Annotated Maps for Autonomous Land Vehicles

Charles Thorpe and Jay Gowdy

Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The new mechanism of Annotated Maps provides an important tool for managing the information needed by autonomous mobile robots. Annotations tie specific information to particular locations or objects in the map, such as the description of a landmark or the proper control strategy for crossing an intersection. "Descriptor" annotations are retrieved on demand. "Triggers" are automatically sent to a specified process when the robot reaches a given location.

We have used Annotated Maps in several mobile robot systems, on board the CMU Navlab. Our most ambitious runs involved navigating through a suburban neighborhood, including image processing to follow roads, 3-D perception for landmark identification, and inertial navigation to turn at intersections. Annotated maps were built by first driving our robot by hand, recording the location of roads and the location and description of objects along the way. During mission planning, triggers were added to specify how and where to drive and what to look for. While executing the mission, the triggers specified when to use image processing; when to slow down and identify landmarks; how to negotiate intersections; and when to stop at the goal.

Introduction

Motivation

Much of the information that mobile robots need is tied directly to particular objects or locations. Maps, object models, and other data structures store useful information, but do not organize it in efficient and useful ways. We have built a new map-based knowledge representation, the "annotated map", to index information to the relevant object and locations. The annotations are used for a wide variety of purposes: describing objects, providing hints for perception or control, or specifying particular actions to be taken. We have provided a query mechanism to retrieve annotations based on their map locations. We have also built "triggers", which cause a specified message to be delivered to a particular process when the vehicle reaches a given location in the map.

These annotated maps serve a crucial role in enabling missions that are otherwise beyond the reach of autonomous systems. Control descriptors allow mission planners to specify what the vehicle is to do at particular locations, reducing the need for onboard planning. Object descriptors contain detailed instructions of how to recognize a particular object, or contain the appearance of this object as seen by a particular sensor on a previous vehicle run. Such information greatly simplifies the problem of seeing and recognizing objects. Geometric queries enable the vehicle to focus its attention on objects in its vicinity, reducing database access and matching time. The trigger mechanism frees individual modules from having to track vehicle position, allowing them to devote their processing to the task at hand or to lie dormant until they receive their trigger message.

Annotated maps do not by themselves solve difficult problems of sensing, thinking, or control for autonomous vehicles. Their contribution is to provide a framework that makes it easy for other modules to cooperate in planning and executing a mission. Annotated maps thus fill a need that is common to many different vehicles, missions, and architectures.

Many analogous annotated maps exist for human use. Aeronautical navigation charts contain symbolic descriptions of routes (airways) and landmarks, and include annotations such as the Morse code call letters of radio navigation beacons. The AAA produces "Triptiks"[1], which include annotations for route, current conditions ("construction", "speed check"), road type (interstate, two lane, etc.), general conditions ("winds through rolling hills"), points of interest (rest areas, gas, food, and lodging) etc. An intelligent person can usually drive a route without such aids; but they do provide a convenient framework for preplanning, and make "mission execution" easier. Furthermore, as we drive a route, we build our own mental representations of landmark appearance, curves in the road, and so forth, which we use to follow the same route more easily at a later time. Our annotated maps provide the same kind of functionality for autonomous mobile vehicles.

[1] Triptik is a registered trademark of the American Automobile Association

90CH2930-6/90/0000-0282$01.00 © 1990 IEEE

Related Work

At CMU, we have developed a family of autonomous mobile robots over the past ten years. Our current main vehicle for outdoor navigation experiments is the CMU Navlab [12, 13]. Our experience, especially with the Navlab, has driven the design of the annotated maps. We already have perception and control modules that can use information from annotated maps, including color vision [4, 8], neural networks [9], 3-D object recognition [6], and planning [10]. We have also built the EDDIE architecture, which provides inter-module communications, control, and system structure for mobile robots [11]. The tools provided by EDDIE are used for the messages that underly queries and triggers in the annotated map.

Many other groups are working on related problems of mobile robots and knowledge representation. Rather than competing with the ideas of annotated maps, most of this research is providing useful tools and ideas that could use or help generate the annotated maps.

Fennema, Hanson, and Riseman at the University of Massachusetts are building world models and maps for their mobile robot, Harvey [5]. They have defined the concepts of "neighborhoods" (topological regions), "locales" (information to decide whether the robot is within a neighborhood), "milestones" (perception for verification), and actions. The UMass map and plan representations are similar to some of the uses of annotations, but have simple, fixed formats, are focused on declarative representations of 3-D object models, and do not provide map-based triggers.

Rod Brooks at MIT has long argued for simple robots with simple control schemes and simple world maps [2]. We concur that simple, sensor-based maps of particular locations are often useful. The lowest levels of our descriptor annotations are designed to contain precisely the sort of information that Brooks' robots use to calculate their position, or to cause a particular action, in a small local area. We disagree with Brooks' contention that this is the only sort of information that a robot should remember. Robots often work in open, featureless environments, and need precise maps and accurate navigation even where no landmarks may be nearby. Annotated maps are designed to keep precise metric information in the geometric levels of annotations, as well as the lower-level cues advocated by Brooks.

Kender gives a much more abstract view of planning for sensor-based navigation [7]. He describes the combinatorial problem of deciding which sensors to use, and which landmarks should be recognized, in order to reach a given goal. The results of analyses such as Kender's should be entered into triggers, to tell the vehicle what to look for, and into object descriptors, to say how to look for those objects.

Blidberg and his associates at the University of New Hampshire's Marine Systems Engineering Laboratory have implemented world models for underwater mobile robots [3]. Most of their work has concentrated on efficient descriptions of space, such as quadtrees. These spatial descriptions are important, but do not include many of the other forms of knowledge (actions, descriptions) for which annotated maps are useful.

Scenario

A typical mission for our Navlab mobile robot is a delivery task on unlined, unmodified suburban streets. The Navlab has specialized perception modules, including color vision for road following on major roads [8], dirt roads [4], and suburban streets [9]. It also has 3-D perception, using a scanning laser rangefinder, for landmark recognition and obstacle detection [6]. Inertial navigation on the Navlab is accurate enough to drive blind for short distances [1].

In order to accomplish its mission, the Navlab must use several of these modules. Road following using color vision will follow streets, but will not be able to recognize intersections. Inertial navigation will drive through intersections, but must have an accurate starting position. Landmark recognition will update vehicle position before intersections, but is too slow to be run continuously. Only a combination of all those modules, each running at the appropriate locations, will produce an accurate and efficient mission.

Knowledge and Organization

In general, planning and executing such a mission requires several types of knowledge: what to look for, and how to see it; what to do, and how to accomplish it; where to go, and how to get there. The knowledge may range from high-level symbols, to low-level raw data. Knowledge is both internal to a single module, and used by controlling modules to switch between knowledge sources. Approaching the intersection, for instance, the perceptual knowledge includes:

- symbolic: intersection
- geometric: size and shapes of intersecting roads
- sensor-specific: use laser range finder to pinpoint the position by landmark identification
- raw data: landmark 2 meters tall, 0.4 meters wide at position (x,y)

Control knowledge can also span a range of levels:

- symbolic: turn left at intersection
- geometric: intersection angle 45 degrees
- vehicle-specific: turn with a circular arc of radius 15m
- raw data: steering wheel position left 1200 clicks

This knowledge must be carefully organized if it is to be useful. If the vehicle has to sort through all bits of information it has about every possible object, it will

overshoot the intersection long before it has figured out how to recognize it or deduced that it was supposed to turn. It is far better to have information tied directly to the map, or automatically retrieved as needed. The landmark recognition module, for instance, must be able to ask for a description of objects within its field of view, and retrieve the knowledge it needs to recognize them.

Annotated Maps

Annotated maps provide the mechanism for organizing this knowledge, by tying information to a map. The annotations contain knowledge about particular objects, locations, or actions. Annotations come in one of two classes: descriptors and triggers. Descriptors are passive, and are retrieved by queries based on geometry and object type. A query for "all objects of type 'intersection' in this polygon" would return the annotation for the requested intersection, if it were in range. Triggers are active, firing when the vehicle reaches a particular location or crosses a certain line. A trigger will send a message to a particular module, such as "controller: start turning hard left in five more feet".

The knowledge in these annotations comes from many sources, including human experts, mission planning software, and even the vehicle's own observations and experiences on previous missions. It is both declarative (data) and procedural (methods and procedures). The level of the annotations depends partly on the vehicle's computational capabilities. Simple vehicles, in known environments, are able to execute simple pre-planned missions by having every object and action completely annotated at low levels. A more challenging environment, with more variation over time, may require higher-level symbolic descriptors in the map and more reasoning at run time. Practical missions will probably require a mix of levels of detail. Even a sophisticated vehicle may, for instance, decide to record the locations of specular reflections from a mailbox, and use those specularities as recognition cues. It may be much more difficult to reconstruct a 3-D model from the observed data, and to later predict the appearance from the model.

Example Runs

Figures 1 and 2 show a typical annotated map. Figure 1 shows a map of a suburban area, including about 0.7 km of road with two T intersections, and a variety of 3-D objects. Object information was collected using the ERIM laser range finder, and the road information was collected by using the inertial navigation system to provide accurate vehicle positions while we traversed the route. Figure 2 shows a detail of the first intersection, including the Navlab's position during a run and several triggers.

The goal of this run was to drive from a house near the beginning of the map to a specified house near the end. Annotations were added to the map to enable the Navlab to carry out this mission. There were annotations to set the speed appropriately: up to 3.0 m/s in straightaways and down to 0.5 m/s in intersections. Other annotations activated and deactivated the module that uses the laser range finder to correct vehicle position based on detected landmarks. Before every intersection there was an annotation that switched driving control from a neural network vision program to a module that used knowledge from the map of the intersection structure and dead reckoning to traverse the intersection. Finally, there was an annotation at the end of the route that caused the vehicle to stop at the appropriate object. The route was successfully traversed autonomously.

In this run, and a variety of other runs, we have successfully used nine different types of trigger annotations:

- set speed
- dead reckon through intersection
- resume vision after intersection
- start landmark matching
- stop landmark matching
- stop at objects
- stop and start fast obstacle detection
- use vision through intersection
- switch perception modules

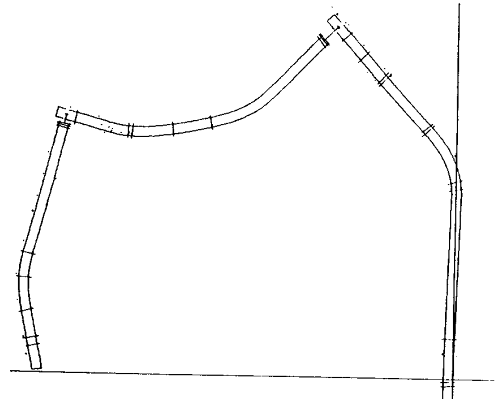

Figure 1: Map built of suburban streets and 3-D objects

Tenets of Map Construction and Use

Several key ideas underly our design for annotated maps, reflecting our experience in building perception and navigation systems for a variety of robots.

Minimize semantic interpretation. No-one can predict all the kinds of knowledge that will be placed in annotations. Moreover, the map module need not understand the annotations. The only common knowledge in annotations should be enough header information to store

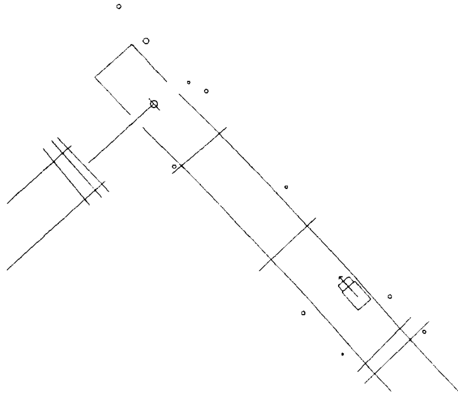

Figure 2: Trigger annotations for sensing and vehicle control

and retrieve the annotation. All the rest of the annotation belongs to the modules that create it and interpret it, with the format to be decided upon by the module creators. The annotated map serves only as a scratchpad.

No specialized query language is needed. The standard queries ask for all objects of type X within polygon Y. Any query more ambitious than that need not be supported. Any more detailed query would require that the map module know the internal details of each type of annotation. It is more efficient, and a better abstraction, to let the querying module sort through the returned objects.

Separate global position tracking from local servoing. Maintaining the current position estimate in local coordinates is a real-time job, and is best done by the low level real-time controller. In order that locations stored in local coordinates will always be consistent, the controller's local coordinates should never be updated. Commanded trajectories, current positions of obstacles to be avoided, and other phenomena that are used once and then discarded, should be kept in local coordinates and never entered into the map. Map-based calculations, such as matching landmarks against a map, or interpreting a position fix, are aperiodic events best done by a separate Navigator module. The Navigator maintains the transform from local to world coordinates. Any module that needs to know current vehicle position in world coordinates must acquire the Navigator's transform, then apply that to the running position reports of the controller. In practice, acquiring the Navlab's current transform is done in one of two ways, specified at start-up:

- The Navigator can send its transform every time it is updated. This is used by fast-running modules that always need the latest update.
- Slower modules, that have a longer cycle time, may not need every updated transform. Worse, receiving too many updates before the module is ready to read them may cause the input queue to overflow. Instead, these modules are notified that a new transform is ready, but do not receive the update until they request it. The Navigator stores which modules have been notified and have not yet requested updates, to avoid sending repeated notifications.

Centralize position tracking. Modules often want to perform specific actions when the vehicle arrives at particular locations in the map. If each module were to continuously poll the Navigator and controller for current position, the controller could become overloaded. Active polling also means that those modules are using computer cycles. Moreover, a Navigator position update may skip the vehicle position estimate past the point for which a module is waiting. For each update, each module would have to figure out if any of its target positions had been passed. We prefer to have a single module, the map manager, doing position tracking for all modules. On reaching the points of interest, it awakens or signals the appropriate module. This is the function of "trigger" annotations.

No master control. The map module is best thought of as an alarm clock (for the triggers) and a scratchpad (for descriptors and trigger messages). It is not some "master" module that controls all thinking, and that therefore can become a major bottleneck. We prefer point-to-point communication between modules, with flow of data and control decided on module by module, rather than forcing all information through a single controller.

Plan incrementally. The map module is designed to be used by many programs, for many purposes, at many times. Some information may be permanent; other annotations may be added to provide directions for only a single mission. It is an advantage to be able to update, add, and delete at various times. In particular, display and user interface modules may read the annotated map from a file, look at it, display the annotations, change things, and write it back out.

Implementation of Annotations

The annotated map needs to provide efficient access, indexed by position. The annotations themselves need to contain an arbitrary amount of data, with a minimum of externally imposed organization on the contents. We have designed and implemented a two-part representation, consisting of a map grid and an annotation database. Each square of the grid contains a list of any annotations that are included in that square's area.

Adding an annotation to the map is a two-step process. First, the actual annotation is added to the annotation database. Secondly, the map grid must be updated. The

location of the annotation is either a point, a line, or a polygon. This location can either be specified directly, for those annotations tied to a location, or retrieved from an object description, for those annotations that describe an object. The location is then scan-converted (converted to a list of cells) into the grid, and a pointer to the appropriate entry in the database is written into each of the corresponding grid cells.

Retrieval of annotations in response to a query is also a two-step process. Queries can specify a polygon and an annotation type. The query polygon is scan-converted into grid cells. The annotations pointed to by each of those cells are collected, checked to see if they match the specified type, and returned.

Triggers work similarly. At each cycle, the map module calculates the current vehicle position. It calculates the line on which the vehicle has moved since the last cycle, and scan converts that line into the grid. Each cell through which the vehicle has moved is checked for trigger annotations. If any are found that have not already been fired, their messages are sent to their destination modules. Since the location of a trigger can be a point, line, or set of lines, a trigger can be fired when the vehicle reaches a certain location or when it enters a given polygon.

Representing Annotations

Annotations are represented with a uniform header format, plus a free-format data field. Typical header fields include:

header:
  {type, destination module, used flag, text description, location, next object, previous object, data size}

data:
  {pointer to data}

The header portion contains all the information that the map module needs to understand. "Type" and "location" are sufficient for answering queries; "destination" is required for sending trigger messages. The "used" flag is set when a trigger is fired, to avoid firing the same trigger repeatedly if the vehicle stays in the area covered by the trigger for more than one cycle. "Text description" is used by graphics display modules. This information is also sent as part of messages, to make it easier to debug receiving modules. The "location" of the annotation is used both in initially setting up the grid pointers, and for the use of the receiving module. "Next object" and "previous object" are used to describe extended linear objects. Extended objects may also have branches, which meet at intersections. Intersections have a center point, and any number of vertices, each of which points to the beginning of an extended object. The most common extended objects are roads, which are represented as short segments pointing to their preceding or following segments, or pointing to intersections.

The data portion of the annotation is, in the view of the map, an undifferentiated field of bytes. Any internal structure need only be understood by the modules that create and read the annotation. Since the headers have a known, fixed size, they can be stored in a random-access file. The data may be stored as a stream of bytes, with the header containing only a pointer to the beginning of the data and the number of bytes.

Implementation Details

Our prototype implementation has tested some of our design decisions, while other details will be decided after further data collection and analysis.

Grid cell size. If grid cells are too small, queries will have to look at large numbers of cells, and map storage will become a problem. But the querying becomes simpler, because any object found in any of the cells can be returned. Larger cells give faster lookups, but are no longer selective enough to answer queries on their own. Instead, objects within grid cells must still be checked to make sure they are within the query polygon. For autonomous land vehicles with sensor ranges of two to thirty meters, a grid with 0.5 to 1.0 meter cell spacing probably provides the right tradeoff; our current implementation uses 0.5 meter cells.

Handling large maps. For a grid with 1.0 m cells, each square kilometer will contain a million cells. Each cell can be represented with at most a few bytes of data, depending on annotation density. The amount of memory required by a grid this size is easily within the capability of today's computer systems, but for missions spanning several kilometers, we will not be able to keep the whole grid in main memory at once. One possible solution is implementing quad-trees to take advantage of sparse data requirements over most of the grid. A more likely strategy is to keep the grid on secondary storage, and only keep a window around the current vehicle position in main memory. The annotation databases themselves may also need to be kept on backing store, and only read in as needed.

Distributed databases. Object descriptions might be most easily implemented in separate databases, internal to the modules that use them. Then the annotations need only return the index of the database entry. The problem with this method is ensuring consistency between databases in the modules, and indices in the grid. At the opposite extreme, the map annotations could contain all the data. The disadvantage of this approach is requiring more traffic between maps and objects. An intermediate approach is to start with all the knowledge in the map annotations, but have it automatically replicated in the

appropriate modules at system initialization time. This ensures consistency while reducing runtime overhead, at the expense of startup costs. The design of distributed databases interacts with the design for handling large maps. Keeping annotations in individual modules would decrease the amount of information needed by the map module, and thus make building large maps somewhat easier.

In the current implementation, the annotation database is static during a run. When the system is initialized, the user adds stop points, turn points, or other triggers to specify the current mission. When the user is ready, the interface module saves the current annotation database and sends the name of the file to the map module. At start up, each module that needs a copy of the annotation database requests the name of the file from the map module. So modules contain a complete, consistent copy of the annotation database. The map module builds the grid, so it can handle geometric queries. It communicates with the other modules by specifying the index in the annotation database of the objects that match the current query. The map module also watches the grid for triggers.

Map update. Changing an annotation during a run is conceptually easy. Moving objects and annotations is more difficult. If a single object moves, it is easy to erase it from one part of the map and write it into another location. But if an entire portion of the map moves, such as discovering that a portion of the road is really longer than previously thought, the changes can be very hard to handle. Many objects would have to move: the road, all objects attached to it, all landmarks that were seen on previous inaccurate runs and indexed to the road, planned mission steps based on following the road or on seeing those landmarks, etc. It is probably better to note the new information, keep running with the flawed map, and build a new map at the end of this run, rather than try to do updates on the fly. Map update strategy is also influenced by the "large maps" and "distributed databases" design issues. If an individual module updates its copy of an object description annotation, it will need to make sure any permanent information is written out when the run is terminated or when that portion of the map is overwritten by a new data window.

Since in the current implementation, each module keeps its own internal copy of the annotation database, map updates must be specially handled while building a new map. Under most circumstances, the map updates refer to objects that the vehicle will not see again on this run, and therefore the updates need not be propagated to all the modules. At the end of a run, all the new objects can be written to a new map file, to be used on succeeding runs. The exception is for building maps of intersections. Our procedure is to drive through the intersection, following one branch, and building a map; then to reposition the vehicle before the intersection, and follow the second branch. In order to register the two branches correctly, the perception and matching systems need to find newly-mapped landmarks. The map manager writes the annotation database to a file and notifies the relevant modules, which read in the updated database.

Interfaces. Conceptually, it is easy to add annotations to the map. A program reads in the annotation database, adds new annotations, and writes the updated files. Machine-generated annotations, such as object descriptions, use interface routines to read and write the map, and to insert annotations into the annotation database. Annotations added by hand require, besides the basic map interface routines, a user interface to point to locations or objects on the map, type or read the annotation data, display the resulting map, and ask for verification. While the format and contents of the annotations will vary, there is still a large body of common functions that use standard modules. We have built an interface, using X windows, that allows a user to add new objects and triggers to the map. The same interface is also used to display the vehicle and map during a run.

Trigger Details

In order for the map module to track vehicle position, it must know both the controller's current local position estimate, and the navigator's transform that relates local to global coordinates. Position queries to our vehicle controllers are efficient, returning in less than 10 milliseconds. Our current implementation uses an efficient process for getting transforms from the navigator, by having the navigator send the transform each time it is updated. Since landmark sightings or position fixes are relatively infrequent, an event-driven transform update is much more efficient than polling.

When the navigator updates position, the map module has to pay special attention to triggers. It may be that the vehicle position estimate will jump forward, skipping some triggers; or it may be that it will move backwards, creating the potential for firing triggers that have already been fired. If the position update is relatively small, it makes sense to use the line of vehicle travel, plus the "used" flag, to make sure that all appropriate triggers get fired once. If the update is large, it may no longer make sense to fire triggers that should have been fired long ago; and it may make sense to refire triggers that were fired very prematurely. Details of these design decisions are yet to be worked out.

The mechanism of notifying a module of a trigger is by sending a message over a port. In the Unix[2] operating

[2]Unix is a trademark of Bell Labs

system, ports can be set up by broadcasting their address and listening to the net to find out who would like to to talk to them. Once connected, ports appear as files, and can be read and written easily. A module can easily check if there are any bytes waiting on its trigger port. If not, it has not received a message, and can continue running. If so, it can read the message header, allocate the memory structure for the message, and read the appropriate number of bytes into its memory. A running module can periodically check to see if a message is waiting. A sleeping module can simply block on read, which will cause it to pause until data arrives. It is possible to set timers, so a module can wait until either a timer expires or a message arrives, whichever occurs first. It is also possible to have an incoming message generate an interrupt, so the module can be notified while running even without checking for incoming data.

Conclusion

Annotated maps provide a framework to organize knowledge storage and retrieval for autonomous mobile robots. The Navlab group at CMU, and other groups around the world, have many of the individual pieces of a complete system: sensing, sensor understanding, local trajectory planning, control, and vehicles. These pieces in themselves are only sufficient to perform limited tasks. Integrating those components into an efficient system is one of the difficult remaining gaps. The annotated map helps fill that gap. By providing generic data handling, it allows diverse modules to communicate their specialized knowledge. By tying this knowledge to specific locations and objects, the annotated map provides a focus of attention, using an efficient grid structure to answer queries about specific parts of the map. And through the automatic triggers, the annotated map eliminates the need for individual modules to attend to vehicle position and map location. We have built our first prototype annotated map, interfaced several modules to it, and used it to store and retrieve data during real Navlab runs. We are currently addressing the issues of large maps, and continue to interface more modules and to use annotated maps to manage a wider variety of knowledge.

Acknowledgements

Our work with autonomous mobile robots, and the Navlab in particular, is done with a host of colleagues in the Robotics Institute and School of Computer Science at CMU. Our thanks especially to Takeo Kanade and William Whittaker, co-principal investigators; to Jill Crisman, Martial Hebert, Dean Pomerleau, Didier Aubert, and Karl Kluge, who built the perception modules that the annotated maps support; and to Jim Frazier, who keeps the Navlab running and happy. This research is sponsored in part by contracts from DARPA, titled "Perception for Outdoor Navigation" and "Development of an Integrated ALV System".

References

1. Omead Amidi. Integrated Mobile Robot Control. Robotics Institute, Carnegie Mellon University, 1990.

2. R. Brooks. "A Robust Layered Control System for a Mobile Robot". *IEEE Journal of Robotics and Automation RA-2*, 1 (1986).

3. Steven G. Chappell. A Simple World Model for an Autonomous Vehicle. Sixth International Symposium on Unmanned Untethered Submersible Technology, June, 1989.

4. Jill D. Crisman and Charles E. Thorpe. Color Vision for Road Following. In *Vision and Navigation: The Carnegie Mellon Navlab*, Charles E. Thorpe, Ed., Kluwer Academic Publishers, 1990, ch. 2.

5. Claude Fennema, Allen Hanson and Edward Riseman. Towards Autonomous Mobile Robot Navigation. DARPA Image Understanding Workshop, May, 1989.

6. Martial Hebert, InSo Kweon and Takeo Kanade. 3-D Vision Techniques for Autonomous Vehicles. In *Vision and Navigation: The Carnegie Mellon Navlab*, Charles E. Thorpe, Ed., Kluwer Academic Publishers, 1990, ch. 8.

7. J. R. Kender and A. Leff. "Why Direction-Giving is Hard: The Complexity of Linear Navigation by Landmarks in One-Dimensional Navigation". *IEEE Transations on Systems, Man, and Cybernetics 19*, 6 (November/December 1989).

8. K. Kluge and C. Thorpe. Explicit Models for Robot Road Following. In *Vision and Navigation: The Carnegie Mellon Navlab*, Charles E. Thorpe, Ed., Kluwer Academic Publishers, 1990, ch. 3.

9. Dean A. Pomerleau. Neural Network Based Autonomous Navigation. In *Vision and Navigation: The Carnegie Mellon Navlab*, Charles E. Thorpe, Ed., Kluwer Academic Publishers, 1990, ch. 5.

10. Anthony Stentz. Multi-Resolution Constraint Modeling for Mobile Robot Planning. In *Vision and Navigation: The Carnegie Mellon Navlab*, Charles E. Thorpe, Ed., Kluwer Academic Publishers, 1990, ch. 11.

11. Charles E. Thorpe. Outdoor Visual Navigation for Autonomous Robots. IAS-2, Den Haag, the Netherlands, 1989.

12. Charles E. Thorpe. *Vision and Navigation: The Carnegie Mellon Navlab.* Kluwer Academic Publishers, 1990.

13. C. Thorpe, M. Hebert, T. Kanade and S. Shafer. "Vision and navigation for the Carnegie-Mellon Navlab". *IEEE PAMI 10*, 3 (1988).