Multiagent Interactions in Urban Driving

Patrick Beeson, Jack O'Quin, Bartley Gillan, Tarun Nimmagadda, Mickey Ristroph, David Li, Peter Stone

***Abstract*—In Fall 2007, the US Defense Advanced Research Projects Agency (DARPA) held the Urban Challenge, a street race between fully autonomous vehicles. Unlike previous challenges, the Urban Challenge vehicles had to follow the California laws for driving, including properly handling traffic. This article presents the modular algorithms developed largely by undergraduates at The University of Texas at Austin as part of the Austin Robot Technology team. We emphasize the aspects of the system that are relevant to multiagent interactions. Specifically, we discuss how our vehicle tracked and reacted to nearby traffic in order to allow our autonomous vehicle to safely follow and pass, merge into moving traffic, obey intersection precedence, and park.**

***Index Terms*—autonomous vehicles, interactive systems, sensor fusion**

I. Introduction

The DARPA Urban Challenge successfully demonstrated the possibility of autonomous vehicles driving in traffic. The main difference between the Urban Challenge and previous demonstrations of autonomous driving was that in the Urban Challenge, robots needed to be prepared to interact with other vehicles, including other robots and human-driven cars. As a result, robust algorithms for *multiagent* interactions were essential. This article introduces Austin Robot Technology's autonomous vehicle (Figure 1), one of 89 entries in the Urban Challenge. The main contribution is a detailed description of the multiagent interactions inherent in the DARPA Urban Challenge and how our team addressed these challenges.

Austin Robot Technology's entry in the Urban Challenge had two main goals. First, the team aimed to create a fully autonomous vehicle that is capable of safely and robustly meeting all of the criteria laid out in the DARPA Technical Evaluation Criteria document [1], including the multiagent interactions that we emphasize in this article. Second, and almost as important, the team aimed to educate and train members of the next generation of computer science and robotics researchers by encouraging and facilitating extensive participation by undergraduate programmers.

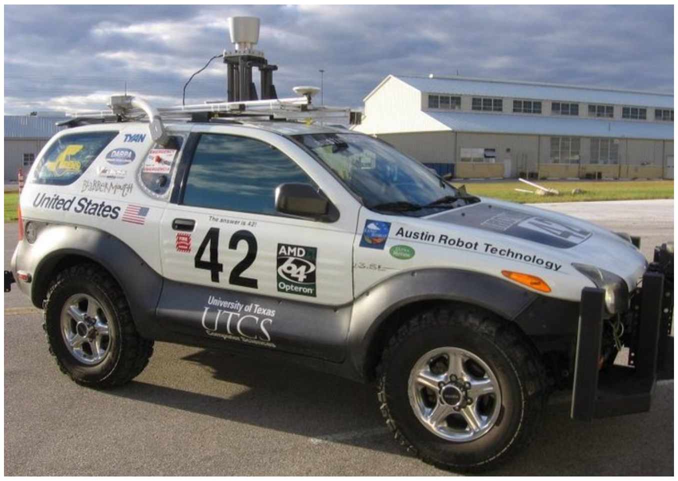

Fig. 1. **The vehicle platform is a 1999 Isuzu VehiCross.** All actuators were developed and installed by the Austin Robot Technology team volunteers.

This article emphasizes the first goal; however, the second goal biases our algorithms to be as straightforward as possible. Nonetheless, the algorithms described here are reliable enough for our team to have placed among the top twenty-one teams at the Urban Challenge National Qualifying Event (NQE). With slightly more luck from prototyped hardware and with a bit more time for testing and verifying code, we believe our autonomous vehicle could have competed well in the final race along with the eleven finalists.

The remainder of this article is organized as follows. Section II provides a brief history of the DARPA autonomous driving challenges. Section III summarizes our specific vehicle platform, including both the hardware and software systems. Section IV presents the main contribution, namely our approach to the multiagent challenges of driving in traffic. Section V summarizes our experience at the Urban Challenge event itself, and Section VI concludes.

II. Background

The first DARPA Grand Challenge was held in 2004 as a competition between academics, military contractors, and amateurs to win a 150 mile autonomous race through the desert. DARPA offered prize money in an effort to spur technological advancements that would lead to one-third of the United States' ground military vehicles being autonomous by 2015. That year, none of the teams made it further than 8 miles. In 2005, the Grand Challenge was held again, and the

Peter Stone is an Associate Professor at The University of Texas at Austin, Department of Computer Sciences. He instructed a Spring 2007 undergraduate course "cs378, Autonomous Vehicles - Driving in Traffic" where undergraduate students became familiar with the robotics problems related to the Urban Challenge.

Patrick Beeson is a doctoral candidate at UT Austin, Department of Computer Sciences. He is the project lead in charge of overseeing undergraduate research and software development on the autonomous vehicle platform. He is teaching the Spring 2008 cs378 autonomous driving course.

Jack O'Quin is a retired IBM operating systems developer and a regular contributor to open source audio programs for Linux. He is a volunteer ART team member that provided immeasurable time and energy designing, programming, and testing most of the control infrastructure. He is also a UT Austin alumnus.

Bartley Gillan, Tarun Nimmagadda, Mickey Ristroph, and David Li were students in Dr. Stone's Spring 2007 course. They all participated in the 2007 DARPA Urban Challenge National Qualifying Event.

course was completed by five teams with Stanford University's team finishing first [2].

Austin Robot Technology (ART) formed as a collection of technologists interested in participating in the 2005 race. In their spare time and with much of their own money, they created an autonomous vehicle that made it to the semi-finals in 2005.

In 2007, DARPA held the Urban Challenge in an attempt to have vehicles race to complete a 60 mile "urban" course in under 6 hours. Carnegie Mellon's Tartan racing team won the race, and six teams completed the course, though only four of these finished under the 6 hour deadline. Unlike the previous races, this race simulated urban (actually more suburban) driving, where the autonomous vehicles had to obey traffic laws and interact with other vehicles on the road.

For the 2007 competition, Austin Robot Technology teamed up with The University of Texas at Austin (UT Austin) via Peter Stone's undergraduate course on autonomous driving. This partnership provided the team with Artificial Intelligence expertise as well as a group of excited undergraduate programmers. It provided the university with an interesting platform on which to offer invaluable undergraduate research opportunities.

The ART team made it to the National Qualifying Event (semi-finals) again in 2007, but was not among the top eleven teams chosen for the final race. Mostly this was due to a shortened development schedule and a few hardware glitches during the qualifying events. In particular the algorithms for interacting with other agents, described in this article, performed well at the event. These other agents may be other robots, or they may be human drivers. Our algorithms make no distinction.

III. Vehicle Overview

Here we give a quick overview of the vehicle's hardware and software before discussing the specific aspects of interacting with other vehicles in Section IV. More hardware details, along with an overview of the software developed by the undergraduate class, can be found in the technical report submitted to DARPA as part of the quarter-final site visit [3].

A. Hardware

The Austin Robot Technology vehicle, in its present configuration, is shown in Figure 1. It is a stock 1999 Isuzu VehiCross that has been upgraded to run autonomously. Austin Robot Technology team members spent much of their time in 2004 and 2005 adding shift-by-wire, steering, and braking actuators to the vehicle. Control of the throttle was achieved by interfacing with the vehicle's existing cruise control system.

In addition to actuators, the vehicle is equipped with a variety of sensing devices. Differential GPS, an inertial measurement unit, and wheel revolutions are combined together by the Applanix POS-LV for sub-meter odometry information. Sick LMS lidars are used for precise, accurate planar range sensing in front and behind the vehicle. A Velodyne High Definition Lidar (HDL) provides 360° 3D range information (see Figure 2).

The vehicle contains three machines with a total of ten AMD Opteron cores, which provides more than enough processing power for the Urban Challenge domain. Additionally a 24V alternator provides power to computers, heavy-duty actuators, and some perception devices, while the vehicle's existing 12V system powers many plug in devices such as network switches, the safety siren, and the safety strobe lights.

B. Software

At the DARPA Urban Challenge, teams were given models of the roadways via Route Network Definition Files (RNDFs). An RNDF specifies drivable road segments, GPS waypoints that make up the lanes in each segment, valid transitions between lanes, stop sign locations, and lane widths. The direction of travel for each lane is provided through the ordering of the GPS waypoints for that lane. An RNDF may also specify any number of "zones", which are open areas of travel with no specific lane information provided. A polygonal boundary consisting of GPS waypoints is given for each zone. Zones are used to describe parking lots, and can have designated parking spots at specified locations. Each RNDF file has one or more associated Mission Data Files (MDFs) that provide an ordered list of GPS waypoints that the vehicle must drive over [5].

Thus, the challenge is to design a system that interacts with the perceptual and actuator devices at the low-level in order to produce a vehicle that can follow the high-level map given by the RNDF and MDF (Figure 3). The vehicle must do so while traveling in legal lanes, using blinkers, following traffic laws, and not colliding with any obstacles.

Key to our success in the DARPA Urban Challenge is the observation that avoiding collisions and obstacles while navigating within a lane is a simpler problem then generic robot navigation. For example, consider driving down a curvy road with another vehicle traveling towards you in an oncoming lane. The other vehicles will pass within perhaps a meter of you, yet the risk of collision is minimal. Without using lane information, the slightest curve in the road could cause even a highly accurate tracking algorithm to predict a head-on collision, necessitating complex path planning. However, under the assumption that any obstacle being tracked will stay within its lane, or within a lane to which it can legally switch, it is safe to continue on in one's own lane. While seemingly risky, this is the implicit assumption of human drivers, and a necessary assumption for any autonomous agent expected to drive like a human.

We discuss navigation and perception using lane information in Section IV. We also discuss the behaviors used for non-lane navigation in parking areas. But first we introduce the software architecture to ground the concepts and ideas that we utilize in modeling other vehicles on the road.

1) Commander: The Commander module operates at the highest level of reasoning. Using the graph built from the RNDF, it determines the optimal route from the current location of the vehicle to the next two goals given by the MDF. We use A* search to find the shortest time path between waypoints, penalizing graph edges that we know are at intersections or that go through parking lots.

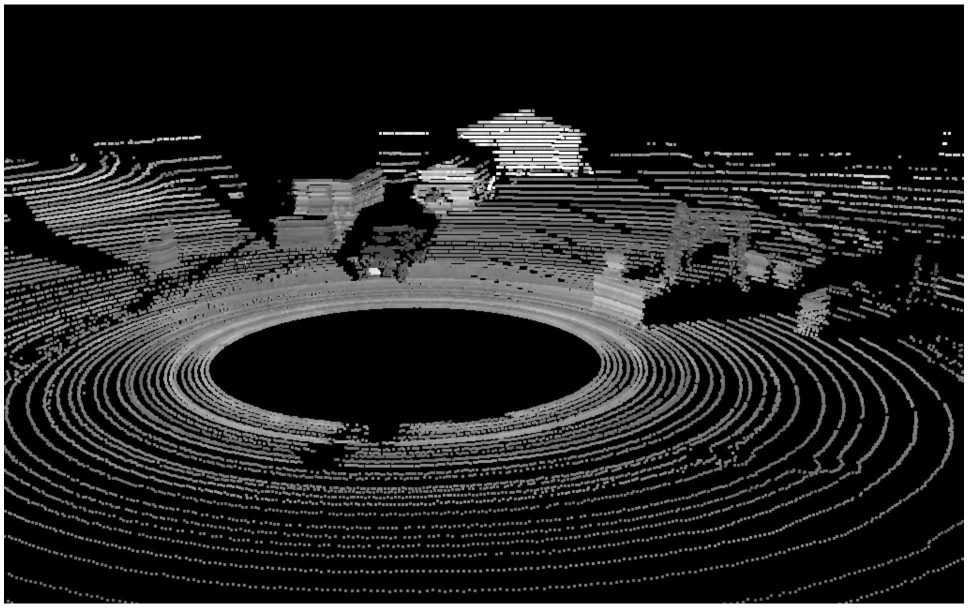

Fig. 2. **Sample snapshot of Velodyne HDL data.** The Velodyne HDL uses lidar technology to return a 360° 3D point cloud of data over a 24° vertical window. Intensity of distance returns are correlated with pixel intensity. Notice that buildings and a truck can be seen among the obstacles.

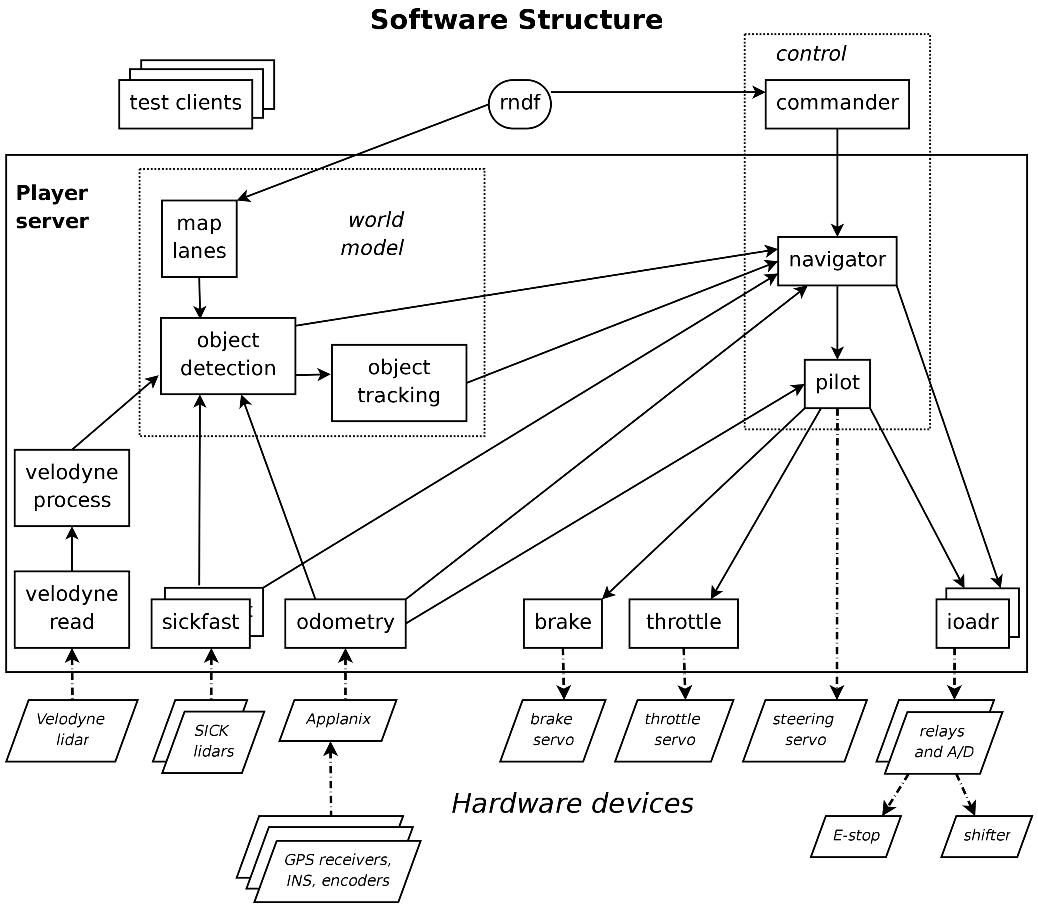

Fig. 3. **Software architecture.** We utilize the Player [4] robot server as our interface infrastructure. Here each module is a separate server, running in its own thread. There is a single client which connects to the top-level server to start the system. Hardware interfaces are shown at the bottom. Perceptual filters and models are on the left side of the diagram while control—planning (commander), behaviors (navigator), low-level actuation control (pilot)—are shown on the right side.

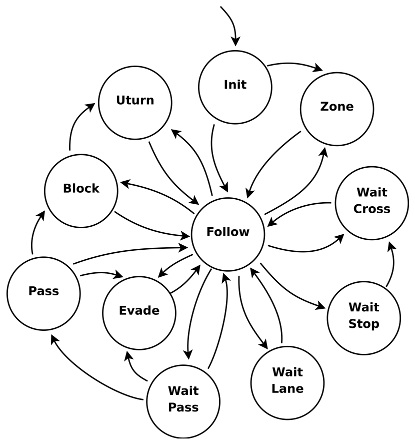

Fig. 4. **A simplified illustration of the vehicle's Run state machine.** Most of the time, the vehicle is in the Follow state. The Navigator module can decide to pass a stalled vehicle, if a passing lane exists, without having the higher-level Commander module replan. Entering other states depends on current traffic conditions or whether the vehicle must enter a parking zone. Many of these are detailed in Section IV.

2) Navigator: The Navigator module is essentially a hierarchical state machine that runs a variety of behaviors on the vehicle. At the highest level is a simple run/pause/disable machine. The Run state contains another state machine, which is illustrated in Figure 4. When running, Navigator uses the next several waypoints in the plan it receives from Commander and runs the appropriate behavior.

Most of the time the vehicle is following its current lane, though many of the interesting behaviors from a multiagent point-of-view occur in the other control states. Due to a shortened development time[1], we did not utilize a traditional model-based route planner for most behaviors. Instead each behavior here uses a snapshot of the world at each cycle to quickly compute a desired travel and turning velocity. The *Pilot* module transforms this velocity command into low-level throttle, brake pressure, and steering angle commands.

3) Velodyne HDL Processing: The Velodyne High Definition Lidar (HDL) provides around one million points of data every second. Following our design principle of trying simple algorithms first, we use "height-difference" maps to identify vertical surfaces in the environment without the need for computationally intensive algorithms for 3D, real-time modeling [6]. Our solution can be thought of as a "slimmed down" version of the terrain labeling method performed by the 2005 Grand Challenge Stanley team [2]. At each cycle (i.e. every complete set of 360° data), we create a 2D (x, y) grid map from the 3D point cloud, recording the maximum and minimum z (vertical) values seen in each grid cell.

Next, a simulated lidar scan is produced from the 2D grid—the algorithm casts rays from the sensor origin, and an obstacle is "detected" whenever the difference between the max and min z values is above a threshold. The result is a 360° 2D simulated lidar scan, which looks very similar to the data

[1]Because the undergraduate class' code was designed as a prototype to pass the regional site visit, we overhauled the software completely between July and the October NQE events.

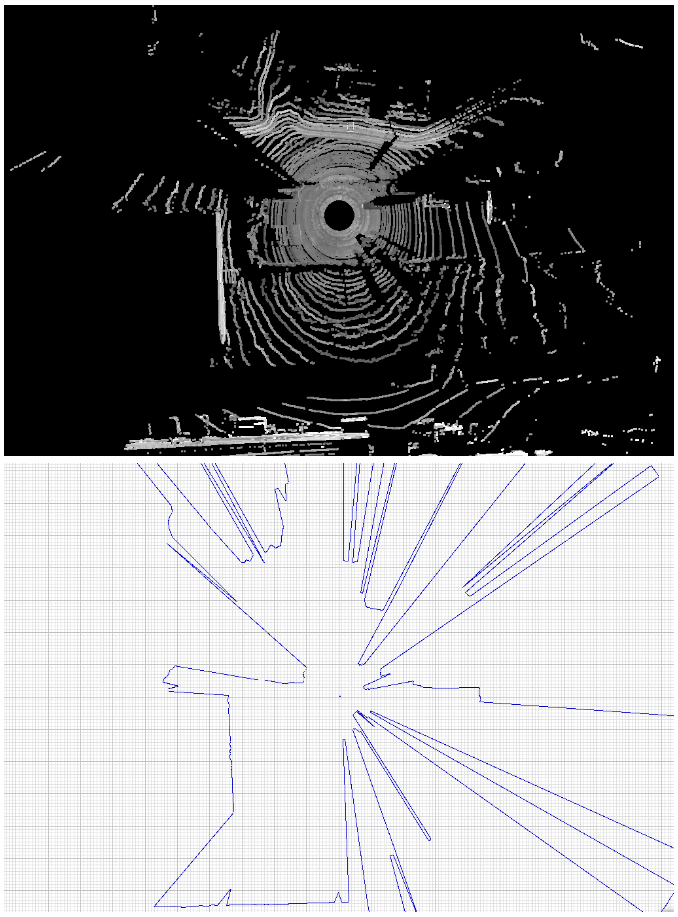

Fig. 5. **Processed Velodyne lidar information.** Raw Velodyne HDL point cloud (bird's eye view of Figure 2 is shown) gets processed into a 2D scan. Notice corresponding features between the two data formulations. This method creates possible occlusions, but allows fast, efficient processing of the million points per second the Velodyne HDL transmits.

output by the SICK lidar devices (see Figure 5); however, this 2D lidar scan is non-planar and only returns the distances to the closest obstacles that have a predetermined vertical measure (currently, 25 cm tall with at least a 45° slope).

4) MapLanes: Initially conceived as a temporary substitute for visual lane recognition, the MapLanes module has become an important piece of our current software infrastructure. MapLanes is designed to parse an RNDF and to create a lane map in the global Cartesian coordinate system provided by the Applanix odometry. Its dual purposes are i) to create lane information useful for vehicle navigation and ii) to provide a way of classifying range data as being in the current lane, in an adjacent lane, or off the road entirely.

The MapLanes road generation algorithm uses standard cubic splines [7], augmented with a few heuristics about roadways, to connect the RNDF waypoints (Figure 6). We first create a C1 continuous Hermite spline [7] from the discrete series of waypoints that define a lane in the RNDF. We chose the Hermite form because its representation allows us to control the tangents at the curve end points. We can then specify the derivatives at the waypoints in such a way that the spline that we create from these curves has the continuity

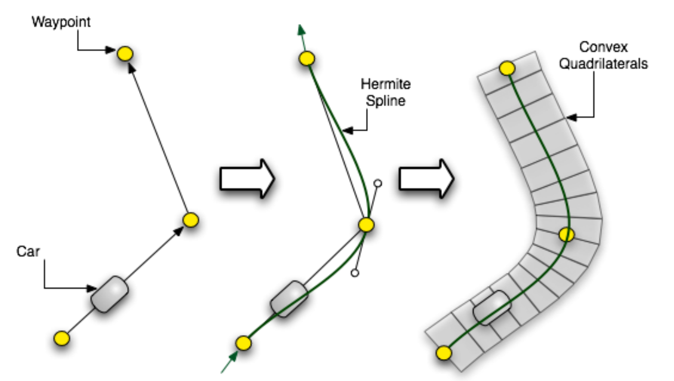

Fig. 6. **Guessing the road shape.** Given waypoints that define a lane, a cubic spline gives a rough approximation of the road. We utilize a few non-standard heuristics to detect straight portions of roadway, which improves the spline tangents at each point. The quadrilaterals that we utilize for the MapLanes module are built on top of the curves. The collection of these labeled quadrilaterals are called *polygons* in our current implementation jargon.

properties we desire.

We then convert the spline from a Hermite basis to the Bézier basis. This step allows us to use any of a large number of algorithms available to evaluate Bézier curves. At this time, we express the curve in terms of nth degree Bernstein polynomials which are defined explicitly by:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \qquad i = 0, \ldots, n.$$

Any point on the curve can be evaluated by:

$$b^n(t) = \sum_{j=0}^{n} b_j B_j^n(t).$$

We set $n = 3$. The coefficients b_j are the Bézier control points.

This spline, along with the lane widths defined in the RNDF, gives the vehicle an initial, rough guess at the shape of the roadway (see Figures 7&8). Each lane is then broken into adjacent quadrilaterals (referred to as polygons in our software) that tile the road model. These quadrilaterals are passed through a Kalman filter where vision-based lane detection can fine tune the lane model or overcome incorrect GPS offsets.[2]

5) Polygon Operations: Each polygon created by MapLanes is placed into a data structure that contains, among other information, the Cartesian coordinates of its four corners, the midpoint of the polygon, the length, the width, the heading of the lane at the polygon's midpoint, the waypoints which the polygon lies between (thus the lane the polygon lies on), the type of lane markings which lie on its boundaries, and a unique ID. An ordered list of polygons for each lane is maintained and published to the perceptual and control modules discussed in Section IV.

We created a polygon library that provides numerous methods for extracting information from the ordered list of polygons. This library performs the bulk of the computation pertaining to the current state of the world surrounding the vehicle. Examples include: filtering out range data not on the road, determining distances along curvy lanes, and determining which lanes will be crossed when passing through an intersection.

[2]For the NQE event, the decision was made to run without vision, as issues such as false positives (shadows, sun flares) and illumination changes are still not adequately handled by our software. To our knowledge only two of the six teams to complete the final course strongly relied on visual lane tracking.

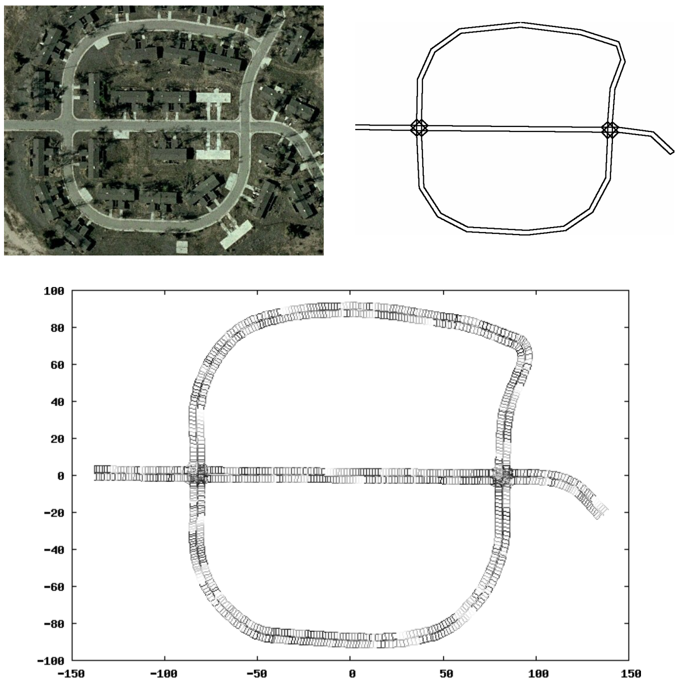

Fig. 7. **MapLanes model of the NQE Area C.** The MapLanes module estimates the continuous shape of the roadway from a simple connected graph of the course extracted from the provided RNDF. Above left is a satellite image of the course. Above right is a graph of the connected RNDF waypoints. The bottom diagram illustrates the MapLanes data structure.

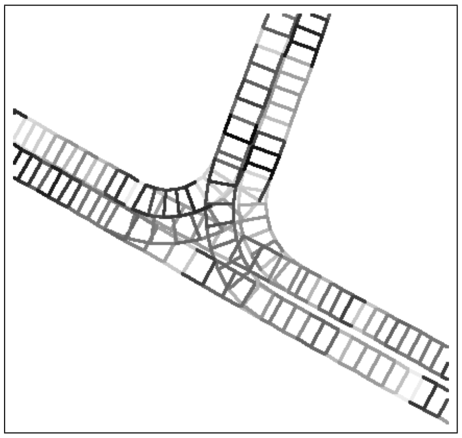

Fig. 8. **Polygons at an intersection.** Lane polygons model each lane while overlapping transition polygons fill in the connections between lanes. This overlapping is how the vehicle determines which lanes to observe when pulling into an intersection: transition polygons of the current lane overlap polygons of lanes that must be assigned obstacle trackers.

IV. Multiagent Interactions

The overall hardware and software architecture sketched in Section III forms the substrate system for the main research reported in this article, namely the vehicle's multiagent interactions. There are two main components to the multiagent interactions that we consider. First, in Section IV-A, we focus on the ability to *perceive* and represent other vehicles. Second, in Section IV-B, we detail our vehicle's *behaviors* based on those perceptions.

A. Perception

We describe the perception necessary for multiagent interaction in two parts. First, we describe obstacle tracking in a lane using the range data received by lidar sensors. Second, we define a set of *observers*, each of which instantiates an obstacle tracker on the appropriate set of nearby lanes, and reports the situation to the control modules. In a sense, each observer is like a "back-seat driver" that continuously calls out the appropriate information: "unsafe to merge left," "your turn to go," etc.

1) Obstacle Tracking: For autonomous driving, a robot needs good approximations of the locations and velocities of surrounding traffic. Recent approaches to obstacle tracking have often utilized a Cartesian-based occupancy grid [8] for spatial and temporal filtering of obstacles. This occupancy grid is used to estimate the state $X = (x, y, \theta)$, $\dot{X} = (\dot{x}, \dot{y}, \dot{\theta})$, and sometimes the shape or extent of surrounding obstacles [9].

Our design differs from omni-directional tracking in that we utilize the MapLanes model of the roadway to solve the obstacle tracking problem. The key insight in simplifying the problem of obstacle tracking in the urban driving domain is that the vehicle only needs to track obstacles that are within lanes.[3] We further reduce the dimensionality of the problem by observing that it is sufficient to track the velocity of each obstacle only along the lane.

By partitioning the world into lanes and defining an order on the quadrilaterals comprising each lane, we impose a linearization on the space. Thus we can easily track the distance to the closest obstacle in each direction of a lane. The distance function in this space is not Euclidean but rather an approximation of the lane distance as reported by the polygon library. The distance computation between two points first projects each point onto the midline of the lane begin tracked. The lane distance is approximated by using the summation of piecewise line segments connecting the lane polygons.

For a particular lane and direction (in front of or behind our vehicle), we build an obstacle tracker using lidar data. The obstacle tracker for each lane receives a laser scan which specifies the positions of all obstacles that are within its lane. It then iterates through all these obstacles to find the closest one in the specified direction. It maintains a history of these nearest observations using a queue of fixed size. We then filter out noise using acceleration and velocity bounds and estimate the relative velocity from the queue of recent observations. Figure 9 illustrates an experiment where our vehicle was sitting still and tracking another vehicle driving in an adjacent lane.

One advantage of this obstacle tracking approach is that it scales linearly with respect to the number of lanes the vehicle attends to, not the number of obstacles. We found that, even though it had a lossy model of the world, it was powerful enough to complete the various requirement that DARPA outlined in the Technical Evaluation Criteria, and therefore sufficient for most urban driving tasks. During the NQE event, we used a 10 frame queue of distances, which reduced the risk of inaccurate measurements from sensor noise, but introduced a lag of about 1 second in the measurements given the 10Hz lidar updates (see Figure 10). We accepted this lag as a trade-off in favor of robust, safe driving over aggressive behavior.

2) Observers: We define an observer as an object focusing on a subset of MapLanes polygons and lidar range data to determine whether that specific area of the world is deemed free from traffic by our vehicle. Think of an observer as a back-seat driver in charge of reporting whether a specific section of the road is safe to enter or occupied by traffic or obstacles. Each observer sends its report to Navigator every cycle. Navigator chooses which observers are appropriate to use for decision making given its current plan. The primary information each observer provides is a single bit, which represents whether its area is clear or unclear. When an observer reports "unclear," it also provides useful quantitative data such as estimated time to the nearest collision.

Our system uses six observers: Nearest Forward, Nearest Backward, Adjacent Left, Adjacent Right, Merging, and Intersection Precedence. In order for the observers to track vehicles and other objects on the road, they need information about the nearby road lanes. Using the current vehicle odometry and the polygon library, each observer determines whether it is applicable or not based on whether lanes exist relative to the vehicle's pose.

a) Nearest Forward Observer: The Nearest Forward observer reports whether the current lane is clear forward of the vehicle's pose. This perception data is from the Velodyne HDL and the front SICK lidars. This observer reports potential collisions in the current lane.

b) Nearest Backward Observer: The Nearest Backward observer is just like the Nearest Forward observer except that it looks behind the vehicle's current pose. This observer is rarely used by Navigator, as often our vehicle ignores anything approaching from behind in the current lane.

c) Adjacent Left Observer: The Adjacent Left observer reports whether the lane immediately to the left of the vehicle's current lane is safe to enter, for example to pass a stopped vehicle. If a vehicle is in the left lane but the time to collision is larger than 10 seconds, the lane is considered to be clear. Unlike the Nearest Forward and Nearest Backward observers, this observer has two trackers, one in front and one behind. It reports the most imminent threat to Navigator.

d) Adjacent Right Observer: The Adjacent Right observer is just like the Adjacent Left observer except that it looks to the lane immediately right of the vehicle. If there is no lane to the right of the observer, then the observer reports that it is not applicable.

e) Merging Observer: The Merging observer is used to check when it is safe for the vehicle to proceed across any type of intersection: driving through an intersection or turning into/across traffic. Like other observers, it uses trackers to estimate relative velocity of obstacles in lanes and makes a binary safe/unsafe decision based on collision time: $t = d/v$,

[3]The 2007 Urban Challenge specifically ruled out pedestrians or any other obstacles that might move into traffic from off the roadway.

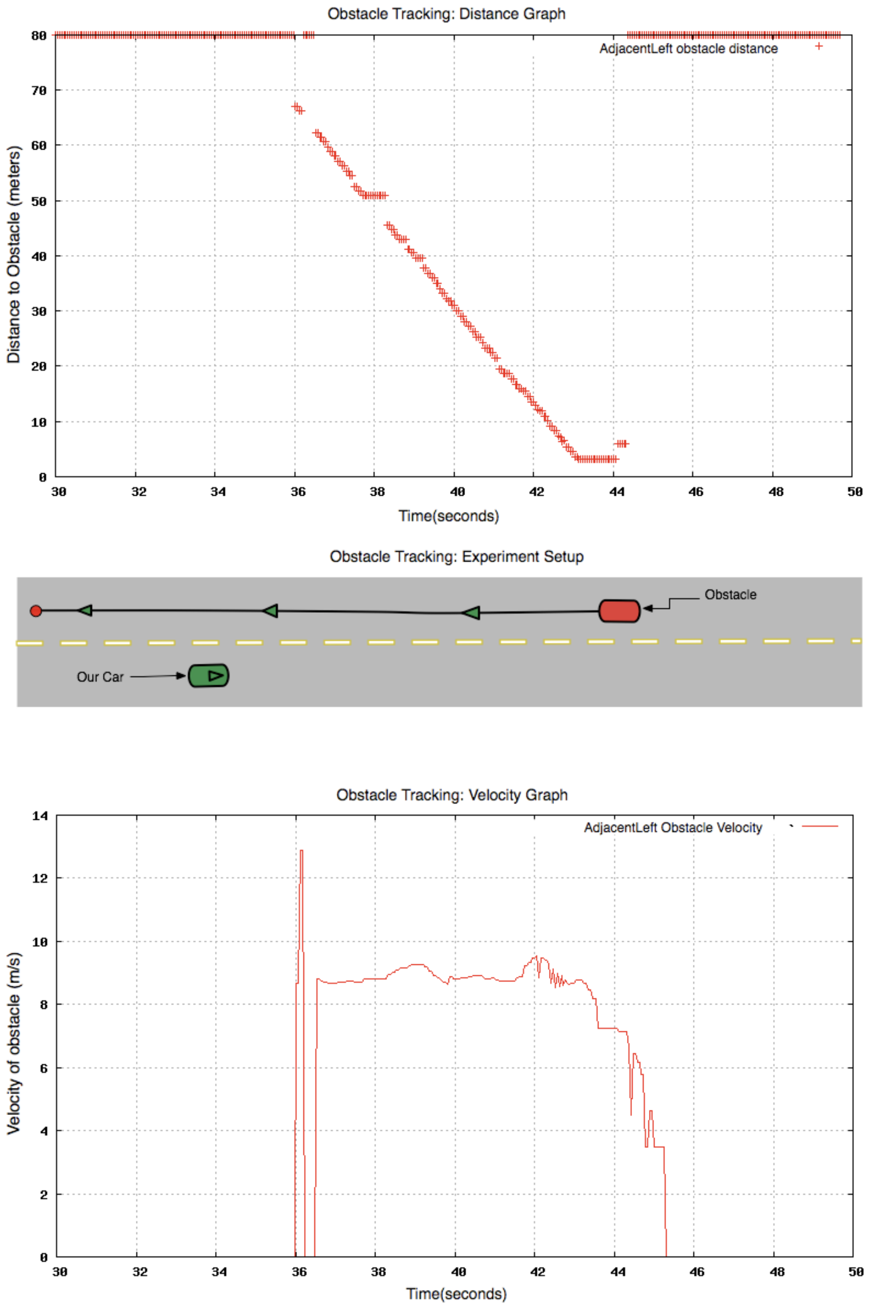

Fig. 9. **Obstacle Tracking Experiment Results.** In this experiment, our vehicle was stopped on a road and tracking another vehicle in the lane to the left. The driver of the tracked vehicle reported an estimated speed of 9 m/s. Other than a brief initial transient, the obstacle tracker accurately models the oncoming vehicle starting from about 60 meters away.

where d is the lane distance, which as described above can be computed between two obstacles in different lanes.

In merging scenarios, the observer currently checks whether all lanes the vehicle *can possibly* traverse are safe. For example, at an intersection, if it is unsafe to turn left, the vehicle will wait, even if it plans to turn right. This behavior is necessitated by the modular design of our system, in that the observers are not aware of the vehicle's plan of action. Note that the effect is more conservative behavior, which may be appropriate given the fact that other vehicles can also be autonomous, thus perhaps unpredictable.

Choosing the set of lanes to assign trackers to is a critical task. A general polygons-based solution was developed for this purpose that looks at *lane* versus *transition* polygons. Lane polygons are the quadrilaterals that are adjacent along the lane (illustrated in Figure 6), while transition polygons connect up different lanes (see Figure 8). We build a table that holds information regarding which lanes overlap each other in the nearby surround. We know a lane crosses another if any transition polygon in the current lane intersects any polygon (lane or transition) of the other lane. Thus a simple point-in-polygon algorithm can be looped over polygons to build the table.

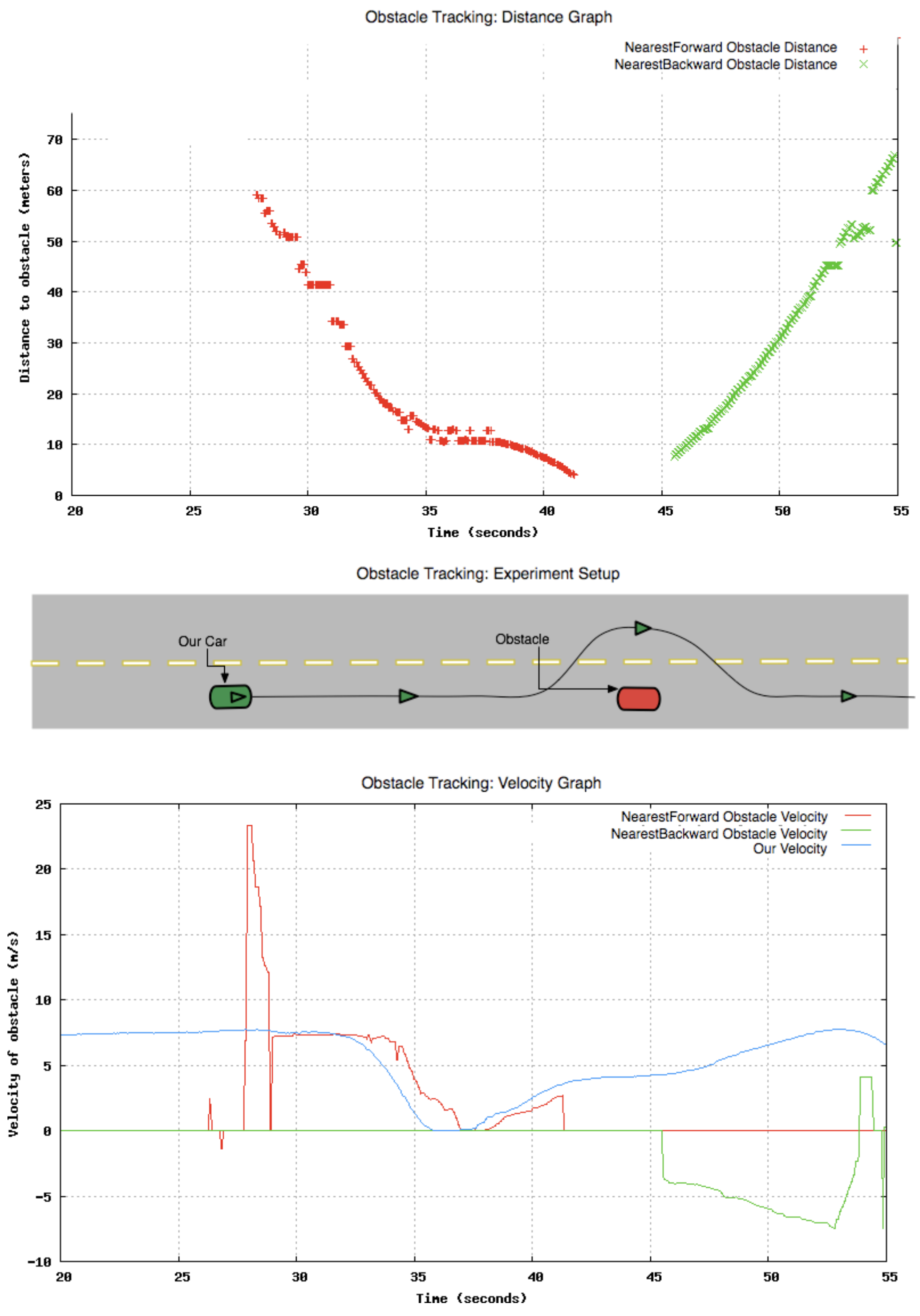

Fig. 10. **Obstacle Tracking Experiment Results.** In this setup, our vehicle drove up to another vehicle that was stopped in the same lane. Our vehicle then passed before returning to the original lane of travel. The above graphs show the relative state of the obstacle estimated by two trackers: one forward and one behind in the current lane.

f) Intersection Precedence Observer: Upon stopping at a stop sign (defined as a special waypoint in the RNDF and marked by a white stop line on the roadway), the vehicle needs to yield to other vehicles that are already stopped at the intersection. The vehicle must then proceed through the intersection when its turn arrives, even if other vehicles have arrived at the intersection later.

The Intersection Precedence observer is used to indicate whether or not it is appropriate to start moving through the intersection. By default, the observer reports that the vehicle should wait. Once it becomes the vehicle's turn to move, the observer reports that it is time to go. This observer does not rely on obstacle tracking as it can be difficult to separate vehicles that pass each other in the intersection. Instead the simple yet effective idea behind this observer is to notice which lanes at the intersection have obstacles (presumably other vehicles) waiting at the instant that our vehicle arrives, and then to notice when these obstacles disappear from each such lane.

There are 3 main components to our intersection precedence observer.

1) *Locate the other stop signs at the same intersection.* The RNDF does not explicitly define the structure of intersections. It can be deduced through complex modeling;

however, we simply locate all stops that are within 20 meters of our vehicle's stop waypoint and are in front of our vehicle.[4]

2) *Determine which stop signs have vehicles with higher priority.* Immediately when the vehicle stops, it needs to determine which stop signs are "occupied."
First, we filter all range readings that fall outside lanes that have stop signs at the intersection. Then we throw out any readings further than 2 meters of a stop sign location—stopped vehicles are supposed to be within 1 meter of the stop sign. Next, we bin all remaining range data for forward facing angles into 180 bins, each representing 1°, and we determine the closest reading in each bin to the vehicle.
The resulting 180 element vector, μ, is our model of the intersection. Most elements in μ will have a zero range (because most range readings have been filtered out); non-zero elements are assumed to indicate vehicles that arrived at the intersection before us, and thus have the right of way. To reduce false positives/false negatives, we collect this data over several cycles immediately upon arriving at the stop sign.

3) *Determine when all higher priority vehicles have gone.* On each subsequent cycle, the vehicle gathers new range readings, and produces a new 180 element vector of the closest ranges at 1° intervals, ν. We update the model of the intersection by declaring that if $\nu_i - \mu_i \geq 2$, then $\mu_i = 0$. Note that once an element of μ is zero, is remains zero. Once all elements of μ equal zero, it is our vehicle's turn to go. This procedure works even when multiple vehicles are queued up in other lanes due to the fact that a small gap always exists between a vehicle leaving and another vehicle pulling up. Note that we actually produce ν using several cycles of range data in order to handle false negatives.

B. Behaviors

Perception and tracking of other vehicles as described in Section IV-A are necessary prerequisites for designing effective multiagent behaviors. This section details the behaviors that we created to interact with traffic in the Urban Challenge domain. With good input from the perception subsystem, these behaviors can be relatively straightforward.

The behaviors are all implemented in the Navigator thread using a hierarchy of controllers. Every 50 msec cycle, Navigator creates a Pilot command indicating a desired velocity v and yaw rate ω for the vehicle. Pilot, running at the same rate in a separate thread, translates those commands into steering, brake, throttle and shifter servo motor instructions.

Each controller in the Navigator module provides an interface that can modify the next Pilot command according to the current situation. Some controllers are finite state machines, others simple code sequences. Every control method also returns a "result" code, which the calling controller often uses to determine its future states and activities.

[4]This does not handle degenerate cases, but it is suitable for most environments.

Figure 4 illustrates how these behaviors are connected in the main Navigator state machine, which is itself a controller. Its states and state transitions all have associated actions, which support the same controller interface. Most Navigator controllers follow lanes in the road network, utilizing MapLanes data to orient the vehicle within its lane. The major exception is the "Zone" controller, which operates in unstructured areas such as parking lots.

1) Follow Lane: The FollowLane controller, designed to follow an open lane, is the behavior that is executed most often. It is executed in the Follow state which appears at the center of Figure 4. Due to a shortened development period between the regional site visit and the NQE, we chose not spend time developing a planner that models vehicle dynamics. Instead we implemented a simple linear spring system.

The spring system is based on an assumed constant velocity v, the lateral offset with respect to the center of the lane Δl, and the heading offset with respect to the lane $\Delta\theta$. The value of v is set before this computation based on local obstacles, the distance to a stop sign, or the curvature of the lane ahead. We gather the lane heading and location using the closest lane polygon just past the front bumper of the vehicle. The vehicle steers in the lane using the following equation:

$$\omega = -k_\theta \Delta\theta - \frac{k_l}{v}\Delta l,$$

where both k_θ and k_l were experimentally determined to be 0.5.

Avoidance of obstacles near the edge of lanes is accomplished by simply changing Δl to edge the vehicle to the other side of the lane. When the obstacle is far into the lane, the vehicle stops with the Blocked result code, which may eventually lead to passing in another lane or performing a U-turn.

2) Follow at a safe distance: While following a lane, this controller checks whether there is another vehicle or obstacle ahead, matching speeds while maintaining a safe distance. Note that the obstacle may be stationary, in which case the vehicle will stop at an appropriate distance behind it, with the controller returning a Blocked result. This behavior can be used in smooth-flowing traffic to maintain at least the standard 2-second following distance, or in stop-and-go traffic. The pseudocode for follow safely is in Algorithm IV.1.

3) Intersection Precedence: When the Follow controller reaches a stop sign way-point, it returns Finished, causing a transition to the WaitStop Navigator state. This transition runs ActionToWaitStop(), followed by ActionInWaitStop() in each subsequent cycle. Algorithm IV.2 gives the pseudocode for these two subcontrollers.

The guidelines for the Urban Challenge specify that if another vehicle fails to go in its turn, the vehicle should wait 10 seconds before proceeding cautiously. Our implementation uses two timers. The *stop_line_timer* gives the Intersection Precedence observer one second to gather information about other vehicles already stopped at this intersection. Meanwhile, the *precedence_timer* starts counting up to 10 seconds each time the number of vehicles ahead of us changes.

When there are none left or the *precedence_timer* expires, we set the *pending_event* class variable to Merge, which

Algorithm IV.1: FOLLOWSAFELY(*speed*)

```
$result \leftarrow$ OK
$distance \leftarrow$ range of closest obstacle ahead in this lane
if $distance \geq maximum\_range$
  then return $(result)$
$following\_time \leftarrow distance/speed$
if { $following\_time \leq min\_following\_time$ or
   { $distance \leq close\_stopping\_distance$
  then { $speed \leftarrow 0$
       { if already stopped
       {   then $result \leftarrow$ Blocked
  else if $following\_time < desired\_following\_time$
  then decrease speed to match obstacle
  else if { already stopped or
          { $following\_time > desired\_following\_time$
  then increase speed to match obstacle
if { Nearest Forward observer reports obstacle approaching and
   { closing velocity is faster than our current velocity
  then $result \leftarrow$ Collision
return $(result)$
```

Algorithm IV.2: WAITSTOP(*speed*)

```
procedure ACTIONTOWAITSTOP(speed)
  set turn signal depending on planned route
  start 1 second stop_line_timer
  start 10 second precedence_timer
  $prev\_nobjects \leftarrow -1$
  return (ACTIONINWAITSTOP(speed))

procedure ACTIONINWAITSTOP(speed)
  $speed \leftarrow 0$
  if stop_line_timer expired and Intersection observer reports clear
    then $pending\_event \leftarrow$ Merge
  $obs \leftarrow$ current Intersection observation data
  if obs.applicable and $obs.nobjects \neq prev\_nobjects$
    then { $prev\_nobjects \leftarrow obs.nobjects$
         { start 10 second precedence_timer
  if precedence_timer expired
    then $pending\_event \leftarrow$ Merge
  return (OK)
```

triggers a transition in the next cycle, in this case to the WaitCross state, which handles Intersection Crossing.

4) Intersection Crossing: When the vehicle has reached an intersection and is ready to proceed, Navigator changes to its WaitCross state. As the state transition diagram in Figure 4 shows, this may either happen from the WaitStop state after intersection precedence is satisfied, or directly from the Follow state if there is no stop sign in our lane (e.g. turning left across traffic). In either case, the vehicle has already stopped. It remains stopped while waiting for the intersection to clear.

The WaitCross control simply activates the appropriate turn signal based on the planned route and waits until the Merging observer reports at least a 10 second gap in surrounding traffic. It then transitions to the Follow controller, which guides the vehicle through the intersection and cancels the turn signals after reaching the next lane of the desired route.

5) Pass: The intersection state transitions provide a simple introduction to the more complex transitions involved in passing a stopped vehicle or other obstacle blocking the desired travel lane.

Our current implementation never passes moving vehicles. In the Follow state, Navigator matches speeds with any vehicle ahead in our lane. As described in section IV-B2, it only returns a Blocked result after the vehicle comes to a complete stop due to a stationary obstacle, which could be a parked vehicle or a roadblock.

A Blocked result in the Follow state initially triggers a transition to the WaitPass state (Algorithm IV.3). Next, Navigator attempts to find a passing lane using the polygon library. If none is available, the situation is treated as a roadblock, causing a transition to the Block state, and initiating a request for the Commander module to plan an alternate route to our next checkpoint, usually beginning with a U-turn. Because a roadblock is a static obstacle not representing a multiagent interaction, we focus more deeply on the case where a passing lane exists, allowing us to pass a parked vehicle blocking our lane.

When waiting to pass, two things can happen. If the obstacle moves within several seconds, Navigator immediately returns to the Follow state. If the obstacle remains stationary, Navigator changes to the Pass state as soon as no vehicle is approaching in the passing lane.

In the Pass state (Algorithm IV.4), Navigator saves the polygon list of the selected passing lane, and invokes the Passing controller. This controller uses the same linear spring system as the FollowLane controller to follow the polygons in the passing lane. It returns Finished when it detects that the vehicle has passed all obstacles in the original lane. Navigator then returns to the Follow state, where the FollowLane controller uses the polygons for the original travel lane to guide the vehicle back.

6) Evade: This controller runs when the main Navigator state machine is in the Evade state. We reach that state after some other controller returns a Collision result, having noticed that the Nearest Forward observer saw something driving towards our vehicle in the current lane. Having a closing velocity with respect to an obstacle does not imply a collision event. Only if the relative velocity is significantly greater than our vehicle's velocity, do we decide to evade.

The Evade controller's job is to leave the lane to the right, wait until there is no longer a vehicle approaching, then return Finished. Navigator then returns to the Follow state. Other evasion techniques could be used. Our approach implements the recommended behavior in the DARPA Technical Evaluation Criteria document [1].

This controller has a simple state machine of its own. Algorithm IV.5 gives pseudocode for each state, the appropriate procedure being selected by the *state* variable in each 20Hz Navigator cycle. When reset on transition to Evade, it begins in the Init state. The Leave state invokes a private leave_lane_right() method, also shown. It calls the LaneEdge controller as long as the lane to the right is clear. That controller steers the vehicle outside the right lane boundary to avoid a head-on collision.

7) Obstacle Avoidance in Zones: In our compressed development schedule, driving in zones was largely put off until

```
Algorithm IV.3: WAITPASS(speed)

procedure ACTIONTOWAITPASS(speed)
  if FIND_PASSING_LANE()
    then { set turn signal for passing direction
           start 5 second passing_timer
           return (ACTIONINWAITPASS(speed))
    else { pending_event ← Block
           speed ← 0
           return (Blocked)

procedure ACTIONINWAITPASS(speed)
  if passing_timer expired and observer reports passing lane clear
    then pending_event ← Pass
  result ← FOLLOWSAFELY(speed)
  if result = OK
    then pending_event ← FollowLane
    else if result = Collision
    then pending_event ← Collision
  speed ← 0
  return (result)
```

```
Algorithm IV.4: PASS(speed)

procedure ACTIONTOPASS(speed)
  if SWITCH_TO_PASSING_LANE()
    then { reset Passing controller
           return (ACTIONINPASS(speed))
    else { pending_event ← Block
           speed ← 0
           return (Blocked)

procedure ACTIONINPASS(speed)
  result ← PASSING(speed)
  if result = Finished
    then { pending_event ← FollowLane
           result ← OK

    else if result = Blocked and blockage has lasted a while
    then pending_event ← Block

    else if result = Collision
    then pending_event ← Collision
  speed ← 0
  return (result)
```

```
Algorithm IV.5: EVADE(speed)

procedure EVADE_INIT(speed)
  set right turn signal on
  state ← Leave
  return (EVADE_LEAVE(speed))

procedure EVADE_LEAVE(speed)
  if still in lane
    then { if Nearest Forward observer reports vehicle approaching
             then result ← LEAVE_LANE_RIGHT(speed)
             else { speed ← 0
                    set left turn signal on
                    result ← Finished

    else { set both turn signals on
           start evade_timer
           state ← Wait
           result ← EVADE_WAIT(speed)
  return (result)

procedure EVADE_WAIT(speed)
  speed ← 0
  if evade_timer expired
    then { set left turn signal on
           state ← Return
           return (EVADE_RETURN(speed))
  return (OK)

procedure EVADE_RETURN(speed)
  cancel evade_timer
  speed ← 0
  if Nearest Forward observer reports vehicle approaching
    then return (OK)
    else return (Finished)

procedure LEAVE_LANE_RIGHT(speed)
  limit speed to 3 meters/second
  result ← Unsafe
  if Adjacent Right observer reports clear
    then result ← LANEEDGE(speed, outside right)
  return (result)
```

just before the NQE. Not having implemented a model-based planner for lane navigation left us with a large piece missing when it came to driving in the less restricted zone areas. Rather than writing a model-based planner, we utilized an off-the-shelf skeleton algorithm called EVG_Thin [10] to get a coarse route that the vehicle could follow between the zone entry, the zone exit, and any parking spots.

Inside of a zone, we use the perimeter polygon given by the RNDF, the parking waypoints, and any observed obstacles to generate a new skeleton every cycle (see Figure 11). The thinning-based skeleton is an approximation of the Voronoi graph [11], thus it connects points that are maximally far from any two obstacles (a criterion we find quite nice in its aversion to danger). Because we use a Voronoi-style skeleton, we also have the distance to the closest obstacle for each point. We call this distance the point's *safety radius*.

Our controller relies on the fact that if two points are within

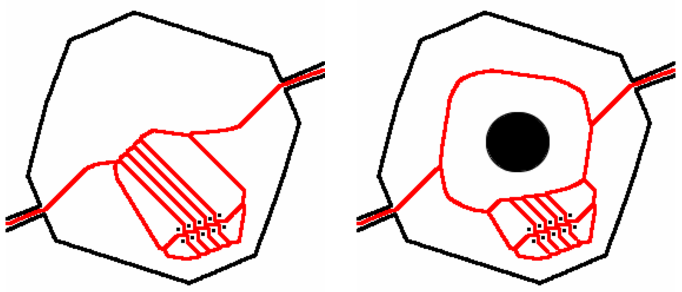

Fig. 11. **Thinning-based skeleton of parking zone.** The black lines connect the perimeter points that define the zone boundaries. The black dots represent the locations of five parking spots. The zone boundaries and parking spots are seen as obstacles along with any obstacles (e.g. the circle in right image) detected by the lidar sensors. The lighter line illustrates the thinned skeleton given local obstacles. This coarse path that avoids obstacles is created at every timestep.

```
Algorithm IV.6: ZONECONTROLLER($zone, obstacles, current\_location, goal$)

procedure PATHTHROUGHZONE()
  $graph \leftarrow$ EVG_THIN($zone, obstacles$)
  $start\_node \leftarrow$ NULL
  $end\_node \leftarrow$ NULL
  for each $node \in graph$
    do
      if $node.safety\_radius \leq minimum\_safety\_radius$
        then remove $node$ from $graph$
        else
          if $current\_location$ within $node.safety\_radius$ of $node$ and
             $goal$ within $node.safety\_radius$ of $node$
            then return (empty path)
          if $current\_location$ within $node.safety\_radius$ of $node$ and
             $node$ is closer to $curent\_location$ than $start\_node$
            then $start\_node \leftarrow node$
          if $goal$ within $node.safety\_radius$ of $node$ and
             $node$ is closer to $goal$ than $end\_node$
            then $end\_node \leftarrow node$
  $path \leftarrow$ A*($graph, start\_node, end\_node$)
  return ($path$)

procedure NEXTAIMPOINTINZONE()
  $path \leftarrow$ PATHTHROUGHZONE()
  if $path$ is empty
    then
      comment: Straight shot to goal.
      return ($goal$)
  $node \leftarrow$ last node in $path$
  while $start\_location$ is not within $node.safety\_radius$ of $node$
    do $node \leftarrow$ previous node in $path$
  if $goal$ is within $node.safety\_radius$ of $node$
    then $aim \leftarrow goal$
    else $aim \leftarrow node$
  return ($aim$)
```

the safety radius of the same skeletal node, the straight line between those two points will not cross any obstacles. This fact allows us to find potentially far away nodes along the skeleton which the vehicle can aim straight for without running into obstacles. In this manner, we ensure the vehicle avoids all obstacles, without going unreasonably far out of its way to follow the Voronoi diagram precisely. Algorithm IV.6 details the procedures that are called 10 times per second, attempting to constantly move the vehicle directly towards the furthest safe point.

This controller does not consider the exact vehicle dynamics when planning a path, and does not respect parking lot navigation conventions, such as passing an approaching vehicle on the right. However, by using the safety radius information to aim at far away points, we still get reasonably smooth control.

8) Park: Some MDFs require the autonomous vehicle to park before it exits a zone. A parking spot is defined in the RNDF by two GPS waypoints (see Figure 12) and a spot width. One waypoint defines the location of the entry to the spot, and the other indicates the desired location of the front bumper when the vehicle is properly parked.

There are three main components to the parking behavior. First, the vehicle must get close to the parking spot. This step is done by using the Voronoi zone planner to get to a point near the entry to the parking spot.

Second, the vehicle must determine the exact location of the spot. Given no surrounding obstacles, the vehicle simply uses its GPS-based odometry to define the spot location. With

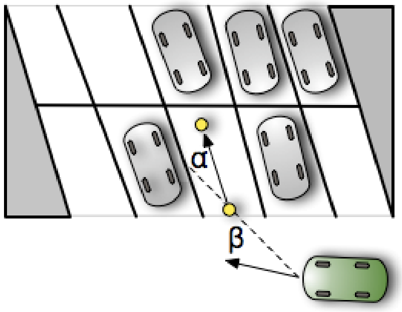

Fig. 12. **Geometric constraints used in parking.** Behavior-based parking tries to minimize both the bearing offset between the front of the vehicle and the beginning of the spot (β) and the heading offset of the vehicle with the parking spot ($\gamma = \alpha + \beta$).

obstacles nearby, the spot location is fine-tuned. The fine-tuning is done by simply defining a rectangle that corresponds to the width of the spot and the length of the spot (the two waypoints plus some extra room determines the length). A discrete search over a predetermined area (1x1 meter offset) is performed to find the spot location that keeps farthest from nearby obstacles.

Third, the vehicle must pull into the spot then reverse out of the spot. This can be broken into four sub-behaviors. i) The vehicle ensures its front bumper is over the GPS waypoint at the spot entry and that it is aligned with the spot. ii) It then pulls straight into the spot until the front bumper is over the

Algorithm IV.7: ALIGN WITH SPOT(*spot_pose*)

```
if too far away from spot entry
  then { comment: go back to zone controller
         return (NotApplicable)
comment: In Figure 12, $\gamma = \alpha + \beta$
$\beta \leftarrow$ Bearing to spot_pose
$\gamma \leftarrow$ Heading offset wrt spot_pose
comment: Angles are normalized to be between (-180,180]
if $|\beta| < 15°$ and $|\gamma| < 15°$
  then { comment: aligned with spot entry
         return (Done)
comment: The space around vehicle is divided into 4 quadrants:
         in front, in back, left, and right of vehicle.
comment: front ≡ [-45°,45°), left ≡ [45°,135°),
         back ≡ [135°,225°), right ≡ [225°,315°)
comment: $v$ is forward velocity; $\omega$ is rotational velocity
$v \leftarrow 0.5$ (m/s)
$\omega \leftarrow \beta$ (degrees/s)
if $\beta$ is in front quadrant
  then { if $\gamma$ is in front quadrant and $|\gamma| > |\beta|$
           then $\omega \leftarrow \gamma$
if $\beta$ is in left quadrant
  then { if $\gamma$ is in front quadrant or $\gamma$ is in right quadrant
           then $v \leftarrow -v$
         else if $\gamma$ is in left quadrant and $|\gamma| > |\beta|$
           then $\omega \leftarrow \gamma$
if $\beta$ is in right quadrant
  then { if $\gamma$ is in front quadrant or $\gamma$ is in left quadrant
           then $v \leftarrow -v$
         else if $\gamma$ is in right quadrant and $|\gamma| > |\beta|$
           then $\omega \leftarrow \gamma$
if $\beta$ is in back quadrant
  then { if $\gamma$ is in front quadrant
           then { $v \leftarrow -v$
                  $\omega \leftarrow 0$
         else if $\gamma$ is in left quadrant or $\gamma$ is in right quadrant
           then { $v \leftarrow -v$
                  $\omega \leftarrow \gamma$
comment: $\eta$ is a tuning parameter
$\omega \leftarrow \eta \cdot \omega$
if $(v, \omega)$ is unsafe
  then { $v \leftarrow -0.5$
         $\omega \leftarrow \eta \cdot \gamma$
         return (Done)
```

second waypoint. iii) It reverses straight back until the front bumper is again over the entry waypoint. iv) Finally it reverses further, turning to face the appropriate direction to continue its plan.

Pulling into and out of the spot once aligned with the spot is straightforward. The pseudocode in Algorithm IV.7 explains the more complex behavior that gets the vehicle aligned to pull directly into the spot.

9) Escape (or Traffic Jam): In cases where the vehicle cannot make progress, it must get unstuck. To do this, we construct a "zone" on the fly. This temporary zone is large enough to encompass our vehicle, the next waypoint in the current plan, and nearby obstacles. We then invoke the same zone controller used in parking lots. In this manner, we continue to make forward progress, though the vehicle may leave the road if there is no alternative path forward. If forward progress is unsafe, the vehicle attempts to reverse. If no safe action can be taken, the vehicle waits until a safe action is applicable.

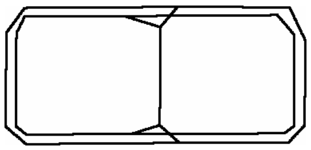
Fig. 13. **Waypoint graph of Area A.** Vertices correspond to GPS waypoints from the RNDF, and edges denote explicit connections from the RNDF.

V. THE URBAN CHALLENGE NQE EXPERIENCE

The multiagent behaviors described in Section IV were put the test at the Urban Challenge National Qualifying Event (NQE), where our vehicle was placed in several challenging multiagent environments. This section describes our experiences at the October NQE as one of the 35 teams invited to participate after having successful site visits in July.

The NQE had three areas meant to test the abilities necessary to safely and effectively navigate the Urban Challenge final course. Area A required the vehicle to merge into and turn across a continuous stream of traffic (eleven human-driven cars operating at 10 mph). This area was the most challenging and was deemed "the essence of the Urban Challenge" by DARPA director Dr. Tony Tether. The challenges in Area B were focused on parking, avoiding static obstacles on the road, and long-term safe navigation. No moving vehicles or dynamic obstacles were encountered in this area. Area C required vehicles to pass a combination of intersection precedence scenarios with human drivers. Roadblocks were added in the middle of the course to force vehicles to perform U-turns and replan routes. This area was similar to the site visit test, which teams were required to complete before being invited to the NQE.

The algorithms described above were reliable enough for our team to place in the top twelve to twenty-one teams at the NQE. With a bit more time for integration testing, we believe that we could have done better. After diagnosing an Ethernet cable failure and tracking down a memory leak in third-party software, we believe we could have competed well in the final race along with the eleven finalists.

A. Area A

The Area A course consisted of a two lane road in a loop with a single lane bypass running north-south down the middle (see Figure 13). Eleven human-driven cars made laps in both directions on the outer loop, while the autonomous vehicles were commanded to perform counter-clockwise laps on the east half of the course. The autonomous vehicle was required to turn across traffic when turning into the bypass, as well as merging back into the main loop when exiting the bypass.

Key to this course was the ability to successfully judge when a sufficient opening was available in traffic. Being overconfident meant cutting off the human drivers and getting

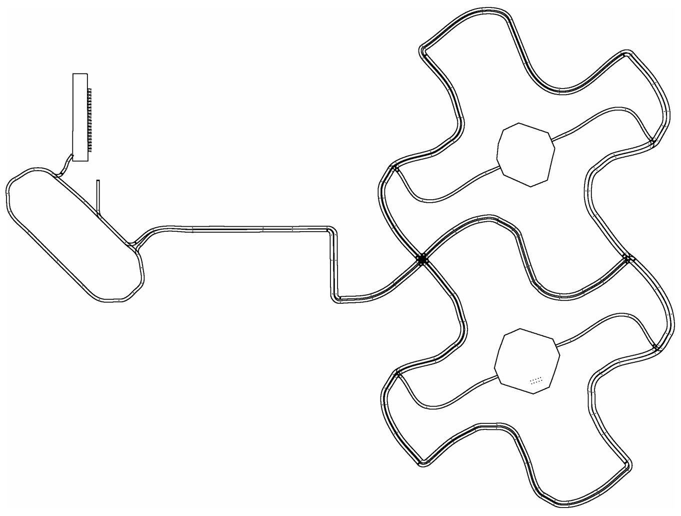
Fig. 14. **MapLanes outline of Area B.** Note the starting chutes in the top left. There are also two large zones, the lower of which has defined parking spaces.

penalized, while being too cautious could result in waiting indefinitely. Our observers were able to successfully detect openings in traffic, resulting in very good multiagent behavior. Our vehicle was able to complete seven laps in the first half-hour run without contact with either vehicles or barriers.

The south edge of the course was lined with k-rails (3 feet high concrete barriers) at the curb for the safety of the judges and spectators. The proximity of the k-rails to the road caused problems for many teams when turning out of the bypass. When making this sharp left turn, these k-rails were seen by our vehicle as obstacles in our path, resulting in our low-level safety controller returning a Blocked event, and thus prohibiting completion of the turn before merging back into the loop (the center bottom of Figure 13). The vehicle eventually called the Escape controller, backed up, and continued down the lane.

This problem was easily overcome by turning down the safety thresholds slightly. However, by the time of our second run in Area A, the race officials had moved these barriers away from the lane boundary by a few feet, since many teams were having problems with this one turn. The vehicle was only able to complete two partial laps during our second run due to a defective Ethernet cable, which dropped most of the data packets from our Velodyne lidar unit. This was an unfortunate hardware failure that was unrelated to our trackers or behaviors.

B. Area B

In Area B, each autonomous vehicle was randomly assigned to one of the start chutes that were used for the start of the final race. The vehicles needed to make their way out of the start zone, down a narrow corridor, and around a traffic circle before proceeding west to the main section of the course, which tested parking and avoiding many static obstacles in the road. Figure 14 illustrates the course.

Our vehicle successfully exited the start zone and made it through the corridor, which had given several prior teams trouble. It then followed the course through several turns before arriving at the first parking lot. The lot contained three parked vehicles surrounding the designated parking spot. Our vehicle parked into and reversed out of the parking space flawlessly. As the vehicle was turning from the path leading out of the parking lot to the main road, a previously undetected memory leak in third-party software caused our control program to crash, ending our run.

Unfortunately, this memory leak occurred in both of our Area B runs, both times just after leaving the parking test area. We eventually traced the root cause back to building Player/Stage [4] with a buggy version of the C++ Standard Template Library.[5] The parking lot was large enough to trigger a memory leak in the STL vector code, which we had never seen in testing. Recompiling Player with the current STL libraries eliminated this memory leak, but unfortunately this solution was not discovered until after both of our Area B runs.

C. Area C

Area C consisted of a large loop with a road running east-west through the center, forming two four-way intersections (Figure 7). Challenges on this course included intersection precedence as well as roadblocks that required replanning the route to the next checkpoint. Human-driven cars provided the traffic at the intersections; however, there was no traffic on other sections of the course. The human drivers retreated to driveways alongside the road when not needed.

Our vehicle made a perfect run, successfully handling intersections with two, three, and four cars queued up when it arrived. Intersections with either two or three cars queued required the vehicle to wait until they had all cleared the intersection before proceeding. The intersection test with four cars had two of the cars queued up in the same lane, which required our vehicle to only wait for the first three cars to clear the intersection and proceed without waiting for the fourth car.

VI. Conclusion and Future Work

This article presented the autonomous vehicle developed by Austin Robot Technology for the 2007 DARPA Urban Challenge. Specifically we focused on the effective multiagent algorithms programmed in part by UT Austin students in only a few months time. We detailed the perceptual algorithms necessary to model other vehicles and the behaviors necessary to drive in the presence of other vehicles. We provided algorithms used in the system to merge, pass, follow, park, track dynamic obstacles, and obey intersection laws.

While our system presents a novel autonomous robot platform, there are many ways to improve the current state by incorporating more sophisticated robotics research into our software. To start, we plan to utilize more of the Velodyne HDL 3D data in order to attempt real-time 3D scene reconstruction and object recognition. Similarly, our system tracks dynamic objects within road lanes, but it does not distinguish

[5]g++-4.1 was updated in Ubuntu Dapper to fix this bug after Player was built on our vehicle's computers.

between vehicles and non-vehicles, nor does it track obstacles outside of the road which may still be on a collision path with our vehicle. Thus, the vehicle cannot yet deal appropriately with pedestrians crossing the road. For more robust navigation, vision needs to be integrated in order to correct MapLanes information due to GPS drift and inaccurate curve estimation. Vision is also needed for dealing with traffic signals, the other main omission from the Urban Challenge scenario (in addition to pedestrians). Finally, we aim to include a model-based planner to provide more human-like local control in open zones and for getting unstuck.

These avenues for future work notwithstanding, we now have an autonomous vehicle that is fully capable of driving in traffic, including the complex multiagent interactions that doing so necessitates. All in all, the research presented in this article takes a large step towards realizing the exciting and imminent possibility of incorporating autonomous vehicles into every day urban traffic.

Acknowledgments

The authors would like to thank the many members of Austin Robot Technology (ART) for building and supporting the vehicle. Arturo Martin-de-Nicolas started Austin Robot Technology in 2004. His brothers Jorge and Juan have provided invaluable support for both hardware and testing. Don McCauley is the electronics and electrical wizard that gave the vehicle a physical brain. Jon Brogden implemented the microcontroller interface to the vehicle's cruise control motor. Dave Tuttle was the 2007 team leader. Dr. Michael Quinlan provided valuable general robotics expertise in the months leading up to the NQE.

Many (non-author) students helped contribute to the project as well, including Ryan Madigan, Richard Edwards, Srinivas Ashok, David Reaves, Evan Archer, Chris Billen, and many more.

The undergraduate course was made possible through the support of the Freshman Research Initiative, directed by Dr. Sarah Simmons, in the College of Natural Sciences at UT Austin. This research is supported in part by NSF CAREER award IIS-0237699 and ONR YIP award N00014-04-1-0545.

References

[1] "DARPA Urban Challenge technical evaluation criteria," 2007. [Online]. Available: http://www.darpa.mil/GRANDCHALLENGE/docs/Technical_Evaluation_Criteria_031607.pdf

[2] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L. E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, "Winning the DARPA Grand Challenge," *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.

[3] P. Stone, P. Beeson, T. Meriçli, and R. Madigan, "Austin Robot Technology DARPA Urban Challenge technical report," 2007. [Online]. Available: http://www.darpa.mil/grandchallenge/TechPapers/Austin_Robot_Tech.pdf

[4] B. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the International Conference on Advanced Robotics (ICAR)*, 2003.

[5] "Route Network Definition File (RNDF) and Mission Data File (MDF) formats," 2007. [Online]. Available: http://www.darpa.mil/grandchallenge/docs/RNDF_MDF_Formats_031407.pdf

[6] D. Wolf, A. Howard, and G. S. Sukhatme, "Towards geometric 3D mapping of outdoor environments using mobile robots," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[7] R. H. Bartels, J. C. Beatty, and B. A. Barsky, *An introduction to splines for use in computer graphics & geometric modeling*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

[8] A. Elfes, "Occupancy grids: A probabilistic framework for robot perception and navigation," Ph.D. dissertation, Carnegie Mellon University, 1989.

[9] J. Modayil and B. Kuipers, "Bootstrap learning for object discovery," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[10] P. Beeson, "EVG-Thin software," 2006. [Online]. Available: http://www.cs.utexas.edu/users/qr/software/evg-thin.html

[11] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Communications of the ACM*, vol. 27, no. 3, pp. 236–239, March 1984.