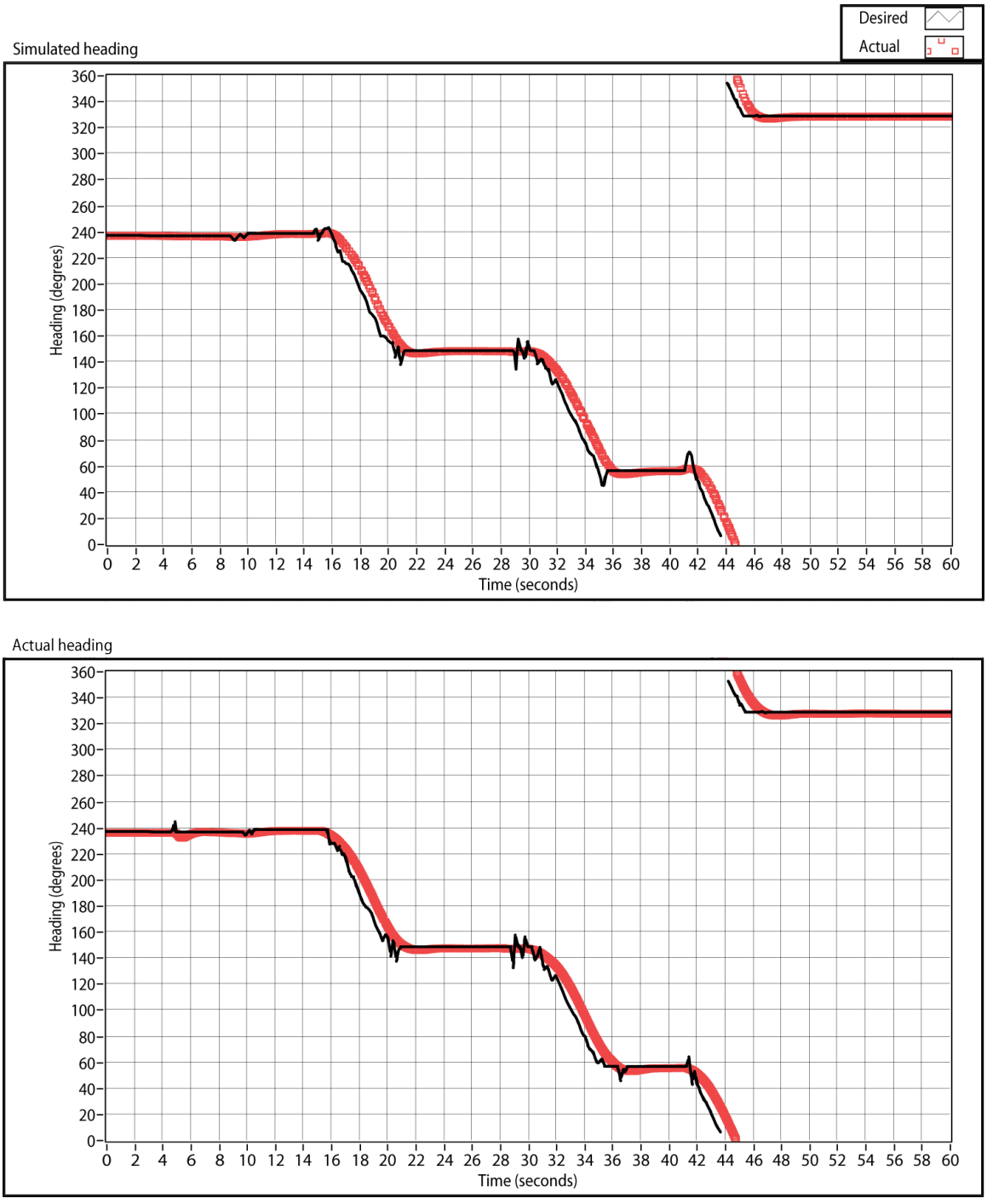



Figure 6.3: GPS heading profiles of simulated (top) vs. actual (bottom) test run.

3000 rpm during the site visit preparations, and thus really was less powerful than the simulated vehicle.

The simulated sensors used for the site visit, namely GPS, IMU, and lidar, output ideal data and therefore did not reflect any of the errors seen in real sensors. At the time, none of Dexter's sensors were redundant, so problems with any sensor would cripple the vehicle, rendering tests of sensor failures not useful. Simulated data matched up well with real sensor data under ideal conditions, with one important exception: during sharp turns, Dexter would tilt and cause the lidars to see the ground. Because Dexsim did not model vehicle tilt, this feature never appeared in simulation, which led to some unfortunate surprises during field tests.

## 6.2 Utility of simulation

The simulator has proven to be a remarkably useful tool on its own. The development cycles of Dexsim and Dexter's AI coincided such that the functionality of the simulator always slightly preceded that of Dexter. The result was that the development team never had to wait for features to be implemented before they could do testing of their current task, which allowed all the team members to develop good testing habits.

Thanks to the loose communication coupling offered by the DataSocket server (see Section 3.2), Dexsim did not need to run in sync with Dexter's AI, or even require an instance of the AI to be running. Instead, it would simply react to commands whenever they were published, or do nothing in their absence. This, coupled with the ability to override Dexter's controls with the mouse when desired, and the ability to pause the simulation to add or delete static and dynamic obstacles, meant that developers rarely needed to close the simulator. Everything short of the RNDF could be changed at whim.

It was not uncommon for a user to open the simulator at the beginning of a

development session and leave it running for an entire day's worth of debugging and code changes. Because it was not a hassle to run simulations, developers did so constantly, and were able to catch large numbers of bugs that otherwise would have remained hidden until field tests. Simulation became a generative activity rather than a retroactive test, and in the process, Dexter racked up hundreds of hours of simulation testing. The developers who responded to the survey estimated that they had used Dexsim an average of 34 hours each, and over 200 hours total during 8 months of development.

All survey respondents noted that the simulator's primary value was its ability to quickly preview behavior. By giving developers the ability to immediately see the effects of new code, without having to run a field test, Dexsim prevented the team from wasting large amounts of time. The exact amount of time saved overall is difficult to judge; survey respondents chose to use phrases such as "weeks", "years", "an order of magnitude", and "incalculable", rather than giving absolute numbers. Individually, developers estimated a four to sixfold improvement in their development time versus testing on Dexter itself. Such responses are too enthusiastic to be of any numerical value, but they do suggest that a simulator should be standard equipment for a team developing an autonomous vehicle. As one respondent pointed out, many other Urban Challenge teams likely have similar simulators, and the lack of one would be a significant disadvantage.

The importance of debugging overlays is also of note. By displaying Dexter's breadcrumbs, the developers were given constant feedback about where Dexter "thought he was going," which immediately revealed most simple bugs. With the position trail added, developers could see how accurately Dexter was driving its breadcrumb trail, which made tuning Dexter's vehicle controller much easier. Visualizing Dexter's localization according to its AI and comparing it against the localization provided by Dexsim made it possible to identify confusions of which lane the vehicle was occu-

pying, and the rendering of the route plan made it possible to identify anomalies in long-term planning or goal finding. The obstacle tracking overlay made it easy to see how well Dexter's perception of obstacles matched up with their true placement, and identify errors in behavior caused by tracking bugs.

In all cases, the overlays made certain classes of bugs intuitively obvious, which might otherwise have required hours of sleuthing to track down. Every survey respondent had different stories about how visualizations helped them identify problems, with the most often-used overlays being Dexter's position trail, breadcrumb trail, and route plan. One developer estimated that 99% of all the project's bugs so far were identified in simulation.

As helpful as the simulation has been, it cannot catch all bugs. The Dexter vehicle itself, being hardware, is subject to any number of potential issues and failures, none of which are modeled in simulation. Furthermore, the simulator represents an idealized representation of Dexter's vehicle dynamics, sensors, and environment, leading to subtle differences in behavior between simulation and the actual vehicle. In most cases, the differences are negligible, but relying too heavily on simulation can sometimes mask important failures. In one example, Dexter's obstacle tracker was working well with the ideal simulated lidar data, but had difficulty with real lidar data from a particularly noisy test. In another example, inaccuracies in the simulator's friction model helped the virtual Dexter stop perfectly at stop lines, while the real vehicle would coast several feet past them. As a result of such inaccuracies, the simulator was not always as useful as it could have been, especially for testing code that relies heavily on vehicle dynamics.

All told, the simulator makes an excellent tool for catching clear and immediate failures of vehicle behavior, and serves as an excellent testing ground for active code changes. Thanks to a few fortuitous design decisions, simulation is a simple enough process that all developers can use it for continual code testing during development.

The results of simulation must be taken with a grain of salt, as they may not represent the exact behavior of the physical vehicle, and no amount of simulation can replace thorough field testing. Nonetheless, the tradeoff made between accuracy and time spent developing the simulator has been a good one. There are relatively few behavioral bugs that cannot be caught in simulation given the current model, and the team was able to have a rudimentary simulator up and running within two months of development.

It is interesting to note that the success of the simulator seems mostly due to usability design. As a consequence of the loosely structured but highly stressful nature of the Urban Challenge project, developers would simply bypass any development tools that required a great deal of setup, training, or patience to use. The simulator suffered such a fate until it was equipped with a set of useful defaults and assumptions such that it required no more than a double-click to use. New features were rarely exercised until they were simplified to the point where they required only a single sentence to describe. Such usage patterns are important to keep in mind when designing tools for a development environment such as the Urban Challenge.

## 6.3 Utility of automated testing

Dexter has run over a hundred hours of automated tests, including a 22 hour, 680 mile long marathon, but as of this writing, the automated system has yet to be adopted by the majority of the development team. The survey responses indicated several possible reasons for this:

- The team had been focused primarily on preparations for the site visit, and were more concerned with the low-level details of vehicle following, obstacle passing, and intersection precedence than the long-term stability of the vehicle.

- Tests for such low-level details are easy to create in the simulator as is, leading

many users to bypass the use of the automated testing system.

- Due to the process involved in starting and shutting down Dexter's AI, the testing system can take upwards of 30 seconds to start a test. As a result, many users simply became impatient and went back to testing in the simulator, creating tests by hand.

- The creation of tests requires specific knowledge of the scripting system used in the simulator, thus requiring test creators to learn a new programming language. Most developers have not taken the time to learn the system, therefore, very few are capable of creating new tests.

- Logs are currently taken by the testing system, but there is no means to play them back in a unified fashion. This makes it difficult to track down problems without re-running the test and watching carefully.

Interestingly, four of these items are usability issues, which again emphasizes the importance of designing such tools for the users. Future development will focus on resolving them.

The survey respondents unanimously held the opinion that while the testing system has not been very useful yet, it will be key to the team's success in the future. Testing of Dexter's basic functionality will need to continue, of course, but as the basic features become more reliable, the long-term stability of the vehicle will need to be addressed. When this happens, the testing system's ability to monitor Dexter for failures over many hours of operation will be crucial.

The bugs that the framework is anticipated to help catch fall roughly into the following categories:

1) Retroactive bugs: New functionality sometimes breaks old in unanticipated ways, and automatic functional tests ensure that such problems will be caught quickly.

2) Compound bugs: Occasionally, a situation arises where two behaviors must be reconciled at the same time, such as performing obstacle avoidance while navigating an intersection, which typically causes unexpected problems with decision making logic. Functional tests can be designed to flex these bugs, but randomized tests can catch situations that even the test designers missed.

3) Decay bugs: Sometimes, large software systems will collapse under their own weight during periods of extended usage, usually due to memory leaks or rare race conditions. Lengthy randomized tests inevitably cause such fragile systems to fail, and logs help track down which components crumbled and why.

4) Bugs from the blue: On rare occasions, problems show up intermittently with no rhyme or reason. Randomized tests usually trigger them after extensive amounts of testing, and while the system can record little information about such a failure, the fact that it exists can tip off developers to start looking for subtle bugs.

So far, the bugs the testing system has helped find fall mostly into the second category. The *Simple Obstacle* tests (see Section 5.3.3), for example, helped identify problems when obstacles were placed close to intersections, stop signs, or corners, and offered a reproducable means of demonstrating the failures.

It is also worth noting that most of the bugs found by the system were identified during the creation of the tests. The greatest benefit of having a regression testing system is that it encourages developers to put their behavioral goals into very explicit form, thus highlighting any bugs preventing those goals from being accomplished by offering a graphic and repeatable demonstration of failures. This leads to an interative development process: the developer adjusts the test to match his expectations, then adjusts the vehicle to match the test, then adjusts his expectations and assumptions based on the vehicle's performance. The cycle repeats until the developer's expecta-

tions, the test, and the vehicle's behavior are all in sync, and everything works the way it should. Once the test is "finished," it is rare for the vehicle to ever fail that test again, barring retroactive bugs introduced down the line.

# Chapter 7

# Conclusions and Future Work

Given the feedback from the development team, we can immediately conclude that the creation of a simulator has a beneficial effect for autonomous vehicle projects. In Dexter's particular case, the simulator has saved hundreds of hours of wasted field time, and dramatically impacted the quality of the resulting vehicle, as evidenced by its excellent performance in the DARPA site visit. The team believes that development without a simulator would have been a crippling handicap, and that creating such a program should be standard practice in autonomous vehicle development.

The regression testing system has potential to tease out and identify many types of difficult bugs in the future, but has not yet been fully utilized by the development team, due to a variety of usability issues. These issues will be addressed in future development, and will become less of a barrier as longer tests become necessary. The process of creating tests and getting Dexter to pass them in the first place is of great utility, as it forces developers to codify their expected behavior into explicit procedures, which can then be evaluated against the vehicle.

That said, there are a large number of potential directions for development of this system to go, both in the immediate future and long-term.

## 7.1 Immediate improvements

A necessary improvement that would help the simulator be more accurate is honing the vehicle dynamics. While the simulation is suitable for now, there is much benefit to be gained from providing better characterization of the vehicle's acceleration, deceleration, and friction. Another useful enhancement would be modeling the vehicle's tilt as it drives around turns, as this has been shown to significantly affect sensor readings such as lidar.

Simulated sensors will also need improvement. As stated before, Dexter does not currently use cameras, meaning that the design described in section 3.7.3 will need to be implemented and tuned before simulations can be performed of GPS outages and sparse RNDF waypoints. Once simulated cameras have been created, simulated GPS outages and IMU wander can be added, as well as lidar errors. This new functionality will test Dexter's sensor fusion, and lead to a whole new class of tests.

As Dexter's development continues and the vehicle becomes more functional, automated tests will need to be created to verify that Dexter conforms to the Advanced Navigation and Advanced Traffic sections of the DARPA rules document [2]. Requirements include zone navigation, parking, sparse waypoint navigation, merges and left turns into moving traffic, dynamic replanning, and freeform driving through traffic jams. These difficult tasks can all be tested using the automated testing system; it is just a matter of creating the functional tests themselves.

To be able to run long-term randomized tests of advanced traffic features, the agent AI in the simulator will need to be enhanced. Specifically, they will need to be taught how to perform u-turns, pass obstacles intelligently, navigate in parking lots, and cope with driving through moving traffic, much like Dexter will. The architecture is flexible enough to accommodate such additions, but it will nonetheless require significant amounts of new development.

With longer and more complicated tests, it will be necessary to offer log playback.

Currently, the system will record logs of significant test events and all aspects of Dexter's AI, but it is incapable of unified playback, making debugging subtle problems a difficult matter of sorting through logs by hand. A flexible, integrated playback system is a natural extension that will greatly assist developers in tracking down tough bugs.

## 7.2 Long-term research

Looking further towards the future, there are a number of directions that the automated testing ideas in this work could be taken. For example, it is easy to imagine what tests might be possible with an extremely high-quality vehicle simulator, or even a simulator for different types of vehicles entirely. One could perform tests on automated racecars, robotic lawnmowers, intelligent cargo ships, or even space-based construction drones. Automatic evaluation is a powerful tool to help developers refine behavior and catch bugs that would normally render automation of such tasks impossible.

A limiting factor of expanded usage of automated testing is the effort required in designing an evaluation system and creating the tests. If an evaluation system offers too much flexibility, it runs the risk of making test creation too difficult; however, offering too little flexibility would make it incapable of expressing all the tests that might be necessary. Thus, it would be of interest to apply machine learning techniques to the problem of behavior recognition. This way, the process of creating a test would be reduced to providing a good set of examples for successes and failures. Depending on the domain, such a procedure could be usable by anybody, not just trained users. Extensive work has been done in training robots to mimic example behavior, thus it should be possible to make a system capable of recognizing it.

Even more extreme is the possibility of creating a system that is itself capable

of exploring the space of tests for interesting cases. Such work has been done in the past with application to unmanned underwater robots [39], in which tests were parameterized as a series of acceptable ranges on computed values, and a genetic algorithm was used to explore the space of tests for cases in which the vehicle controller failed. Provided that a proper parameterization is chosen, such a system would be capable of stochastically guaranteeing the reliability of a controller, even against cases the designers had never anticipated.

# Appendix A

# Sample Test Script

A test script is written in the Lua language [26], and is loaded by the simulator at the start of a test. Once the test has started, the Update() function of the test script will be called six times per second to perform the evaluation. The return value of the Update() function determines whether the test is passed, failed, or needs to continue. The test script below is used for the u-turn functional test (see Section 5.3.2);

Every object in the object file (see Section 5.3), has a name which can be referenced from the test script. In this example, "sensor_out", "sensor_in_uturn", and so forth refer to sensors placed at various positions around the map (see Figure A.1). Dexter can be referenced as "dex" in scripts.

Driving evaluation criteria (Section 5.2) are exposed to scripts as functors, making extensive use of Lua's treatment of functions as objects. For example, "f_hit_sensor(*sens*)" in the example script will return a function that will return true whenever Dexter is in contact with the sensor named *sens*, and false otherwise.

The test evaluation itself is driven by a state machine. Whenever the state machine is updated, it will pronounce the test a success if it transitions into the "success" state, or a failure if it transitions into the "failure" state. Otherwise, it will let the test continue.

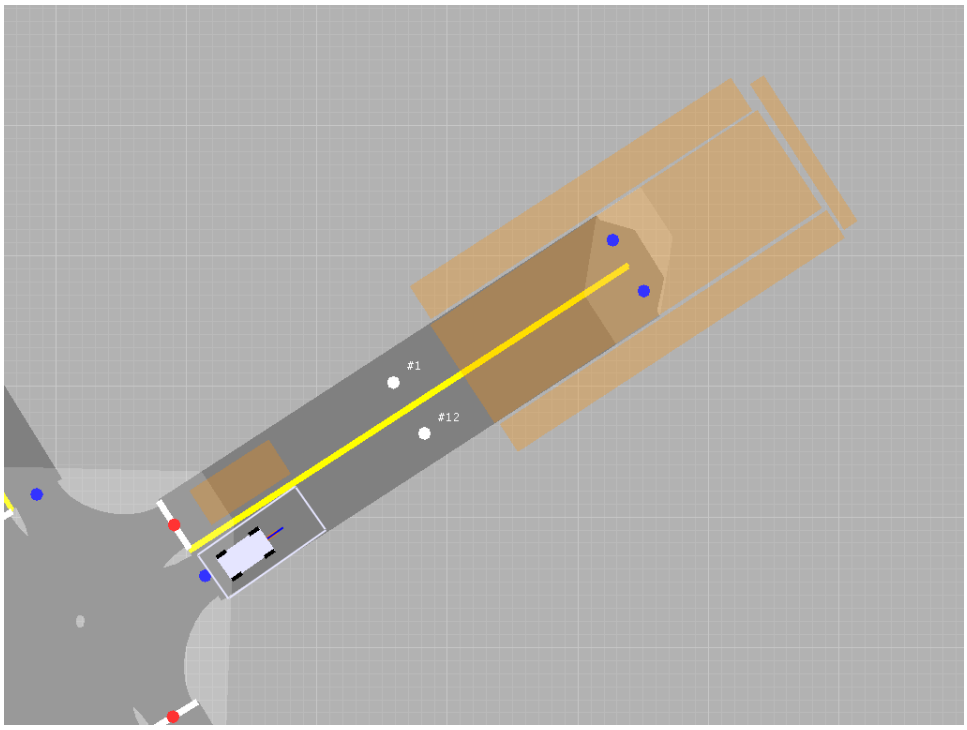



Figure A.1: A zoomed-in view of the u-turn test. The sensors are: sensor_end on the far end of the segment, sensor_right and sensor_left on their respective sides of the segment, sensor_in_uturn covering the u-turn area, and sensor_out in the lane of traffic opposite Dexter.

The machine is specified as a series of edges, in the format "{start state, condition function, end state}", as well as a set of so-called "instafails", which will cause the machine to transition into the failure state regardless of its current state. Instafails are commonly driving evaluation criteria that spell doom for Dexter, such as losing localization (i.e. running off the road) or colliding with an object. In the u-turn test, instafails are also used to fail the test if Dexter leaves the specified u-turn area.

The u-turn test will succeed if Dexter travels into the u-turn area (contacts sensor_in_uturn), then makes it out of the area again and contacts the goal sensor (sensor_out). The only way for Dexter to hit both sensors in order while avoiding the sensors surrounding the u-turn area is to perform a proper u-turn.

Screenshots are taken whenever a state machine changes states; thus, screenshots will be taken whenever Dexter enters the u-turn, hits the goal sensor, or triggers an

instafail. For this test, a second state machine has been set up with no bearing on the test results, which alternates between states whenever Dexter changes direction. This generates a screenshot at each stage of the u-turn, allowing the developer reviewing the test to examine Dexter's performance at each stage of the turn.

---

```
-- Test: u-turn
-- Author: Christian Miller
-- Date: 5/26/07
-- Description: Verifies that dexter performs a u-turn without
-- going outside of bounds.

-- condition and instafail definitions

hit_left = f_hit_sensor(sensor_left)
hit_right = f_hit_sensor(sensor_right)
hit_end = f_hit_sensor(sensor_end)
in_uturn = f_hit_sensor(sensor_in_uturn)
hit_out = f_hit_sensor(sensor_out)

-- state machine definitions

tm = StateMachine:new("start", "vehicle")

instafails = {
    ifail_lost_localization_5s,
    ifail_stop_and_stare_10s,
    ifail_runtime_10m,
    ifail_collided,
    {hit_left, "Ran off road during u-turn"},
    {hit_right, "Ran off road during u-turn"},
    {hit_end, "Ran off road during u-turn"},
}

-- [currstate (str), condition (bool fn), newstate (str)]
edges = {
    {"start", in_uturn, "in_uturn"},
    {"in_uturn", hit_out, "success"},
}

tm:addInstafails(instafails)
tm:addEdges(edges)

direction_machine = StateMachine:new("forward", "direction")

function going_forwards()
    if (dex.speed > 0.0) then
        return true
    else
        return false
    end
```

```
end

function going_backwards()
    if (dex.speed < 0.0) then
        return true
    else
        return false
    end
end

direction_edges = {
    {"forward", going_backwards, "backward"},
    {"backward", going_forwards, "forward"},
}

direction_machine:addEdges(direction_edges)

function Update()
    ret = tm:update()

    direction_machine:update()

    return ret
end
```

---

# Bibliography


[1] Defense Advanced Research Projects Agency. Urban Challenge overview. Available online: *http://www.darpa.mil/grandchallenge/overview.asp*, 2007.

[2] Defense Advanced Research Projects Agency. Urban Challenge rules. Available online: *http://www.darpa.mil/grandchallenge/rules.asp*, 2007.

[3] Defense Advanced Research Projects Agency. Urban Challenge teams list. Available online: *http://www.darpa.mil/grandchallenge/teamlist.asp*, 2007.

[4] S. Alles, C. Swick, S. Mahmud, and F. Lin. Real time hardware-in-the-loop vehicle simulation. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, pages 159–164, May 1992.

[5] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of a hybrid system model of multiple merge junction highways and semi-automated vehicles. In *Proceedings of the IEEE conference on Systems, Man, and Cybernetics*, October 1997.

[6] T. Barrera, A. Hast, and E. Bengtsson. Minimal acceleration Hermite curves. In K. Pallister, editor, *Game Programming Gems 5*, pages 225–231. Charles River Media, Hingham, Massachusetts, 2005.

[7] S. Brennan and A. Alleyne. A scaled testbed for vehicle control: The IRS. In *Proceedings of the IEEE International Conference on Control Applications*, pages 327–332, August 1999.

[8] F.P. Brooks. *The mythical man-month.* Addison-Wesley, 1995.

[9] Team Case. Urban Challenge wiki: Dexsim documentation. Private team wiki: *http://reducto.case.edu/projects/urbanchallenge/wiki/DexsimDocs*, 2007.

[10] G.L. Chang and T. Junchaya. Simulating network traffic flows with a massively parallel computing architecture. In *Proceedings of the 25th conference on Winter Simulation*, pages 762–770, December 1993.

[11] R. Deyo, J.A. Briggs, and P. Doenges. Getting graphics in gear: graphics and dynamics in driving simulation. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 317–326, August 1988.

[12] Polyphony Digital. Gran Turismo 4. Playstation 2 video game, 2005.

[13] P.A.M. Ehlert and L.J.M. Rothkrantz. Microscopic traffic simulation with reactive driving agents. In *Proceedings of the IEEE Intelligent Transportation Systems Conference*, pages 860–865, August 2001.

[14] A. Elci and A. Zambakoglu. City traffic simulation package and its utilization. *ACM SIGSIM Simulation Digest*, 13(1–4):7–11, January 1982.

[15] D. Beazley et al. SWIG. Available online: *http://www.swig.org*, 1996–2007.

[16] S. Thrun et al. Stanley: the robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, 2006.

[17] W. Newman et al. Team Case and the 2007 DARPA Urban Challenge. Submitted to DARPA for site visit, June 2007.

[18] R. Finkel and J.L. Bentley. Quad trees: a data structure for retrieval of composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[19] G. Genta. *Motor Vehicle Dynamics: Modeling and Simulation*. World Scientific, 1996.

[20] D.L. Gerlough. Simulation of freeway traffic by an electronic computer. In *Proceedings of the Highway Research Board*, pages 543–547, 1956.

[21] H.H. Goode, C.H. Pollmar, and J.B. Wright. The use of a digital computer to model a signalized intersection. In *Proceedings of the Highway Research Board*, pages 548–557, 1956.

[22] E.G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 205–214, 1992.

[23] H. Holzmann, O. Nelles, and R. Isermann. Vehicle dynamics simulation based on hybrid modeling. In *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 1014–1019, September 1999.

[24] A.R.W. Huang and C. Chen. A low-cost driving simulator for full vehicle dynamics simulation. *IEEE Transactions on Vehicular Technology*, 51(1):162–172, January 2003.

[25] M.P. Hunter, R.M. Fujimoto, W. Suh, and H.K. Kim. An investigation of real-time dynamic data driven transportation simulation. In *Proceedings of the 38th conference on Winter Simulation*, pages 1414–1421, December 2006.

[26] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[27] National Instruments. LabVIEW 8.2 Program Documentation. Online help files accessible in the LabVIEW development environment, 2007.

[28] J.H. Katz. Simulation of a traffic network. *Communications of the ACM*, 6(8):480–486, August 1963.

[29] K.R. Laughery, T.E. Anderson, and E.A. Kidd. A computer simulation model of driver-vehicle performance at intersections. In *Proceedings of the 22nd national ACM conference*, pages 221–231, January 1967.

[30] W.S. Lee, J.H. Kim, and J.H. Cho. A driving simulator as a virtual reality tool. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 71–76, May 1998.

[31] R. McHaney. Bridging the gap: transferring logic from a simulation into an actual system controller. In *Proceedings of the 20th conference on Winter Simulation*, pages 583–590, December 1988.

[32] J.V. Mierlo and G. Maggetto. Innovative iteration algorithm for a vehicle simulation program. *IEEE Transactions on Vehicular Technology*, 53(2):401–412, March 2004.

[33] OKTAL. SCANeR 2 Driving Simulation Software. Available online: *http://www.scaner2.com*, 2003.

[34] P. Paruchuri, A.R. Pullalarevu, and K. Karlapalem. Multi agent simulation of unorganized traffic. In *Proceedings of the joint conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 176–183, July 2002.

[35] H.J. Payne. A critical review of a macroscopic freeway model. In *ASCE Research Directions in Computer Control of Urban Traffic Systems*, pages 251–265, 1979.

[36] B. Raney, N. Cetin, A. Vollmy, and K. Nagel. Large scale multi-agent transportation simulations. In *Proceedings of the Annual Congress of the European Regional Science Association (ERSA)*, August 2002.

[37] K.A. Redmill and U. Ozguner. VATSIM: a vehicle and traffic simulator. In *Proceedings of the IEEE/IEEJ/JSAI International Conference on Intelligent Transportation Systems*, pages 651–661, October 1999.

[38] R. Schmidt, H. Weisser, P. Schulenberg, and H. Goellinger. Autonomous driving on vehicle test tracks: overview, implementation, and results. In *Proceedings of the IEEE Intelligent Vehicles Symposium*, pages 152–155, October 2000.

[39] A. Schultz, J. Grefenstette, and K. De Jong. Adaptive testing of controllers for autonomous vehicles. In *Proceedings of the IEEE Symposium on Autonomous Underwater Vehicle Technology*, pages 158–164, June 1992.

[40] F.P. Testa and M. Handelman. Simulation of Garland, Texas, vehicular traffic using current and computed optimal traffic settings. In *Proceedings of the 6th conference on Winter Simulation*, pages 580–589, December 1973.

[41] D.J. Verburg, A.C.M. van der Knaap, and J. Ploeg. VEHIL: developing and testing intelligent vehicles. In *Proceedings of the IEEE Intelligent Vehicles Symposium*, volume 2, pages 537–544, June 2002.

[42] D. Ward, T. Bertram, and M. Hiller. Vehicle dynamics simulation for the development of an extended adaptive cruise control. In *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 730–735, September 1999.

[43] K. Yamada and T.N. Lam. Simulation analysis of two adjacent traffic signals. In *Proceedings of the 17th conference on Winter Simulation*, pages 454–464, December 1985.