An Intelligent Predictive Controller for Autonomous Vehicles

Alonzo Kelly

CMU-RI-TR-94-20

The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

May 2, 1994

©1994 Carnegie Mellon University

This research was sponsored by ARPA under contracts “Perception for Outdoor Navigation” (contract number DACA76-89-C-0014, monitored by the US Army Topographic Engineering Center) and “Unmanned Ground Vehicle System” (contract number DAAE07-90-C-RO59, monitored by TACOM).

The views and conclusions expressed in this document are those of the author and should not be interpreted as representing the official policies, either express or implied, of the US government.

Abstract

Performance of high speed rough terrain autonomous vehicles is compromized by inadequate real time response characteristics, poor models of vehicle dynamics and terrain following, and suboptimal allocation of limited computational resources. A new approach to high speed rough terrain autonomy is presented which solves these problems through a real-time managed response time strategy, a 3D state space dynamics and terrain following model, and minimum throughput adaptive perception and planning strategies. The system which has been implemented is called RANGER. This report overviews the implementation of the RANGER code.

Table of Contents

List of Figures

1. Introduction

RANGER is an acronym for Real-time Autonomous Navigator with a Geometric Engine. It is a computer program which allows a vehicle to intelligently drive itself over rugged outdoor terrain. It has been implemented for the HMMWV, a military off-road vehicle, and is designed for

- the highest possible speed of continuous motion
- on the roughest possible terrain
- for the longest possible excursion

1.1 Components

The system is composed of:

- A state space controller, described in [29], which implements local planning, path planning, path generation, path following, goal seeking, and coordinated control of steering, brake, throttle, and sensor pan/tilt actuators.
- A navigation Kalman filter, described in [31], which generates a vehicle position estimate based on all available information.
- An adaptive perception algorithm, described in [30], which forms a model of the surrounding environment in a minimal throughput manner that is tightly coordinated with the controller.
- A graphical simulation and development environment.

1.2 Commentary

RANGER takes a new view of the high speed autonomy problem which enables it to achieve higher speeds. Historically, the view has been one of image processing on the one hand sometimes generating maps, sometimes not, planners on the other searching a configuration space for safe trajectories and generating commands, and control on the other doing its best to follow those commands. This view is not optimal for high speed rough terrain work.

The fundamental requirement of safe vehicle control is that of maintaining safety, and ultimately, this has only an indirect relationship to the contents of a map, and an even more indirect relationship to the contents of an image. Image processing is only important to the degree that it predicts safety. It is not an end in itself, and considering it as such has been shown to be responsible for wholesale waste of limited computational bandwidth. Ultimately, high fidelity models of vehicle safety must be explicit models of vehicle safety, and not image operators because the mapping from one to the other is very complicated. In rough terrain, the emphasis shifts from the spatially localized obstacle to an explicit evaluation of tipover propensity and various kinds of collision with the terrain.

1.3 Acknowledgments

Tony Stentz, Martial Hebert, Barry Brumitt and R Coulter made significant creative contributions to this work and they deserve a large measure of credit for the groundbreaking work which lead to it. The overall system arises from the synergy of the team and the free exchange of ideas in an academic setting.

1.4 Features

Some of RANGER's features include:

- A removable real time kernel for embedded applications.
- An extensive library of development support tools. These tools operate at arms length to the kernel and still permit image processing to proceed at 10 Hz on a SPARC II in embedded applications. They include:
    - Event server and data logger
    - Animated X11 based wireframe and image graphics
    - Configuration parameter C language interpreter
    - Visible surface ray tracing sensor simulator
    - State space vehicle kinetic simulator
    - Fractal geometry terrain simulator
    - Command line configuration parser
- A highly parameterized, extendible object oriented architecture which permits ease of interoperation across different vehicle testbeds.
- A simple elegant design which, for the real time kernel, can be implemented in about 1000 lines of C code.
- A voting mode which allows it to operate as a steering behavior inside a larger system.
- Generalized interfaces for sensors of different modalities, with different projective kinematics, reporting results in different coordinate systems.
- High fidelity feedforward system dynamics models.
- Simultaneous integrated perception of geometry and terrain type.
- Simultaneous integrated control of steering, speed, brake and sensor pan/tilt actuators.
- Integrated position estimation and real time identification Kalman filter.

2. Procedural View

When the various development features are removed, the remaining modules are suitable for embedded real time environments. These core modules are called the **real time kernel.** The real time kernel is a layer around the control laws which connect them to the vehicle. This section provides an overview of the real time kernel which follows the data flow diagram convention of Ward and Mellor for real time systems [46].

The diagram convention used here is the **data flow diagram**. In these diagrams, the bubbles represent transformations of signal flows, and arrows signify data. Each diagram has an implicit update frequency or characteristic cycle time associated with it. For the present purpose, consider that double arrows indicate time continuous data and single arrows indicate intermittent data flows. Rectangles indicate external entities and parallel lines are called stores. A store indicates memory of data beyond a single characteristic cycle time.

2.1 Real Time Kernel

At the highest level, the system can be considered to consist of 4 modules as shown in the following data flow diagram.

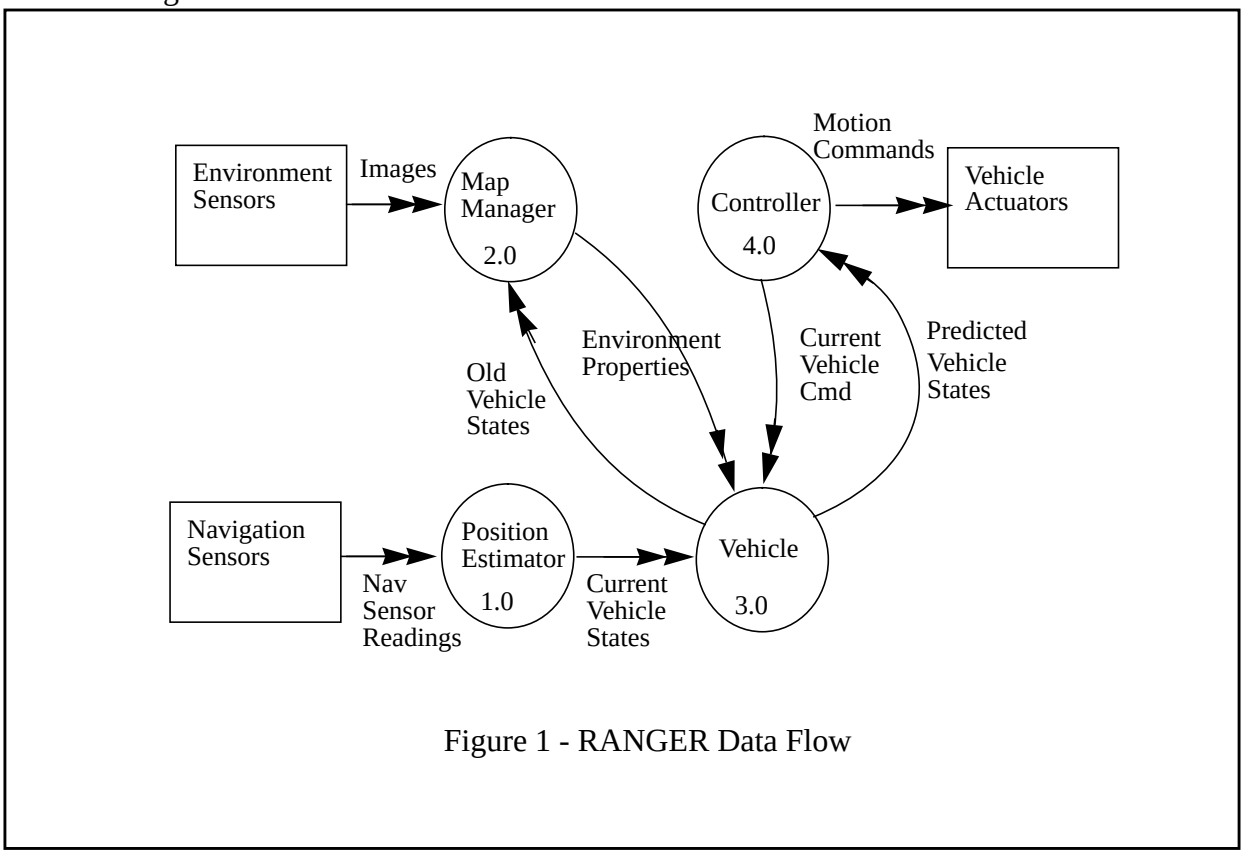

Figure 1 - RANGER Data Flow

2.1.1 Position Estimator

The **Position Estimator** is responsible for integrating diverse navigation sensor indications into a single consistent indication of vehicle state. Vehicle state information includes the positions of all actuators and some of their derivatives, and the 3D state of motion of the vehicle body. This module may be the built-in navigation Kalman filter or another system which generates the same output.

2.1.2 Map Manager

The **Map Manager** integrates discrete samples of terrain geometry or other properties into a consistent terrain map which can be presented to the vehicle controller as the environmental model. It maintains a current record of the terrain immediately in front of the vehicle which incorporates all images necessary, and which automatically scrolls as the vehicle moves.

The map manager forms an abstract data structure when combined with the terrain map itself. Images are transitory input flows at this level of conceptualization. Consumers read or write the map in navigation coordinates and are ignorant of the underlying transformation and scrolling operations.

2.1.3 Vehicle

The **Vehicle** object is both the control loop feedforward element and an abstract data structure which encapsulates the vehicle state. This module incorporates FIFO queues which store a short time history of vehicle states and vehicle commands. Old state information is required by the map manager in order to register images in space. The current vehicle state is used as the initial conditions of the feedforward simulation. Old commands are used in the feedforward simulation as well. This module also provides daemons which compute derived properties of vehicle state such as the planning range window and the maximum safe speed and curvature. Another set of daemons is used to convert curvature, steer angle, and turn radius to any of the other two based on the vehicle state and the wheelbase.

2.1.4 Controller

The **Controller** object is responsible for coordinated control of all actuators. This module includes regulators for sensor head control, an obstacle avoidance tactical controller and a path following strategic controller. It also incorporates an arbitrator to resolve disagreements between the latter two controllers.

2.2 Map Manager

The Map Manager is distinguished from a perception system in the following way. It is considered that the generation of images is the function of perception and the processing of them is the function of the map manager. Image generation may be a simple as a hardware function for color, intensity, or laser range images, or may involve nontrivial processing for stereo or terrain type images. Many Map Manager functions are implemented as macros for speed.

This module converts range image sequences to a large number of samples of the geometry of the vehicle's immediate environment specified in the navigation cartesian coordinate system. The module incorporates an image processing element which converts a raw image into an ideal image, an adaptive perception element which determines which region of the image contains new information and therefore warrants the computational cost of coordinate conversion, and finally the coordinate transformation itself.

2.2.1 Coordinate Transformation

The coordinate system conversion proceeds from image space all the way to global coordinates. Therefore the vehicle position estimate specified in the vehicle state is required in order to do this transformation. Old vehicle positions are used in the coordinate transformation in order to remove the distortion associated with the delay of the perception and communication electronics.

The data flow diagram for the coordinate transform component of the map manager object is given below. The mathematics of these transforms are documented elsewhere [26]. This section takes a range image as input and produces samples of geometry in navigation coordinates as output.

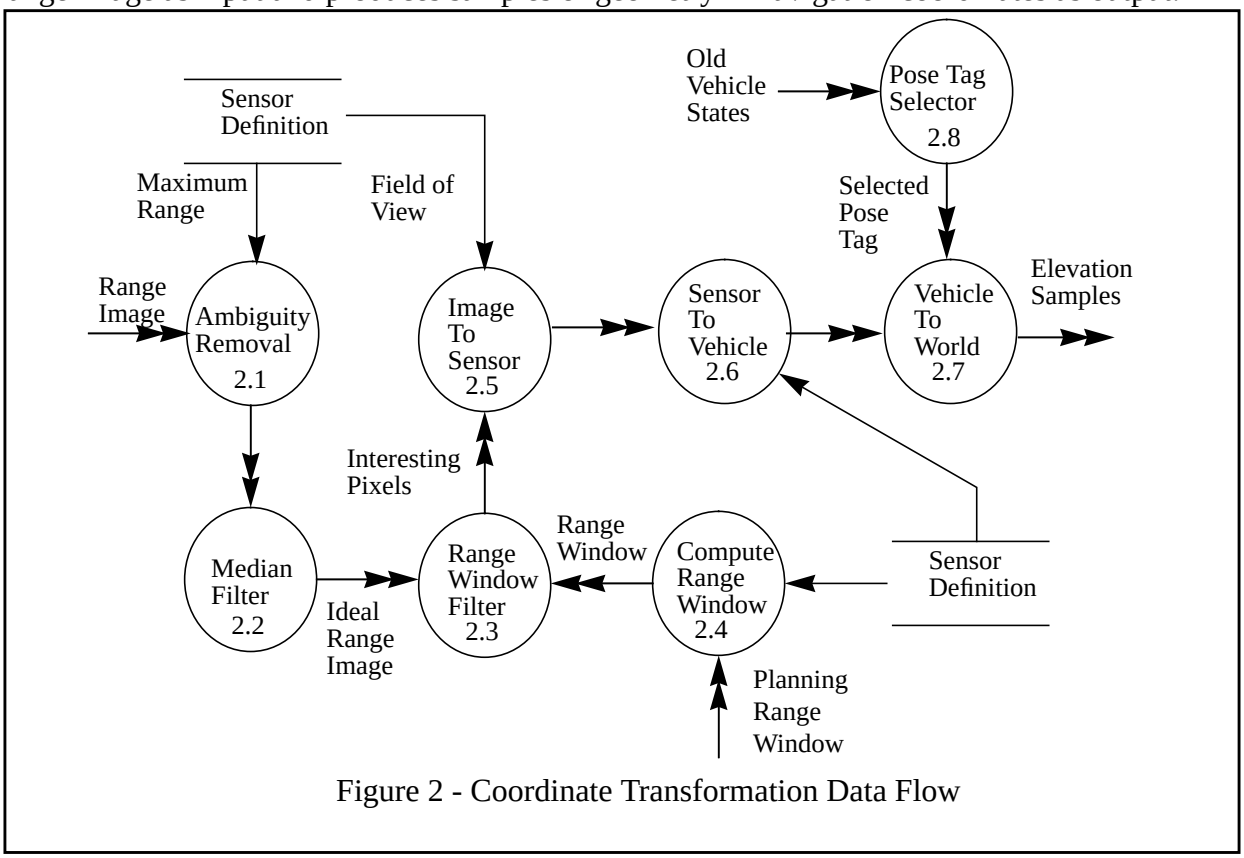

Figure 2 - Coordinate Transformation Data Flow

2.2.1.1 Ambiguity Removal

For amplitude modulated sensors, range is available modulo some effective modulator wavelength. This module reconstructs an ideal range image from a range staircase generated by an AM laser rangefinder.

2.2.1.2 Median Filter

This module uses the well known median filter algorithm to reduce image noise without smoothing the entire image as a side effect.

2.2.1.3 Range Window Filter

This module implements the adaptive perception algorithm. For each column of the image, the algorithm walks up from the bottom to the top to determine the range pixels which are within the current region of interest, and only those pixels are passed onto later stages in the processing.

2.2.1.4 Range Window Computation

This module transforms the planning range window, supplied by the vehicle object, into a range window in sensor coordinates. The transform accounts for the position of the sensor on the vehicle, the latency of the sensor, and the size of the vehicle wheelbase.

2.2.1.5 Image to Sensor

Based on the projection model for the sensor, which may be azimuth polar, elevation polar, or perspective, this module converts coordinates from (range, row, col) to (x,y,z) in a sensor fixed cartesian coordinate system. A lookup table called the projection table is used which specifies the direction cosines of the ray through each pixel for efficiency reasons.

2.2.1.6 Sensor to Vehicle

Based on the position and orientation of the sensor frame on the vehicle, this module converts coordinates from the former to the latter via multiplication by the relevant homogeneous transform. If the sensor is mounted on a head, the head transform is incorporated in real time. If not, the transform is fixed and is computed once at start-up.

2.2.1.7 Vehicle to World

This module uses the position of the vehicle ***when each pixel was measured*** in order to convert coordinates into the global frame of reference via the relevant homogeneous transform. Elements of this transform change as the vehicle moves, so it is recomputed each cycle.

2.2.1.8 Pose Selector

This module interpolates linearly between old vehicle states based on the known physical scanning pattern of the rangefinder (or a single pose can be used for stereo) and assigns a unique pose to every pixel in the image. In this way, both motion distortion and aliasing in the terrain map are avoided. The transforms required to form the range image (such as disparity to range for stereo) are considered to exist outside the system.

2.2.2 Map Management

The map management portion of the map manager deals with the transformation of data into and out of the discrete representation of the terrain map. The data flow diagram for this component of the map manager module is given below.

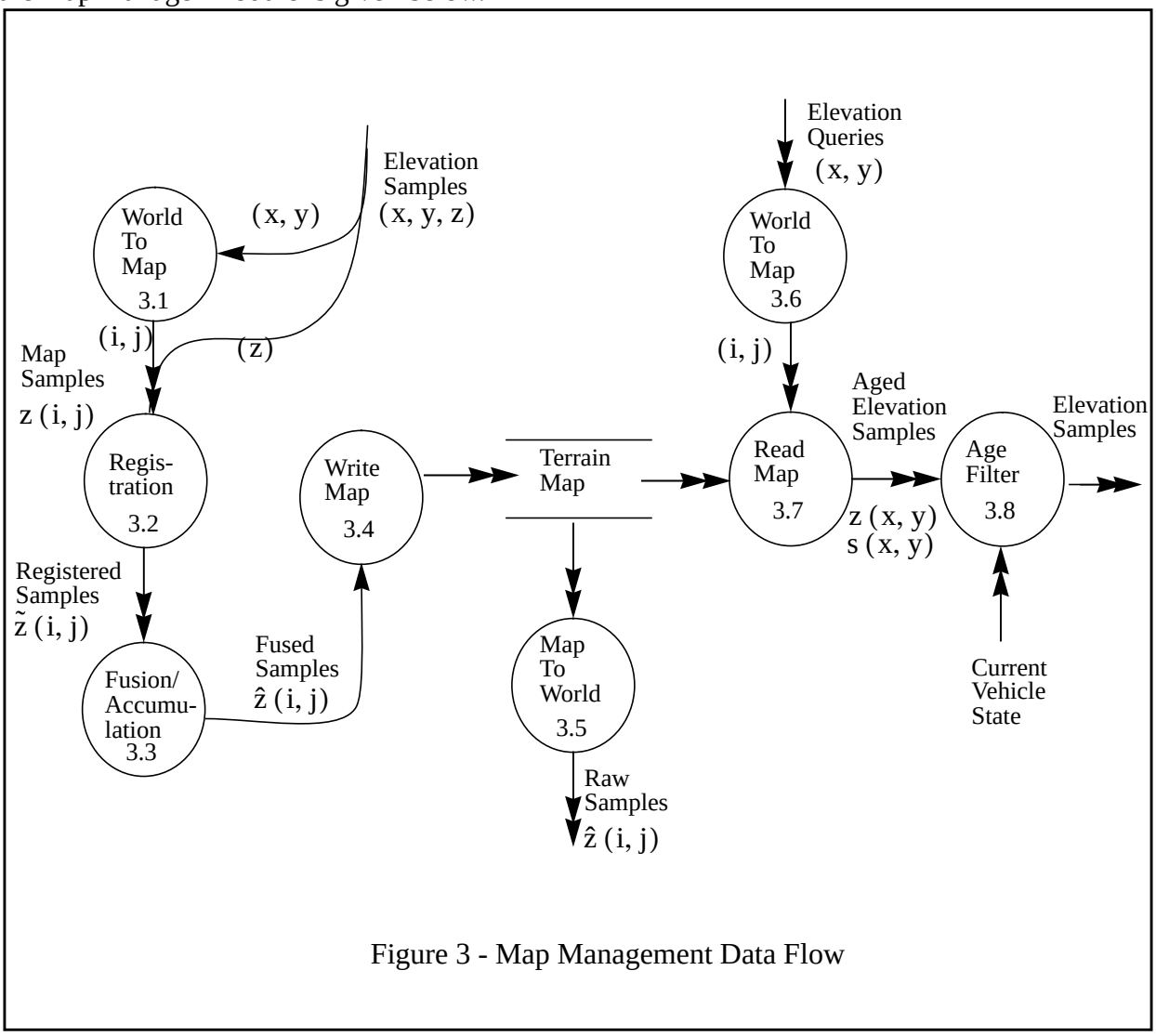

Figure 3 - Map Management Data Flow

2.2.2.1 World to Map

This module converts coordinates from (x,y) to (i,j) in order to process terrain samples into a map. A single block of memory is used to store elevation and the (i,j) coordinates of a point are computed by modulo arithmetic on the input (x,y) tuple.

2.2.2.2 Map to World

This provides the inverse transformation to the above for debugging purposes.

2.2.2.3 Registration

This module recovers the vehicle excursion between images by matching the overlapping regions of consecutive images. Currently, only the elevation (z) coordinate is matched and this works best in practice due to errors which are, as yet, unidentified. When the z deviation is computed, it is applied to all incoming samples in order to remove the mismatch error.

2.2.2.4 Fusion/Accumulation

After the mean mismatch error is removed, there are still random errors in the elevation data. This module computes mean and standard deviation statistics for map cells in situations when two or more range image pixels from the same image (or optionally, from consecutive images) fall into the same map cell.

2.2.2.5 Write Map

This module performs the physical memory write operation. When pixels are placed in the map, they are tagged with the current distance travelled in order to implement a **cell aging feature** that prevents overlap of separate pages of geometry. This is a very efficient mechanism for avoiding the problems of page overlap. It is easy to implement because of the finite radius of curvature of the vehicle.

Aging is performed on a cell by cell basis in the map so that old pages do not poke through the holes in new images. They will be correctly discarded based on age even though they are still physically in the map. The mechanism works correctly for a stationary vehicle because distance is used instead of time as a measure of age.

2.2.2.6 Read Map

This module performs the physical read of memory when a query is received.

2.2.2.7 Age Filter

When the map is read, this module checks the age of the information against the current distance of the vehicle. If the information is too old, the cell is treated as unknown outside the map manager.

2.3 Vehicle

The **Vehicle** consists of models for each of the submodels discussed earlier. These are suspension, propulsion, steering, and dead reckoning. It also provides queue access functions and various daemons for accessing the state information conveniently. The data flow diagram for the vehicle simulator is given below.

2.3.1 Simulation Feedforward

The simulation component of the vehicle object serves double duty as the feedforward element in the tactical controller and as a simulation tool for simulated runs in a simulated world. The commanded trajectory is specified in terms of curvature and speed and the predicted vehicle states are the predicted vehicle response to that command given in terms of the six axis position and orientation of the vehicle, for each command, as a function of time.

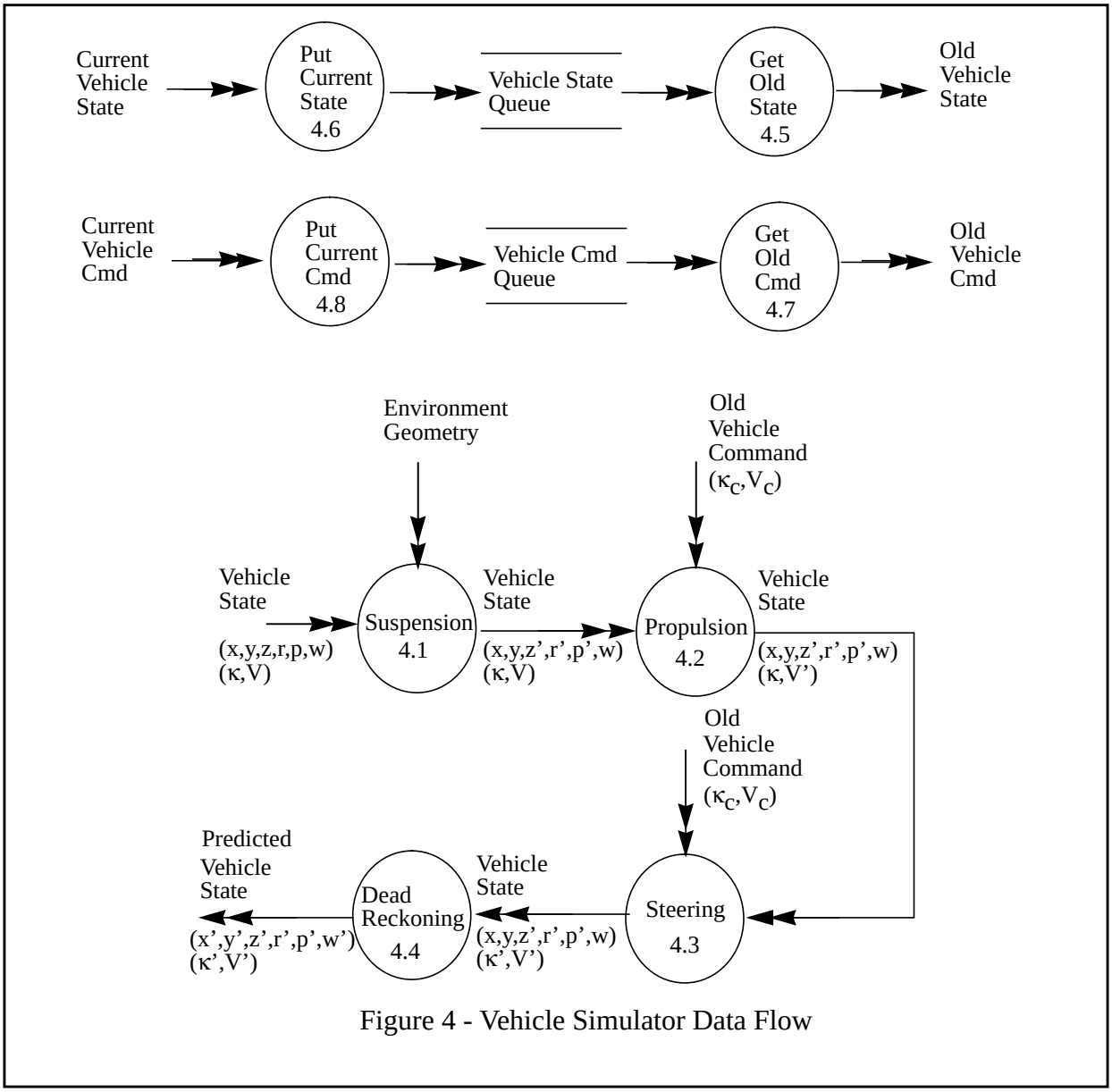

Figure 4 - Vehicle Simulator Data Flow

2.3.1.1 Suspension

This module uses the current estimate of position and heading and conceptually, drops the vehicle slowly onto the terrain map in order to determine the attitude it achieves when it settles. Strictly speaking, the attitude determination problem is nonlinear and nonlinear techniques are necessary to solve it. However, the simple technique used here works well in practice.

This module is also responsible for **on demand interpolation** of the terrain map. It computes the elevation of the terrain under each wheel and if the map is unknown, it attempts to propagate elevation forward from the last known wheel elevation along the response trajectory. Interpolation is limited to a small distance which corresponds to the assumed smoothness of the terrain.

2.3.1.2 Propulsion

This module computes the speed for the current cycle from the speed of the last cycle and the speed command.

2.3.1.3 Steering

This module implements the differential equation which models the transfer function of the steering column as a delayed, limited, first order system in velocity. Basically, the current steering position and velocity are the initial conditions for the differential equation and a straightforward rectangular rule integration scheme gives the response of the steering to the input command. A **bicycle model** approximation to the Ackerman steering model is used.

2.3.1.4 Dead Reckoning

This module computes the Fresnel integrals which convert the speed and curvature signals into position and heading. Correct 3D linear and angular dead reckoning equations are used.

2.3.2 Vehicle Daemons

The daemons perform simple transformation functions on data on an as-required basis.

2.3.2.1 Put Old State

This module stores incoming time tagged state information in the state FIFO. Normally this data is generated from an external sensor and the position estimator.

2.3.2.2 Get Old State

This module looks up old state data. Normally, this data is needed by the map manager when processing images.

2.3.2.3 Put Old Command

This module stores incoming vehicle commands and time tags them. Normally, this data is generated by the controller. After a command is issued to the real vehicle, a copy is stored in the queue.

2.3.2.4 Get Old Command

This module looks up old command data. This data is needed by the vehicle itself in implementing feedforward of delays.

2.3.2.5 Get Planning Window

This module uses the current vehicle velocity to determine the lookahead required in the planning algorithm. Basically, the idea is that there is no point looking for obstacles that the system has already failed to avoid or in looking where the vehicle cannot go. Only those places where there is currently an option of going need be perceived. Based on the current velocity, the terrain to wheel coefficient of friction, the maximum curvature, and the turning reaction time, the region of interest is specified in polar coordinates as the range window.

2.3.2.6 Get Adaptation Speed

The problem of computing the planning window is circular because the trajectory traversed is not known until the speed is, and the speed is not known until the trajectory is. This module uses either the current speed when a human controls throttle or the mean predicted speed when the vehicle has a propulsion controller for the purposes of computing the planning window.

2.3.2.7 Get Turn React Time

This module estimates the turning reaction time from an estimate of the dynamics of the steering column for the purposes of calculating the planning window.

2.4 Controller

Most aspects of the **control laws** (with the exception of feedforward itself) are implemented as controller modules as illustrated below.

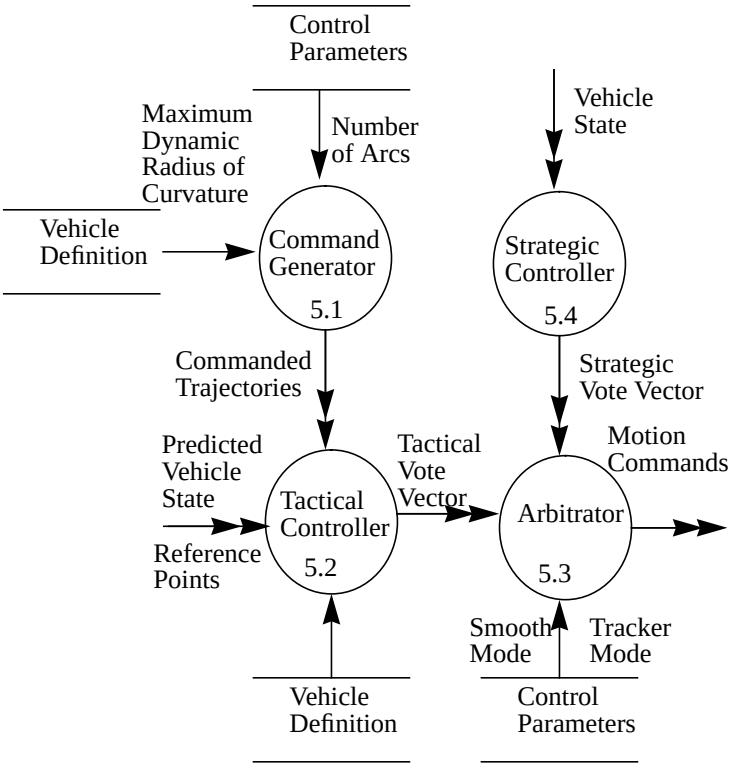

Figure 5 - Controller Data Flow

2.4.1 Command Generator

The **Command Generator** module uses knowledge of the vehicle kinematic steering constraints and the dynamics of rollover to compute a set of equally spaced command trajectories which are presented to the vehicle simulator. The data flow for this module is given below.

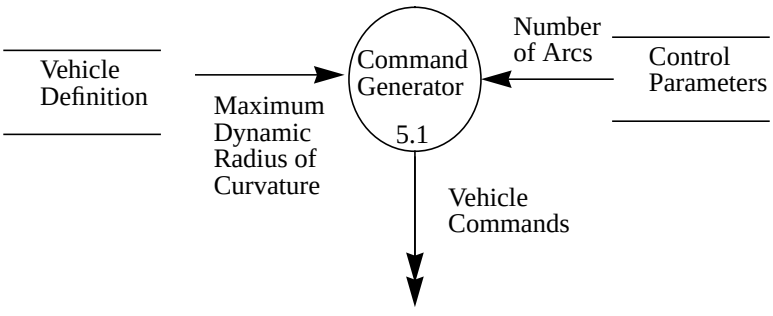

Figure 6 - Command Generator Data Flow

2.4.2 Tactical Controller

The **Tactical Controller** is a module which attempts to maintain nonzero forward velocity while avoiding tipover, collision, or other hazardous situations. Based on the response state signals, the tactical controller forms a holistic noise immune estimate of the overall merit of turning in each direction, chooses the best alternative, and sends a command vote vector to the arbitrator for integration with the strategic vote. The data flow diagram is given below.

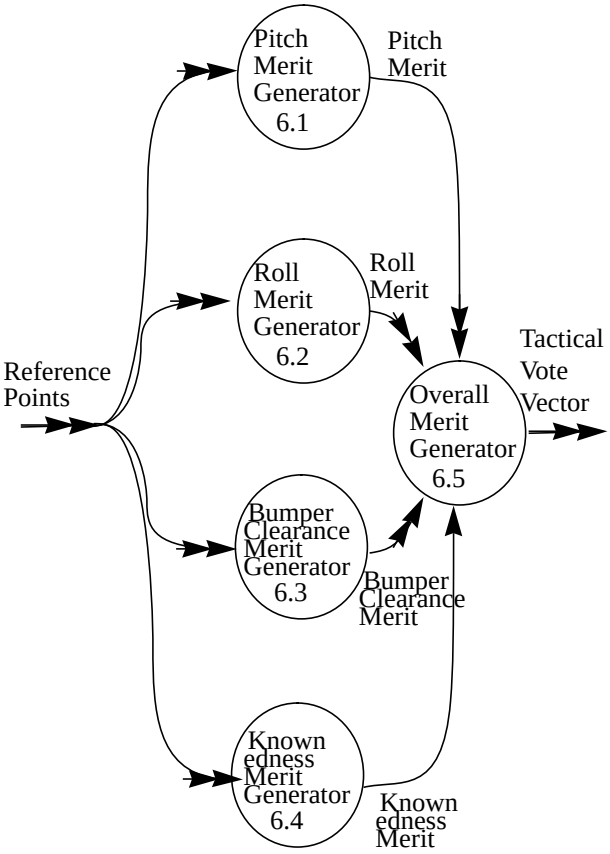

Figure 7 - Tactical Controller Data Flow

A basic premise is that the hazard signals, for some suitable hazard vector, encode all information necessary to assess safety in a minimalist manner. The hazard signals themselves are computed in this module from the reference point coordinates.

The hazards currently computed are static stability and the information density of the signals, called knownedness (until a better name arises). Extensions to body collision are straightforward. Bumper collision was once implemented and removed because of the poor signal to noise characteristics of the signal. This module also stores a time history of vehicle state in signal form for graphical display purposes.

Conceptually, this module operates by integrating a three dimensional scalar field into a single decision. The field can be thought of as merit(time, hazard, curvature). The speed dependence is accounted for implicitly through the time dependence. Thus, there are a number of hazards considered, and each is a time signal, and each hazard time signal is computed for each candidate

trajectory. The first two dimensions are integrated out in this module to produce the tactical vote vector which is passed to the arbitrator for the final decision.

2.4.2.1 Pitch Merit Generator

This module computes the signal norm of the pitch turnover hazard signal for each curvature.

2.4.2.2 Roll Merit Generator

This module computes the signal norm of the roll turnover hazard signal for each curvature.

2.4.2.3 Bumper Clearance Merit Generator

This module computes the signal norms for the bumper collision hazard signal for a few points on the front bumper for each curvature.

2.4.2.4 Knownedness Merit Generator

It is considered very undesirable for the vehicle to drive onto unknown terrain. This module computes a norm of the knownedness signal which is basically the degree to which terrain is unknown for sampling or occlusion reasons.

2.4.2.5 Overall Merit Generator

This module computes the minimum of the merit factors of the above four hazards. In principle, the existence of any one of the above hazards is undesirable, so it is appropriate to rate the corresponding command by the worst of these.

2.4.3 Strategic Controller

The **Strategic Controller** module accepts a strategic goal which biases the system behavior in the long term. The strategic goal may be to follow a heading, a fixed curvature, a single point, or a convoluted path. The path tracker component is the only nontrivial one. It accepts a specification of a path to be followed in global coordinates and generates steering preferences which would cause the vehicle to track the path. The tracking algorithm is a modification of **pure pursuit**. The output of this module is overridden by the arbitrator if tracking the path would lead to a hazardous situation. The data flow diagram for the tracker module is given below.

2.4.3.1 Goal Point Generator

This module uses the current vehicle position and heading to determine the closest point on the path which is outside the pure pursuit **lookahead distance**. The lookahead distance is adaptive to the current tracking error - increasing as the error increases. In this way, when an obstacle avoidance maneuver causes significant path error, the algorithm will search to reaquire the path on the other side of the obstacle instead of causing a return to the front of the obstacle.

2.4.3.2 Curvature Generator

This module computes a preference curvature for the vehicle based on where it is now, and the position of the goal point on the path. This algorithm is a kind of pure pursuit implementing a proportional controller on heading where the error signal is generated from the angle between the vehicle longitudinal axis and the line from the vehicle to the goal point. The feedforward version chooses the curvature corresponding to the arc command which is closest to the goal point.

2.4.3.3 Path Recorder

The tracker incorporates a recording facility which permits the recording of paths which are traversed when a human is driving. This permits convenient generation of test trajectories. Raw positions are recorded each cycle to a file which is further processed later when it is read during playback.

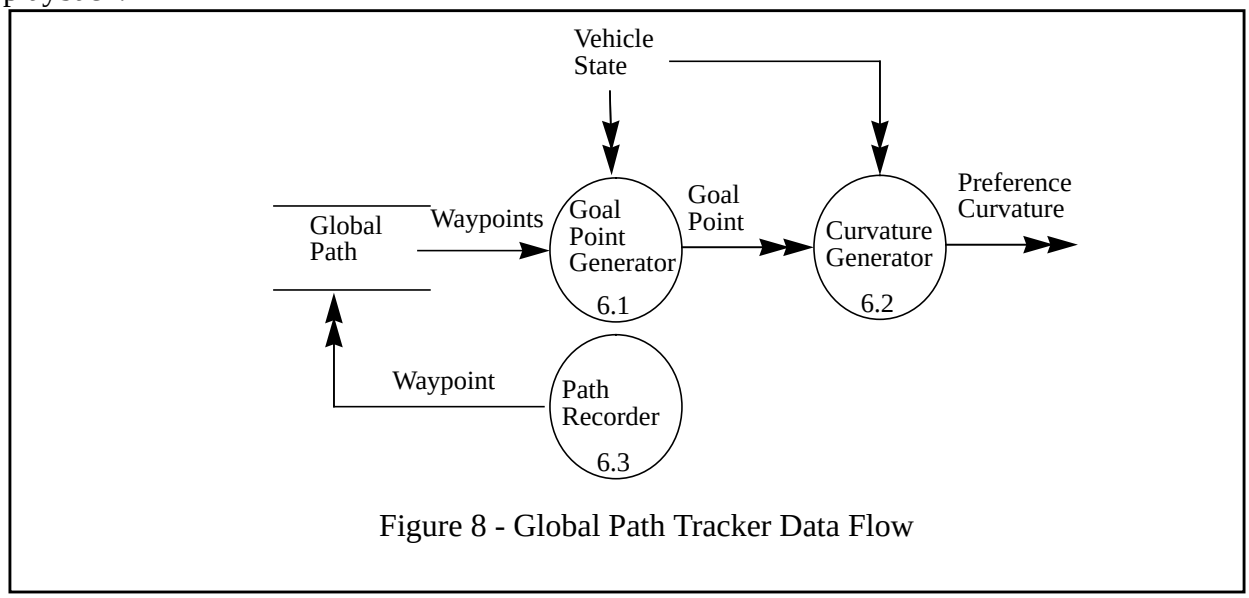

Figure 8 - Global Path Tracker Data Flow

2.4.4 Arbitrator

The arbitrator integrates out the curvature dimension of the vote vectors of both the tactical and strategic controllers to generate a single command to the hardware. When the system operates as a voting element in a larger system, no physical i/o takes place and the vote vectors are available externally. The data flow diagram for the arbitrator module is given below.

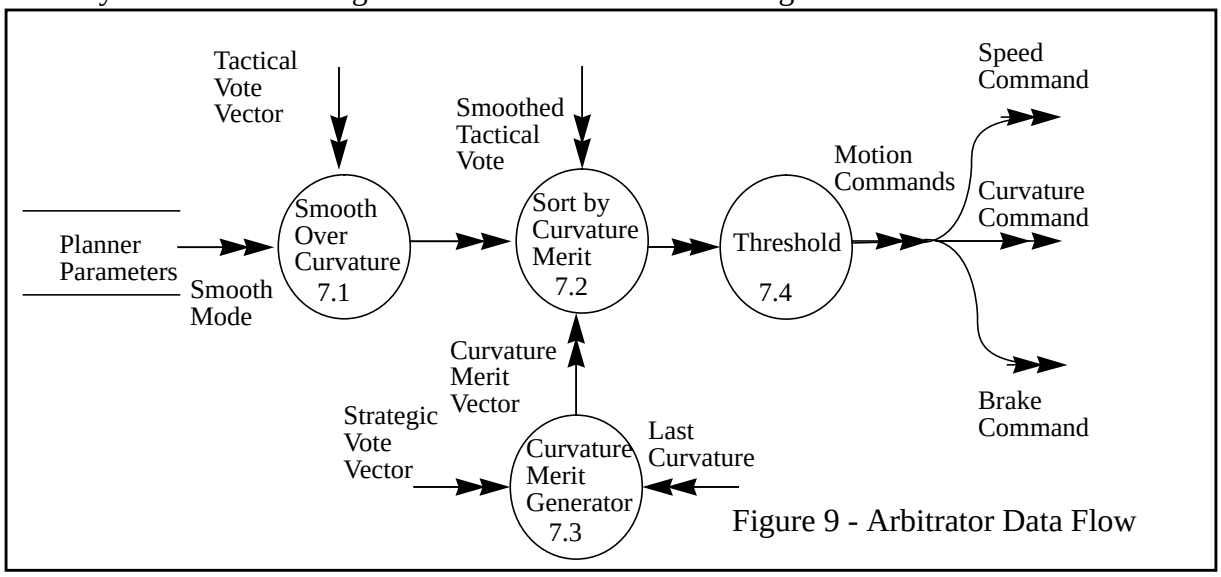

Figure 9 - Arbitrator Data Flow

2.4.4.1 Smooth Over Curvature

At this point, the tactical vote vector has reduced the 3D merit field to a sampling of the vector merit(curvature). The environment exhibits a degree of smoothness, so it is appropriate at times for poorly rated trajectories to degrade the ratings of their spatial neighbors. This mechanism will reduce the impact of errors in both feedback and actuation by causing the vehicle to give obstacles a wide berth. At times, this module is disabled depending on the spacing of the candidate trajectories investigated. The algorithm itself is simple **Gaussian filtering** in one dimension.

2.4.4.2 Curvature Merit Generator

This module computes a merit for the strategic vote vector elements along the last remaining dimension of the field based on the distance of each command from the maximum of the strategic vote vector measured in curvature space. Effectively, this measures the degree of disagreement between the two controllers for each element of the tactical vote.

2.4.4.3 Sort by Curvature Merit

This module sorts the tactical vote vector using the curvature merit of the corresponding strategic votes as the key. Candidate commands are then effectively presented to the final decision process in order of preference.

2.4.4.4 Threshold

This module implements the final decision. The first trajectory in the sorted list which exceeds a merit threshold is chosen and sent to the vehicle controller. If no steer angle exceeds the threshold, a panic stop command is issued.

3. Object Oriented View

The second highest level of conceptualization of the system is the **object library**. Users of components of the system normally interact with it at this level. In order to simplify interfaces and enhance generality and portability, an object oriented hierarchy is incorporated which is structured for real time use. It is intended that the modularity of the object hierarchy tree match as closely as possible both the physical modularity of vehicle degrees of freedom and the typical systems modularity of contemporary robot vehicles.

3.1 Objects in Vanilla C

C lacks many of the refinements necessary for “pure” object oriented programming. However, an object oriented design can still be accomplished and some degree of object oriented programming is still possible.

3.1.1 Principle

By and large, all objects are C structures and the procedures that process them have no static memory, so they are effectively re-entrant. All information associated with the object is retained externally and supplied through either the handle or inheritance. This has the advantage that any number of copies of any object are possible, and no extra external variables are needed to distinguish them from each other. It also has the advantage that memory is reduced because only one copy of the instructions is retained.

3.1.2 Lineage and Inheritance

Objects point to their parents as a rule, and some point to their children if this link is useful.

3.1.3 Lexical Conventions and Messages

As a general rule, an object of type Map is declared as:

```
Map mymap;
```

The operator destroy is applied to the object thus:

```
map_destroy(mymap);
```

After which the handle points to garbage and should not be used. Function names start with the type of object and the operator is appended to the name. the handle is normally passed as the first argument and subsequent arguments modify the message.

3.1.4 Creation and Handles

Objects are created by calling the creation operator. This function returns a handle to the newly created object and all subsequent manipulation will use the handle to uniquely identify it:

```
Map map_create(args);
```

Most creation operators perform reasonableness checks on the supplied arguments. There is considerable difficulty involved in implementing variable numbers of arguments in the creation functions, though it can be done. For this reason, system revisions will often appear as changes in the argument list supplied to the creation functions.

3.1.5 Configuration and Delayed Binding

Most objects are allocated and configured at run-time. This **delayed binding** principle permits a single pure code segment to be used for all configurations of the system without recompilation or relinking. There is a very slight computational cost associated with delayed binding, because it implies conditionals executed at run-time rather than compile or link time, but it is not considered worth bothering about. Typically, there are system level efficiency issues which far outweigh the cost of delayed binding.

3.2 Object Hierarchy

The use of an object hierarchy has several advantages. The system can be implemented so that it makes few assumptions about many of the specifics of the vehicle. For example, any number of sensors of any modality can be used and the system does not know or care. Some can be mounted on movable sensor heads and some not etc.

The implementation consists of three main specialized objects, the vehicle, map manager, and controller objects[1]. These objects use the services of a lower level generic object substrate which is not as specific to the application, and which provides services for:

- kinematics
- image and map processing
- simulation
- graphics
- data logging
- text parsing

These can be visualized in the following figure.

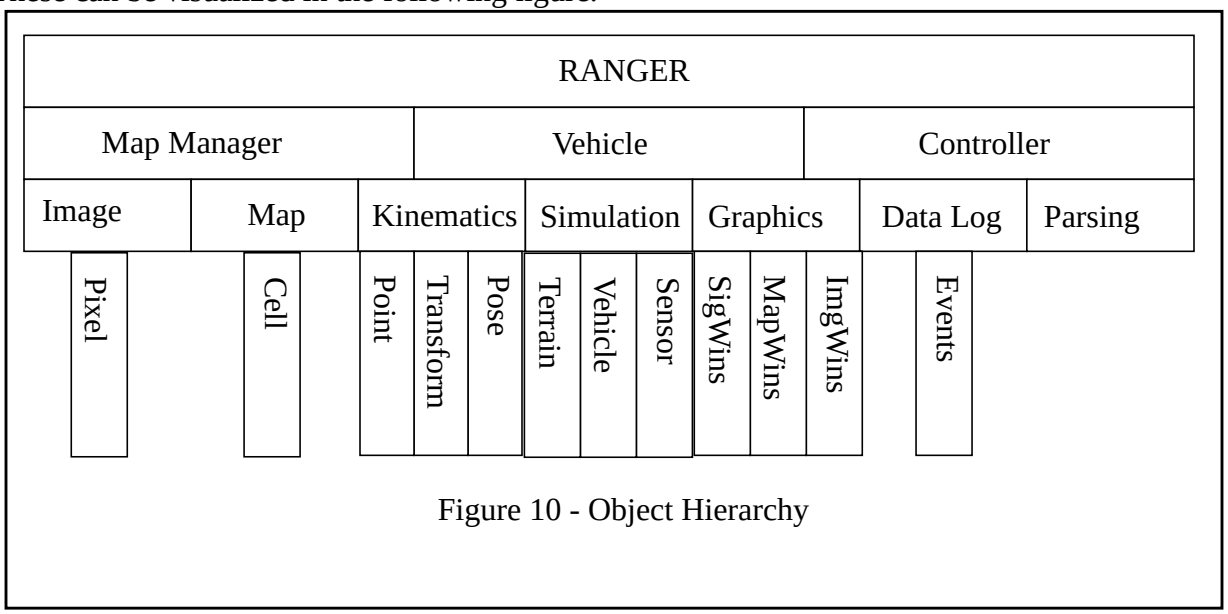

Figure 10 - Object Hierarchy

1. A Kalman filter is a fourth arms-length object.

3.3 Timing Requirements

The primary reason for the separation into three top level objects is that each has different timing requirements. Ideally, each would be implemented as a separate process, and they would all run asynchronously. Three main data streams connect the system to the physical world. These are:

- image input stream
- vehicle state input stream
- vehicle command output stream

Additionally there may be sensor head state input streams and sensor head command output streams. The real time clock is also considered to be an external device. The architecture is indicated in the following figure:

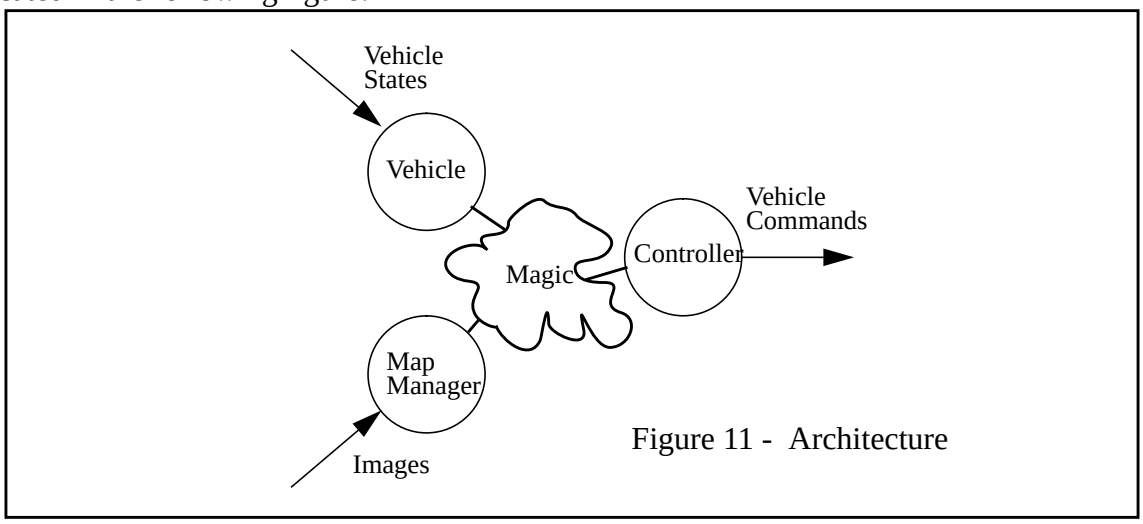

Figure 11 - Architecture

3.3.1 Image Input Stream

The frequency requirement of the image input stream is governed by the rate at which images are generated. However, because images generally overlap on the ground, it is not necessary to process all of them. The adaptive perception mechanisms will minimize the amount of image information processed if they overlap anyway. Thus, the practical frequency requirement of the image input stream is the guaranteed throughput requirement.

3.3.2 Vehicle State Input Stream

The frequency requirement on this stream is governed by the need to measure the environment at sufficient fidelity because small errors in vehicle attitude act over the radius measured from the sensor to the environment. Normally, this frequency is much higher than the image frequency requirement. The result of this that ***it is not feasible to consider image processing to be indivisible*** and measure vehicle state only between images. Some options for managing this requirement include:

- use an **interval timer** to interrupt image processing to allow a handshaked asynchronous measurement of state
- use a synchronous buffered communications channel which is written to the system by an agent close to the vehicle state indicators
- implement the vehicle object in a separate task from image processing

3.3.3 Vehicle Command Output Stream

The frequency requirement on this stream is governed theoretically by the control fidelity requirement for the vehicle to track commands faithfully. However, vehicle natural frequency may be much slower than this and little is achieved against control fidelity by attempting to drive the vehicle faster than it can respond. Thus, the practical requirement on this stream is the vehicle natural frequency. The controller need not run any faster than this frequency.

3.4 Object Definitions[2]

In general, all objects know how to create themselves and destroy themselves. Most can read and write themselves from files. Many can draw themselves in windows. Most can copy themselves. Many objects which represent physical reality can simulate themselves. Objects at the top level are specialized to the application.

3.4.1 Terrain Maps

Terrain maps are root level entities implemented as two dimensional array data structures which encode elevation statistics, rgb color, and terrain type information. Conceptually, terrain maps are the “world object”. Terrain maps intrinsically incorporate an explicit assumption that the environment can be represented in 2-1/2 D with elevation oriented along the assumed “vertical” direction. Typically, one terrain map is used to compute an evolving environment model from perceptual sensor data. Another can be used to represent ground truth information in simulated environments. A third can be used to represent single images in the vehicle coordinate system.

Terrain maps include information about their resolution and they know how to map x, y coordinates into their array structure. Terrain map access routines can incorporate a wrapped world model so that they are effectively infinite. Terrain maps know how to draw themselves in a map window on a display. Terrain maps know how to wrap or scroll themselves. They know how to simulate terrain geometry. The creation operator is:

```
Map map_create
(
int rows,   /* number of rows */
int cols,   /* number of columns */
double x0,  /* x coordinate of cell (0,0) */
doule y0,   /* y coordinate of cell (0,0) */
double dx,  /* x resolution */
double dy   /* y resolution */
)
```

Maps are accessed in the coordinate system specified in the creation operator. The association of cell (0,0) with a point is done for speed reasons, since map access is a deeply nested process. It is expensive to clear a map, and there are times when it would affect performance and is unnecessary, so this must be done explicitly via map_clear() if this is required (it usually is) at start-up.

2. Readers should not rely on the accuracy of these definitions. RANGER is a research prototype. The code is the final authority. These definitions are intended to provide an overview only.

3.4.2 Map Manager

There is usually one map manager object for each terrain map which will be updated from sensor readings. The map manager maps know how to process various types of images into the map representation. The creation operator is:

```
Map_mgr map_mgr_create
(
Map map,          /* map to be managed */
Map lmap,         /* local map */
Vehicle veh,      /* the vehicle object */
Cntrl cntrl,      /* the controller object */
double nom_cyc    /* nominal cycle time */
)
```

If the local map pointer is not NULL, a terrain map will be constructed in vehicle coordinates as well as the more usual world coordinate map. The nominal cycle time is used to avoid the problem that cycle time cannot be measured the first time the map manager cycles. Subsequent cycles use the actual cycle time of the last cycle. Normally, one provides an image to be processed as follows:

```
map_mgr_process_image
(
Map_mgr map_mgr, /* handle */
Image img,        /* image handle */
)
```

This operator processes the image into the terrain map based on:

- whether it is a true image or a stereo range map
- whether it encodes geometry, color, intensity, or type information
- whether it comes from a sensor on a head or not

It converts coordinates from the vehicle to the world based on the latest vehicle state and integrates the image into the map.

3.4.3 Controller

Controllers know how to control the whole vehicle. Vehicle controllers create speed, steering, and brake commands to control the vehicle. The tactical controller object is an expert in evaluating the existence of hazards in the environment. A strategic controller knows how to control a vehicle to cause it to achieve a strategic goal. A head controller knows how to control a sensor head. Controllers are created incrementally because there are so many parameters. They are also

destroyed incrementally:

```
Cntrl cntrl_create
(
Vehicle veh, /* vehicle handle */
int ncmds,  /* number of commands */
int vote_only,/* vote_only mode */
double thresh /* merit threshold*/
)
```

Which creates a controller to control the indicated vehicle. The number of commands is the number of curvature commands which will be feedforward simulated. In vote-only mode, physical command i/o to the hardware is suppressed. The threshold is the merit threshold below which a panic stop will be issued.

A tactical controller must be created as well. The creation operator is:

```
Tc_Cntrl tc_cntrl_create
(
Cntrl cntrl,      /* parent controller handle */
int smooth,       /* do gaussian smoothing */
double max_pitch, /* pitch tangent limit */
double max_roll   /* roll tangent limit */
)
```

If smooth is set, the controller will gaussian filter the tactical merit vector. The max_roll and max_pitch parameters specify the attitude angles which are considered to constitute extreme rollover hazard. A strategic controller must be created. The creation operator is:

```
enum TrackMode_e {TRACK_NONE,TRACK_RECORD, TRACK_PLAYBACK};
typedef enum TrackMode_e TrackMode;
enum Sc_Type_e {SC_NONE, SC_PATH, SC_POINT, SC_CURV, SC_HEADNG};
typedef enum TrackMode_e TrackMode;

St_Cntrl st_cntrl_create
(
Cntrl cntrl,      /* parent controller handle */
Sc_Type type,     /* strategic goal type */
double curv,      /* goal curvature */
double headng,    /* goal heading */
TrackMode mode,   /* path tracker mode */
char file[],      /* path filename */
double lookahead, /* nonadaptive lookahead */
double gain       /* proportional gain */
)
```

Depending on the type parameter, other parameters can become irrelevant. Depending on the track mode, the track path file will be recorded, played back or ignored. A controller will run both

subcontrollers, arbitrate, and generate a command to its associated vehicle via:

```
cntrl_cmd_vehicle
(
Cntrl cntrl,      /* controller handle */
)
```

This can be done at any time independent of when images arrive.

3.4.4 Vehicles

Vehicles are entities which reflect the physical characteristics of the vehicle. Vehicles point to the map in which they reside. They incorporate a FIFO buffer which remembers a short history of vehicle state so that perceptual information can be time registered with the state of the vehicle when the measurement took place. They also incorporate a FIFO buffer which remembers a short history of vehicle commands which can be used in feedforward control models.

They include a steering model, a propulsion model, and a brake model. Vehicles know how to draw themselves in a map window on a display. Vehicles know how to simulate their response to a command and generate state information over a specified time period. They are created via:

```
Vehicle veh_create
(
Map    map,             /* parent map */
double img_dens,        /* imaging density */
int    speed_cntrl,     /* propulsion controller flag */
double nom_speed_cmd,   /* nominal speed */
double min_turn_radius, /* minimum turn to command */
double max_acc,         /* max lateral acceleration to command */
double wheelbase,       /* longitudinal wheelbase */
double width,           /* transverse wheelbase */
double roff,            /* rear tire to bumper offset */
double foff,            /* front tire to bumper offset */
double tire_width,      /* width of the tires */
double tire_radius,     /* radius of the tires */
double steer_alp_dot_max,/* maximum rate of steering wheel */
double steer_delay,     /* complete delay from state measurement */
double state_latency,   /* complete delay to steering actuator */
double turn_lookahead   /* percent of complete turn to use in lookahead
*/
)
```

Normally, when a position estimate arrives, one updates the vehicle position with:

```
veh_put_state(veh,state)
(
Vehicle veh,            /* vehicle handle */
Vehicle_State state     /* vehicle state handle */
)
```

3.4.5 Vehicle States

Vehicle states incorporate time variable information associated with the physical degrees of freedom of the vehicle body and its actuators. When placing a state in the vehicle, all of the following fields must be updated:

```
struct Vehicle_State_s
 {
 double distance_travelled;
 double curvature;
 double turn_radius;
 double steer_angle;
 double speed;
 double time;
 Pose   pose;
 }
```

The curvature, turn radius, and steer angle are all redundant specifications of the same thing. These can be updated consistently as a unit by calling ***just one of***:

```
veh_set_curvature(state,model,K)
Vehicle_State state;
Vehicle_Model model;
double K; /* curvature */

veh_set_turn_radius(state,model,R)
Vehicle_State state;
Vehicle_Model model;
double R; /* turn radius */

veh_set_steer_angle(state,model,alp)
Vehicle_State state;
Vehicle_Model model;
double alp; /* steer angle */
```

If veh is a vehicle, then the vehicle model is available as &(veh->veh_model).

3.4.6 Poses

Poses are simply 6-vectors of doubles in a standard order:

```
#define X 0
#define Y 1
#define Z 2
#define R 3
#define P 4
#define W 5
typedef double Point[3];
typedef double Pose[6];
```

Use the defined index constants to improve readability.

3.4.7 Sensor Heads

Sensor head models are entities which reflect the physical characteristics of a sensor head. They incorporate a FIFO buffer which remembers a short history of head state so that perceptual information can be time registered with the state of the head when the measurement took place. They incorporate a FIFO buffer which remembers a short history of head commands which can be used in feedforward control models. They include a dynamics model. Heads know how to draw themselves in a map window on a display. They have an associated communications channel which supplies them with sensor data. Sensor heads know how to simulate their response to a command and generate state information for a time period. Details are available in the code.

3.4.8 Sensors

Sensor objects contain field of view and imaging projection information. Sensors point to either the sensor head or the vehicle upon which they are mounted. They may measure rgb color, intensity, range, or 3D position. Sensors are created with:

```
typedef enum {AZIMUTH,ELEVATION,PERSPECTIVE} ProjType;
enum Sensor_Dim_e {SENSOR_1D, SENSOR_3D};
typedef enum Sensor_Dim_e Sensor_Dim;
enum Sensor_Type_e {SENSOR_GEOM,SENSOR_INTEN,SENSOR_TYPE};
typedef enum Sensor_Type_e Sensor_Type;
enum Sensor_Frame_e {SENSOR_SENSOR,SENSOR_HEAD,SENSOR_VEHICLE};
typedef enum Sensor_Frame_e Sensor_Frame;

Sensor sensor_create
(
Vehicle     veh,              /* host vehicle */
Head        head,             /* host sensor head */
int         head_mounted,     /* head mounted flag */
Sensor_Dim  dim,              /* image dimension */
Sensor_Type type,             /* sensor type */
Sensor_Frameframe,            /* coordinate system of image */
ProjType    proj,             /* sensor imaging projection */
double      latency,          /* sensor latency */
double      scan_time,        /* time to scan an image */
double      max_range,        /* range associated with max pixel value*/
double      max_usable_range, /* maximum really useful range */
double      scale,            /* pixel to range or intensity scale */
double      offset,           /* pixel to range or intensity offset */
double      hfov,             /* horizontal field of view */
double      vfov,             /* vertical field of view */
int         rows,             /* rows in an image */
int         cols,             /* cols in an image */
Pose        pose              /* position and attitude wrt host */
)
```

3.4.9 Images

Images point to a sensor object. They are array data structures which may encode intensity, color, range, 3D position, or terrain type information.

In order to support various sensor types, responsibility for the conversion to vehicle coordinates may or may not rest with the sensor interface and control software. This is because it is expected that sensor pointing heads and active scan controllers will be more tightly coupled to the sensor than Map Manager is, and are best able to supply the computational bandwidth necessary to convert coordinates. For convenience, images can have windows around the edges ignored and may be subsampled automatically. This permits permanent removal of bad pixels caused by rangefinder internal reflections, convolution operator support, or stereo baseline separation. Subsampling is useful for square pixel sensors and is critical to high speed performance.

```
Image image_create
(
Sensor sen,       /* parent handle */
int rows,         /* number of rows */
int cols,         /* number of columns */
int start_col,    /* subwindow start col */
int end_col,      /* subwindow end col */
int start_row,    /* subwindow start row */
int end_row,      /* subwindow end row */
int rskip,        /* row subsample factor */
int cskip,        /* column subsample factor */
int max_pix       /* maximum pixel value */
)
```

Subsampling also gives dramatic improvements in the performance of the ray tracing sensor simulator.

3.4.10 Pixels

Pixels store intensity, range, red, green, blue, terrain type or 3D position information. Access macros for individual pixels are provided in the code include files.

3.5 Substrate Objects

The substrate objects are implemented for general use.

3.5.1 Transforms

Transform objects are position and attitude information which relate two cartesian coordinate systems. They can also be considered to be the position and attitude of a physical object expressed in a coordinate system. Transforms know how to convert themselves to and from poses, how to multiply themselves to produce other transforms, how to invert themselves, and how to project their position components onto the axes of other transforms.

3.5.2 Buffers

FIFO buffers are used to store time histories of states and commands. Buffers can buffer any type of object.

3.5.3 Points

Point objects are 3D floating point vectors. They may be positions, or linear or angular velocities. They know which type they are and therefore can transform themselves based on a transform.

3.5.4 Angles

Angle objects are floating point numbers between -PI and PI. They know how to find their cosines and sines very efficiently from lookup tables. They know how to add and subtract themselves and ensure that the answer is an angle in the valid range.

3.5.5 Signals

Signal objects are scalar floating point signals which may or may not be known at any particular point in time. They know how to draw themselves in a signal window on the display.

3.5.6 Events

Events are objects which are associated with the arrival of sensory data. They know how to log themselves in a log file and how to read themselves from a log file.

3.5.7 Clocks

Clock objects are a layer around a physical real time clock. Clocks can be suspended so that simulation cycles do not corrupt the measurement of time used in the real-time components of the system.

3.5.8 Timers

Timers are convenient layers around the interval timer facilities of the underlying operating system. Timers generate software interrupts and have an associated interrupt service routine. Timers are used to fake synchronous data inputs by periodically issuing asynchronous demands for data.

3.5.9 Displays

Displays are the parent objects which own windows.

3.5.10 Windows

Windows are objects that own a portion of a physical screen. Other objects draw themselves in windows. Windows know how to convert coordinates from a hypothetical viewplane onto physical screen coordinates. Windows know how to scroll themselves based on the position of a distinguished object. Mapwins, imgwins, and sigwins are a kind of window which associate a map, an image, or a signal with a window, and keep the screen updated to monitor changes in these central data structures.

3.5.11 Graphics Objects

Various graphics objects know how to animate themselves inside a window. Images, polylines, polymarks, rectangular quadmeshes, rectangles, circles, and arcs are some graphics objects.

4. Test Environment Layer

The **test environment** is the highest level of conceptualization of the system. System interfaces are externalized in order to simplify porting to different environments. Ideally, the system is ported by changing only these interfaces.

In order to simplify system installation, the system modules are not confined to files based on the object hierarchy. Thus, all graphics is done in one file, even though the functionality is shared among all objects. For use in an embedded real-time environment, graphics, data logging, parsing and simulation tools may not be desirable, or supportable in hardware. The relevant functions can be replaced with stubs and then the nonexistent system libraries involved (X11 for instance) can be removed from the Makefile. This is easy to do if it is done at the highest possible level.

The test environment provides a buffer layer which lies between the real-time kernel and the physical i/o drivers. The elements of this layer are:

- graphics facilities
- simulators
- data logger
- configuration parser
- command line parser
- world tree
- physical i/o
- main event loop

This layer exists in order to provide a consistent, capable test environment and to permit reconfiguration for different vehicle configurations. The main() function may reside in this layer, if necessary. The main event loop is usually implemented in this layer.

It optionally:

- intercepts data for viewing purposes
- generates simulated data for nonexistent hardware modules
- records data to the file system
- plays data back for off-line review and analysis

It is the responsibility of the test environment in general and the physical i/o routines in particular, to supply sensor measurements and to dispatch commands. The existence of simulation, parameter editing, and data logging facilities is optional. None are needed to run the system.

Conceptually, the layer insulates the system from the knowledge of whether it is embodied in real hardware or executing in real time. It also insulates the system from the knowledge of whether or not it is the highest level control loop.

4.1 Layout Compromise

The file layout of this layer is a compromise which attempts to satisfy several conflicting requirements. Users who wish to port the system will find that a single file is all that need be edited to hook the system to a vehicle's physical i/o and specify the world. User's who wish to maintain the system will find that conceptual layers are spread over several files and the hierarchy of layers is compromised.

Conceptually, the event loop is the highest level loop in the system, and hierarchy dictates that it belongs in the same file as main(), but because it depends on the actual sensors available it is pushed into the physical i/o file so that users who port the system can change it without knowledge of other system internals.

Information hiding dictates that all functions related to a particular piece of data reside in the same file, but because many users may not want to be bothered with graphical display of i/o, graphics is pushed into its own file and it communicates with the i/o filter via global variables.

For simulation and data logging, if the hierarchy of layers is true, the question arises of whether logging simulated data or simulating logged data will be possible. The latter was chosen so that the data logger can be tested in simulation. However, because the event loop resides in the physical layer, the lowest layer, aspects of simulation must be visible in this layer at least to the extent that external devices not be opened in simulation mode.

Somehow, it should be possible to achieve all things at once, but time limitations prevent revisiting this design question.

4.2 I/O Filter

In principle, any external data interface can be implemented in simulation, from logged data on the file system, or from the hardware and the knowledge of which is being used is very localized. This data dispatching mechanism can be visualized in the following figure:

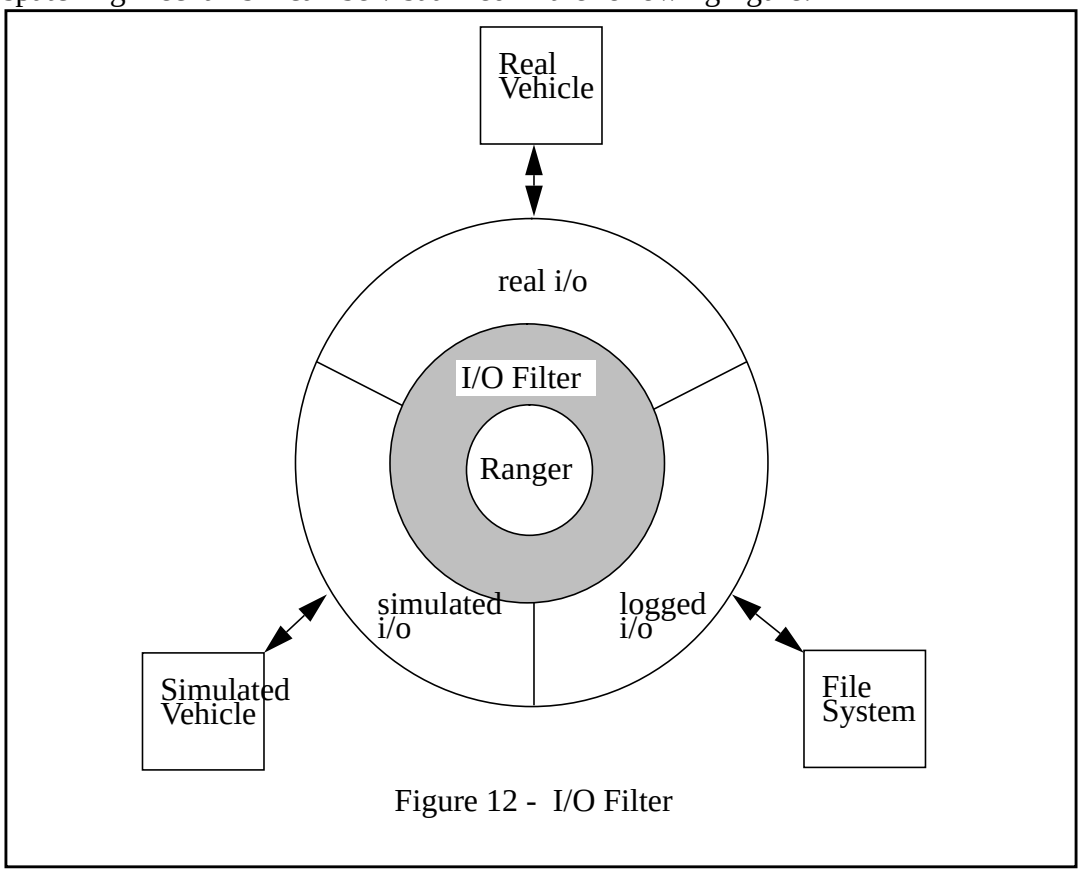

Figure 12 - I/O Filter

4.3 World Tree

The object library and all layers below it makes no assumptions about the configuration of the vehicle. Therefore, it is the responsibility of the test environment layer to specify the configuration of the world at start up. The initialization step involves the creation of a world tree that represents the physical connections of objects with each other and their associated attributes. A typical code fragment for a vehicle incorporating a fixed mount laser rangefinder might be as shown below:

```
world_create()
{
Pose pose;
ranger_terrain_map = map_create(args);
map_clear(ranger_terrain_map);
ranger_vehicle = veh_create(ranger_terrain_map,args);
ranger_controller = cntrl_create(ranger_vehicle,args);
ranger_map_manager = map_mgr_create(ranger_terrain_map,
                                    ranger_vehicle,
                                    ranger_controller,args);
ranger_tactical_controller = tc_cntrl_create(ranger_controller,args);
ranger_strategic_controller = st_cntrl_create(ranger_controller,args);
pose[X] = 0.0; pose[Y] = 2.5; pose[Z] = 2.13;
pose[R] = 0.0; pose[P] = -0.2269; pose[W] = 0.0;
ranger_range_sensor = sensor_create(ranger_vehicle,pose,args);
ranger_range_image = image_create(ranger_range_sensor,args);
}
```

4.4 Events

To enhance portability, the physical i/o interface is defined in terms of a generic io_datum object which serves double duty as a place holder for events because, currently, all events are associated with external i/o. This permits, for instance, automatic management of several sensors without having to assume a fixed number in the interface specification. It also permits the data logger to read the datum type from the log file and discover its size so that it knows how to read the data itself.

The real-time kernel is bound to a physical i/o interface through the event loop in the physical i/o layer. This layer knows the world tree, owns all top level objects, performs physical i/o and

dispatches events. An event object is created as follows:

```
enum Io_Datum_Type_e {IO_DATUM_IMAGE,
 IO_DATUM_VSTATE,
 IO_DATUM_HSTATE,
 IO_DATUM_TIME,
 IO_DATUM_VCMD,
 IO_DATUM_HCMD};
typedef enum Io_Datum_Type_e Io_Datum_Type;

Io_Datum io_datum_create
(
Io_Datum_Type type;
int sim;
)
```

The io_datum structure contains placeholders for handles to all possible types of externally communicated data. It is the responsibility of the physical i/o layer to allocate memory for these and point the event structures to them.

```
phys_io_init(sim,cntrl_sim)
int sim;
int cntrl_sim;
{
vehicle_state_event = io_datum_create(IO_DATUM_VSTATE,sim);
vehicle_command_event = io_datum_create(IO_DATUM_VCMD,cntrl_sim);
range_image_event = io_datum_create(IO_DATUM_IMAGE,sim);
color_image_event = io_datum_create(IO_DATUM_IMAGE,sim);
/*
** Point events to allocated memory.
*/
vehicle_state_event->datum.vs = &(ranger_vehicle->veh_state);
vehicle_command_event->datum.vc = &(ranger_vehicle->veh_cmd);
range_image_event->datum.img = ranger_range_image;
color_image_event->datum.img = ranger_color_image;
/*
** Open hardware communications, if necessary
*/
if(!sim)
 {
 erim_init();
 Cnt_Init("anchor","alonzo");
 }
}
```

The input sim flag is the overall simulation mode. When it is on, physical i/o channels should not be opened. The cntrl_sim flag is used to disconnect actuator command output only.

4.5 Main Event Loop

The organizing principle of the test environment layer is that there is a central event loop called the **main event loop**, which:

- reads vehicle state information at high, roughly synchronous rate and stores it in the vehicle queues for later processing
- reads images from diverse environmental sensors as soon as their data is available
- reads the clock from time to time as needed

The event loop model is used for two reasons:

- determinism can be guaranteed for data logging purposes because all system i/o must pass through it.
- response time can be minimized by an event-reaction model

When an image is available, the event loop calls the map manager to request integration of the image into a single consistent description of the local environment. Perhaps after the image is processed, or in general at any time, the event loop calls the controller to cause it to update the commands being sent to the vehicle.

The architecture is intended to be separable into a number of independent tasks, perhaps implemented on separate processors, at some future time. In particular, the terrain map is exclusively written by perception sensors and read by control modules. It can be implemented in shared memory and perception and control modules may run at different rates.

System dependent issues are managed in the main event loop, so this is a custom loop which may be altered for each application. This loop, and the setup code before it, is the only code fragment which knows the number of sensors, their nature, the associated image processing functions, whether the sensor is mounted on a pan/tilt head, how many communications channels there are,

etc. A typical code fragment for the input portion of an event loop might be as shown below:

```
phys_io_get_next_event(event)
Io_Datum *event;
{
static int sim_switch=0;

if( !simulation_mode )
  {
  while(1)
    {
    if(phys_io_read_vehicle_state(vehicle_state_event->datum.vs))
     {
     *event = range_image_event;
     break;
     }
    else if(phys_io_read_image(range_image_event->datum.img))
     {
     *event = range_image_event;
     break;
     }
    }
  }
else
 {
  if( sim_switch == 0 )
    {
    *event = vehicle_state_event;
    sim_switch = 1;
    }
  else if( sim_switch == 1 )
    {
    *event = range_image_event;
    sim_switch = 0;
    }
  }
```

Only this layer knows the typical sequence of events, so it must remember where it is in the typical sequence. The sim_switch variable is only one way of doing this.

Event soliciting and dispatch are separated to provide the test environment layer an opportunity to intercept events for its many services. Objects are invoked after the test environment invokes the

following function:

```
phys_io_event_dispatch(event)
Io_Datum event;
{
if( event->type == IO_DATUM_IMAGE)
  {
  if( event->datum.img == ranger_range_image )
    {
    map_mgr_process_image(ranger_map_manager,event->datum.img);
    cntrl_cmd_vehicle(ranger_controller);
    tenv_event_dispatch(vehicle_command_event);
    }
 else if( event->datum.img == ranger_color_image )
    {
    map_mgr_process_image(ranger_map_manager,event->datum.img);
    }
 }
else if( event->type == IO_DATUM_VSTATE )
  {
  veh_put_state(ranger_vehicle,event->datum.vs);
  }
else if( event->type == IO_DATUM_VCMD && !event->sim )
  {
  phys_io_command_vehicle(vehicle_command_event->datum.vc);
  }
}
```

4.6 Data Logger

The **data logger** is one service that may transparently intercede between the generation of an event and its dispatch. The data logging facilities are subject to the following simplifying design decisions:

4.6.1 Determinism

The data logger is implemented in the main event loop because it is the only code fragment which is guaranteed to see events in the order in which they occur during recording. A fundamental assumption is that, given the same input, the same output is generated by the system. In a system with multiple asynchronous inputs, they must also occur in the same time sequence in order to ensure the same output. A single data file is used for all data logging in order to ensure that the time sequence is also maintained intact.

4.6.2 Zero State Start-up

It is assumed that whenever the data logger is started for playback, the system is in exactly the same state it was during recording. Hence, the data logger cannot be turned on in the middle of a run because the state is not saved. Only the initial state is guaranteed to be repeatable.

4.6.3 Parameter Constancy

It is assumed that some mechanism outside the data logger will ensure that system parameters are consistent between record and playback. There are system parameters which are not associated with any logged data, so this must be ensured outside the data logger.

4.6.4 File Layout Constancy

Any changes to the layout of logger data files render old data files unusable.

4.7 Simulator

The simulator is another service that may intercede between the generation of an event and its dispatch. If simulation is on, the physical i/o layer generates a pointer to an empty event which is the correct event is its simulation sequence, and the simulator will fill in the event with simulated data. Various simulators are implemented inside their associated objects. The vehicle simulator is necessary to run the system, but sensor and terrain simulation are not.

4.8 Parameter Editor

A C language interpreter is available as a general tool for parsing configuration command strings. This can be very useful for avoiding recompilation in the field and simplifies software configuration control.

4.9 Graphics Tools

Graphics tools are available for displaying images, maps, and vehicles. These are not necessary to run the system.

4.10 Physical I/O Layer

The physical I/O layer binds the real-time kernel to the physical sensors and effectors. The system does not know if network, backplane, or board level communications are involved in this connection. There are a few general principles imposed on the input/output services. This section outlines these requirements.

4.10.1 General Rationale

Image digitization for scanning laser rangefinders may take significant time and the vehicle moves up to 5 meters over this time at 10 m/s. Therefore, the vehicle state must be sampled while this is occurring and the state data must be supplied later on. Some agent close to the position and attitude sensors must write vehicle state information at high frequency to the state communication channel. This data must be queued until the system has time to read it. The physical i/o layer must flush this state queue on every read and the Vehicle will store the data internally. The Vehicle queue may be used for this purpose by synchronous i/o channels because the queues can be accessed while the system is running concurrently.

Image digitization is normally a synchronous process and although the Map Manager may not have time to process all images, they typically overlap on the ground so this is not a serious concern. Some agent close to the image formation hardware must write images to one of the image communication channels as soon as they are generated. The physical i/o layer must flush this image channel on every read and provide only the most recent image to the Map Manager.

4.10.2 Time Tags

All sensory inputs are ideally time tagged at the source. A consistent system time standard is assumed, however it is generated. Consistency of time stamps is necessary within each data stream, and between data streams. This allows the system to register contemporary events in time even though may have arrived at disparate times, and to reason explicitly about delays and their effect on the requirement to maintain high fidelity control.

Images are tagged with the time that image formation commenced. States are tagged with the instant in time to which the states correspond. Tagging agents must account for their own latency in their time tags.

In some cases, any consistent monotonic function of time can be used instead of time. For instance, image frame numbers can masquerade as time provided states are tagged to the same standard.

4.10.3 Coordinate Systems

The world z axis is positive upwards, x to the right, and y forward. Angles are counterclockwise positive according to the right hand rule. ***Euler angles are in z-x-y order*** and this accounts for the nonstandard linear axis conventions.

4.10.4 Unit Systems

In all cases length is expressed in meters. Time is expressed in seconds. Angles are expressed in radians. Derived units such as speed, for example, are derived from these. The signs of curvatures, steer angles and radii of curvature are consistent with the sign of heading. Thus, all are positive for a left turn.

4.10.5 Sample Rate

Sensory inputs must be sampled at the Nyquist rate, and this rate differs for different sensors. The system is not fast enough to do this itself, so i/o channels are responsible for a guarantee of high resolution sampling. Timers hidden in the physical i/o layer can accomplish this by sampling at regular frequency.

The map manager and controller objects copy the entire vehicle data structure at the start of their cycles, so it is admissable to queue data into the vehicle queues while these objects are running concurrently.

5. Tuning and Performance

This section provides some useful rules of thumb for tuning the system controller for a particular vehicle. The importance of tuning cannot be overstated. The system will not work at all unless many of the parameters are set correctly to inform the system of the nature of the vehicle and the sensors. There is a point at which no software can make up for a suboptimal vehicle and sensor suite, but this section provides pragmatic measures which help manage the situation somewhat.

The system meets all safety requirements at 10 mph except for acuity with a 10 mrad acuity range sensor. In theory, roughly a 1 mrad acuity sensor is necessary for a HMMWV. There is a prevailing question about the existence of systematic sensor errors on contemporary vehicle testbeds which also make it impossible to meet the fidelity requirement in practice. The response requirement implies that the most direct limitation on vehicle speed is the useful range of the perception sensor and this implies that achieving speeds beyond about 20 mph for a vehicle mounted perception sensor is a problem of a whole different order of difficulty for both resolution and accuracy reasons.

The following heuristics sound rather complicated, but little can be done. The vehicle goes where it goes, and if the sensor does not provide the data necessary, there is not much that software can do about it. If excessive playing with the numbers is necessary, it suggests an underlying structural problem that is a fundamental limitation. Among other things, it was the observation that there was no way for the ERIM sensor to see where the vehicle goes that prompted the development of RANGER. See [28] for a discussion of the underlying requirements.

5.1 Map Resolution

The map resolution should ideally be chosen to be about half the wheel radius. However, a minimum acuity setup works best because most sensors can only provide this. There is a trade-off between elevation noise and smoothing of legitimate terrain geometric frequencies. In practice, 0.25 meters, then 0.5 meters, then 0.75 meters have become the favorite resolutions for a 10 mrad sensor as speeds have increased. This unfortunately implies that small obstacles cannot be avoided and this arises from sensor limitations. Practically, this has also implied a migration to smooth terrain with only large obstacles. In some scenarios, it is legitimate to physically remove small rocks from the test area.

The sampling problem causes a trade-off between map resolution and system robustness. In general, the map resolution should be chosen based on the target speed, the lookahead which arises from it, then the size of a pixel at that lookahead. As a rule of thumb, choose a map resolution which is as large as can be sustained in the test environment because this also improves the efficiency of the control algorithms and the perception algorithms. The speed/reliability trade-off is managed by choosing the map resolution. Adaptive sweep implies that higher speeds will give more uniform sampling at higher speeds. It also implies, apparently paradoxically, that the software runs faster as the vehicle drives faster because the same field of regard subtends a smaller angle in an image. This behavior results from a fixed angular resolution.

5.2 Image Subsampling

The map is not hierarchical for efficiency reasons, and there is a contemporary problem that sensor pixels are too large and the wrong shape. The vehicle simulator will interpolate a few map cells forward over small holes in the map, so complete coverage is not necessary.

Sometimes, reducing the image column skip provides better coverage, but this is a suboptimal solution because square pixels are really too close together in azimuth. In general an image pixel subsampling equal to the mean value of the sensor height to range over the groundplane projection is a good theoretical subsampling of square pixel image columns. However, this has never been achieved in practice due to the size of the pixel itself. Row skip of 1 and col skip of 1 to 4 have been used at various times. As pixels become smaller than 10 mrad, the theoretical roughly 10:1 ratio should be possible.

5.3 Field of View

It is known that 3m/s vehicles are just about the worst case for requirements on the horizontal field of view of a geometry sensor. However, no existing sensor provides the 120 or so degrees that are necessary to see all of the terrain that a HMMWV can reach. In practice, RANGER must be tuned to a fixed field of view sensor.

5.4 Detection Zone

An optimal strategy is to start with the maximum lookahead distance that can be achieved from the sensor before quantization noise becomes a problem. From this range, subtract the vehicle wheelbase. This number should be the maximum planner range ever computed and it implies a limitation on vehicle speed. At this range, the sensor field of view gives a certain distance in the crossrange direction. By and large, all of the candidate commands should at one time or another fall within the map width at the maximum range minus the width of the vehicle. There is no simple formula that can be given without unjustified assumptions because the underlying relationship is a differential equation that depends heavily on the speed and the initial steering angle.

Too small a field of view implies system failure during turning maneuvers. Long lookahead helps as does fast cycle time, but the feedforward will compute where the vehicle will go, and if the geometry is not known there, the system will stop the vehicle.

Candidate trajectories which regularly drive off the known area of the map will always be discarded by the system, so they waste resources. Artificially confining the trajectories to what the sensor generates, will severely compromise vehicle maneuverability. Conversely, narrow field of view will cause failure to avoid large obstacles solely because it cannot see the “way out”.

When a human sees the way out, and the system does not, the culprit is usually the horizontal field of view, the limited maximum range, or the pitch of the body on rough terrain. Many contemporary vehicle testbeds have **tunnel vision**, **stabilization**, and **myopia** problems.

5.5 Wheelbase & Minimum Turn Radius

The vehicle wheelbase is used in the kinematic transforms of steering, so it cannot be faked without causing other problems. The minimum turn radius affects only planning so it can be increased as far as is necessary until all candidate trajectories fall within the geometry provided by the sensor.

The position of the endpoints of the "arcs"[3] depend heavily on speed, so if they are tuned for one speed, they will not be correct at another. For this reason, it is best not to confine all high speed arcs to the sensor projection by increasing the minimum turn radius, because this limits low speed maneuverability too severely.

Once the minimum turn radius is chosen, the turn lookahead parameter is adjusted so that the planner lookahead keeps the vehicle within the useful bounds of the map. Note that any value less than 1.0 implies that the vehicle will not be able to turn 90 degrees if it sees a large obstacle but sensor performance often requires a value less than 1.0. Note also that the trajectories drawn on the screen reflect the position of the center of the rear axle, so geometry must be known one wheelbase beyond the extent of the arcs on the screen.

5.6 Latency Models

The sensor latency, steering delay, and maximum steering wheel rate numbers all affect the planning window computation and should be reasonably accurate. However, there is ample opportunity to fudge things with other parameters.

5.7 Number of Paths

The number of paths argument supplied to the tactical controller is adjusted finally so that the spacing of the arcs is reasonable. Too close wastes resources and too separated implies inability to drive through narrowly separated hazards.

5.8 Map Size

The maximum range of the planning window provides the basis for choosing the size of the map. Normally, the map should be no smaller than about 30 meters square. Too small a map will cause failure of the **cell aging** mechanism. The vehicle **imaging density** parameter affects the extra width of the range window which is used as a safety margin. A number greater than 1 is always necessary. A value of 2.0 gives more robust, but slower speed navigation.

5.9 Speed Control

The speed control parameter adjusts whether the current speed or the nominal speed is used in the range window computation. If there is no vehicle propulsion controller or if it performs poorly at the usual speed, it is best to turn this bit off and allow the system to use the current speed in its computations.

3. The system response curves are actually similiar to segmented clothoids.

5.10 Hazards

The maximum roll and pitch of hazards are normally set around 15 to 20 degrees. Too high a number risks real rollover and too low a number risks false alarms due to map noise. The controller merit threshold is usually set around 0.4. Too high a number risks real encounters with hazards while too low a number risks a refusal of the system to ever move at all.

5.11 Path Tracking

The tracker lookahead distance is normally about 15 meters with a gain of 5.0. Too high a lookahead implies excessive smoothing of the response to legitimate path frequencies. Too low a lookahead implies instability of the heading servo. Similar arguments apply to the gain in reverse. For stability, the ratio of gain to lookahead is the real parameter of importance. There is a fundamental trade-off of path following fidelity and loop stability.

5.12 Cycle Time

As well as limited sensor range, the overall system cycle time can severely limit performance. The maximum sensor range is reduced by the distance travelled between processed images and the extent of the vehicle. In practice, cycle times below 1 second are necessary for even fairly poor navigation at 3 meters/second. At cycle times over 2 seconds, the whole system is not viable at all with contemporary sensors except at painfully slow (<1 m/s) speeds.

5.13 Graphics

The graphics output is provided for simulation and development purposes and is based on vanilla X calls so it is not computationally efficient. In practice, all but a very small map window is generated in a real-time scenario.

5.14 Sensors and the Environment

Rangefinders tend to generate noisy range data when the spaces between vegetation branches are about the size of a pixel footprint. This is **mixed pixel** problem. Dense vegetation tends to be treated as a solid obstacle. High grass may be treated as a raised terrain surface if it is everywhere dense enough. Usually it is not.

Rangefinders also have problems with **specular reflection** of the laser beam from optically shiny surfaces. Median filters helps both of the above problems, but large unknown geometry regions may still occur. Conversely, the ERIM performs exceptionally well on completely snow covered surfaces if the snow is not shiny.

Stereo perception requires texture throughout the field of view. Some systems cannot compute correspondence on paved roads. Bodies of still water may cause stereo to range to the reflection instead of the surface of the water. Both lidar and stereo can be sensitive to even moderate rain and snow precipitation.

Rough terrain has the same impact as limited field of view because the attitude of the vehicle body points the sensor in useless directions.

5.15 Failure Modes

There is some obstacle size that cannot be resolved at all. With a 10 mrad sensor, it is surprisingly large.

There is some speed that cannot be safely achieved. With 30 meter lookahead, this speed is about 10 mph.

Terrain map fidelity is a problem with a decade of history. It is usual that higher speed work must attempt to succeed with a very poor quality terrain map.

Image registration has problems when the terrain causes excessive vibration of the sensor. Adaptive regard should guarantee that the edge resulting from this is not traversed in feedforward simulation, but other concerns may make this impossible.

There is some angle of incidence to a large obstacle that cannot be achieved. Limited horizontal field of view also causes failure during turns whether or not an obstacle is involved because the sensor is not pointed far enough to image the region of space that can be reached. Artificially limiting maneuverability solves this problem at the expense of causing failure to avoid large obstacles due to limited lookahead.

Playback of recorded paths is sensitive to long term drift of the position estimation system. Without landmark feedback of some sort to damp drift, recorded paths become useless in as little as half an hour.

Dense obstacles may cause failure either due to poor fidelity of the feedforward or due to poor terrain map acuity which fails to see the spaces between obstacles.

Poor attitude indications can cause some sharp grades to appear level in one direction and sheer in the other. The system has at times preferred going downhill to level due to gyro drift and has at times refused to traverse a grade in all but one of two opposite directions. The signal windows provide an internal view of the predicted pitch and roll of the vehicle Normally, the same grade viewed in opposite directions should switch the sign of the signals and the degree to which this is not true is likely bias in the calibration of sensor tilt or vehicle attitude indications.

Wheel collision and body collision hazards are not implemented in some versions of the system.

6. Future Work

This section outlines some ideas for future development.

6.1 Generalized Commands

Currently the system uses a fixed set of constant speed arcs for feedforward. However, the system is intrinsically able to handle arbitrary functions of time as command input signals. Later revisions may investigate the usefulness of S shaped curves and various speed profiles in feedforward. This would allow fast recovery from obstacle avoidance and an ability to function more reliably with a narrower sensor field of view.

6.2 Speed Control

An intelligent propulsion controller may result from the work of other students at CMU. If such a controller required knowledge of the terrain geometry in front of the vehicle, this module could be tightly integrated into RANGER and both systems could share the state space model and simulation facilities.

6.3 Terrain Typing

It should be possible to integrate terrain type images into the terrain map data structure and then generate a vegetation hazard in the tactical controller.

6.4 Learned Arbitration

A central notion of the system is that safety, or alternately, hazard is a multidimensional continuum. There is, as yet, no theoretical basis for the hazard arbitration mechanisms. For example, if 12 degrees pitch is bad, how bad is 8 degrees exactly? Further, does any norm of the hazard vector have more significance than another. Is the manhattan length (L_1 norm), cartesian length (L_2 norm), or maximum projection (L_∞ norm) appropriate? How does the distance from the vehicle reduce the severity of a hazard. These issues seem difficult to manage when combined with the poor quality of terrain maps and perhaps time will permit the investigation of learned arbitration mechanisms to solve the problem.

7. Bibliography

[1] O. Amidi, “Integrated Mobile Robot Control”, Robotics Institute Technical Report CMU-RI-TR-90-17, Carnegie Mellon University, 1990.

[2] M. G. Bekker, “The Theory of Land Locomotion”, The University of Michigan Press, 1956.

[3] M. G. Bekker, “Off-the-Road Locomotion”, The University of Michigan Press, 1960.

[4] R. Bhatt, L. Venetsky, D. Gaw, D. Lowing, A. Meystel, “A Real-Time Pilot for an Autonomous Robot”, Proceedings of IEEE Conference on Intelligent Control, 1987.

[5] W. L. Brogan, “Modern Control Theory”, Quantum Publishers, 1974

[6] R. A. Brooks, “A Hardware Retargetable Distributed Layered Architecture for Mobile Robot Control”, Proceedings of IEEE International Conference on Robotics and Automation, 1987.

[7] B. Brumitt, R. C. Coulter, A. Stent, “Dynamic Trajectory Planning for a Cross-Country Navigator”, Proceedings of the SPIE Conference on Mobile Robots, 1992.

[8] T. S. Chang, K. Qui, and J. J. Nitao, “An Obstacle Avoidance Algorithm for an Autonomous Land Vehicle”, Proceedings of the 1986 SPIE Conference on Mobile Robots, pp. 117-123.

[9] M. Daily, “Autonomous Cross Country Navigation with the ALV”, Proceedings of the 1988 IEEE International Conference on Robotics and Automation, pp. 718-726.

[10] E. D. Dickmanns, “Dynamic Computer Vision for Mobile Robot Control”, Proceedings of the 19th International Symposium and Exposition on Robots, pp. 314-27.

[11] E. D. Dickmanns, A. Zapp, “A Curvature-Based Scheme for Improving Road Vehicle Guidance by Computer Vision”, Proceedings of the SPIE Conference on Mobile Robots, 1986.

[12] J. C. Dixon, “Linear and Non-Linear Steady State Vehicle Handling”, Proceedings of Instn Mechanical Engineers, Vol. 202, No. D3, 1988.

[13] R. T. Dunlay, D. G. Morgenthaler., “Obstacle Avoidance on Roadways Using Range Data”, Proceedings of SPIE Conference on Mobile Robots, 1986.

[14] D. Feng, “Satisficing Feedback Strategies for Local Navigation of Autonomous Mobile Robots”, Ph.D. Dissertation at CMU, May, 1989.

[15] D. Feng, S. Singh, B. Krogh, “Implementation of Dynamic Obstacle Avoidance on the CMU Navlab”, Proceedings of IEEE Conference on Systems Engineering, August, 1990.

[16] U. Franke, H. Fritz, S. Mehring, “Long Distance Driving with the Daimler-Benz Autonomous Vehicle VITA”, PROMETHEUS Workshop, Grenoble, December, 1991.

[17] J. Gowdy, A . Stentz, and M. Hebert, “Hierarchical Terrain Representation for Off-Road Navigation”, In Proc SPIE Mobile Robots 1990.

[18] D. Langer, J. K. Rosenblatt, M. Hebert, “A Reactive System For Off-Road Navigation”, CMU Tech Report

[19] M. Hebert and E. Krotkov, “Imaging Laser Radars: How Good Are They”, IROS 91, November 91.

[20] M. Hebert. “Building and Navigating Maps of Road Scenes Using an Active Sensor”, In Proceedings IEEE conference on Robotics & Automation, 1989; pp.36-1142.

[21] M. Hebert, T. Kanade, and I. Kweon. “3-D Vision Techniques for Autonomous Vehicles”, Technical Report CMU-RI-TR-88-12, The Robotics Institute, Carnegie Mellon University, 1988

[22] B.K Horn and J. G. Harris, “Rigid Body Motion from Range Image Sequences”, Image Understanding, Vol 53, No 1, January 1991, pp 1-13

[23] R. Hoffman, E. Krotkov, “Terrain Mapping for Outdoor Robots: Robust Perception for Walking in the Grass”, Submitted to IEEE International Conference on Robotics and Automation, 1993.

[24] D. Keirsey,D. Payton, J. Rosenblatt, “Autonomous Navigation in Cross-Country Terrain”, proceedings of Image Understanding Workshop, 1988.

[25] A. Kelly, A. Stentz, M. Hebert, “Terrain Map Building for Fast Navigation on Rough Terrain”, Proceedings of the SPIE Conference on Mobile Robots, 1992.

[26] A. J. Kelly, “Essential Kinematics for Autonomous Vehicles”, CMU Robotics Institute Technical Report CMU-RI-TR-94-14.

[27] A. J. Kelly, “Modern Inertial and Satellite Navigation Systems”, CMU Robotics Institute Technical Report CMU-RI-TR-94-15.

[28] A. J. Kelly, “A Partial Analysis of the High Speed Autonomous Navigation Problem”, CMU Robotics Institute Technical Report CMU-RI-TR-94-16.

[29] A. J. Kelly, “A Feedforward Control Approach to the Local Navigation Problem for Autonomous Vehicles”, CMU Robotics Institute Technical Report CMU-RI-TR-94-17.

[30] A. J. Kelly, “Adaptive Perception for Autonomous Vehicles”, CMU Robotics Institute Technical Report CMU-RI-TR-94-18.

[31] A. J. Kelly, “A 3D State Space Formulation of a Navigation Kalman Filter for Autonomous Vehicles”, CMU Robotics Institute Technical Report CMU-RI-TR-94-19.

[32] A. J. Kelly, “Concept Design of A Scanning Laser Rangefinder for Autonomous Vehicles”, CMU Robotics Institute Technical Report CMU-RI-TR-94-21.

[33] In So Kweon, “Modelling Rugged Terrain by Mobile Robots with Multiple Sensors”, CMU PhD Thesis, 1990

[34] T. Lozano-Perez, and M. A. Wesley, “An Algorithm for Planning Collision Free Paths Among Polyhedral Obstacles”. Communications of the ACM, Vol. 22, Num. 10, October 1979, pp. 560-570.

[35] M. Marra, R. T. Dunlay, D. Mathis, “Terrain Classification Using Texture for the ALV”, Proceedings of SPIE Conference on Mobile Robots, 1988.

[36] L. S. McTamaney, “Mobile Robots: Real-Time Intelligent Control”, IEEE Expert, Vol. 2, No. 4, Winter, 1987.

[37] H. P. Moravec, “The Stanford Cart and the CMU Rover”, Proceedings of the IEEE, Vol. 71, Num 7, July 1983, pp. 872-884.

[38] K. Olin, and D. Tseng, “Autonomous Cross Country Navigation”, IEEE Expert, August 1991, pp. 16-30.

[39] D. A. Pomerleau, “Efficient Training of Artificial Neural Networks for Autonomous Navigation”, Neural Computation, Vol. 3, No. 1, 1991, pp88-97.

[40] J. Rosenblatt,D. Payton, “A Fine-Grained Alternative to the Subsumption Architecture”, Proceedings of the AAAI Symposium on Robot Navigation, March, 1989.

[41] S. Singh et. al, “FastNav: A System for Fast Navigation,”, Robotics Institute Technical Report CMU-RI-TR-91-20, Carnegie Mellon University, 1991.

[42] S. Shafer, A. Stentz, C. Thorpe, “An Architecture for Sensor Fusion in a Mobile Robot”, Proceedings of IEEE International Conference on Robotics and Automation”, April, 1986.

[43] H. C. Stone, “Design and Control of the Mesur/Pathfinder Mocrorover”, JPL

[44] D. H. Shin, S. Singh and Wenfan Shi. “A partitioned Control Scheme for Mobile Robot Path Planning”, Proceedings IEEE Conference on Systems Engineering, Dayton, Ohio, August 1991

[45] A. Thompson, “The Navigation System of the JPL Robot”, Proceedings of the International Joint Conference for Artificial Intelligence, 1977, pp749-757.

[46] P. J. Ward & S. J. Mellor, “Structured Development for Real-Time Systems”, Yourdon Press.

[47] B. Wilcox et al. “A Vision System for a Mars Rover. SPIE Mobile Robots II”, November 1987, Cambridge Mass., pp. 172-179.

Index